Project 1: Timeout Queue (4%)

ENEE 447: Operating Systems — Spring 2012
Assigned: Monday, Feb 1; **Due:** Friday, Feb 12

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 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 many 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.

Build a Timeout Queue

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

```
void create_timeoutq_event( int timeout, int repeat, pfv_t function, unsigned int data );
```

This function takes a pointer to a function (which returns void, i.e., a pfv_t) as well as a simple piece of 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. If the *repeat* value sent into the function 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 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.

```
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), 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:

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

create_timeoutq_event( ONE_SEC, 2 * ONE_SEC, blink_led, RED );
create_timeoutq_event( 6 * ONE_SEC, 2 * ONE_SEC, blink_led, GRN );
create_timeoutq_event( 11 * ONE_SEC + 500 * ONE_MSEC, ONE_SEC, blink_led, GRN | RED);

while (1) {

        if (handle_timeoutq_event()) {
                continue;
        }

        timeout = bring_timeoutq_current();
        wait(timeout);

}
```

Again, this is programmed in C and does not interact with the ARM peripherals, so you can first build and test the facility on your laptop, before trying to run it on the Raspberry Pi board. You will have to "fake" time-related facilities such as wait() and gettime() … but it will allow you to debug your code.

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?*