CS 452 Kernel 2

Louis A. Burke (laburke) and Taras Kolomatski (tkolomat)

June 22, 2017

Overview

Early childhood development is markedly impacted by the name given to a child by its parents. This moniker is imbued on the child not by any material process, but by repeated, direct, and structured communication. While a child may make noise to communicate to anyone who will listen, the adults utilize these sigils to directly and efficiently pass ideas to each other. By means of repeated messages, parents emboss a child's name on it from an early age. When the child has grown they have internalized their handle to such a degree that it becomes tied to their identity. Humanity is thus naming and messaging.

Our realtime kernel now also supports naming, making use of an underlying messaging system to set the names of tasks and associate them with their task IDs.

Structure

Message Passing. The message passing functionality is merely added to the existing scheduler functionality. Negative priorities are now used to indicate blocked tasks, so that the scheduler knowns not to reschedule them when it ignores them on the top of the active queue. In order to know if a blocked task has to be rescheduled or not, the scheduler marks each task with whether it is currently in the scheduling system or not.

As for message passing itself, each task descriptor now contains a queue of tasks waiting to send to it. When it calls receive it will accept one of those tasks and advance the queue. If the queue is empty it will block until it is not.

As for sending, tasks merely contain two pointers to buffers that can be written to by other tasks. Replying is pretty similar to send, except it also has to unblock the target.

The way we setup our kernel interface, kernel commands can only take 3 arguments. This allows all arguments to be passed in registers, but is not enough arguments for the message passing calls. To solve this we had two options. The first was to arbitrarily wrap all arguments in a struct and pass a pointer to it, but that seemed clunky. Instead we created a buffer utility type. By replacing buffer/size pairs with a single buffer pointer we reduce even send to just three arguments. This also allows us to write the optimized and advanced (truncation checks) buffer copier in a separate module, with minimal overhead.

Name Server. Our nameserver is just a user task like any other. It is created early and takes up only a very small task space, but runs at high priority. The name server is fairly rudimentary, as it doesn't need to deal with a large dynamic load.

It consists of a buffer of stored names and their associated task ids. This buffer is filled from the top down with names as they are reported to the task. Instead of searching for old names to replace in the buffer, newly registered tasks merely continue to write names lower in the buffer. Meanwhile when searching for a name, the buffer is traversed from low to high, in reverse order. In this manner the newer names will always register before the older ones.

While this leaves us with a fixed total number of "RegisterAs" calls, the number we are allowed is easily modified at compile-time. Since we can know in advance exactly how many named services we will need, we can precompute this value reliably.

Game Task Priorities

The server was given a priority of 1, while the clients were given priorities of 4 so as to allow the server time to register itself and setup its state. Servers can in general be a higher priority task than the tasks they serve since they spend most of their time blocked. Their runtime is simply an extension of that of the tasks they serve.

Results

Length	Caches	SRR	O2	Time
4	off	yes	off	265
64	off	yes	off	401
4	on	yes	off	19.8
64	on	yes	off	29.0
4	off	no	off	356
64	off	no	off	478
4	on	no	off	25.8
64	on	no	off	35.0
4	off	yes	on	101
64	off	yes	on	139
4	on	yes	on	8.07
64	on	yes	on	10.5
4	off	no	on	143
64	off	no	on	176
4	on	no	on	10.7
64	on	no	on	13.2

Based on the reward for optimizing different sections of the code, we believe that the main performance bottlenecks likely reside in three locations.

The first, and hardest to mitigate, is in data copying. When optimization is turned on, we over-copy. We assume everything is aligned to 4-byte boundaries and copy as such. This leads to some bugs when optimizations are enabled, but provides a significant speed up by avoiding an unnecessary branch and divide (checking the modulo of the size).

Second is the scheduler. When a real-time task sits active it is very fast, however whenever any task with a lower priority needs to run, we have to shift the scheduler stack. This shift was very costly when there were a full 32 priorities, so we reduced them to just 8. The effect was pretty small, but still noticeable.

Finally mode switching itself is still a bottleneck. Despite involving mostly hand-written assembly, these context switches happen so frequently and incur such frequent cache misses, that they are a large amount of time sink. In order to improve this, we optimized a few key instructions. We also inspected the elf file to see that the assembly instructions were being placed far away from the rest of the instructions. This lead to a full organization of the linker script in order to place critical functions near each other, and long functions near the end.

Game Task Output

The output of the game is:

```
src/tasks/rps_client.c:27        Joined
src/tasks/rps_client.c:27        Joined
Signed up!
Signed up!
P1: R
P2: S
p2: I lost.
```

```
P2: P
P1: I won!
Press any key to continue...
P1: P
p1: Great minds think alike.
Press any key to continue...
P1: S
p2: Great minds think alike.
P2: P
p2: I lost.
P2: Q
p1: I won!
Press any key to continue...
P1: R
src/tasks/rpsserver.c:94        QUIT
p1: Sore loser.
Press any key to continue...
```

Note that first the clients join and sign up using the name server and the game server (name server debug shown as expanded debug message). Next they play a few rounds, announcing their moves (capital letters) and sending them to the server. When they get a response they parse it and emote as appropriate. They don't get rescheduled again until they send their next move to the server, so their moves come immediately after they find out what happened. The three cases of winning, losing and tying are shown. Finally one of the tasks quits. The other task continues playing but the server notifies it that the other player has quit (again shown using expanded debug messages). The result is that the player also quits, ending the program.

SHA of commit

The commit hash of the submission commit (which is on master) is:
`5d9b5b7e633d551df48fc81ebe0cf06ad2072b4c`
The repository can be cloned from:
`gitlab@git.uwaterloo.ca:laburke/cs452_kernel.git` (SSH)
or
`https://git.uwaterloo.ca/laburke/cs452_kernel.git` (HTTPS)