ECE 254/MTE 241 Lab1
Inter-process Communication by Message Passing
Last updated: 2012/10/01

Objective

This lab is to learn about, and gain practical experience in message passing for the purpose of inter-process communication. In particular, you will use the POSIX message queue facility in a general Linux environment and the mailbox APIs in RL-RTX on the Keil LPC1768 board.

After this lab, students will have a good understanding of, and ability to program with

- the `fork()` and `exec()` system calls, and their use for creating a new child process on the Linux platform;
- the `wait()` family system calls, and their use to obtain the status-change information of a child process;
- the POSIX message queue facility (`<mqueue.h>`) on the Linux platform for inter-process communication;
- the creation of tasks using RL-RTX; and
- the use of mailbox APIs in RL-RTX for inter-task communication.

Starter files

Download the `lab1_example.zip` file from the lab website. It contains the following:

- Linux child-process example code [1],
- Linux POSIX message queue API example code, and
- LPC1768 frequently used example code.

Pre-lab Preparation

1. Read Section 3.2.2 about `fork()` and `exec()` and Sections 3.4.1 and 3.4.2 about `wait()` in [1].
2. Complete the example code of `wait()` on page 57 in [1] by including necessary header files and compile it. Execute it to examine the difference of the output from the output of the code in Listing 3.4,
3. Read the man page (section 7) of `mq_overview` (`http://linux.die.net/man/7/mq_overview`),
4. Build and execute the example code in the `mqueue` sub-directory in the `lab1_example.zip` file, and
5. Read the Keil μ`Vision4` IDE getting started guide (posted on the lab web site). Install the MDK-ARM on your own personal computer.

Lab Tutorial

- How to create your first Keil RL-RTX application, and
- How to view and control the RTX operation in debug mode.

Lab1 Assignment

Solve the producer-consumer problem by message passing both on Linux and on the Keil LPC 1768 board.

This is a classic multi-tasking problem in which there are one or more tasks that create data (these tasks are referred to as "producers") and one or more tasks that use the data (these tasks are referred to as "consumers"). For the purpose

of this assignment, we will use a single producer and single consumer. The producer will generate a fixed number, N, of random integers, one at a time. Each time a new integer is created, it is sent to the consumer using the system message-passing facility (POSIX queues for Linux, mailboxes for Keil). The consumer task receives the the data using the system's message-passing facility, and prints out the integer it has read[1].

Note that because we are using message passing, the two processes will not share any memory. Consequently, we do not need to worry about conflicting operations in shared-memory access: the operating system's message-passing facility takes care of the shared access within kernel space. However, kernel memory is finite, and thus there cannot be an unbounded number of messages outstanding; at some point the producer must stop generating messages and the consumer must consume them, otherwise the kernel's memory will be completely consumed with messages, blocking the sender from further progress. What is needed, therefore, is to set up the correct queue size. When the queue is full, the producer is blocked by the system and cannot continue to send messages until a message is consumed.

Define B as the number of messages that the producer may send without the consumer having consumed them before the producer must stop sending (in general N > B.). This is the total number of messages that the operating system must be able to store, *e.g.,* within a message queue or a mailbox.

Just as the producer may block if it has sent B unconsumed messages, the consumer may block if there are currently no messages to consume but more are expected from the producer. The program terminates when the consumer has read and displayed all N integers from the producer.

Figure 1 is taken from [2]. The figure lists the pseudo code of solving multiple producers and consumers problem by using message passing, which also applies for a single producer and single consumer problem solution. You may want to consider to implement this solution, though you are not limited to it.

Requirements:

- On the Linux platform, create a producer-consumer pair of processes with a fixed-size message queue in which the consumer is a child process of the producer (use the `fork()` system call). The system is called with the execution command

  ```
  ./produce <N> <B>
  ```

  where N is a parameter specifying the number of integers the producer should produce and B is the number of integers the message queue can hold. Note that in general N > B. The command will execute per the above description and will then print out the time it took to execute; the time for forking is fixed relative to the operation and there fore should be kept separate in your timing measurement. Therefore you should measure the time before the fork, before the first integer is generated, and after the last integer is consumed and displayed. You should then print the time differences. On Linux, use `gettimeofday()` for to measure the time and the terminal screen for display. Thus your output should be something like:

  ```
  Time to initialize system: <whatever the result is>
  Time to transmit data: <whatever the result is>
  ```

- On the Keil LPC 1768 board, create your own timer task to measure the time and use the LCD screen for display. The system initialization time is the time for creating a task by RL-RTX. Similar to measuring the time before the `fork()` on Linux, you should measure the time before calling the `os_tsk_create` family APIs on the LPC1768 board. Since there are no keyboards on the board, you can hard code the values of N and B in your code. You may want to use arrays to stores all the <N,B> values and write a task to go through each <N,B> value for each experiment and run a set of experiments in batch rather than re-compile and download your code for each different <N,B> pair.

[1] If it helps, you can think of the producer as a keyboard device driver and the consumer as the application wishing to read keystrokes from the keyboard; in such a scenario the person typing at the keyboard may enter more data than the consuming program wants, or conversely, the consuming program may have to wait for the person to type in characters. This is, however, only one of many cases where producer/consumer scenarios occur, so do not get too tied to this particular usage scenario.

```
const int
    capacity = /* buffering capacity */ ;
    null = /* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce,pmsg);
     pmsg = produce();
     send (mayconsume,pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume,cmsg);
     consume (cmsg);
     send (mayproduce,null);
    }
}
void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1;i<= capacity;i++) send (mayproduce,null);
    parbegin (producer,consumer);
}
```

Figure 5.21 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

Figure 1: Pseudo code screen shot taken from [2]

Post-lab Deliverables

Submit the following items to the course book system before the deadline. Please *only submit once* and submit by group (*i.e.* not individually). We will grade the last (NOT the first) submission when there are multiple submissions of the same item.

1. Source code with a `README` containing build instructions.
   Zip the entire source code together with a `README` file and name the `.zip` file `lab1_src.zip`.
2. A lab report named as `lab1_rpt.docx` or `lab1_rpt.pdf` which contains the following items.
   - The results of your program on one of the `ecelinux` machines.
     - Use tables or graphs (or both) to show how the average initialization time and average data-transmission time varies as `(N, B)` varies.
       Table 1 shows the `(N, B)` values you are required to present in the tables or graphs. You are free to present more timing data from `(N, B)` values not listed in Table 1 if you wish. If you want to use graphs, you may want to consider log-log plots.
     - A table to show the standard deviation of the data transmission time for given `(N, B)` values in Table 1.
     - A histograms of the average data transmission time given `(N, B) = (320, 10)`.
   - The results of your program on Keil LPC1768 board.
     - Use tables or graphs (or both) to show how the average initialization time and average data-transmission time varies as `(N, B)` varies.
       Table 1 shows the `(N, B)` values you are required to present in the tables or graphs. You are free to present more timing data from `(N, B)` values not listed in Table 1 if you wish. If you want to use graphs, you may want to consider log-log plots.

		B				
		1	2	4	8	10
N	20	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
	40	t_{21}	t_{22}	t_{23}	t_{24}	t_{25}
	80	t_{31}	t_{32}	t_{33}	t_{34}	t_{44}
	160	t_{41}	t_{42}	t_{43}	t_{44}	t_{45}
	320	t_{51}	t_{52}	t_{53}	t_{54}	t_{55}

Table 1: Timing measurement data table for given (N, B) values on Linux.

  - Discuss whether you need to execute the program multiple times for a given `(N, B)` pair in order to compute the average transmission time.

- A comparison of your testing results on Linux and LPC1768. Discuss the cost difference between these two platforms for solving the producer-consumer problem by message passing. Your conclusion should be based on comparing the performance difference (if any) by using the timing measurement data.
- An appendix that contains your source-code listing. You only need to include the header file and the C file you wrote. You do not need to include system pre-defined files such as `startup_LPC17xx.s` in the appendix unless you have made modification to these files.

References

[1] M. Mitchell, J. Oldham, and A. Samuel. Advanced linux programming. *Available on-line at http://advancedlinuxprogramming.com*, 2001.

[2] W. Stallings. *Operating systems: internals and design principles*. Prentice Hall, seventh edition, 2011.