Kernel 3
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, 9^{th}, 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 kernel3 .
                          or
git clone file:////u7/mqchen/cs452/cs452-microkern -b kernel3 .
```

You will now have a working instance of our kernel2 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/assn3.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/assn3.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/assn3.elf` and `/u/cs452/tftp/ARM/hkpeprah/assn3.elf`.

1.3 Command Prompt

After the startup tasks have finished running, the user will reach a command prompt, where they will be able to enter commands. The following commands are supported:

Command	Description
q / quit	Exit the prompt and shut down the kernel.
rps	Start a Rock-Paper-Scissors game.

2 Kernel Structure

2.1 Refactoring

Send-Receive-Reply was modified to better handle the Receive-Before-Send case. In the last kernel submission, our `Receive` implementation did not store the user supplied parameters when no message was available, and simply had a second call if the first were to fail. This incurred an overhead of an extra context switch. The issue was fixed in this version of the kernel, and the receive parameters are now stored and `Send` directly copies into them if they are present.
In addition, the way a user task's result is stored was changed. Since hardware interrupts require scratch registers to be saved, the user task result must now be saved on to its stack. This is done by writing the result into the location where r0 is stored on the user stack based on its SP. This method also saves us a word in our task descriptor since the result is no longer stored there.

2.2 Hardware Interrupts

The clock interrupt is enabled by:

1. Loading 50800 into the clock load register (interrupt every 1/10 second), then starting the clock in 508 KHz, pre-loaded mode

2. Enable the clock interrupt - interrupt 51 - by setting bit 19 of the VIC2 enable register (32-63 in VIC2, 51-32 = 19)

2.2.1 Context Switch

The updated context switch which supports the handling of hardware interrupts now saves the scratch registers. That is, r0-r12, lr (r13 = sp, r15 = pc) are now pushed and popped from the user stack. There is a separate handler for IRQ and SWI that do some slightly different things, but share a good portion of code that is re-used through assembly macros. The first step in both handlers is to save the user state:

1. Switch to SYS mode with interrupts disabled (write 0x9F to CPSR).
2. Save r0-r12, lr to the user stack, move the SP to r0.
3. Switch back to the service mode (SWI/IRQ).
4. Push SPSR, LR of the respective service mode to r0 (user stack).

IRQ Only: The IRQ handler subtracts 4 from the LR when storing it to the user stack (interrupted instruction instead of next instruction).
SWI Only: THe SWI handler pops the top of the kernel stack as an address and writes r1 into it. This is the method of passing the arguments (a struct on the user stack) of the sys call to the kernel, as described in K1.
After the user state is saved, the processor is returned to the kernel by popping off the stored kernel registers from its stack. Since the kernel starts in SVC mode, the IRQ handler must first switch to SVC mode before making the pop from the stack. Both handlers will enter the kernel in the exact same way, returning the task's SP (since it's in r0) as the result of the `swi_exit` call.
`swi_exit` has the same function as it did in K1, switching from the kernel back to the user task. The main difference is that now all registers are restored, and the result is no longer passed in as a parameter to be assigned to r0 (it should already be stored as r0 on the user stack). In addition, the LR received from SVC/IRQ is now stored on the user stack and returned to, instead of the short-cut taken in K1 where the kernel would immediate return to the user's LR. While the shortcut was okay for SWI (the `swi_call` function stub meant that the LR for each SWI call would point to the same instruction - mov pc, lr), it would not work for interrupts since the interrupted task could have been executing at any point.

2.2.2 Handling Interrupts

Something something

2.3 ClockServer

2.3.1 Implementation

The ClockServer registers with the NameServer then creates the ClockNotifier which handles processing of timer generated interrupts. We use the $508kHz$ 32-bit timer. It then writes 5080 to the timer load register; using the $508kHz$ timer and with 10 milliseconds, this corresponds to 5080 in the timer load register. It then writes to the timer control register to enable it, to set the mode to periodic so that the count resumes back at 5080 after counting down, and to set the timer to the $508kHz$ timer. The ClockServer then blocks on `Receive` to handle messages from other tasks. When the ClockServer recieves a message of type `Delay` or `DelayUntil`, it adds the task to its delay queue sorted by the number of ticks that the

task is waiting; the array is sorted in ascending order. When it receives a message of type `Tick` it increments its internal counter, replies to the sending task immediately, then iterates through its delay queue waking up every task with a delay less than the current count by replying to them, and stopping as soon as it reaches a task with a delay greater than its current tick count. This ensures that we do not needlessly check tasks that will not wake up as the queue is sorted.

2.3.2 ClockNotifier

The ClockNotifier is responsible for notifying the ClockServer when a tick occurs; a tick is defined as ten milliseconds passing on the 32-bit hardware timer. Since the $508kHz$ clock is used, ten milliseconds is equivalent to setting the value of the clock to $508 \cdot 10 = 5080$ upon which it will countdown to 0 then generate an interrupt. The ClockNotifier blocks on the timer interrupt and on return it sends a message to the ClockServer indicating a tick took place, which the ClockServer immediately replies to, and then the ClockNotifier blocks on AwaitEvent again. This task never exits unless the system is shutting down.

2.3.3 API

Method	Prototype	Description
Time	`int Time()`	Sends a message to the ClockServer to get the current tick count.
Delay	`int Delay(int ticks)`	Sends a message to the ClockServer to block the current task until number of ticks passed.
DelayUntil	`int DelayUntil(int ticks)`	Sends a message to the ClockServer to block the current task until the time ticks has been reached.

2.4 System Calls

System Call	Prototype	Description
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.

2.4.1 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.

2.4.2 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 User Tasks and Output

3.1 Program Output

Our kernel implements larger number as being a higher priority, but gives the same values to the requesting tasks based on their priority.

Priority (larger is higher)	Delay Times (tics)	Number of Delays	Task ID
3	10	20	4
4	23	9	5
5	33	6	6
6	71	3	7

where the last column gives the id of the task that was created with those parameters for reference in the output. Then, the output is:

```
Tid: 4      Delay Interval: 10     Delays Complete: 1
Tid: 4      Delay Interval: 10     Delays Complete: 2
Tid: 5      Delay Interval: 23     Delays Complete: 1
Tid: 4      Delay Interval: 10     Delays Complete: 3
Tid: 6      Delay Interval: 33     Delays Complete: 1
Tid: 4      Delay Interval: 10     Delays Complete: 4
Tid: 5      Delay Interval: 23     Delays Complete: 2
Tid: 4      Delay Interval: 10     Delays Complete: 5
Tid: 4      Delay Interval: 10     Delays Complete: 6
Tid: 6      Delay Interval: 33     Delays Complete: 2
Tid: 5      Delay Interval: 23     Delays Complete: 3
Tid: 4      Delay Interval: 10     Delays Complete: 7
Tid: 7      Delay Interval: 71     Delays Complete: 1
Tid: 4      Delay Interval: 10     Delays Complete: 8
Tid: 4      Delay Interval: 10     Delays Complete: 9
Tid: 5      Delay Interval: 23     Delays Complete: 4
Tid: 6      Delay Interval: 33     Delays Complete: 3
Tid: 4      Delay Interval: 10     Delays Complete: 10
Tid: 4      Delay Interval: 10     Delays Complete: 11
Tid: 5      Delay Interval: 23     Delays Complete: 5
Tid: 4      Delay Interval: 10     Delays Complete: 12
Tid: 4      Delay Interval: 10     Delays Complete: 13
Tid: 6      Delay Interval: 33     Delays Complete: 4
```

```
Tid: 5      Delay Interval: 23     Delays Complete: 6
Tid: 4      Delay Interval: 10     Delays Complete: 14
Tid: 7      Delay Interval: 71     Delays Complete: 2
Tid: 4      Delay Interval: 10     Delays Complete: 15
Tid: 4      Delay Interval: 10     Delays Complete: 16
Tid: 5      Delay Interval: 23     Delays Complete: 7
Tid: 6      Delay Interval: 33     Delays Complete: 5
Tid: 4      Delay Interval: 10     Delays Complete: 17
Tid: 4      Delay Interval: 10     Delays Complete: 18
Tid: 5      Delay Interval: 23     Delays Complete: 8
Tid: 4      Delay Interval: 10     Delays Complete: 19
Tid: 6      Delay Interval: 33     Delays Complete: 6
Tid: 4      Delay Interval: 10     Delays Complete: 20
Tid: 5      Delay Interval: 23     Delays Complete: 9
Tid: 7      Delay Interval: 71     Delays Complete: 3
```

3.2 Explanation of Output

In the context of this description, we define "waking up" as "unblocking and printing." We get this output because task 4 (priority 3) has the lowest delay interval, so it wakes up twice before the next task with the lowest delay interval (task 5) wakes up as $10 * 2 = 20 < 23$. Now, after that task wakes up, task 4 wakes up a third time before task 6 wakes up as $10 * 3 = 30 < 33$. This process repeats a second time before task 4 wakes up for a seventh time, followed by task 7 waking up for the first time as it delays for 7 ticks. Task 6 is the first to finish as it has the lowest total delay time; $6 \cdot 33 = 198$ ticks. Then task 4 with $10 \cdot 20 = 200$ ticks, then task 5 with $23 \cdot 9 = 207$ ticks followed lastly by task 7 with $3 \cdot 71 = 213$ ticks. The output corresponds to each task delaying the number of ticks it was given before waking up and delaying again until its completed the required number of delays it was passed. When a task delayed, it could only be woken up by a subsequent tick, which for tasks with small delays, meant they would wake up faster as they would be waiting for a smaller amount of time.

4 MD5 Hashes

Source files can be accessed at either `/u7/mqchen/cs452/cs452-microkern` or `/u8/hkpeprah/cs452-microkern` on the `kernel3` branch. The listing of all the fields being submitted alongisde their MD5 hashes and locations are as follows: