MTE 241 - LAB 4

Andrew Morton and Bernie Roehl, Fall 2018

Introduction

Message queues are a common form of inter-process communication, and are widely used in embedded real-time systems. In this lab we'll be introducing you to the use of message queues, and show how they're used in a client-server context.

Learning Objectives

In this lab, you will…

- Become familiar with the basics of queueing theory
- Learn how to simulate queues in software
- Learn how to generate an exponential distribution of arrival times from a uniform random variable

About Queues

Queueing theory is a way of modeling the behaviour of client-server systems. In many real-world scenarios, the arrival of items in a queue is non-uniform and is described as *stochastic* (i.e. random). Examples include connection requests at cell towers, cars arriving at a roundabout, and people at a movie theatre waiting in line to buy tickets.

We describe a queue using *Kendall notation*, which consists of (at least) three parameters -- the arrival distribution, the departure distribution and the number of servers. For this lab, the arrival and distribution processes will both be Markovian, so the queues will be described in Kendall notation as M/M/*n* where *n* is the number of servers.

Consider a movie theatre. There's a single queue, and patrons arrive at the theatre more or less randomly. There may be multiple servers (ticket sellers), who take a variable length of time to service requests. If there are five ticket sellers, this would be a single M/M/5 queue.

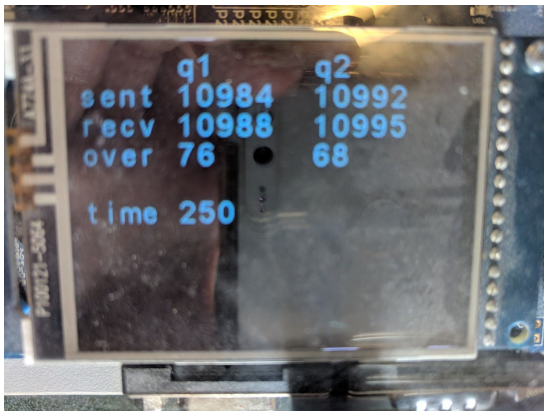
An alternate approach is to have lineups similar to those at the Canadian National Exhibition. There are a number of separate ticket wickets, and as people arrive they choose which wicket to line up at (usually based on the number of people waiting at each wicket). It is this approach that we'll be using in this lab. You will be creating two queues, each with its own server, so each queue will be M/M/1.

What You Are Expected to Do

You will need to create two message queues using the `osMessageQueue` support in the Keil RTOS. You will create a client thread that sends messages into the queues, alternating between them, and following a Markovian distribution of inter-arrival times (i.e. the time between messages follows an exponential distribution). You will create a server thread for each queue, which will pull messages out of the queue. The content of the messages is not important; they just represent work requests from client to server.

You will need to keep count of the number of messages sent to each queue, the number received from each queue, and the number discarded for each queue because the queue was full. You should also display the elapsed time in tenths of seconds.

You should have an additional thread that displays these statistics on the LCD screen, as shown below:



Keil RTOS Message Queues

The Keil RTOS provides *message queues* for inter-thread communication. The full API for Keil RTOS message queues can be found here:

https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__Message.html

Remember that in order to access the RTOS functions, you must have the following in your source code:

```
#include <cmsis_os2.h>
```

To create a new message queue with space for 10 messages each of which is an integer, you would do this:

```
osMessageQueueId_t q_id = osMessageQueueNew(10, sizeof(int), NULL);
```

The third parameter is a pointer to a struct that defines the message queue attributes. Passing NULL accepts the default message queue attributes. The message queue identifier is returned on success. You need to use this identifier in subsequent get and put operations.

To send `int msg` with try semantics (without blocking), use the following function:

```
osMessageQueuePut(q_id, &msg, 0, 0);
```

The third parameter is the priority, and the fourth parameter is the timeout. With try semantics, the call will return `osErrorResource` if there is not enough space in the queue.

To receive a message from a queue with blocking, you use the following function:

```
osMessageGet(q_id, &msg, NULL, osWaitForever);
```

The third parameter points to a buffer to record the message priority or NULL to discard it.

Client and Server Threads

You will create a total of four concurrent threads for this project -- a client, two servers (one for each queue) and one monitor thread that displays the system status on the LCD screen.

The client thread will wait a random period of time (see below for details) and then send a message to one of the two queues, alternating between them. It repeats this indefinitely.

Each server thread will start by determining which queue it's watching, then wait for a random period of time before retrieving a message from its queue. It repeats this indefinitely

The Monitor Thread

The monitor thread should run once a second, and should display the following for each of the two queues:

- Total messages sent successfully
- Total messages received
- Total overflows (this occurs when `osMailAlloc()` returns NULL)
- Elapsed time in seconds

Delays and Randomness

The client and server threads will both need to delay for a random period of time. The function for delaying is simply:

`osDelay(nticks):`

The `nticks` value is the number of system ticks to delay before returning.

For this lab, you will use the following values:

- Average arrival rate at ***each*** of the two queues is 9 Hz
- Average service rate for each queue is 10 Hz
- The size of each queue is 10
- Use osKernelGetTickFreq() to get ticks/second

Note that the arrival and service rates are *average* values. The actual inter-arrival times and service times will be different for every iteration through your thread's loop, and will vary exponentially.

Since the delays vary exponentially, we must compute the next delay value by taking a random number in the range (0,1] and mapping it to an exponential distribution via the following equation:

$$F^{-1}(x) = -\ln(1-x)$$

We would then take the result and divide it by the average arrival or service rate (the values given above) to generate the next delay value.

Fortunately for you, we provide code that does most of the work. The reason we do this is that we won't be using floating point.

On many embedded processors, including the one we're using on the Keil board, there is no hardware support for floating-point operations, which makes them very slow. Instead of using floating-point, we use a pseudo-random number generator (PRNG) algorithm called `lfsr113()` that returns values in the range `[0,UINT32_MAX]`.

We provide you with a file called `random.c` (which you must add to your source group in uVision) and an include file called `random.h`, (which you must `#include` in your code). The `random.c` file contains a `next_event()` function that returns the actual delay in seconds, handling all the random number generation and the conversion to an exponential distribution.

However, there's a catch. The `next_event()` function converts a pseudo-floating-point value into an integer that's been multiplied by 2^{16} (i.e. left shifted 16 bits) and rounded to the nearest integer value. To convert this to a value suitable for `osDelay()`, you must convert it from seconds to ticks, divide by the average rate (described above), then right shift it by 16 bits.

Assigning Queues to Servers

When you create a server thread, you must pass it a parameter so it knows which queue to process.

Getting the Code and Getting Set Up

The setup procedure is the same as with previous projects, and you can refer to the earlier lab manuals if you're uncertain. Note that in addition to the files you've seen in previous projects, there are several new ones: `random.c, random.h, lfsr113.c,`

`lfsr113.h`. Do not add `make_table.c` to your project. It was used to generate the lookup table in `random.c`.