
# Project 5: Distributed Scheduling (4%)

**ENEE 447: Operating Systems** — Spring 2012

**Assigned:** Monday, Feb 29; **Due:** Friday, Mar 11

## Purpose

In this project you will implement trivial multicore load balancing on the Raspberry Pi, using a combination of the last three projects. In your last three projects you have implemented the following facilities:

- IRQ and FIQ interrupt handlers
- An IPC mechanism to communicate between "client" and "server" cores
- A context-switch mechanism

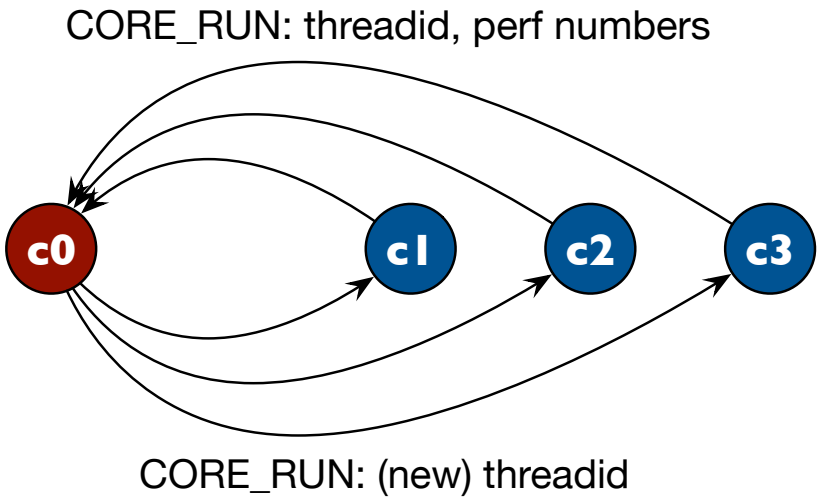
In this project, you will combine these mechanisms into one system, in which the "server" core, **core0**, will tell the other cores which threads to run. The "client" cores (cores 1, 2, and 3) will periodically update **core0** with their status, at which time they present the following pieces of information:

1. Core status {CORE_ERROR, CORE_RESET, CORE_IDLE, CORE_RUN}
2. If CORE_RUN, the thread ID of the thread currently running
3. A measure of the thread's performance (for now, a hard-coded value that we will ignore)

Core0 uses this information to determine which cores should be running which threads, in a load-balancing fashion as described in class.

## Multicore Load-Balancing

In general, because the incremental cost of silicon is low, future systems can easily have enough cores to run numerous threads with minimal thread-switching. In such a scenario, the job of the operating system becomes one of load balancing more than thread scheduling: i.e., it is most important to determine how best to divvy up the processing resources, instead of figuring out how to get numerous jobs done quickly.




To this end, we will have the code running on **core0** act as a lowest-level kernel, and the code running on the other cores will be subservient to it. They will periodically let **core0** know what they are doing (what

thread number they are running), and in response, the kernel on **core0** will tell them what to do next. This is illustrated in the figure above. The figure shows how cores 1, 2, and 3 register with **core0**, with the CORE_RUN status value, and arguments of **threadid** and that thread's most recent **performance numbers**. The kernel code on **core0** responds with a (potentially new) state as well as a potentially new **threadid**.

## Implement Distributed Scheduling

Your task is to use the components that you have already developed:

- Implement interrupt handlers for the kernel and the ukernel (for now, what we're calling the kernel-like code running on cores 1, 2, 3)
  - The kernel's handler should respond to message by giving a new thread ID (I have provided a handler that simply toggles the lowest bit of the thread ID between even/odd)
  - The ukernel's handler should look like the client code in your IPC project: when it wakes up it should save process state, send a message to the kernel, and, when the kernel responds, it should switch to the ID given by the kernel
- Use your IPC mechanism to communicate between cores
- Use your context-switch code to round-robin back and forth between two "user" codes as directed by the kernel

You have been given the following code:

**add the following to ipc.c**

```
void _init_ipc()
{
        unsigned int i;

        for (i=0; i<NUM_CORES; i++) {
                // clear the mailbox
        }

        #define INT_IRQ 0x0F
        #define INT_FIQ 0xF0
        #define INT_NONE 0

        // mailboxes & interrupts
        PUT32(0x40000050, INT_FIQ);    // mbox 0 interrupts by FIQ; note: IRQ used by timer
        PUT32(0x40000054, INT_IRQ);    // mboxes 1..3 interrupt via IRQ
        PUT32(0x40000058, INT_IRQ);
        PUT32(0x4000005C, INT_IRQ);

        return;
}
```

This file simply changes the set-up so that all of the mailboxes interrupt … the kernel will use this to interrupt cores 1, 2, 3 periodically (later, we will use timer-controlled interrupts, but I figured we would stick with things you have already seen, because it is already going to be difficult enough).

Next are modifications to the **hype.h** #include file.

**add the following to hype.h:**

```
enum core_status {
        CORE_ERROR=0,
        CORE_RESET,
        CORE_IDLE,
        CORE_RUN,
```

```
        // add new ones above here
        NUM_CORESTATUS
};

#define SET_REGMSG( status, thread, ipc1000 )    \
(((status << 24) & 0x0F000000) | ((thread << 16) & 0x00FF0000) | (ipc1000 & 0x0000FFFF))

#define REGMSG_STATUS( msg )          ((msg >> 24) & 0xF)
#define REGMSG_THREAD( msg )          ((msg >> 16) & 0xFF)
#define REGMSG_IPC1000( msg )         (msg & 0xFFFF)
```

Here, you simply add some message support to the file. These primitives are used to send status messages back and forth between the **ukernels** on cores 1, 2, and 3, and the main **kernel** on **core0**. The enumeration lists a number of possible **status** values that the core can have. The SET_REGMSG macro packs into a 28-bit field three values:

1. a 4-bit **status** value
2. an 8-bit **threadid** number
3. a 16-bit measure of **performance** (measured in *instructions per 1000 cycles*)

The REGMSG_ macros extract these fields from a 32-bit integer.

The following is the kernel code. You need do nothing to it, unless you want to try other scheduling mechanisms. Right now, as written, it simply toggles between odd and even numbers, pseudo-randomly. The code assumes that, on each core, you are running two different threads with adjacent IDs (even and odd) such that the odd one is 1 larger than the even one, and not the other way around.

For example, **core1** can run threads 0 and 1 but not threads 1 and 2. If **core1** runs threads 0 and 1, then **core2** can run threads 2 and 3, but not threads 1 and 2 or threads 3 and 4, etc.

**kernel.c**

```
#include "hype.h"

extern unsigned int interrupt_core( unsigned int );

void
kernel()
{
        unsigned int then, now, delta;

        #include "initf.auto"

        then = now_usec();
        while (1) {
                now = now_usec();
                delta = usec_diff( now, then );
                if (delta > ONE_SEC) {
                        then = now;
                        interrupt_core(1);
                        interrupt_core(2);
                }
        }
}

extern unsigned int krecv();

void
incoming_kmsg()
{
        unsigned int msg = krecv();
        int id, thread;
        int swap = now_usec() & 2;
```

```
        if (MSG_VALID(msg) && (id = MSG_SENDER(msg)) != 0) {
                msg = MSG_DATA(msg);

                if (REGMSG_STATUS(msg) == CORE_RUN) {
                        thread = REGMSG_THREAD(msg);
                        if (swap) {
                                if (thread & 0x1) {
                                        thread &= 0xfffffffe;
                                } else {
                                        thread |= 0x1;
                                }
                        }
                        msg = SET_REGMSG( CORE_RUN, thread, 0 );
                } else {
                        msg = SET_REGMSG( CORE_RESET, 0, 0 );
                }
                send(id, msg);
        }
}
```

Here, the time value is used to decide whether or not to change threads.

The **u_kernel** code is based on what was previously in your **z_applications.c** file for the IPC project. You will keep the **z_applications.c** file from your context-switch code: your thread should be running the client applications in that file, and the code in **u_kernel.c** should do two things:

1. when awoken due to an interrupt, send a message to **core0**
2. upon receipt of an ACK from **core0**, switch to the indicated thread (or reset, or whatever the kernel on **core0** tells the client core to do)

The code is shown below:

**u_kernel.c:**

```
#include "hype.h"

unsigned int
ukernel_status(void)
{
        unsigned int inmsg, outmsg;

        outmsg = SET_REGMSG( CORE_RUN, current_thread, 0xcafe );

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

        //
        // do whatever the kernel says to do:
        //
        // CORE_RESTART -- only in error situations (optional extra credit)
        // CORE_RUN -- specific thread number
        //

}
```

The response that returns from the kernel may have your code restore the previously saved thread context, or it may have you change to a different thread, or it may have you reset yourself if it detects an error. The resetting is optional, for extra credit; only do it if you get everything else working.

Note that this should all look very familiar to you — it is simply the first time you have put multiple pieces together.

## A Few Things to Note

There are a few considerations … for instance, each core must have two thread IDs, and none of the thread IDs can be the same (otherwise, things might get weird).

You can have each core run the same **z_application.c** code, because all of the state is kept on the stack. If you use global variables anywhere, you will want to have each core run a different set of functions specifically written for it.

## Build It, Load It, Run It

Once you have it working, show us.