CS452: Real-Time Programming

Kernel (Part 3): Clock Sever and Idle Task

January 30, 2020

Yuhui Pan
20868342

Leanora McVittie
20513649

Usage

The git repository can be found at https://git.uwaterloo.ca/lmcvitti/trains_kernel, and the exact commit hash will be emailed to Professor Karsten; it can also be found as the final commit on the `k3` branch in that repository.
The executable file, `k3.elf`, can be compiled by cloning the above repository and then running `make` in the root directory. It can be run by loading it into RedBoot and run using RedBoot's `go` command.
A pre-existing executable, `k3.elf` can be found in `/u/cs452/tftp/ARM/y247pan/` and can also be loaded into RedBoot and run.
The executable will run until all four user tasks have been delayed and printed out their delay time and ticks. No further user input is needed to trigger this. Once all the delaying tasks have finished running, the kernel will exit.
The idle time percentage will be printed in the top right corner of the terminal. We expect that the terminal window will be at least 96 columns wide in order to view the idle percentage. Once the kernel exits, the RedBoot prompt will push the idle percent counter off the top of the terminal, so the user will have to scroll up to see it.

Clock Server

Execution

Similarly to the Name Server implemented in the last assignment, the Clock Server executes within an infinite while loop. On each iteration of the loop, it calls `Receive()` and waits for a task to want to send a message to it. After receiving a message, it determines what type the message is, and then executes accordingly. If the message is a `TIME` request, it simply replies with the value of its internal ticks counter. If the message is a `DELAY` request, it calculates the tick value when that task should be replied to, and then adds the task id to the queue of delayed tasks. It does the same for a `DELAY_UNTIL` request, except that instead of calculating the delayed tick value, it checks that the delay time is greater than the current tick count, and then sets the requested time as the tick value, and adds the task id to the delayed tasks queue. Finally, if the message is a `TICK` request, then it is from the clock notifier, we then update the value of the internal ticks, and reply back to the clock notifier to unblock it. We release any tasks that are on the delayed priority queue waiting for this particular time, sending them the current number of ticks. The server expects that only the Clock Notifier will send `TICK` requests.

Data Structures

Clock Messages

Clock messages are sent as a struct with two integers: the message type, and a time field. The message type is an enum, which defines the type of message. This is the field that that clock server checks to determine how it needs to respond to the message. The time field is used both to send tick amounts to the clock server, and for the server to send the current times back to the sending tasks.

Min Heap

The tasks that have been delayed and are waiting for their ticks to be up are stored in a min-heap, which is implemented as two arrays, one which orders the task ids as a min-heap (sorted by their wake time), with the children of the task at index i stored in `i*2` and `i*2+1`, and the other of which is a mapping from the ids to their wake times. The `add` and `remove` methods automatically keep the minheap sorted. We generalize priority queue from k1 by adding function pointers to make it more flexible. This implementation was already discussed in k1 as our implementation of the ready queue, so we won't go into it in detail here.

Clock Notifier

The clock notifier is a very simple task, started by the clock server, which simply loops endlessly, first waiting for a timer event, which is set up (in the kernel initialization) to trigger an interrupt every 10ms, and then sending a message to the clock server to inform it of the update.

Interrupt Handler

The interrupt handler checks that the interrupt that has been triggered is a timing interrupt; if it isn't then it returns. This can easily be expanded to include more interrupt types and checks as more interrupts are needed. For this assignment we only need the timing interrupts. Then, if the interrupt is for a timing event, all tasks that are waiting on the `TIMER_EVENT` queue are woken up and put back on the ready queue. Finally the timer interrupt is cleared.
The interrupt handler registration process during the setup is similar to that of the swi handler except that the memory location is `0x38` instead of `0x28`. The details of entering interrupt can be found in `src/kernel/swi.S`, `enter_interrupt` section. When entering the interrupt assembly routine, we store the user registers first by going into system mode, we then switch interrupt mode to save the `lr`/return address. An important thing here is to get `lr` by `lr - 4` instead of lr itself because of the pipelining effect of architecture. And we also have to store `spsr_usr` and `lr_usr`. And go back to supervisor mode and change `pc` to enter kernel.

Tasks that want to wait for an event, can call `AwaitEvent()`, which enters kernel mode, and the kernel puts the task into the `EVENT_WAIT` state and adds them to the queue of the event they wish to wait for. If the event they are waiting for is unrecognized, the task is placed back on the ready queue with an error code as its return value.

Idle Task

The idle task sets itself up by first unlocking the `DeviceCfg` location (`0x80930080`) by setting a flag in the `SysSWLock` location (`0x809300C0`), and then setting a flag in the DeviceCfg location which allows the `Halt` location (`0x80930008`) to be read. Once those are set up, the idle task enters an infinite while loop which simply reads from the `Halt` memory location once per loop. Reading from this location puts the CPU into low power mode until an interrupt is triggered.
The idle percentage counter computes the percentage of time that the CPU spends within the idle task (halted) during the execution of the program, and outputs it to the terminal. In the main kernel loop, immediately before a task is switched to and immediately after a task is switched from, the `task_performance` function is called. This function determines how much time it has been since the function was last called, and then adds that to the `total_ticks` counter. If the most recent run task was the idle task, it also adds that time to the `idle_ticks` counter. It then computes and stores the tick information of the idle task and kernel. The percentage of time that the program has spent in the idle state is computed by dividing the `idle_ticks` by the `total_ticks`.

Output

This output analysis does not include the idle counter, as that discussion is within the Idle Task section of this report.

```
Task Id <4>, Delay Interval <10>, Delay Number <1>, Ticks <10>
Task Id <4>, Delay Interval <10>, Delay Number <2>, Ticks <20>
Task Id <5>, Delay Interval <23>, Delay Number <1>, Ticks <23>
Task Id <4>, Delay Interval <10>, Delay Number <3>, Ticks <30>
Task Id <6>, Delay Interval <33>, Delay Number <1>, Ticks <33>
Task Id <4>, Delay Interval <10>, Delay Number <4>, Ticks <40>
Task Id <5>, Delay Interval <23>, Delay Number <2>, Ticks <46>
Task Id <4>, Delay Interval <10>, Delay Number <5>, Ticks <50>
Task Id <4>, Delay Interval <10>, Delay Number <6>, Ticks <60>
Task Id <6>, Delay Interval <33>, Delay Number <2>, Ticks <66>
Task Id <5>, Delay Interval <23>, Delay Number <3>, Ticks <69>
Task Id <4>, Delay Interval <10>, Delay Number <7>, Ticks <70>
Task Id <7>, Delay Interval <71>, Delay Number <1>, Ticks <72>
Task Id <4>, Delay Interval <10>, Delay Number <8>, Ticks <80>
```

```
Task Id <4>, Delay Interval <10>, Delay Number <9>, Ticks <90>
Task Id <5>, Delay Interval <23>, Delay Number <4>, Ticks <92>
Task Id <6>, Delay Interval <33>, Delay Number <3>, Ticks <99>
Task Id <4>, Delay Interval <10>, Delay Number <10>, Ticks <100>
Task Id <4>, Delay Interval <10>, Delay Number <11>, Ticks <110>
Task Id <5>, Delay Interval <23>, Delay Number <5>, Ticks <115>
Task Id <4>, Delay Interval <10>, Delay Number <12>, Ticks <120>
Task Id <4>, Delay Interval <10>, Delay Number <13>, Ticks <130>
Task Id <6>, Delay Interval <33>, Delay Number <4>, Ticks <132>
Task Id <5>, Delay Interval <23>, Delay Number <6>, Ticks <138>
Task Id <4>, Delay Interval <10>, Delay Number <14>, Ticks <140>
Task Id <7>, Delay Interval <71>, Delay Number <2>, Ticks <143>
Task Id <4>, Delay Interval <10>, Delay Number <15>, Ticks <150>
Task Id <4>, Delay Interval <10>, Delay Number <16>, Ticks <160>
Task Id <5>, Delay Interval <23>, Delay Number <7>, Ticks <161>
Task Id <6>, Delay Interval <33>, Delay Number <5>, Ticks <165>
Task Id <4>, Delay Interval <10>, Delay Number <17>, Ticks <170>
Task Id <4>, Delay Interval <10>, Delay Number <18>, Ticks <180>
Task Id <5>, Delay Interval <23>, Delay Number <8>, Ticks <184>
Task Id <4>, Delay Interval <10>, Delay Number <19>, Ticks <190>
Task Id <6>, Delay Interval <33>, Delay Number <6>, Ticks <198>
Task Id <4>, Delay Interval <10>, Delay Number <20>, Ticks <200>
Task Id <5>, Delay Interval <23>, Delay Number <9>, Ticks <207>
Task Id <7>, Delay Interval <71>, Delay Number <3>, Ticks <214>
```

A few things about this output: the `Task Id` is the id of the task that was delayed and is printing the line. The `Delay Interval` is the amount of time that task is delaying for. The `Delay Number` is the delays that the task has completed, so it counts up incrementally for each task. Finally, `Ticks` is the tick count that the task was awakened from its delay on. From this output it is clear that each task executes a certain interval of delay a certain number of times.
The most interesting part of this output is that for the tasks with delays `10`, `23`, and `33`, the ticks on which they are woken up is the multiple of the `Delay Interval` and the `Delay Number`. However, for the task with the `71` tick delay, the tick value is that multiple plus `1`. This is because the first request that that task makes, to delay `71` ticks, occurs after `10` milliseconds from the clock server's initialization have already passed. Therefore, it requests a `71` tick delay when the ticks are at `1`, not `0`, and so the time to wake up the task is calculated as `72` ticks. This delay effect continues through the rest of the delay requests from this task, and so each tick count is off by one.