Thread2 Demonstration

Files

The files for this demonstration can be found in the rlogin cluster in the directory

`/web/courses/cs3214/spring2014/butta/examples/thread-demo/thread2`

The files are `bounded-buf.c bounded-buf.h int-pipe1.c int-pipe2.c int-pipe3.c int-pipe4.c int-pipe.h, int-pipe1.h, int-pipe2.h, int-pipe3.h, int-pipe4.h, Makefile prime.c prime.h rngs.c rngs.h thread-ipipe.c thread-pipe.c`

The `"make" command` by default will create four executables named `tp, tip1, tip2, tip3, and tip4.` The programs for these executables have different ways of performing the same simple task – generating a sequence of random numbers and testing whether each number is a prime. Prime random numbers are used in cryptographic security systems but their purpose here is simply to have work for threads to perform.

The files are named to help indicate what code is used in each executable as shown in this table.

executable	main code	uses
tp	thread-pipe.c	Unix pipe
tp1	thread-ipipe.c	int-pipe1.c, int-pipe1.h
tp2	thread-ipipe.c	int-pipe2.c, int-pipe2.h
tp3	thread-ipipe.c	int-pipe3.c, int-pipe3.h
tp4	thread-ipipe.c	int-pipe4.c, int-pipe4.h

The files for the random number generation (`rngs.c` and `rngs.h`) and prime testing (`prime.c` and `prime.h`) are used in all versions. All of the versions of `int-pipex.c` implement the same interface for a circular queue but they use different approaches for synchronization.

Purpose

The purposes of this demonstration are

- to see how Unix pipes can be used for inter-thread communication
- to see how arguments are passed to threads and how values are returned from threads
- to see how to use locks to guarantee mutual exclusion of critical sections
- to see an example of an atomicity violation
- to see how to use locks for signaling
- to see how to use condition synchronization

Step 1

1. Use the `Makefile` to create the executable programs with the command "`make`".
2. At the shell prompt execute `tp`. Observe the output that is produced. Execute `tp` several times and observe the output in each case. Examine the code in `thread-pipe.c`. Briefly read the description of the producer-consumer problem in http://en.wikipedia.org/wiki/Producer-consumer_problem. Answer questions 1 and 2.

Step 2

1. Examine the code in `bounded-buf.h` and `bounded-buf.c`. This code implements a simple circular queue. Read the code sufficiently to be convinced of this. Examine the code in `int-pipe1.h` and `int-pie1.c`. Note the comments at the beginning `int_pipe_write` and `int_pipe_read` referring to "busy waiting." Read briefly http://en.wikipedia.org/wiki/Busy_waiting for a description of busy waiting. Note also the two assert statements in the two functions `int_pipe_write` and `int_pipe_read`. Use "`man 3 assert`" at the shell prompt to see the man page about `assert`. Answer question 3.
2. Examine the code in `thread-ipipe.c` and compare it to the code in `thread-pipe.c`. Answer question 4.
3. At the shell prompt execute `tip1` which uses `thread-ipipe.c` and `int-ipipe1.c`. Observe the output that is produced. Execute `tip1` several times and observe the output in each case. Answer questions 5 and 6.

Step 3

1. Examine the code in `int-pipe2.h` and `int-pipe2.c`. In particular, notice the `int_pipe_read` function. Answer question 7.
2. If you execute `tip2` some number of times you will likely see output similar to this:

   ```
   tip1: int-pipe1.c:51: intp_pipe_read: Assertion
   '!int_pipe_empty(intp) && !int_pipe_closed(intp)'
   failed.
   ```

   Answer question 8.

Step 4

1. Examine the code in `int-pipe3.h` and `int-pipe3.c`. In particular, notice the `int_pipe_read` function and the `int_pipe_write` function. Answer questions 9 and 10.
2. If you execute `tip3` some number of times you will not see the assertion failure seen in Step 3 above. Look carefully at the code in `int_pipe_read` in `int-pipe3.c` and answer question 11.

Step 5

1. Examine the code in `int-pipe4.h` and answer question 12.
2. Compare closely the while loop in the `int_pipe_read` function in `int-pipe3.c` and `int-pipe4.c`. Also note the one line change in the `int_pipe_write` function between the two versions. Answer question 13.

Questions

Based on your observations above, answer these questions:

1. How does the description of the producer-consumer problem differ from the code in `main()`? How many threads play the role of the producer? How many threads play the role of the consumer? What plays the role of the bounded buffer? What code does a producer thread execute? What code does a consumer thread execute? What argument is passed to a producer thread? What argument is passed to a consumer thread? Explain how a random number generated in `rand_gen` is made available to be tested for being a prime number in `prime_test`?
2. Does each execution of `tp` produce results that account for all of the numbers that are produced? Though we do not see the actual numbers, would you believe that the executable performs its function of producing random prime numbers? Does the execution of `tp` appear to be deterministic? If not, to what do you attribute the non-determinism? Does this analysis give you a different understanding of non-determinism?
3. What does "busy waiting" mean? What is a major disadvantage of "busy waiting? What does an `assert` statement do in general? What role do they play in the implementation of `int_pipe_write` and `int_pipe_read`?
4. What is the major difference between `thread-pipe.c` and `thread-ipipe.c.`
5. Does each execution of `tip1` produce results that account for all of the numbers that are produced? Does the execution of `tip1` appear to be deterministic? If not, to what do you attribute the non-determinism?
6. Are there critical sections in the code of the `bounded_buffer_add` and `bounded_buffer_remove` functions in `bounded_buf.c`? If so, what shared variables are manipulated in these critical sections? Can you describe a sequence where two threads execute `int_pipe_read` and both retrieve the same value from the bounded buffer? Can you describe a sequence where two threads execute `int_pipe_read` and the size of the bounded buffer is left in an incorrect state?
7. What synchronization element has been added to the implementation of the bounded buffer in `int-pipe2`? Will the use of this synchronization element in the `int_pipe_read` and `int_pipe_write` functions guarantee that the internal variables in the bounded buffer are protected against concurrent execution? Explain why this is or is not the case? In the `int_pipe_read` function in `int-pipe2.c` why is

the while loop before the lock operation? Should this not be inside the critical section so that the state variables of the pipe are protected from concurrent access?

8. How is it possible for this assertion in `int_pipe_read` to have failed given the conditions on the while loop and if statements that immediately precede the assertion? Can you identify an atomicity violation? How would you summarize the two major deficiencies of the `int-pipe2.c` implementation?
9. What synchronization element has been added to the implementation of the circular queue in `int-pipe3`? How does this new element help to synchronize the actions in `int_pipe_read` and `int_pipe_write`?
10. What significance is there to this code sequence in `int_pipe_read`:

```
pthread_mutex_unlock(&intp->mutex);
pthread_mutex_lock(&intp->avail);
pthread_mutex_lock(&intp->mutex);
```

    Why is this code in the body of the while loop? What is true when the loop is terminated? How does this code related to the code in `int_pipe_write` that unlock the mutex variable `avail`?

11. Does the `int-pipe3.c` implementation use busy waiting for reading?
12. Looking in `int-pipe4.h` how has the "`avail`" synchronization element changed from `int-pipe3.h`?
13. Based on your comparison of the code can you form a hypothesis about the meaning of the `pthread_cond_wait` operation? Can you also form a hypothesis about the meaning of the `pthread_cond_signal` operation?

Extensions

1. Use condition synchronization to eliminate the busy waiting in the `int-pipe-write` function. You will need to introduce another `pthread_cond_t` variable.
2. Test your implementation by creating two or more threads that write random numbers to the circular queue.