Named Pipe for Exchanging Lines

Objective

You will implement a kernel-level pipe for exchanging lines among user-level processes. You will learn about concurrency, synchronization, and various kernel primitives. You may find the following resources helpful in addition to any other online resources.

- Slides: Kernel API for Semaphores and Waitqueues
- Linux Device Drivers book
- Any other online resources

Task A

Learn what is a named pipe. Learn how to create and use a named pipe using command line in Linux (or any UNIX system).

Task B

Read, understand, and run, following two user-level C-programs for a consumer and a producer.

- Consumer C program (Updated April 20, 5:20PM)
- Producer C program (Updated April 20, 5:20PM)
  You can ignore the bash scripts that were released earlier (linked below for reference). Explanation: While these scripts are fine by themselves, the `read line` command in consumer was reading the device byte-by-byte, causing confusion when used with a character device.
  - Old Consumer Script
  - Old Producer Script
- Run one consumer and one producer concurrently.
  - Kill the producer with Ctrl-C. Leave consumer

    - running. What happens and why?
    - Kill the consumer with Ctrl-C. Leave producer running. What happens and why?
- Run one consumer and multiple producers concurrently.
- Run multiple consumers and one producer concurrently.
- Run multiple consumers and multiple producers concurrently.

Explain in your report, what you see, and why.

Task C

Now solve the problem you observed in Task B as follows. Replace the UNIX named pipe in the producer and consumer scripts above with your own implementation of a miscellaneous character device (say /dev/linepipe) as a kernel module in the Linux Kernel. This device must maintain a FIFO queue (i.e. a pipe) of maximum N lines, where N is configured as a module parameter. A line is a sequence of characters terminated by newline character '\n'.

- Producers write lines to /dev/linepipe.
- Consumers read lines from /dev/linepipe and print it on the screen.
- When the pipe is full, i.e. when there are N lines are stored in /dev/linepipe, then any producer trying to write will block.
- When the pipe is empty, i.e. when there are no lines in /dev/linepipe, then any consumer trying to read will block.
- When a consumer reads from a full pipe, it wakes up all blocked producers. In this case, no blocked consumer should be woken up.
- When a producer writes to an empty pipe, it wakes up all blocked consumers. In this case, no blocked producer should be woken up.
- No deadlocks. All producers and consumers make progress as long as at least one of each is running.
- No race conditions. Different lines should not get mixed up (interleaved), as it did in Task B. Each line that is written by producers is read EXACTLY once by one consumer. No line is lost. No line is read more than once.

What you need to learn to complete this assignment

You can use either the semaphore-version of the solution to producer-consumer problem or a monitor-type solution, both of which were covered in class. It is likely the semaphore version may be easier to implement. You will need to learn the following kernel mechanisms.

- Using semaphores in kernel using the following functions: sema_init(), DEFINE_SEMAPHORE, down_interruptible() (preferred over down()), and up().
- For alternative implementations using mutexes and waitqs: mutex_init(), DEFINE_MUTEX, mutex_lock(), mutex_unlock(), init_wait_queue_head(), wait_event_interruptible() (preferred over wait_event), and wake_up_interruptible() (or wake_up).
- Memory allocation in kernel using kmalloc or kzalloc.

Frequently asked questions

Q: Is there any locking in user space?
A: No, all synchronization happens in kernel space.

Q: Do we implement our own producers and consumers in user space?
A: No, use the two scripts given above.

Q: How to I terminate producers and consumers?
A: After fixed number of iterations OR using Ctrl-C.

Q: Why should I use *_interruptible versions of kernel functions?
A: So that your producer/consumer code can be terminated using Ctrl-C in user space. We'll test for this during demo.

Q: How does the producer generate a unique line?
A: Please see scripts above. "Producer [pid] Line [i]", where [pid] is the process ID of the producer and [i] is a number that increments each iteration.

Q: How long should be each line? Can we assume fixed

length?
A: Whatever length is needed to store the above line.

Q: How to I run multiple producers and consumers concurrently?
A: Open multiple terminals. Run one consumer or producer in the foreground each terminal.

Grading Guidelines

- 20 - Code works for one producer and one consumer.
- 30 - Code works for multiple concurrent producers and consumers.
- 20 - No deadlocks. All producers and consumers make progress as long as at least one of each is running.
- 20 - No race conditions. Each line is read EXACTLY once by one consumer. No line is lost. No line is read more than once.
- 10 - Producers block on write when pipe is full.
- 10 - Consumers block on read when pipe is empty.
- 10 - Blocked producers are not woken up by other producers. Blocked consumers are not woken up by other consumers.
- 10 - Blocked producers and consumers can be terminated cleanly using Ctrl-C
- 20 - Handle all major error conditions.
- 10 - Clean, modular, and commented code. Clean readable output. No "giant" functions. Minimal use of global variables.