Kernel 4

CS452 - Spring 2014

Real-Time Programming

Team
Max Chen - mqchen
mqchen@uwaterloo.ca

Ford Peprah - hkpeprah
ford.peprah@uwaterloo.ca

Bill Cowan
University of Waterloo
Due Date: Monday, 23rd, June, 2014

Table of Contents

1 Program Description

1.1 Getting the Program

To run the program, one must have read/write access to the source code, as well as the ability to make and run the program. Before attempting to run the pogram ensure that the following three conditions are met:

- You are currently logged in as one of `cs452`, `mqchen`, or `hkpeprah`.
- You have a directory in which to store the source code, e.g. `~/cs452_microkern_mqchen_hkpeprah`.
- You have a folder on the FTP server with your username, e.g. `/u/cs452/tftp/ARM/cs452`.

First, you must get a copy of the code. To to this, log into one of the aforementioned accounts and change directories to the directory you created above (using `cd`), then run one of

```
git clone file:////u8/hkpeprah/cs452-microkern -b kernel4 .
                              or
git clone file:////u7/mqchen/cs452/cs452-microkern -b kernel4 .
```

You will now have a working instance of our kernel4 source code in your current directory. To make the application and upload it to the FTP server at the location listed above (`/u/cs452/tftp/ARM/YOUR_USERNAME`), run `make upload`.

1.2 Running the Program

To run the application, you need to load it into the RedBoot terminal. Ensure you've followed the steps listed above in the "Getting the Program" settings to ensure you have the correct directories and account set up. Navigate to the directory in which you cloned the source code and run `make upload`. The uploaded code should now be located at

```
/u/cs452/tftp/ARM/YOUR_USERNAME/assn4.elf
```

To run the application, go to the RedBoot terminal and run the command

```
load -b 0x00218000 -h 10.15.167.4 ``ARM/YOUR_USERNAME/assn4.elf''; go
```

The application should now begin by running through the game tasks before reaching a prompt. The generated files will be located in `DIR/build` where `DIR` is the directory you created in the earlier steps. To access and download an existing version of the code, those can be found at `/u/cs452/tftp/ARM/mqchen/assn4.elf` and `/u/cs452/tftp/ARM/hkpeprah/assn4.elf`.

1.2.1 Bootup Sequence

Before the program can actually begin accepting input from the user, it has to put the kernel into a state that is ready to begin working. To this end, the boot sequence of the application does the following sequentially

1. Turns on the cache.
2. Initializes UART2.
3. Initializes UART1.
4. Initializes the debugging and display interface.
5. Initializes the memory.
6. Initializes the task handling.
7. Sets up the SWI handler.
8. Initialzies the logging system.
9. Turns on the train set.
10. Writes the display to the screen.
11. Seeds the pseudo random number generator.
12. Enables interrupts.

These are done using busy-wait before the application has been running any of the user tasks and moves the kernel into a state ready to begin running tasks.

1.2.2 Shutdown Sequence

The application does not terminate until the user issues a shutdown command ('q' on the shell). The purpose of the shutdown operations is to leave the kernel into a clean state such that another application can be reloaded without having to issue a hardware reboot to the box. To shutdown, the application busy-waits performing the following tasks sequentially

1. Disable interrupts.
2. Turns off the train set.
3. Clears any remaining tasks.
4. Disablees the idle timer.
5. Dumps the log.

1.2.3 Command Prompt

After the startup tasks have finished running, the user will reach a command prompt where they will be able to enter commands. A list of available commands the syntax are listed below in the `Shell` section.

2 Kernel Structure

3 System Primitives

System Call	Prototype	Description
Create	`int Create(int, void(*)())`	Creates a task with the specified priority to run the given code.
MyTid	`int MyTid()`	Returns the task ID of the calling task.
MyParentTid	`int MyParentTid()`	Returns the task ID Of the parent task of the calling task.
Pass	`void Pass()`	Calling task gives up control to another task of the same priority.
Exit	`void Exit()`	Calling task exists.
WhoIs	`int WhoIs(char*)`	Queries nameserver for task ID of task with the given name.
RegisterAs	`int RegisterAs(char*)`	Registers the given name to the TID of the calling task with the nameserver.
UnRegister	`int UnRegister(char*)`	Unregister the current task iff its name and tid match what is in the NameServer.
Send	`int Send(int, void*, int, void*, int)`	Sends a message to the specified task. Blocks on Reply.
Receive	`int Receive(int*, void*, int)`	Calling task blocks until it receives a message.
Reply	`int Reply(int, void*, int)`	Replies to the specified task with the given message.
Log	`int Log(const char*, ...)`	Logs a variable length formatted message to the logger.
Logn	`int Logn(const char*, n)`	Logs a fixed length message to logger.
AwaitEvent	`int AwaitEvent(int eventType)`	Blocks until the event identified occurs then returns.

WaitTid	`int WaitTid(unsigned int tid)`	Waits until the task specified by the `tid` exits, then returns.
Delay	`int Delay(int)`	Blocks the current task until the specified number of ticks have elapsed.
DelayUntil	`int DelayUntil(int)`	Blocks the current task until the specified time has been reached.
Time	`int Time()`	Returns the current number of ticks that have elapsed.
Getc	`int Getc(int)`	Returns first unreturned character from the given UART.
Putc	`int Putc(int, char)`	Transmits the given character to the given UART.
WaitOnSensor	`int WaitOnSensor(char, unsigned int)`	Blocks the calling task until the specified sensor triggers.
CpuIdle	`int CpuIdle()`	Returns the percentage of time CPU was idle.
SigTerm	`void SigTerm()`	Signals to halt the system (kill the NullTask).

3.0.4 Create

`Create` creates a new Task Descriptor with the given priority and code, giving it a stack and assigning it an ID. The created tasks has all the satate needed to run and is added to the priority queues so that it can run the next time it is scheduled. Returns:

- `tid` - Positive integer id of the newly created task, unique.
- -1 - if the priority is invalid.
- -2 - if the kernel is out of task descriptors.

3.0.5 MyTid

Returns the task id of the calling task.

3.0.6 MyParentTid

Returns the task id of the task that created the calling task.

- `tid` - Positive integer id of the parent task.
- 0 - If task was created by kernel / has no parent.

3.0.7 Pass

Moves the calling task from being active back onto the priority queue in a ready-to-run-state. Has no return value.

3.0.8 Exit

Ceases execution of the current task and eliminates it. Has no return values.

3.0.9 WhoIs

Lookup the task with the given name in the NameServer and returns its tid if found. Does not block waiting for registration. Returns

- `tid` - Non-negative task identifier on success.
- -1 - NameServer hasn't been created.
- -2 - Error in send.

3.0.10 RegisterAs

Register the current task with the name `name` in the NameServer. Returns

- 0 - Successfully registered.
- -1 - NameServer hasn't been created.
- -2 - Error in send.

3.0.11 UnRegister

Unregister the calling task iff its task ID and name exist in the NameServer. Does not remove the found task if the tid found at the hash location does not match the tid of the caller. Returns

- 0 - Successfully unregistered.
- -1 - NameServer hasn't been created.
- -2 - Error in send.

3.0.12 Send

On a Send, an `Envelope_t` is retrieved from the available pool, and if none are available, an error is returned (A more elegant solution could be implemented by blocking the task until one of structs is available). The msg, msglen, reply, and replylen parameters from the `Send` call are copied into the respective fields of the `Envelope_t`. In addition, the current task pointer is added to the envelope as the sender, and the envelope is set as the outbox pointer of the sender.
The sender is moved to the `RECV_BL` state and the envelope is added to the tail of the receiver's inbound message queue, inboxTail.
If the receiver is in the `SEND_BL` state, it is added back to the ready queues.

- `size` - Non-negative integer representing the number of bytes copied.
- -1 - task ID is not possible
- -2 - task ID does not correspond to a valid task
- -3 - transaction incomplete

3.0.13 Receive

On a Receive, the inbox of the current task is checked. If there is no inbound messages, the task is moved to the `SEND_BL` state, and nothing else is done. In this case, the task will be unblocked by Send when a message is actually available, and the user task will have to make another call to Receive to get the message. This process is transparent to the user and is done through the user-mode Receive function.
If there is a message available, then the corresponding values in the envelope are copied into the provided pointers, and the sender task is moved to the `REPL_BL` state. The envelope is then removed from the head of the queue so the next message can be received.

- `size` - Non-negative integer representing the number of bytes copied.

3.0.14 Reply

For Reply, the intended sender task's outbox parameter is used to find the envelope, and the provided reply message is copied into the provided pointer. The sender is then added back to the ready queue and the envelope is released back into the envelope pool.

- `size` - Non-negative integer representing the number of bytes copied.
- -1 - task ID is not possible
- -2 - task ID does not correspond to a valid task
- -3 - target task is not reply blocked.

3.0.15 Log

Allocates a block of memory from the current memory space, and copies the passed string to that location in memory.

- 0 - Successful write
- -1 - Out of memory from where to write.

3.0.16 Logn

Allocates a block of memory from the current memory space, and copies a fixed sized string to that location in memory.

- 0 - Successful write
- -1 - Out of memory from where to write.

3.0.17 AwaitEvent

`AwaitEvent` blocks until the event identified by the passed integer, `eventType`, occurs as an interrupt then returns with the value generated by the interrupt. The value is non-zero. In the event that the passed integer is not a valid event, it returns -1 or if the queues are full it returns -2. Since we do not use event buffers, the previous correspondence for 0, -2 and -3 are irrelevant to our implementation.

3.0.18 WaitTid

`WaitTid` blocks on the wait queue of the specified task and returns when that task exists with the status of the exit. Returns -1 if the task does not exist.

3.0.19 Delay

Send a message to the ClockServer to block the curren task until the number of ticks have passed.

3.0.20 DelayUntil

Send a message to the ClockServer to block the current task until the given number of ticks have been reached.

3.0.21 Time

Returns the curren tick count by querying the ClockServer.

3.0.22 Getc

Returns the first unreturned character from the given UART; a wrapper for send to the serial IO server. Blocks until it receives a character.

- `character` - On success returns the read character.
- -1 - Serial IO server does not exist
- -2 - Error in send

3.0.23 Putc

Queues the given cahracter for trasmission to the specified UART; character may not have been transmitted on return. Is a wrapper to the IO serial server.

- 0 - On success
- -1 - Serial IO server does not exist
- -2 - Error in send

3.0.24 WaitOnSensor

Blocks the calling task until the specified sensor has been triggered. Parameters are passed as `(module, index)` where index is a positive integer corresponding to the identifier within that module. Returns

- 0 - On success and unblocked.
- -1 - If the given sensor does not exist.

3.0.25 CpuIdle

Returns the amount of time that the CPU has been idle (running the null task).

- `idle` - Non-negative integer corresponding to the time the CPU has been idle as a fraction of the total CPU time.

3.0.26 SigTerm

Tells the Kernel to kill the NullTask and cease execution. Has no return value.

4 Servers

5 Algorithms and Data Structures

6 Shell Task

6.1 Commands

When inputting commands, it is important to note that they must exactly match their prototype or they will be interpreted as garbage by the terminal and equivalent report back `command: error command not found`. For example, to move train 1 at speed 10, the following command would be entered "tr 1 10" followed by a `RETURN` or `ENTER`. All commands must be terminated by a `RETURN` or `ENTER` for the terminal to process them. The syntax for issuing a command is `''command_name ARGUMENT_1 ARGUMENT_2 .... RETURN''`. The following table lists the commands supported by this Kernel's shell:

Command	Argument 1	Argument 2	Description
go	N/A	N/A	Start the train controller (if not started).
stop	N/A	N/A	Stop the train controller (if started).
tr	1 - 80	0 - 14	Set the train specified by the first argument to the speed specified by the second argument.
ax	1 - 80	16 - 31	Run the auxiliary function specified by the second argument on the train specified by the first argument.
rv	1 - 80	N/A	Reverse the direction of the train specified by the first argument.
li	1 - 80	N/A	Turn on the lights on the train specified by the first argument.
sw	0 - 255	S or C	Throw the switch specified by the first argument ot straight (S) or curved (C) specified by the second argument.
ho	1 - 80	N/A	Turn on the horn on the train specified by the first argument.
add	1 - 80	N/A	Add a train to the track (useful to introduce new trains to the track).
rps	N/A	N/A	Play a game of Rock-Paper-Scissors.
time	N/A		Reports the current time when the command was called.
q	N/A	N/A	Halt the system and return to RedBoot.

6.2 Introducing New Trains

By default, the `TrainUserTask` that listens to the `Shell` assumes that the trains 45, 48, 49 and 50 exist on the track to allow the kernel to have some idea of what trains are currently running and where they might be on the track. Should you wish to use another train that currently is not in the above list, that train can be added to the track by running the `add` command listed in the above subsection.

6.3 Shell Display

The following picture highlights overlay layout of the terminal display: Below the top two description lines at the top of the display are the current time (to within 10 milliseconds) and the percentage of time that the CPU has been idle (calculated as fraction of the time spent doing nothing, running the null task, over the total time the CPU has been running). Below those two are the current state of the switches ('C' for curved and 'S' for straight),

```
CS 452 Real-Time Microkernel (Version 0.1.9)
Copyright <c> Max Chen (mqchen), Ford Peprah (hkpeprah).  All rights reserved.
Time: XX:XX:XX     CPU Idle: XX%

====Switches====
001: C   002: C   003: C   004: C   005: C   006: C   007: C   008: C   009: C
010: C   011: C   012: C   013: C   014: S   015: C   016: C   017: C   018: C
151: C   152: S   153: C   154: S

====Sensors=====
A01  A12  B01


> ENTER COMMANDS HERE
```

Figure 1: CS452 Microkernel Shell Display

and below those are the recently triggered sensors, read from left to right; the most recent triggered sensor is the farthest left and they scroll horizontally. Finally, below all of that information is the shell prompt where the user can enter information.

7 Known Bugs and Errors

There are known and unknown bugs and errors in our kernel system. Below we have listed some of the known bugs and errors and grouped them categorically. Recoverable errors are ones the system can compensate for at runtime in order to correct them and continue the state of the kernel. Unrecoverable errors are those that would require reprogramming in order to function properly. Lastly, User Interface Bugs are those tricky bugs that occur due to some configuration of the Kernel and generally do not affect how the application runs, but would affect how the user sees the interface.

7.1 Recoverable Errors

- Switch 14 is broken on th second track; to compensate this, the kernel switches it to the straight state which does not cause a derail, though a user could switch it back to curve and allow a derail to become possible again.

7.2 Unrecoverable Errors

- Train derails because of a switch state in the middle switches; in order to remedy this, the train functions should never allow a switch state involving the center switches that would cause a derail.

- Known broken switches (e.g. switch 14 on the second track), and known faulty sensors can cause derails or locations to not accurately be picked up; to compensate for this, the kernel would have to know which sensors are broken and to compensate by looking one sensor ahead or behind that instead to estimate distance.

7.3 User Interface Bugs

- Depending on the state of the box prior to the application being loaded, if the previous group had not exited gracefully, cleared their FIFOs, or otherwise, there may be garbage input read by the IO servers when the kernel is loaded; this is characterized by random printing at the top of the terminal repeatedly that do not seem to be valid characters being typed by the user. To rectify this issue, reboot the box and reload the program onto it.

- During the boot operation, newlines may sometimes be printed erroneously by the terminal causing the display to be printed incorrectly, characterized by a display state not resembling the one listed above in the `Shell` section. This can be remedied by quitting the program, and starting it again.

8 MD5 Hashes