Project 3: UMBCTube and the Bounded Buffer

Due November 28th, 11:59:00 PM
Version 1.0.1 (11/8/2012)

11/8/2012: Updated GitHub instructions to use the "main" branch.

Objectives

- Solve the Bounded Buffer problem, also known as producer-consumer, using a **circular buffer** and **semaphores** in user space. Information can be reviewed in the following sources:
    - Silberschatz textbook chapters 6 and 7
    - https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem
    - https://en.wikipedia.org/wiki/Circular_buffer
    - https://en.wikipedia.org/wiki/Semaphore_(programming)
- Implement a set of system calls to provide a circular buffer available to *concurrent* consumer and producer threads in kernel space.

Background

In the First Browser War of 1995, Bill Gates' Microsoft Corporation and the Netscape Communications Company competed to determine who could dominate the emerging market for web-browsers, ultimately resulting in Netscape's disappearance from popular culture and the marketplace. By bundling Microsoft's browser, Internet Explorer, with Windows, one of the most popular operating systems in the world, Microsoft enjoyed several advantages, including the peculiar ability to execute portions of Internet Explorer's functionality directly in the Windows kernel. Leveraging the high performance of kernel code, Microsoft released Internet Explorer 4.0 in 1997, demonstrating a decisive performance advantage over Netscape and creating a public perception that the Netscape browser was slow. This perception, in addition to Microsoft's aggressive business strategy, saw Netscape drop from a 90% market share in 1995 to virtually 0% by the 21st century.

Unburdened by the overhead of system calls and security routines, code running in kernel space can run significantly faster than equivalent user space code.

Links
https://en.wikipedia.org/wiki/Browser_wars
http://www.nethistory.info/History%20of%20the%20Internet/index.html

Setup

This project has two parts. Part 1 is to code a producer consumer process in user space. Part 2 is to implement this process in kernel space. You will require two sets of files from GitHub:

(1) A clean copy of Linux kernel 5.10.70
(2) A copy of Project 3's files, including a working buffer similar to Project 2, ready for you to modify.

To prepare your project directory, please execute the following commands on your Debian 11.0.0 VM. If you need instructions on how to create this VM, please refer to the Project 2 documentation on Discord.

You will need to create a repository for your project. See the Submission Instructions further in this document.

```
cd /usr/src
mkdir proj3
cd proj3
git init
git remote add origin https://github.com/CMSC421-Fall21/linux5.10.git
git pull origin main
git remote rm origin
git remote add origin <your project 3 GitHub repository>
git push --set-upstream origin main
```

Following these steps, your project 3 repository is ready for you to modify, commit, and push.

To obtain the Project 3 files, please use the following download link (**Note: Do not clone this repository -- just download the files and use them in userspace and/or in your custom kernel.)**
https://github.com/CMSC421-Fall21/project3-files/archive/refs/heads/master.zip

Header File

NOTE: If you downloaded the project 3 files in the previous step, you already have this header.

In this section, we provide a header file that specifies two structures that you will use for your circular buffer -- `struct_node_421` and `ring_buffer_421`. **Use this header file exactly as is -- do not modify the header file.** We cannot properly test and grade your code if you change

any function signatures or structures defined in this header. You are allowed to create additional header files with your own helper functions.

You will use this header file for both your user space and kernel space implementations.

```
#ifndef BUFFER_H
#define BUFFER_H

#ifdef __cplusplus
extern "C" {
#endif

#ifndef __KERNEL__
#include <semaphore.h>
#else
#include <linux/semaphore.h>
#endif

#define SIZE_OF_BUFFER 20
#define DATA_LENGTH 1024

typedef struct node_421 {
    struct node_421 *next;
    char data[DATA_LENGTH];
} node_421_t;

typedef struct ring_buffer_421 {
    int length;
    node_421_t *read;
    node_421_t *write;
} ring_buffer_421_t;

#ifndef __KERNEL__
long init_buffer_421(void);
long enqueue_buffer_421(char *data);
long dequeue_buffer_421(char *data);
long delete_buffer_421(void);
void print_semaphores(void);
#endif

#ifdef __cplusplus
}
```

```
#endif

#endif
```

Part 1 User Space

This project presents a concrete example of the bounded buffer problem, requiring you to implement the following:

(1) The circular buffer from project 2.

(2) A producer that inserts data into the buffer. The producer will insert entries comprising of 1024 characters starting with zeros, then 1s, then 2s, and so on.

(3) a consumer will then take each entry out of the buffer and display the characters to the screen.

To simulate the unpredictable nature of the internet, and the timing of when the producer process would receive the next stream of bytes to load into the buffer, use the rand() function. You will need to iterate 100,000 times in a **for** loop using the `rand()` function to put random amounts of time between each iteration. The consumer process will also loop 100,000 times. Use a different seed to generate different random numbers from the producer process.

Use the POSIX pthread API for the following implementations. You will need to read the linux documentation on adding the pthread library when compiling.

The Buffer

- The buffer comprises a *circular, singly-linked list* consisting of 20 nodes, each storing 1024 characters of data and a pointer to the next node. **Note: This is the same type of data structure that you implemented in Project 2.**
    - **Note:** In your source code, make use of the defined values in buffer.h: `SIZE_OF_BUFFER` and `DATA_LENGTH`
- Users **shall** interact with the buffer through a set of system calls, which you will implement and test in user space first.
    - `long init_buffer_421(void);`
        - Initializes a new circular buffer in kernel memory, allocating memory for each node. **(Use `malloc` in user space, `kmalloc` in kernel space.)**
        - **Note: Do not reinitialize the buffer if it already exists.**
        - Return 0 on success, -1 on failure.
    - `long enqueue_buffer_421(char *data);`
        - Copies 1024 bytes from `data` into the `write` node's data variable.
        - Correctly update the buffer's length and `write` pointer.
        - **Note: We cannot insert data into an uninitialized buffer.**

        - **Note: We cannot insert data into a full buffer. When this happens, we should BLOCK the caller (Hint: use a semaphore).**
        - Return 0 on success, -1 on failure.
    - `long dequeue_buffer_421(char *data);`
        - Copies 1024 bytes from the `read` node into the provided buffer data.
        - Correctly update the buffer's length and `read` pointer.
        - **Note: We cannot dequeue an uninitialized buffer.**
        - **Note: We cannot dequeue an empty buffer. (length == 0). When this happens, we should BLOCK the caller (Hint: use a semaphore).**
        - Returns 0 on success, -1 on failure.
    - `long delete_buffer_421(void);`
        - **If it exists**, delete the buffer.
        - Free any memory (`free` in user-space, `kfree` in kernel-space) that you allocated in `initialize_buffer_421`.
        - Returns 0 if successful, -1 if not.

Additionally, when testing your code in user space, we make the function `print_semaphores(void)` available for printing the status of your semaphores.

Note: Implementing the producer and the consumer in user space is best done with the pthread library. Use semaphores for your locking mechanism. (Hint: You will probably need three of these for a proper solution.)

pthread: http://lemuria.cis.vtc.edu/~pchapin/TutorialPthread/pthread-Tutorial.pdf
semaphores (user space): https://linuxhint.com/posix-semaphores-with-c-programming/

The Producer

- Comprises a single thread or process that writes sequential blocks of characters. Each block contains 1024 char bytes. The first block will be the "0" character and incrementing with each iteration. The second block will contain 1024 “1” chars, and so on. After the character “9”, start over with “0”.
- The producer should randomly wait between 0-1 seconds prior to each insert.
- Must execute **concurrently** with the consumer.

The Consumer

- Comprises a single thread or process that retrieves sequential blocks of characters from the buffer and then displays them on the screen.
- The consumer should randomly wait between 0-1 seconds prior to each dequeue.
- Must execute **concurrently** with the producer.

Part 2 Kernel Space

In Part 2 of this project, you will convert your user space buffer functions to system calls.

Use the following system call numbers for each corresponding buffer function.

```
#define __NR_init_buffer_421 442
#define __NR_enqueue_buffer_421 443
#define __NR_dequeue_buffer_421 444
#define __NR_delete_buffer_421 445
```

In essence, the steps will be very similar to Project 2 -- we recommend reviewing the documentation on adding system calls if needed.

In kernel space, you will need to use an alternate semaphore implementation. Review the following link for details on the kernel semaphores: https://www.hitchhikersguidetolearning.com/2021/03/05/semaphore-structures-and-semaphore-apis-in-linux-kernel/

If you have a working test file with a producer and a consumer, you should need to make minimal changes to adapt to kernel space. Specifically, you should only need to convert your buffer function calls into system calls.

Reminder: You must use `kmalloc`, `kfree`, `printk`, and other kernel variations of common functions when working in kernel space.

IMPORTANT: In system calls that read/write using pointers from user space, you must use appropriate, safe functions from the kernel memory access API. For example:

- `copy_from_user`
- `copy_to_user`

Blindly trusting a user's pointers in kernel space presents serious security vulnerabilities.

See https://developer.ibm.com/articles/l-kernel-memory-access/

Test Files

This section is applicable to both Part 1 and Part 2.

Your test file should instantiate two threads -- a **producer** thread and a **consumer** thread. Each thread should use the relevant functions (or system calls, if testing kernel code) to add (producer) or consume (consumer) data from your buffer.

Submission Instructions

Submit the project using your GitHub repository that you initialized in the earlier "Setup" section.

Prior to any commit, run `make clean` and `make mrproper` to eliminate build artifacts from your project directory. Whenever you run `make mrproper`, you will need to run `make localmodconfig` and `make xconfig` again to compile your kernel. See the Project 2 instructions for details.

To submit your project:

- **Step 1:**

Accept the assignment using the links below. Please make sure you use the one for your section.
Section 1: https://classroom.github.com/a/ccfDpNkd
Section 2: https://classroom.github.com/a/X0S064a4
Section 3: https://classroom.github.com/a/SYzN3cnw
Section 4: https://classroom.github.com/a/HesrsLq7
Section 5: https://classroom.github.com/a/Tc_xEiGJ

Please use the following format to name your repository :
project-3-YOURUMBCUSERNAME
For example:
For umbc username AAAA, the repository's name will be:
project-3-AAAA

- **Step 2:**

Add your project files in the relevant directories in your proj3 kernel, commit your changes, and push to your remote GitHub classroom repository for Project 3 (after you accepted the assignment as in step 1).
If GitHub complains about large files, you likely need to clean build artifacts from your source tree -- see the previous paragraph.

These are the files that will contain the code you have created for this project. You will push your kernel to github. Make sure that the correct versions of these files are included in the kernel that you submit to github. Your project must compile. Projects that do not compile will receive a zero.

(1) **buffer.h**

(a) This should exactly match the header we provide you. **Do not modify this header.** If you need to add additional headers, feel free. (See #6)

(2) buffer.c

(a) Your **kernel** implementation of the UMBCTube circular buffer. If you are unable to get your code to run in kernel space, you may omit this file.

(3) **buffer_user.c**

(a) Your **user space** implementation of the UMBCTube circular buffer. **You must submit this, even if you have a kernel space implementation working.**

(4) README

(a) Tell us how you tackled each aspect of the problem -- the buffer, the consumer, the producer, and your test file. Include instructions on how to compile and run your test file.

(5) **test.c**

(a) A test file that exercises your buffer. See the previous section for details. You may include more than one of these files, but tell us about them in your README.

(6) Any Makefiles, additional headers, etc.

(a) You must include any Makefiles that you modify or create, as well as any custom header files your project needs. If you fail to do this, likely your project will not compile when we go to grade it.

Project Tips

1. **Implement this project in user space first!** You can score a large portion of the points for this project by producing a functional user space implementation.

2. Concurrent code is inherently difficult to implement and debug. Give yourself lots of time here.

3. The user space and kernel space semaphores are not exactly the same. Be sure to reference the APIs for each.

4. Compile often with small increments of new code. It is much harder and more time-consuming to debug larger code blocks with more errors.