Authors

Sebastian Gaweda (20143503) and Shashwat Pratap (20638036)

Make and Run

checkout - git clone https://git.uwaterloo.ca/sgaweda/cs452-rtos-aka-trains.git
make - cd cs452-rtos-aka-trains/k3/ && make
run - in gtk terminal run - load ARM/{user-name}/k3.elf; go

Description of Program

Interrupt Handling

The first major change in k3 was a massive overhaul of the SWI context switch. Global "registers" (c variables) were removed, and all important information is now stored on the user stack. The SWI context switch now saves all the registers of the program that triggered the SWI, which makes it possible to use the same exit when returning from a hardware interrupt. The hardware interrupt handler is of largely the same form as the SWI context switch with some minor differences in what modes the handler switches to and what value it pushes onto the stack as a flag for the kernel. This flag is used by the kernel to determine if a hardware interrupt has occurred so that it can handle it appropriately.

Clock Server

One of the first things required for the Clock Server was AwaitEvent(EventId). AwaitEvent(...) is a Software Interrupt call, which takes in an EventId and based on this EventId, stores the caller on a queue. This queue contains all the tasks waiting on the same event / interrupt to occur, to continue their execution. Once the event / interrupt occurs, all the tasks on the queue are popped and their execution state is set to Ready and they are pushed on the Ready Queue. A RingBuffer is used for creating the queue to store tasks waiting on the same event.

The Clock Server creates a Clock Notifier which runs at the highest priority.

First thing the Clock Notifier does is start one of the timers (we used timer 1) and sets it to run at 2KHz and sets the initial value of the clock to 20 and makes it run in periodic mode. This gives the kernel periodic interrupts every 10 ms which is the resolution of our system clock. Once it sets this clock up, Clock Notifier goes into an infinite loop where it keeps Awaiting on Interrupts generated by timer 1. Whenever it resumes its execution after

the AwaitEvent, it sends a message to the Clock Server to signify that a tick has passed. On receiving a reply it goes back to waiting on Timer Interrupts.

The Clock server runs on priority 1, and whenever it Receives a message from Clock Notifier it increments an internal variable which stores how many ticks have passed and Replies back to the Clock Notifier to let the Clock Notifier Await on a timer interrupt. It handles 3 types of inputs other than the input from Clock Notifier.

It offers functionality for Time, Delay and DelayUntil syscalls. All 3 of these functions are wrappers for Send to the Clock Server. When Clock Server gets a Send message from a task for Time, it replies with the internal count of ticks that have passed (which it tracks and increments every time it gets a Send message from the Clock Notifier).

The final 2 functions, Delay and DelayUntil are quite similar in the terms of what happens when they are called. Both Delay and DelayUntil send to the Clock Server the number of ticks they want to sleep for / sleep until. The Clock Server maintains a RingBuffer where it stores a list of the tasks which want to sleep for certain ticks. The clock server takes the ticks from the Delay and DelayUntil call and waits for the specified number of ticks to pass before replying to the task which called Delay / DelayUntil.

Halting

When no tasks are ready, the halt program is queued in the kernel. When the kernel sees that the halt program is about to execute, it captures timing data using timer 3 and reports idle time statistics to the terminal. We wanted to have the idle task be entirely responsible for this, but making this task kernel controlled greatly simplified the logic around timing idle time.

Program Output

Output

1. TID: 4 DELAY INTERVAL: 10 delays: 1
2. TID: 4 DELAY INTERVAL: 10 delays: 2
3. TID: 5 DELAY INTERVAL: 23 delays: 1
4. TID: 4 DELAY INTERVAL: 10 delay: 3
5. TID: 6 DELAY INTERVAL: 33 delays: 1
6. TID: 4 DELAY INTERVAL: 10 delays: 4
7. TID: 5 DELAY INTERVAL: 23 delays: 2
8. TID: 4 DELAY INTERVAL: 10 delays: 5

9. TID: 4 DELAY INTERVAL: 10 delays: 6
10. TID: 6 DELAY INTERVAL: 33 delays: 2
11. TID: 5 DELAY INTERVAL: 23 delays: 3
12. TID: 7 DELAY INTERVAL: 71 delays: 1
13. TID: 4 DELAY INTERVAL: 10 delays: 7
14. TID: 4 DELAY INTERVAL: 10 delays: 8
15. TID: 5 DELAY INTERVAL: 23 delays: 4
16. TID: 4 DELAY INTERVAL: 10 delays: 9
17. TID: 6 DELAY INTERVAL: 33 delays: 3
18. TID: 4 DELAY INTERVAL: 10 delays: 10
19. TID: 4 DELAY INTERVAL: 10 delays: 11
20. TID: 5 DELAY INTERVAL: 23 delays: 5
21. TID: 4 DELAY INTERVAL: 10 delays: 12
22. TID: 6 DELAY INTERVAL: 33 delays: 4
23. TID: 4 DELAY INTERVAL: 10 delays: 13
24. TID: 5 DELAY INTERVAL: 23 delays: 6
25. TID: 7 DELAY INTERVAL: 71 delays: 2
26. TID: 4 DELAY INTERVAL: 10 delays: 14
27. TID: 4 DELAY INTERVAL: 10 delays: 15
28. TID: 5 DELAY INTERVAL: 23 delays: 7
29. TID: 6 DELAY INTERVAL: 33 delays: 5
30. TID: 4 DELAY INTERVAL: 10 delays: 16
31. TID: 4 DELAY INTERVAL: 10 delays: 17
32. TID: 5 DELAY INTERVAL: 23 delays: 8
33. TID: 4 DELAY INTERVAL: 10 delays: 18
34. TID: 4 DELAY INTERVAL: 10 delays: 19
35. TID: 6 DELAY INTERVAL: 33 delays: 6
36. TID: 4 DELAY INTERVAL: 10 delays: 20
37. TID: 5 DELAY INTERVAL: 23 delays: 9
38. TID: 7 DELAY INTERVAL: 71 delays: 3

Explanation

The bootloader creates NameServer, ClockServer and the 4 ClockClients into existence, and the kernel creates the idletask. The ClockServer creates the ClockNotifier which in turns starts the clock and timer interrupts. After that the 4 clock clients start Delaying themselves for certain specified periods of times:

1. The task with priority 3 Delays 20 different times each time for 10 ticks.

2. The task with priority 4 Delays 9 different times each time for 23 ticks.

3. The task with priority 5 Delays 6 different times each time for 33 ticks.

4. The task with priority 6 Delays 3 different times each time for 71 ticks.

The order in which they print signifies when their delay times get over and going through it, and counting the ticks which would have happened, we can make sense of the data. For example, we get 7 delays of 10, 3 delays of 23 and 2 delays of 33 before we get 1 delay of 71.