Project 3: James Webb Telescope Communication and the Bounded Buffer

Due 17 April 2022, 11:59:00 PM
Version 1.0.3 (03/15/2022)

Background

Named in honor of the trailblazing astronomer Edwin Hubble, the Hubble Space Telescope is a large, space-based observatory, which has revolutionized astronomy since its launch and deployment by the space shuttle Discovery in 1990. Far above rain clouds, light pollution, and atmospheric distortions, Hubble has a crystal-clear view of the universe. Scientists have used Hubble to observe some of the most distant stars and galaxies yet seen, as well as the planets in our solar system.

Hubble's capabilities have grown immensely in its over 30 years of operation. This is because new, cutting-edge scientific instruments have been added to the telescope over the course of five astronaut servicing missions. By replacing and upgrading aging parts, these servicing missions have greatly extended the telescope's lifetime.

The Hubble's domain extends from the ultraviolet through the visible (which our eyes see) and into the near-infrared. This range has allowed Hubble to deliver stunning images of stars, galaxies, and other astronomical objects that have inspired people around the world and changed our understanding of the universe.

The James Webb telescope was launched earlier this year. It is the replacement for the Hubble telescope. The Webb telescope is observing the infrared spectrum instead of the Hubble's is in space doing deep space research. It will be sending back images to earth of what it is observing. The hope is to observe what happened just after the big bang and before the galaxies formed.

The Webb telescope will be sending back images which are made up of data. This is where you the student can assist. The data will beam to earth and be collected in an antenna. The antenna will forward the data into a computer for processing. Coincidentally, this is on the campus of The Johns Hopkins University in Baltimore. Let us keep it simple and assume the images are simply ASCII characters (0,1,2,3,4,5,6,7,8,9). It is your job to develop an application that will receive the data, control the flow of data through the buffer and display it. I am sure you can see this is the Producer / Consumer problem.

Student Work Flow

As a starting point, we are breaking this problem into parts which are below. The recommended way forward is to start in user space and move the code to the kernel space. User space is more forgiving than the kernel.

1. Circular Buffer
    1.1. Create Circular Buffer in User Space
    1.2. Create Circular Buffer in Kernel Space
2. Webb Telescope Video and the Bounded Buffer using Semaphores
    2.1. Semaphore in User Space
    2.2. Semaphore in Kernel Space
3. Webb Telescope Video and the Bounded Buffer using Monitors
    3.1. Use a Monitor instead of a Semaphore for locking in User Space

It is recommended you get the part 1 (circular buffer) working correctly before moving to part 2 (adding a semaphore).

Setup

You will require two set of files:

1. A clean copy of Linux kernel 5.10.70
2. Project3_files.zip (collection of header files for part 1,2,3)

To prepare your project directory, please execute the following commands on your Debian VM. If you need instructions on how to create this VM, please refer to the Project 1 documentation on Blackboard.

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

```
cd /usr/src/
git clone git@github.com:CMSC421-SP22/linux5.10.git cmsc421-project3
git remote rm origin
git remote add origin <SSH LINK TO YOUR REPOSITORY>
git push --set-upstream origin main
```

Following these steps, your project 3 repository is ready for you to modify, commit, and push. **project3_files.zip** is provided with the project description on the Blackboard.

Note: Each part of this project has its own header file. Part 1 and Part 2 header files are designed in a way that it can be used for user-space and kernel-space both. Part 3 header file is for user-space only. It is strongly recommended to read the entire project documentation before you start working on the project.

Part 1: Circular Buffer

In this project, we implement a set of four *system calls* in the Linux kernel that enable a user to perform the following tasks:

1. Create a circular buffer in user space and kernel space.
2. Fill the buffer with integers.
3. Print the buffer's contents to the stdout/kernel log.
4. Delete the buffer and free any allocated memory.

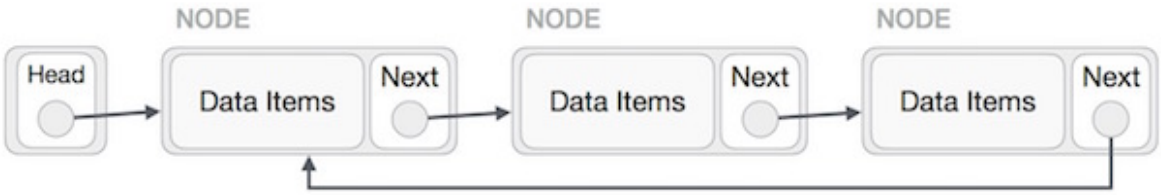
Our circular buffer comprises a singly linked list consisting of twenty nodes, each holding an integer and a pointer to the next node, that form a circular structure in kernel memory. The buffer accepts a maximum of 20 values, after which it should notify the sender that there is no additional space. In this part of the project, we do not require a function for removing items from the buffer.

We recommend beginning your development in user-space, then subsequently migrating your code into system calls in kernel-space. A later section explicitly addresses some of the differences in developing in user-space and kernel-space.

Singly Linked List Example

In this section, we provide some useful resources related to circular buffers and singly linked lists. While this project expects students to be familiar with singly linked lists, we strongly recommend reviewing how to implement singly linked lists in the C programming language.

https://en.wikipedia.org/wiki/Circular_buffer
https://en.wikipedia.org/wiki/Linked_list



(https://www.tutorialspoint.com/data_structures_algorithms/circular_linked_list_algorithm.htm)
(Note that the algorithm is flexible -- you are not required to follow the algorithm at the above link for your insertion operations.)

Header File

In this section, we provide a header file that specifies two structures that you should 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 deviate from the function signatures or structures defined in this header. You are welcome to create additional headers of your own for your project.

You can use this header for both your user space and kernel space implementations. When implementing your system calls in kernel space, you will need to specify your function prototypes in `include/linux/syscalls.h`, as we did in project 1 for our `hello` system call.

You can get this below given **buffer.h** file from **project3_files.zip** under the **buffer** directory.

```
#ifndef BUFFER_H
#define BUFFER_H

#ifdef __cplusplus
extern "C" {
#endif

#define SIZE_OF_BUFFER 20

typedef struct node_421 {
   struct node_421 *next;
   int data;
} 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 insert_buffer_421(int i);
long print_buffer_421(void);
long delete_buffer_421(void);
#endif

#ifdef __cplusplus
}
#endif
#endif
```

Part 1.1 User Space

In this section, implement the circular buffer in User Space. These are the functions that need to be implemented.

`long init_buffer_421(void)`

- Initializes a new circular buffer in memory (kernel memory for kernel space), allocating memory for each node. **(Use `malloc` in user space, `kmalloc` in kernel space.)**
- There should be 20 nodes total.
- The final node's next pointer should point to the first node to create a circle.
- Each node should store an initial integer value of 0.
- Return 0 if we successfully initialize, -1 if we fail for any reason, including our buffer already being initialized. **Do not reinitialize the buffer if it already exists.**

`long insert_buffer_421(int i)`

- Inserts the integer ***i*** into the next available node.
- Note: Increment the buffer length when you insert.
- **Note: We cannot insert data into an uninitialized buffer.**
- Inserting fails if the buffer is already full.
- Returns 0 if the insert is successful, -1 if it fails.

`long print_buffer_421`

- Writes a string representation of every node in the buffer, along with the data in each node, to the kernel log. If in user-space, print to stdout instead.
- Begins at the buffer's read pointer.
- **Note: We cannot print an uninitialized buffer.**
- Returns 0 if successful, -1 if not.

`long delete_buffer_421`

- **If it exists**, delete the buffer.
- Free any memory (`free` in user-space, `kfree` in kernel-space) that you allocated in `init_buffer_421`.
- Returns 0 if successful, -1 if not.

Part 1.2 Kernel Space

Now, implement the circular buffer in Kernel Space. In this part, You will create system calls for each of the functions mentioned in part 1.1. These system calls will work the same as functions, but in kernel space. You must use the following system call numbers for each of the system call:

```
#define __NR_init_buffer_421 442
#define __NR_insert_buffer_421 443
#define __NR_print_buffer_421 444
#define __NR_delete_buffer_421 445
```

Kernel Programming Tips

Due to the overhead of compiling your custom Linux kernel, we **strongly recommend building and testing your logic in user space first.**

When migrating your source-code into your kernel, there are several important differences to note:

1. You cannot use `malloc` and `free` in kernel space. Rather, we use the following functions:
    a. `kmalloc` (Use GFP_KERNEL for the second argument.)
    b. `kfree`
    c. See https://www.kernel.org/doc/htmldocs/kernel-hacking/routines-kmalloc.html
2. Many standard C libraries are not directly available in kernel space. To print to the kernel log, use `printk` from the `linux/kernel.h` header.

Test Files

Your test file should check for edge cases such as trying to enter data into a buffer before initialization, entering data into a deleted buffer and more.

Part 2: Webb Telescope Video and the Bounded Buffer using Semaphores

Objectives

- Until now, you have hoped there were no issues with the producer putting data in the buffer and the consumer removing data at the same time. Let us resolve this issue. Solve the Bounded Buffer problem, also known as producer-consumer, using a **circular buffer** and **Semaphores** in both user-space and kernel-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/Semaphore_(programming)
- Implement a set of system calls to provide a circular buffer available to concurrent consumer and producer threads in kernel space.

Header File

The header file below is similar to the one that is provided in part 1. It differs in terms of structure names and type of data it holds. This file (**buffer_sem.h**) is also available in **project3_files.zip** under the **prodcon_sem** directory.

```
#ifndef BUFFER_SEM_H
#define BUFFER_SEM_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
```

```c
typedef struct bb_node_421 {
    struct bb_node_421 *next;
    char data[DATA_LENGTH];
} bb_node_421_t;

typedef struct bb_buffer_421 {
    int length;
    bb_node_421_t *read;
    bb_node_421_t *write;
} bb_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 2.1: User Space

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

(1) The circular buffer from Part 1.
(2) A producer that inserts data into the buffer. The producer will insert entries consisting 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 **1000** times in a **for** loop using the rand() function to put random amounts of time between each iteration.

The consumer process will also loop 1000 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 Part 1.**
    - **Note:** In your source code, make use of the defined values in buffer_sem.h: `SIZE_OF_BUFFER` and `DATA_LENGTH`
- The producer and consumer **shall** interact with the buffer through a set of system calls(functions), which you will implement and test in user space first.
    - `long init_buffer_421(void);`
        - Initializes a new circular buffer in user/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 `init_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 0-10 milliseconds prior to each enqueue.
- The producer 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 0-10 milliseconds prior to each dequeue.
- The consumer must execute **concurrently** with the producer.

Part 2.2: Kernel Space

In this part, 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_sem_421 446
#define_NR_enqueue_buffer_sem_421 447
#define_NR_dequeue_buffer_sem_421 448
#define_NR_delete_buffer_sem_421 449
```

In essence, the steps will be very similar to Project 1 -- 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 in user-space with a producer and a consumer, you will need to make minimal changes to make it work in kernel space as well. Specifically, you should only 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 should use appropriate and 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 will present serious security vulnerabilities.

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

Test Files

Your test file should create 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 or consume data from your buffer.

Part 3: Webb Telescope Video and the Bounded Buffer using Monitor

Objectives

- Solve producer-consumer problem using a **circular buffer** and **Monitor** in user-space only.
- Implement Monitor on a circular buffer using mutex and conditional variables (You will need POSIX pthread API in this part)
- This part will differ from part 2 only in terms of synchronization mechanism. So, requirements are same as Part 2.1 with only difference that you will use Monitor for locking instead of Semaphores.
- Information can be reviewed in the following sources:
    - Section 6.7 of "Operating System Concepts", pages 266-280
    - https://en.wikipedia.org/wiki/Monitor_(synchronization)

Header File

For implementing Monitor, you will need the **buffer_mon.h** header file provided in the **project3_files.zip** under **prodcon_mon** directory.

Test Files

Test files will be the same as the user-space test file for part 2 but with Monitor setup.

Points Distribution

- Table given below indicates points associated with successful completion of each task. Detailed breakdown can be found in the rubric that is uploaded on blackboard.

Task	Environment	Approx. Points
Circular Buffer	User-Space	9
Circular Buffer	Kernel-Space	20
Producer-Consumer- Semaphores	User-Space	20
Producer-Consumer- Semaphores	Kernel-Space	21
Producer-Consumer- Monitors	User-Space	25
Personal Academic Conduct	-	5

Submission Instructions:

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 Project 1 instructions for details.

To submit your project:

- **Step 1:** Accept the assignment using the link below. Please make sure you use the one for your section.

  Section 1 (*Larson TuTh 11:30am - 12:45pm***):**
  https://classroom.github.com/a/U45c5WOK

  Section 2,3, and 4 (*Tompkins TuTh 2:30pm - 3:45pm, TuTh 10:00am - 11:15am, MoWe 11:30am - 12:45pm* **):**
  https://classroom.github.com/a/zJXJFRjF

  Section 99 and 100 (*Larson TuTh 4:00pm - 5:15pm***):**
  https://classroom.github.com/a/zMPXGGPN

  Please use the following format to name your repository :
  project3-YOURUMBCUSERNAME
  For example, If your umbc username is **AAAA**, then repository's name will be: **project3-AAAA**

- **Step 2:**
  Add your project files in the relevant directories in your project 3 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.

- **Step 3:**
  You need to upload a pdf placeholder document in blackboard, as you did for project 1.

Directory Structure and Files

These are files that will contain the code you have created for this project. You will push your kernel to github. Make sure that the correct version 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. Look through the directory structure below to understand how your files should be organized.

```
.
└── project3-username/
    ├── buffer/ (part 1)
    │   ├── buffer.h (provided)
    │   ├── buffer.c (kernel-space file)
    │   ├── test.c (to test kernel-space)
    │   ├── buffer_user.c (user-space)
    │   ├── test_user.c (to test user-space)
    │   └── README.MD
    ├── prodcon_sem/ (part 2)
    │   ├── buffer_sem.h (provided)
    │   ├── buffer_sem.c (kernel-space file)
    │   ├── test_sem.c (to test kernel-space)
    │   ├── buffer_user_sem.c (user-space)
    │   ├── test_user_sem.c (to test user-space)
    │   └── README.MD
    ├── prodcon_mon/ (part 3)w
    │   ├── buffer_mon.h (provided)
    │   ├── buffer_mon.c (user-space file)
    │   ├── test_mon.c (to test user-space)
    │   └── README.MD
    └── other kernel folders
```

(1) buffer.h / buffer_sem.h / buffer_mon.h

(a) This should exactly match the header we already included. **Do not modify this header.** If you need to add additional headers, feel free. (See #13)

(b) Each part must have its own buffer.h / buffer_sem.h / buffer_mon.h inside its directory.

(2) buffer.c

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

(4) buffer_user_sem.c

(a) Your **user space** implementation of the producer-consumer problem using the circular buffer with **Semaphores** (as described part 2)**.**

(5) buffer_sem.c

(a) Your **kernel space** implementation of the producer-consumer problem using the circular buffer with **Semaphores** (as described part 2).

(6) buffer_mon.c

(a) Your implementation of the producer-consumer problem using the circular buffer with **Monitor** which is only required in user space.

(7) test.c

(a) A test file that exercises your **kernel-space** implementation of **circular buffer**. See the previous section for details.

(8) test_user.c

(a) A test file that exercises your **user-space** implementation of **circular buffer**. See the previous section for details.

(9) test_sem.c

(a) A test file that exercises your **kernel-space** implementation of producer-consumer with **semaphores.**

(10) test_user_sem.c

(a) A test file that exercises your **user-space** implementation of producer-consumer with **semaphores**.

(11) test_mon.c

(a) A test file that exercises your **user-space** implementation of producer-consumer with **monitors**.

(12) 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.

(b) Each part should have its own README in the part directory, with clear instructions on how to compile and run user and kernel space implementation.

(13) 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.

(b) Make sure to include description of all of the additional test files and header files in the respective README.

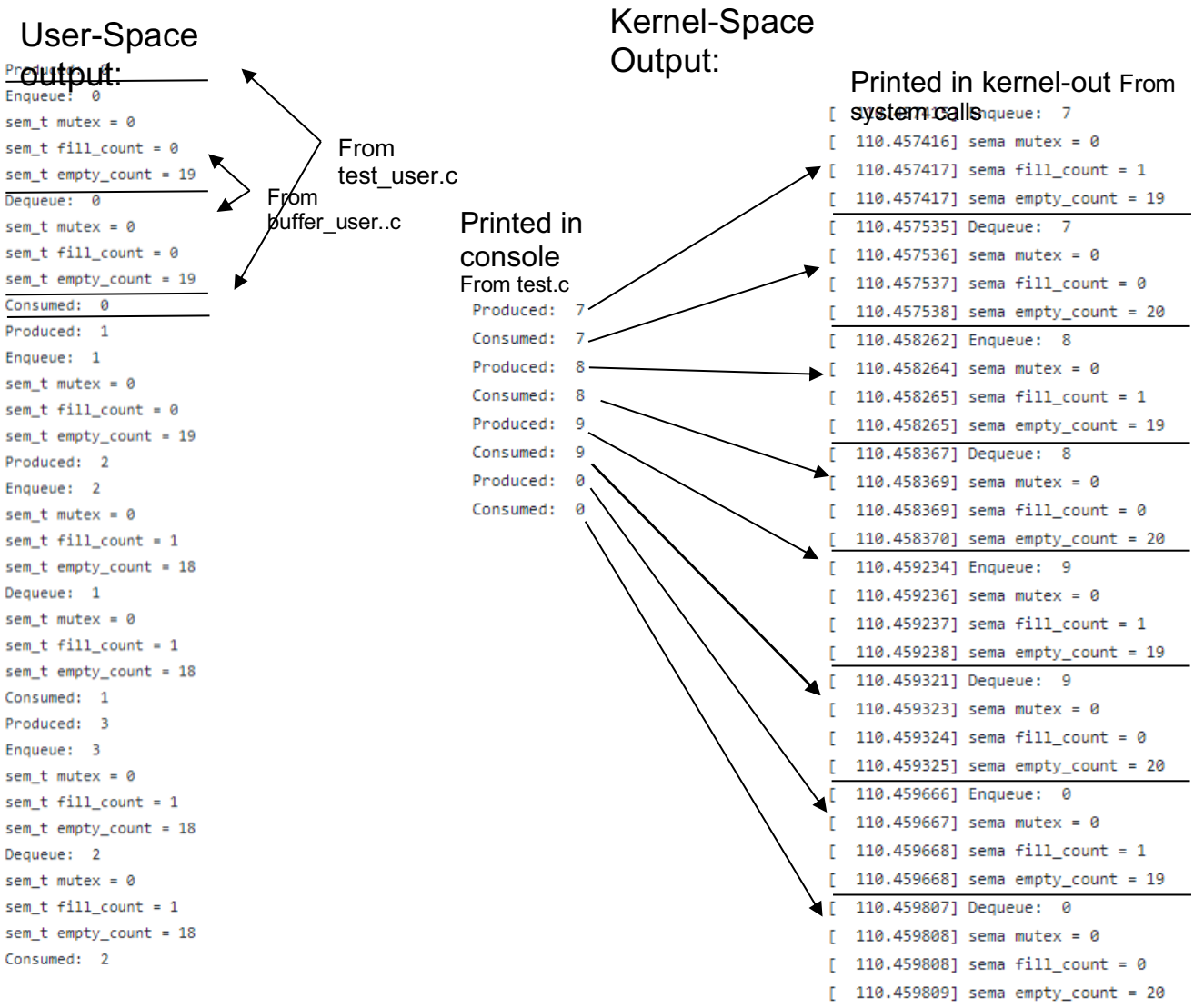
Extra credit: the project is worth 100 points. For each day that you turn in this project early, you can earn 1 point extra credit, up to 5 points (5 days early). For example, if a project is due on Apr. 4 and you are finished on Apr. 2, you will earn 2 points extra credit, the highest possible score would be 102.

Late Policy: The late policy is firm. Projects are either 2, 3, or 4 weeks long. You need to start working on the projects early in the project time period. Students who wait until the last few days usually do not do well. Running into problems during the last several days before a project is due will not be a valid excuse for waiving the late policy.

The discord channel is the place to post questions on the projects and where GTAs will hold office hours. Students should read the project descriptions several times thoroughly before posting questions. If a student posts a question where the answer is in the project description, the GTAs have been directed to tell the student to find the answer in the project description.

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.
5. Some suggestions for testing your code:
    a. Verify the order of production and consumption. If you enqueue, E.g., items 1, 2, 3, and 4, verify that we dequeue items 1, 2, 3, and 4 in that order, irrespective of sleep time.
    b. Verify that you initialize your semaphores correctly, and ensure you are decrementing (wait) and incrementing (post) them correctly.
    c. Call delete after joining your producer and consumer threads. Verify that the buffer deletes correctly.
    d. It should not matter which thread you start first. If you begin the consumer thread first, dequeue should block it until data enters the buffer.
6. Use print_semaphores to observe your semaphores in user space. If desired, include a system call for printing semaphores in kernel space. See the following illustration for an example (Note: You are not required to mirror this output.):



Sample Output

In this section, we provide some examples of output from a test program.

When printing from your producer and consumer threads, you must lock around the print statements, especially if retrieving your buffer's length.

NOTE: The order of acquiring locks is *non-deterministic*. Depending on how you print, you may find that operations print out of order. This is an expected outcome -- as long as there are corresponding enqueue and dequeue operations with the correct post-condition (size) of the buffer, things are likely working.

Scenario 1: An empty buffer.

```
:: Enqueueing element into buffer. ::
5555555555...
1 items in buffer.
:: Enqueueing element into buffer. ::
6666666666...
2 items in buffer.
:: Dequeueing element from buffer. ::
5555555555...
1 items in buffer.
:: Dequeueing element from buffer. ::
6666666666...
0 items in buffer.
:: Enqueueing element into buffer. ::
7777777777...
1 items in buffer.
:: Dequeueing element from buffer. ::
7777777777...
0 items in buffer.
```

Scenario 2: A full buffer.

```
:: Enqueueing element into buffer. ::
3333333333...
20 items in buffer.
:: Dequeueing element from buffer. ::
4444444444...
19 items in buffer.
:: Enqueueing element into buffer. ::
4444444444...
20 items in buffer.
:: Dequeueing element from buffer. ::
5555555555...
19 items in buffer.
:: Enqueueing element into buffer. ::
5555555555...
20 items in buffer.
:: Dequeueing element from buffer. ::
6666666666...
19 items in buffer.
:: Enqueueing element into buffer. ::
6666666666...
20 items in buffer.
```