1 Administrivia

2 Sample Output

Please run with `go -c`.

2.1 Timings

Syscall	No Caches (`go`)	Caches (`go -c`)
`Create()`	217 μs	16 μseconds
`MyTid()`, `MyParentTid()`, `Pass()`	157 μs	14 μs
`SRR` 4 bytes	492 μs	39 μs
`SRR` 64 bytes	1100 μs	79 μs

3 REAME

See `docs/`. Our extra commands include “q” to return to RedBoot, “reboot” to reboot the ARM board, “honk n” to make a train honk, and “lights n” to turn train headlights on and off.

4 Kernel Description

4.1 Kernel

Our kernel’s `main` function is organized as follows:

```
bootstrap ()

while (there is a task to run) {
  req = activate (activet_ask)
  req = handle(req, task_descriptor)
  active_task = schedule ()
}
```

`activate` is our context switch and exception handler. It will restore a user’s state and then jump to user code. When a software or hardware interrupt is triggered by a system call the cpu will jump back into the code for activate. The user state is saved and the kernel’s is restored. The value returned by activate is a flag indicating whether or not we have come from a system call or an interrupt. We then extract the system call number from the the user’s code or examine the active interrupts, respectively.

`handle` is really a switch statement, and takes the system call opcode and the relevant information for this task and does the appropriate thing. We have implemented all system calls as described in the kernel specification document.

`schedule` will take the current task and make its state Ready if its current state is Active. If not, it leaves the task in a blocked state. Next, it will pick the highest priority task to run. This is described in more detail in the following section.

4.2 Scheduling

The type of scheduling used is round robin with priority levels. The task which schedule chooses is garunteed to be in the state Ready and of the highest possible priority given that. The task descriptors have a next pointer in them of type pointer-to-task-descriptor. In this way a group of tasks can be stored in a singly-linked list. The task management system uses a pair of arrays of pointers to task descriptors. The first array, p, points to the last activated task of a given priority. The second array,head, points to the most recently created task of a given priority. In this way we are able to have circular buffers. The `schedule` function starts at priority 0. If at any point in any priority it finds a task in the Ready state it sets the state to Active, updates the p pointer for that priority (for round robin scheduling) and returns a pointer to this task. If it cannot find a task of priority 0 that is `READY` it then tries level 1 and so on until it has exhausted all levels. If there is no task available to run, schedule will return 0 and the main kernel loop will exit, causing the kernel to exit and RedBoot to take over. However in practice this will never happen, as we have an idle task running at the lowest priority in an infinite loop. There are 6 priority levels. The justification for each level is given below. The names are considered self explainatory. We have yet to find a need for new priority levels.

```
enum PRIORITY {
  INTERRUPT = 0,
  SYSCALL_HIGH = 1,
  SYSCALL_LOW = 2,
  USER_HIGH = 3,
  USER_LOW = 4,
  IDLE = 5,
};
```

4.3 Task States

As described in the Kernel Description document, we have the following possible states for tasks:

```
enum STATE {
  ACTIVE,
  READY,
  DEFUNCT,
  SEND_BLOCKED,
  RECEIVE_BLOCKED,
  REPLY_BLOCKED,
  EVENT_BLOCKED,
};
```

4.4 Message Passing

To temporarily store messages sent between tasks we added some queueing variables to the Task Descriptors. The heart of the queues is the `struct mq` which holds the length of a message, a pointer to a message and the relevent TID. For messages sent to a task, there is a circular buffer (implemented with an array and two index variables) of size `MQSIZE = 10`. For messages sent as replies from a recipient to the sender there is a single `struct mq` for holding the address and size of the sender's reply buffer. We are aware that the assignment says that messages should be copied into the recipient's address space. However, we choose to interpret this literally: all tasks share the same address space, so saving the messages anywhere will satisfy the condition. We are confident that this will not cause errors because a sending task is blocked until the message is received, therefore the recipient will have a chance to copy the message before the sender could corrupt it. Similarly, if a sender is reply-blocked, it will be restored to the ready state as soon as a single task replies to it. Thus only a single-sized queue is necessary.

4.5 Events

For `AwaitEvent()`, we chose to change our kernel and to make interrupts disabled when the notifier is called and for only one task to be able to wait on any single event. Therefore the event queue can be represented by an array of pointers to TD structs. When a task is waiting on the event a pointer to its TD struct will be in the queue. The pointer is removed when the task is woken up. For the delay queues in `Delay()`, two seperate mechanisms are needed. First, to support n tasks being delayed at once, `MAXTASKS` "delay" structures are allocated statically. These are then setup as a ring buffer of "dynamically allocatable" nodes. This will be improved in the future when we find more time. As a task gets delayed, a structure is allocated and inserted into a min-heap, taking $O(\log n)$ time. Sadly this takes O(n) time. Tasks are popped off the heap with the same time complexity upper-bound. This means that when the clock ticks, Delay can compare the current time and the wakeup time of the first node in the heap and pop it off accordingly, possibly popping multiple delayers. The downside to this allocation scheme is that if there is a single long-delaying task, and many shorter delaying tasks, we may wrap around the ring buffer of list nodes and reusing the long sleeping tasks's list node.

4.6 Interrupts

`activate` starts a user task and through careful use of the `movs` and `ldm/stm` version 3 (load/store from/to user mode registers from supervisor mode) instructions we change to user mode in an atomic fashion. This is to prevent an edge case where an interrupt occurs before all of the user state is restored. We `movs pc, lr` on line x. At line $x + 1$ we have our IRQ handler. This carefully changes to supervisor mode (making sure to transport `SPSR_irq` and `lr_irq`) A magic byte is stored in a magic location. Now the code “falls through” to the regular system call handler. We process things as if it was a regular system call (this is where our code goes on such a system call.) This code passes the value stored at the magic location to the kernel and sets it to 0. The kernel can tell that it is processing an interrupt by whether the magic byte is 0 or not. The context switch is under 20 instructions of code for a regular system call.

4.7 Servers and Task Structure

There is one serial server for each UART. Each has different clients, and different performance profiles due to the speeds of the UARTs. While each server has a similar overall structure, each is different in details. This makes each server smaller, and lets us turn on BWIO for a single UART if we need or want to debug our code. The train server directly handles all train related communication, and exposes data to user clients. The Window Manager deals with all output to the screen.

4.8 Notifiers

We decided it was a better design choice to leave interrupts off when running our notifiers. This leaves our kernel virtually free of hardware specific code apart from some timing code and identifying the nature of interrupts. We have a notifier for every kind of interrupt, for a total of 5.

4.9 Servers

Need more here?