# Named Pipe for Exchanging Numbers

## Objective

You will implement a kernel-level pipe for exchanging numbers 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.

- Linux kernel module programming
- Miscellaneous Character Devices
- Kernel API for Semaphores and Waitqueues
- Linux Device Drivers book

## Task A - Learn kernel modules, character devices, and named pipes

1. Learn how to create kernel modules from chapters 1, 2, and 3 of Linux kernel module programming website.
2. Try the Example module code and understand how it works.
3. Learn how to write a *character device kernel module* from Chapter 4 of Linux kernel module programming website.
4. Learn what is a *named pipe* in UNIX.
5. Figure out how to create and use a named pipe using command line in Linux using the `mkfifo` command.

## Task B - Study userspace producer consumer programs.

1. Read, understand, and run, following two user-level C-programs for a consumer and a producer.
    - Consumer C program
    - Producer C program
2. Run one consumer and one producer concurrently. To do this, open two text terminals. Run a consumer in one and a producer in another at the same time.)
    - 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?
3. Run one consumer and multiple producers concurrently.
4. Run multiple consumers and one producer concurrently.
5. Run multiple consumers and multiple producers concurrently.
6. Note down what you observe, and why, in your README file. Do you see any race conditions? If not, can you reason about what could go wrong?
7. Learn the Kernel API for Semaphores and Waitqueues.

## Task C - Creating a Number Pipe

Replace the UNIX named pipe in the producer and consumer scripts from Task B with your own implementation of a character device (say /dev/numpipe) as a kernel module in Linux. (You can also implement a miscellaneous character device for this part, instead of a regular character device.) This device must maintain a FIFO queue (i.e. a pipe) of maximum N numbers. N must be given as a module parameter during the insmod command.

1. Producers write numbers to /dev/numpipe.
2. Consumers read numbers from /dev/numpipe and print it on the screen.
3. When the pipe is full, i.e. when there are N numbers are stored in /dev/numpipe, then any producer trying to write will block.

4. When the pipe is empty, i.e. when there are no numbers in /dev/numpipe, then any consumer trying to read will block.
5. When a consumer reads from a full pipe, it wakes up all blocked producers. In this case, no blocked consumer should be woken up.
6. When a producer writes to an empty pipe, it wakes up all blocked consumers. In this case, no blocked producer should be woken up.
7. No deadlocks. All producers and consumers make progress as long as at least one of each is running.
8. No race conditions. Each number that is written by producers is read EXACTLY once by one consumer. No number is lost. No number 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. 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 for Part C?
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 C programs given above.

Q: How do 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 number?
A: Please see C programs above.

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

# Submission Guidelines

- For Parts A and B: Submit any code that you wrote and README file as one zipped file.
- For Part C: Submit your code, README, and Makefile for Part C only.

# Grading Guidelines

- Parts A and B : 30

- Part C: 70
  - Code works for one producer and one consumer.
  - Code works for multiple concurrent producers and consumers.
  - No deadlocks. All producers and consumers make progress as long as at least one of each is running.
  - No race conditions. Each number is read EXACTLY once by one consumer. No number is lost. No number is read more than once.
  - Producers block on write when pipe is full.
  - Consumers block on read when pipe is empty.
  - Blocked producers are not woken up by other producers. Blocked consumers are not woken up by other consumers.
  - Blocked producers and consumers can be terminated cleanly using Ctrl-C
  - Handle all major error conditions.