Project 2: Timeout Queue (4%)

ENEE 447: Operating Systems — Spring 2019

Assigned: Tuesday, Feb 12; **Due:** Sunday, Feb 24

Purpose

This project has you build a timeout-queue facility, based on the classic Unix *callout table*, and using the timer library that you built in Project Zero. A timeout queue is a simple and elegant way to keep track of deadlines and to create periodic jobs.

Your implementation will make use of doubly-linked lists, which will be provided to you … the doubly-linked list is a technique used to keep track of things, and it is very flexible in that you can walk the list in either direction, and deleting an item from the list only requires a pointer to the item to delete (as opposed to a singly-linked list, which requires a pointer to the item as well as a pointer to the preceding item in the list).

As will be the case in most of the semester's projects, this is programmed in C and does not interact with the ARM peripherals, so you can first build and test it on your laptop, before trying to run it on the Raspberry Pi board.

On Timers and Access to System Registers

There is a very powerful facility in the latest instantiations of the ARM instruction set. The following comes from toe document *ARMv8 Instruction Set Overview:*

5.8.2 System Register Access

`MRS Xt, <system_register>`

Move `<system_register>` to `Xt`, where `<system_register>` is a system register name, or for implementation-defined registers a name of the form "`S<op0>_<op1>_<Cn>_<Cm>_<op2>`", e.g. "`S3_4_c13_c9_7`".

`MSR <system_register>, Xt`

Move `Xt` to `<system_register>`, where `<system_register>` is a system register name, or for implementation-defined registers a name of the form "`S<op0>_<op1>_<Cn>_<Cm>_<op2>`", e.g. "`S3_4_c13_c9_7`"..

`MSR DAIFClr, #uimm4`

Uses `uimm4` as a bitmask to select the clearing of one or more of the `DAIF` exception mask bits: bit 3 selects the `D` mask, bit 2 the `A` mask, bit 1 the `I` mask and bit 0 the `F` mask.

`MSR DAIFSet, #uimm4`

Uses `uimm4` as a bitmask to select the setting of one or more of the `DAIF` exception mask bits: bit 3 selects the `D` mask, bit 2 the `A` mask, bit 1 the `I` mask and bit 0 the `F` mask.

`MSR SPSel, #uimm4`

Uses `uimm4` as a control value to select the stack pointer: if bit 0 is set it selects the current exception level's stack pointer, if bit 0 is clear it selects shared EL0 stack pointer. Bits 1 to 3 of `uimm4` are reserved and should be zero.

What this means is that, for all of the I/O registers that are part of the core (as opposed to those that are defined at the SoC level), you need not know the addresses where they live; you need only know their names. So, for example, here is a write-up of some of the system registers available in ARMv8 cores (64-bit cores), taken from the document *ARM® Architecture Reference Manual—ARMv8, for ARMv8-A architecture profile:*

© Copyright 2016, 2019 Bruce Jacob, All Rights Reserved

B1.2.3 System registers

System registers provide support for execution control, status and general system configuration. The majority of the System registers are not accessible at EL0.

However, some system registers can be configured to allow access from software executing at EL0. Any access from EL0 to a system register with the access right disabled causes the instruction to behave as an UNDEFINED instruction. The registers that can be accessed from EL0 are:

Cache ID registers The CTR_EL0 and DCZID_EL0 registers provide implementation parameters for EL0 cache management support.

Debug registers A debug communications channel is supported by the MDCCSR_EL0, DBGDTR_EL0, DBGDTRRX_EL0 and DBGDTRTX_EL0 registers.

Performance Monitors registers

See *Performance Monitors support* on page B1-64.

Thread registers The TPIDR_EL0 and TPIDRRO_EL0 registers are two thread ID registers with different access rights.

Timer registers In ARMv8 the following operations are performed:

- Read access to the system counter clock frequency using CNTFRQ_EL0.
- Physical and virtual timer count registers, CNTPCT_EL0 and CNTVCT_EL0.
- Physical up-count comparison, down-count value and timer control registers, CNTP_CVAL_EL0, CNTP_TVAL_EL0, and CNTP_CTL_EL0.
- Virtual up-count comparison, down-count value and timer control registers, CNTV_CVAL_EL0, CNTV_TVAL_EL0, and CNTV_CTL_EL0.

Performance Monitors support

The ARMv8 architecture defines optional Performance Monitors.

The basic form of the Performance Monitors is:

- A 64-bit cycle counter.
- Up to a maximum of 32 IMPLEMENTATION DEFINED event counters, where the number is identified by the PMCR_EL0.N field.
- System register access to the cycle counter and event registers, and related controls for:
  - Enabling and resetting counters.
  - Flagging overflows.
  - Generating interrupts on overflow.

  Software can enable the cycle counter independently of the event counters.

Software executing at EL1 or a higher Exception level, for example an operating system, can enable access to the counters from EL0. This allows an application to monitor its own performance with fine grain control without requiring operating system support. For example, an application might implement per-function performance monitoring.

For details on the features, configuration and control of the Performance Monitors, see Chapter D6 *The Performance Monitors Extension*.

EL0 access to Performance Monitors

To allow application code to make use of the Performance Monitors, software executing at a higher Exception level must set the following bits in the PMUSERENR_EL0 system register:

EN When set to 1, access to all Performance Monitors registers is allowed at EL0, except for writes to PMUSERENR_EL0, and reads/writes of PMINTENSET_EL1 and PMINTENCLR_EL1.

ER When set to 1, read access to event counters is allowed at EL0. This includes read/write access to PMSELR_EL0, so that the event counter to read through PMXEVCNTR_EL0 can be set.

CR When set to 1, read access to PMCCNTR_EL0 is allowed at EL0.

SW When set to 1, write access to PMSWINC_EL0 is allowed at EL0.

Note

Register PMUSERENR_EL0 is always read-only at EL0.

© Copyright 2016, 2019 Bruce Jacob, All Rights Reserved

Let's focus for the moment just on the timer registers, in particular, CNTPCT_EL0. Here is the documentation write-up for that register (from the same doc):

D8.5.10 CNTPCT_EL0, Counter-timer Physical Count register

The CNTPCT_EL0 characteristics are:

Purpose

Holds the 64-bit physical count value.

This register is part of the Generic Timer registers functional group.

Usage constraints

This register is accessible as shown below:

EL0	EL1 (NS)	EL1 (S)	EL2	EL3 (SCR.NS=1)	EL3 (SCR.NS=0)
Config-RO	Config-RO	RO	RO	RO	RO

This register is accessible at EL0 when CNTKCTL_EL1.EL0PCTEN is set to 1.

If EL2 is implemented, this register is accessible at Non-secure EL1 and EL0 when CNTHCTL_EL2.EL1PCTEN is set to 1.

Configurations

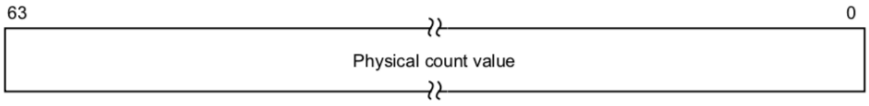
CNTPCT_EL0 is architecturally mapped to AArch32 register CNTPCT.

CNTPCT_EL0 is architecturally mapped to external register CNTPCT.

Attributes

CNTPCT_EL0 is a 64-bit register.

The CNTPCT_EL0 bit assignments are:



Bits [63:0]

Physical count value.

Accessing the CNTPCT_EL0:

To access the CNTPCT_EL0:

```
MRS <Xt>, CNTPCT_EL0 ; Read CNTPCT_EL0 into Xt
```

Register access is encoded as follows:

op0	op1	CRn	CRm	op2
11	011	1110	0000	001

How can you use this? The document itself tells you under the heading "Accessing the CNTPCT_EL0" toward the end of the section. For example, let's say you type the following assembly code into a text file called time.s:

```
.globl get_time
get_time:
        mrs     x0, cntpct_el0
        ret
```

This will give you the *get_time()* function, which reads the 64-bit time counter into the register **x0** and returns it (in ARMv8, function return values are passed through the **x0** register).

© Copyright 2016, 2019 Bruce Jacob, All Rights Reserved

If you really don't want to mess with assembly code, you can type the following in a C-language file:

```
unsigned long get_time()
{
    register unsigned long t;

    // read the current counter
    asm volatile ("mrs %0, cntpct_el0" : "=r"(t));

    return t;
}
```

We can argue about which of the two is easier. The main point is that these two representations are equivalent — they end up producing exactly the same binary executable code.

With a 64-bit architecture, there is no need to use separate 32-bit reads to access the timer, and there is no need to check to make sure the top 32 bits are the same before and after reading the bottom 32 bits, and there is no need for a special *time_diff()* function—to tell the difference between two times, just subtract the earlier from the later (perhaps checking for negative values as a possible error scenario).

Build a Timeout Queue

You should implement the following functions for handling a timeout queue facility. They have the following definitions and behaviors:

```
int bring_timeoutq_current( void );
```

This function calculates the time difference between now and when the timeout queue was last updated, and it subtracts that difference from the head of the list. It returns the amount of time to wait, which can be the value of the next-to-fire event, or perhaps some MAX_WAIT value if you choose that the kernel should never go to sleep for too long.

```
void create_timeoutq_event( int start, int repeat, pfv_t function, namenum_t data );
```

This function takes a pointer to a function (which returns void, i.e., a pfv_t) as well as a piece of generic 8-byte data (in general, this could be more sophisticated, like a pointer to a dynamically allocated data structure), and it inserts the function into the timeout queue with the specified timeout. This is done by taking an event structure off the free list, initializing its values, and inserting it into the timeout queue with the appropriate timing. The function is intended to run at time *start* microseconds from now. The function assumes that the calling function has already brought the timeout queue's internal notion of time to be current [through the function *bring_timeoutq_current()*…] If the *repeat* value is non-zero, then when the event is handled (by the function `handle_timeoutq_event`), it will be re-inserted into the timeout queue instead of being put back onto the free list.

```
int handle_timeoutq_event( void );
```

This function looks at the front of the timeout queue and, if the timeout has expired, or is about to expire within the next microsecond or so, then the event's function is executed, and the corresponding data value is passed to it. If this happens, then the corresponding event structure is removed from the list. It is either placed back onto the free list, or it is re-inserted into the timeout queue, depending on the value of the event's *repeat* variable, which was initialized in the `create_timeoutq_event` function. The function returns a Boolean value representing whether or not an event was handled. The outer loop will use this to decide whether or not to keep checking the queue for expired events. We will return control to the outer loop in this example (as opposed to having the `handle_timeoutq_event` function walk the list, handling every single event that has reached its timeout, and only stopping once it reaches the end of the list or an event with a still-positive timeout value),

© Copyright 2016, 2019 Bruce Jacob, All Rights Reserved

because we want to handle not only timed events but asynchronous events, and we do not want to have to write reentrant list-handling code.

Your functions should work together in the following code (this is taken from kernel.c):

```
init_timeoutq();       // given to you

// create some timeout events
namenum_t data;

data.num = 3;
bring_timeoutq_current();
create_timeoutq_event( 2 * ONE_SEC, 4 * ONE_SEC, do_hex, data );

data.num = 10;
bring_timeoutq_current();
create_timeoutq_event( 3 * ONE_SEC, 4 * ONE_SEC, do_blink, data );

data.num = 0xabcde0123456789;
bring_timeoutq_current();
create_timeoutq_event( 4 * ONE_SEC, 4 * ONE_SEC, do_hex, data );

data.name[0] = 'H';
data.name[1] = 'e';
data.name[2] = 'l';
data.name[3] = 'l';
data.name[4] = 'o';
data.name[5] = '.';
data.name[6] = 0;
data.name[7] = 0;
bring_timeoutq_current();
create_timeoutq_event( 5 * ONE_SEC, 4 * ONE_SEC, do_string, data);

while (1) {

        if (handle_timeoutq_event()) {
                continue;
        }

        timeout = bring_timeoutq_current();
        wait(timeout);

}
```

As with a number of the projects to come, the timeout queue is programmed in C and does not interact with the ARM peripherals directly, so you can first build and test the facility on your laptop, before trying to run it on the Raspberry Pi board. If you do so, you will have to "fake" the time-related facilities such as *wait()* and *get_time()* … but it will allow you to debug your code.

Optional Extra Credit

When you look at the code, you may notice that the *blink_led()* function has been renamed to *blink_led_stall()* to indicate that it performs its operation by stalling between turning the LED on and off. Through this implementation, which is simple but effective, it *also* effectively blocks the CPU from doing anything useful in the mean time until it's done blinking the LED—which is important because it means that it also blocks the *kernel* from doing anything useful in the mean time until it's done. Therefore there are two versions of the *do_blink()* function called in the code above. The first version looks like one would expect:

```
void do_blink( namenum_t data )
{
   blink_led_stall(data.num);
}
```

The second version is quite a bit more involved:

```
void do_blink( namenum_t data )
{
    int i;
    namenum_t foo;
    foo.num = 0;
```

© Copyright 2016, 2019 Bruce Jacob, All Rights Reserved

```
        bring_timeoutq_current();
        for (i=0; i<(data.num*2); i++) {
            if (i & 0x1) {
                create_timeoutq_event( 50 * i * ONE_MSEC, 0, led_off, foo );
            } else {
                create_timeoutq_event( 50 * i * ONE_MSEC, 0, led_on, foo );
            }
        }
    }
```

The two versions of the code are enabled/disabled by setting the #if statement in the code to either 0 or 1.

Among other things, the second example shows why the function *create_timeoutq_event()* does not need to bring the time current itself—if it is desired to create a string of events, all relative to *now(),* then you should only update the time once at the beginning. If instead you bring the timeout queue current each time, you are effectively compacting time for those events.

What this second version accomplishes is a way to do a string of events that will not block the kernel from doing other things. By using the timeout queue instead of just spin-waiting, the kernel is free to interleave various things together to achieve a globally consistent timing of events. For example, what do you think would happen if the *do_blink()* function were given an input value of more than 10? Note that it works by blinking on and off every 10th of a second.

Getting your timeout queue implementation working with the first version is a bit easier than getting it to work with the second version, which is running things at 0.05sec (50ms) time granularities instead of time granularities of 1 second. Also, the blinking should be regular, and the on/off time periods should be the same—if not, it will be noticeable. And, lastly, using the timeout queue implementation would allow you to have blinking times in excess of one second, without weirdness happening.

There is also a *do_butter()* function for you to try in place of *do_blink(),* if you are interested. This causes significant overlap between the various tasks, and it does the age-old musical timing of putting a 4-beat on top of a 3-beat.

So, the Extra Credit Part?

To get extra credit, run your code with the more complex version of *do_blink(),* which turns the LED on and off through your timeout queue, and set the number of blinks in the kernel to be 20 instead of 10 (replace ***data.num = 10;*** with ***data.num = 20;*** in the kernel.c code above). If you make this attempt, you'll get a maximum of 5pts on the project, instead of a maximum of 4pts (an additional 25%).

Build It, Load It, Run It

Once you have it working, show us.

One question: *why does the software sometimes start working just fine, and then all of a sudden stop working?*

© Copyright 2016, 2019 Bruce Jacob, All Rights Reserved