Project 3: IPC (4%)

ENEE 447: Operating Systems — Spring 2012

Assigned: Monday, Feb 15; **Due:** Friday, Feb 26

Purpose

In this project you will design and build an inter-process communication (IPC) facility. The goal is to enable simple client/server style communications within a multicore chip. It will make use of your interrupt-handling facility that you are building for Project 2.

Mailboxes in ARM

As you have seen in Project 2, the ARM architecture has these things called "mailboxes," which are 32-bit registers that, when written to, generate interrupts to the receiving core. They lie at the following addresses:

4.8 Core Mailboxes

The system has sixteen mailboxes. They are accessed through 32 register.

- There are 16 write-to-set-bits registers
  These registers are write-only
- There are 16 write-to-clear-bits registers
  These register can also be read.

Address	Mailbox-set
0x4000_0080	Core 0 Mailbox 0 Set
0x4000_0084	Core 0 Mailbox 1 Set
0x4000_0088	Core 0 Mailbox 2 Set
0x4000_008C	Core 0 Mailbox 3 Set
0x4000_0090	Core 1 Mailbox 0 Set
0x4000_0094	Core 1 Mailbox 1 Set
0x4000_0098	Core 1 Mailbox 2 Set
0x4000_009C	Core 1 Mailbox 3 Set
0x4000_00A0	Core 2 Mailbox 0 Set
0x4000_00A4	Core 2 Mailbox 1 Set
0x4000_00A8	Core 2 Mailbox 2 Set
0x4000_00AC	Core 2 Mailbox 3 Set
0x4000_00B0	Core 3 Mailbox 0 Set
0x4000_00B4	Core 3 Mailbox 1 Set
0x4000_00B8	Core 3 Mailbox 2 Set
0x4000_00BC	Core 3 Mailbox 3 Set

Address	Mailbox-clear
0x4000_00C0	Core 0 Mailbox 0 Rd/Clr
0x4000_00C4	Core 0 Mailbox 1 Rd/Clr
0x4000_00C8	Core 0 Mailbox 2 Rd/Clr
0x4000_00CC	Core 0 Mailbox 3 Rd/Clr
0x4000_00D0	Core 1 Mailbox 0 Rd/Clr
0x4000_00D4	Core 1 Mailbox 1 Rd/Clr
0x4000_00D8	Core 1 Mailbox 2 Rd/Clr
0x4000_00DC	Core 1 Mailbox 3 Rd/Clr
0x4000_00E0	Core 2 Mailbox 0 Rd/Clr
0x4000_00E4	Core 2 Mailbox 1 Rd/Clr
0x4000_00E8	Core 2 Mailbox 2 Rd/Clr
0x4000_00EC	Core 2 Mailbox 3 Rd/Clr
0x4000_00F0	Core 3 Mailbox 0 Rd/Clr
0x4000_00F4	Core 3 Mailbox 1 Rd/Clr
0x4000_00F8	Core 3 Mailbox 2 Rd/Clr
0x4000_00FC	Core 3 Mailbox 3 Rd/Clr

A write to one of the "Mailbox-set" locations interrupts the receiving core, the core number that is common within the color scheme—i.e., writes to addresses 0x40000080, 84, 88, and 8C all interrupt Core 0; writes to addresses 0x40000090, 94, 98, and 9C all interrupt Core 1; etc. Reads to these locations are not permitted (they either do nothing or hang).

A read to one of the "Mailbox-clear" locations returns the value previously written, or, if no value was previously written, and the mailbox is set up to generate an interrupt, the read hangs. A write to one of

the "Mailbox-clear" locations clears whatever interrupt is pending, or does nothing if there is no interrupt for that mailbox.

The interrupt vector/s for each of the four mailbox groups is set using the following I/O registers:

4.7 Core Mailboxes interrupts

There are four Mailbox interrupt control registers.

Address: 0x4000_0050 Core0 Mailboxes interrupt control **Address: 0x4000_0054 Core1 Mailboxes interrupt control** **Address: 0x4000_0058 Core2 Mailboxes interrupt control** **Address: 0x4000_005C Core3 Mailboxes interrupt control** **Reset: 0x0000_0000**	
Bits	**Description**
31-8	<Reserved>
7	Mailbox-3 FIQ control. If set, this bit overrides the IRQ bit (3). 0 : FIQ disabled 1 : FIQ Enabled
6	Mailbox-2 FIQ control. If set, this bit overrides the IRQ bit (2). 0 : FIQ disabled 1 : FIQ Enabled
5	Mailbox-1 FIQ control. If set, this bit overrides the IRQ bit (1). 0 : FIQ disabled 1 : FIQ Enabled
4	Mailbox-0 FIQ control. If set, this bit overrides the IRQ bit (0). 0 : FIQ disabled 1 : FIQ Enabled
3	Mailbox-3 IRQ control. This bit is only valid if bit 7 is clear otherwise it is ignored. 0 : IRQ disabled 1 : IRQ Enabled
2	Mailbox-2 IRQ control. This bit is only valid if bit 6 is clear otherwise it is ignored. 0 : IRQ disabled 1 : IRQ Enabled
1	Mailbox-1 IRQ control. This bit is only valid if bit 4 is clear otherwise it is ignored. 0 : IRQ disabled 1 : IRQ Enabled
0	Mailbox-0 IRQ control. This bit is only valid if bit 4 is clear otherwise it is ignored. 0 : IRQ disabled 1 : IRQ Enabled

This allows the writing of each of the 16 mailboxes to be tied to the IRQ vector, the FIQ vector, or no vector at all (i.e., writing to that mailbox will not raise an interrupt).

In Project 2, you will see some brief hints of how these can be used: in my code, I used this mechanism to have the kernel periodically interrupt the user process through the **interrupt_core**() primitive. The

interrupted core would need to use the **clear_interrupt**() to prevent itself from taking the interrupt repeatedly *ad infinitum*, but at least the code shows how the facility can be used.

One can test the status of a mailbox through the following facility, which tells the software whether any of the mailboxes are responsible for an interrupt.

4.10 Core interrupt sources

The cores can get an interrupt or fast interrupt from many places. In order to speed up interrupt processing the interrupt source registers shows what the source bits are for the IRQ/FIQ. As is usual there is a register for each processor.

There are four interrupt source registers.

Address: 0x4000_0060 Core0 interrupt source **Address: 0x4000_0064 Core1 interrupt source** **Address: 0x4000_0068 Core2 interrupt source** **Address: 0x4000_006C Core3 interrupt source** **Reset: 0x0000_0000**	
Bits	**Description**
31-28	<Reserved>
17:12	Peripheral 1..15 interrupt (Currently not used)
11	Local timer interrupt
10	AXI-outstanding interrupt <For core 0 only!> all others are 0
9	PMU interrupt
8	GPU interrupt <Can be high in one core only>
7	Mailbox 3 interrupt
6	Mailbox 2 interrupt
5	Mailbox 1 interrupt
4	Mailbox 0 interrupt
3	CNTVIRQ interrupt
2	CNTHPIRQ interrupt
1	CNTPNSIRQ interrupt
0	CNTPSIRQ interrupt (Physical Timer -1)

Bits 4–7 of the IRQ-interrupt-source register (there is an analogous register also for the FIQ interrupt sources) indicate which if any of the mailboxes for a given core are responsible for a pending interrupt. This can, in theory, be used to make sure a mailbox has valid data in it before attempting to read it.

Inter-Process Communication in the BCM2836

Your task is to develop a mechanism that can deliver 28-bit integer values from one core to another (we are going to send a 32-bit word which has a header with a valid bit and a 2-bit sender ID (the core ID), plus the 28-bit data payload). This can be through the mailboxes or the memory system. One advantage of the mailboxes is that they are set up for precisely this sort of thing. There are some significant disadvantages with using them, however:

- The fact that a read to an unwritten mailbox hangs makes it important for your code never to read a mailbox unless you are absolutely certain that it contains a written value.
- Using the mailbox facility for both data delivery and notification (interrupts) means that it might interact poorly with your countdown timer, so one will have to be on a separate line (e.g., use both theIRQ interrupt and the FIQ interrupt).
- There are race conditions in which an interrupt notification can be lost if a mailbox is written while the recipient is handling a mailbox-related interrupt.

An alternative is to use the memory system and write data into known memory locations, using the mailbox facility solely to notify the recipient that something has arrived. In this case, the recipient should sweep through all memory locations assigned to it, because regular memory will not cause the read to hang if there is a NULL value present. The main disadvantage of this scheme is that ache coherence is absolutely essential; otherwise, the first time you receive a message, all will be fine, but the second time you go to receive a message, you will simply read the previously received one.

The obvious disadvantage of all these schemes is that what is desired—what is *required*, actually—is the ability to transfer data directly from one core to another, and none of these schemes do that. All are slow, and the one that uses memory requires the cache-coherence engine for data delivery, and the fact that data delivery and notification are handled through two separate mechanisms means that it is theoretically possible for the notification to arrive ahead of the message. That means one could get the notification that data is in a message somewhere, and the recipient would read stale message data, because the updated message data arrives later.

Unfortunately, ARM does not have what we need, so we will make do with what we have. Something to note about the mailbox scheme:

- The fact that a read to an unwritten mailbox hangs makes it important for your code never to read a mailbox unless you are absolutely certain that it contains a written value. ***However***, a read to an empty mailbox should ***not*** hang if that mailbox is set up ***not*** to deliver interrupts when written.

What this means is that we can set up a client/server type IPC mechanism, in which only the server gets a notification (an interrupt) when something is written to one of its mailboxes, and the clients need to keep trying to communicate with the server (send a message, recv a message) until they get an actual response.

The problem still remains that the server can hang if it accidentally reads the wrong mailbox. For instance, if we use the four mailboxes for Core0 to handles messages from Core0, Core1, Core2, and Core3, respectively, then the handler on Core0 has to make sure that it reads the mailbox that generated the interrupt, otherwise the code on Core0 can hang, which would be really bad if that is the kernel. The status registers at 0x40000060 and 0x40000070 should be able to solve this, such that the kernel code first checks the status register to see which mailbox caused the interrupt, and then the kernel gets its message from the corresponding mailbox.

The advantage of that approach is that you can send full 32-bit values. For those of you up for a challenge, try implementing that (in theory, it should work just fine). I went an easier route, because I kept getting weirdness when using the status register (probably a bug in a completely different part of my code, but whatever, this is what finally worked): I funneled all of the requests through the same mailbox. This has several implications:

- The mailbox read & write routines are slightly simplified.
- The mailbox does not identify the sender (all requests go through the same mailbox), so we have to explicitly put the sender information into the message itself; thus, we can't send full 32-bit messages.
- It is possible for two different senders to step on each other; i.e., one sender can write to the kernel's mailbox, and then, before the kernel has the chance to get the message and send a reply, another core overwrites the exact same message. By setting it up such that the clients need to keep trying to send/receive until they get a response, this problem is solved.
- Cache coherence is a non-issue because the I/O registers are not supposed to be cacheable.

Your Implementation

You are to set up two things:

1. A handler for the kernel process that is invoked whenever a message arrives at the kernel's mailbox

   The handler should build off of your Project 1 solution: set up the vector table and populate the RESET entry with a pointer to the provided **reset** code (in **1_boot.s**), and populate the FIQ entry with a pointer to the code that you write, which should do some basic save/restore operations as you implemented in Project 1, and then call the **incoming_kmsg**() routine in **kernel.c**.

2. An IPC facility that reads and writes the ARM mailboxes

   Your IPC facility should have the following components:

```
//
// sets the top bit (bit 31 [32]) of the outgoing message
// sets the sender ID, from calling cpu_id(), in bits 28-29 [29-30]
// puts the user's message into bits 0-27 [1-28]
// returns a Boolean
// 0 - msg not delivered (if dest MBOX is full) [kernel ignores this]
// 1 - otherwise
//
unsigned int
send( unsigned int dest, unsigned int msg )
{
        // your code goes here
}

//
// returns the message received in MBOX
// return NACK (zero) if a timeout occurs - i.e., if
// time expires and there is still no valid message to read
//
unsigned int
recv( unsigned int timeout )
{
        // your code goes here
}

//
// this is what the kernel code can call instead of recv, because it has no
// timeout, and you don't care if it returns a valid message or not
// ... if you think it's simpler, you can just have one recv() function
// that both kernel and apps call ... for me, this was easier
//
unsigned int
krecv( )
{
        // your code goes here
}
```

The code is set up right now such that the kernel (in **kernel.c**) receives messages that arrive to its mailbox, through the handler code which calls **incoming_kmsg**(). This routine verifies that the message is valid and came from someone other than itself, and, if so, it returns the incoming message data incremented by one. The code looks like this:

```
void
incoming_kmsg()
{
        unsigned int msg = krecv();
        int id;
```

```
        if (MSG_VALID(msg) && (id = MSG_SENDER(msg)) != 0) {
                msg = MSG_DATA(msg);
                msg += 1;
                send(id, msg);
        }
}
```

The user code sends a message about once per second, keeps trying send/recv pairs until it gets a valid response, and then verifies that the return value is indeed one greater than the value it sent out originally. The code looks like this:

```
void
client1(void)
{
        unsigned int now, inmsg, outmsg, diff;

        while (1) {
                oldwait(50);
                now = now_usec();
                outmsg = now & 0x000FFFFF;

                do {
                        unsigned int backoff = 1;
                        while (!send(0, outmsg)) {
                                oldwait(backoff);
                                backoff *= 2;
                        }
                } while (( inmsg = recv(USER_TIMEOUT) ) == NACK);

                diff = MSG_DATA(inmsg) - outmsg;
                if (diff == 1) {
                        blink_led(GRN);
                }
        }
}
```

There are two cores doing this simultaneously, so you should get each one blinking a different color.

You will perhaps note the exponential backoff in the inner **send** loop. This is traditional networking practice; if you are trying to get a server's attention, and your first attempt fails, it is not immediately obvious whether it is due to network traffic, server load, or simple bad timing (two things arrived at the server at the exact same time). So the most common approach is to try again, but a little while later, and if the second try fails, then wait even longer, and so on.

You'll also note that I'm using **oldwait**() here instead of **wait**(). This is because the **wait**() function uses the one system-wide timer, and we have that timer dedicated to serving the kernel. Later, I will show you how to set up additional timers for Cores 1–3.

Build It, Load It, Run It

Once you have it working, show us.

