HW 3: **Spooler**

CS 162

Due: 9:00 PM October 16, 2019

Contents

1 Introduction

This homework is intended to give you the opportunity to explore more advanced synchronization in data structures. In this assignment, you will be building a circular buffer, conceptualized in this homework as a "spooler". A spooler is a queue with multiple producers and multiple consumers, which supports the functionality of printing the items in its queue in-order. Our spooler in this assignment operates on string types.

Note: The bounded buffer example was covered in Lecture 7, "Synchronization Operations." We recommend you review this lecture before starting this homework or read 5.3.2 "Case Study: Thread-safe Bounded Queue" from the textbook.

In this assignment, you will first develop an API for a generic queue utilizing a circular buffer with basic functionalities of insert and remove. In the next part, you will add locking to make your queue thread-safe. Finally, you will use condition variables as a more efficient means of handling an empty or full queue and implement termination detection.

Please be aware that you will be building off of your code for previous parts as you progress in the homework, so you should make a special effort to write clean code that is easy to understand and modify. Of course, we'd encourage you to always do this, but know that it's particularly painful to debug the later parts of this homework if the code from earlier parts is buggy as well as difficult to follow.

This assignment is due at 9:00 pm on Wednesday, October 16th, 2019.

1.1 Getting Started

Log in to your Vagrant Virtual Machine and run:

```
$ cd ~/code/personal/
$ git pull staff master
$ cd hw3
```

Run `make` to build the code.

Six binaries should be created: `unit-tests`, `spooler1`, `spooler2`, `spooler2_killed`, `spooler3`, and `spooler4`.

1.2 Overview of Source Files

Below is an overview of the starter code:

`spooler.h`

This file contains the struct definition for the queue. Please read through this file carefully to understand the changes in struct members as you iterate on your spooler implementation throughout this homework.

`spooler.c`

This file implements a wrapper for the spooler API. This allows us to produce separate executables for each part.

`consumer.c | consumer.h | producer.h | producer.c | main.c`

These files define an example usage of a program that utilizes our spooler. In the producer file, each producer spawns a new thread that writes to the queue. The consumer then reads off the queue and prints it.

`main_killed.c`

This file presents the same main file as main except the program exits after all producers are finished. This is not safe and will be used to profile your part 2.

`spooler_shared.c | spooler_shared.h`

These files implement an API that will be shared by thread safe and unsafe versions.

`spooler_unsafe.c | spooler_unsafe.h`

These files implement an API for a version of the queue that is not thread safe.

`spooler_safe.c | spooler_safe.h`

This files implement the API necessary to add mutual exclusion to the queue, making it thread safe.

`spooler_one_cv.c | spooler_one_cv.h`

These files implement the API necessary to allow for efficiently wait for either space in the queue to add items or items in the queue to remove and perform termination detection using a single condition variable.

`spooler_two_cv.c | spooler_two_cv.c`

These files implement the API necessary to add a second condition variable.

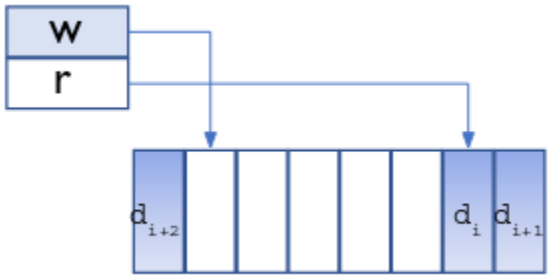

Figure 1: An example of a circular buffer with 3 elements and room for 5 more. The next element to be read is at index 6 and the next write location should be index 1.

2 Building a Queue Backed by a Circular Buffer

For this first part of the homework, we'll implement a queue backed by a circular buffer.

A circular buffer allocates a fixed amount of memory to store data and keeps track of its `read` and `write` index. The `read` index gives the next element in the spooler that may be read and removed. The `write` pointer gives the next available space in the spooler where data may be written to. The buffer is circular because if space is still available after we write to the last allocated address, we may have to write back to the front of the buffer.

An example of a circular buffer is shown in Figure 1. This struct is implemented for you in `spooler.h` - be sure to understand its definition specific to `PART 1`.

You will need to implement the following API in `spooler_shared.c`:

- `void spool_init_shared(spool_t *sp, size_t size)`
- `bool spool_empty(spool_t *sp)`
- `bool spool_full(spool_t *sp)`
- `void spool_insert_unchecked(spool_t *sp, char *item)`
- `char *spool_remove_unchecked(spool_t *sp)`
- `char **reveal_buffer(spool_t *sp)`

As well as the following API in `spooler_unsafe.c`:

- `bool spool_insert_unsafe(spool_t *sp, char *item)`
- `char *spool_remove_unsafe(spool_t *sp)`

For this section the queue should **not** be thread safe; we will add that in the coming sections. Instead, assume that there is only one thread.

We have provided you a series of tests with the `unit_tests` executable. To run these tests, you should run:

```
$ make
$ ./unit_tests
```

If you see `All tests passed` print, then you have passed all tests.

3 Threading and Mutual Exclusion

Next, we will add support for multiple threads. Run the executable produced by our thread-unsafe queue with the following commands:

```
$ make
$ ./spooler1 2 1 3
```

This should create three threads: two producers which each add three strings, and one consumer that removes and prints these strings.

If this were thread safe, this program would print `C0: Prod # String i`, for `i` values of 0 through 2, and for producer `#`s 0 and 1. However, with our current thread-unsafe implementation, even if we run this many times we will rarely - or perhaps never - see six such strings. Therefore, we need to add mutual exclusion through the use of a lock.

We have modified the `spool_t` struct to include a new member, `pthread_mutex_t buf_lock`, which will be used starting in this part of the homework, under the `PART 2` section.

You will need to add the following functions to `spooler_safe.c`:

- `void spool_init_safe(spool_t *sp)`
- `bool spool_insert_safe(spool_t *sp, char *item)`
- `char *spool_remove_safe (spool_t *sp)`

When you have completed the implementation of these functions, run the following commands:

```
$ make
$ ./spooler2 2 1 3
```

You should notice that all six strings print... but now your terminal is stuck! In fact, if you increase the number of strings, producers, or consumers, the correct number of strings should always print, but the terminal should still hang. In the next section, we will explore how to fix this issue.

3.1 Assessing Locking Efficiency

Answer the following questions in "hw3.txt":

1. Open `consumer.c` and look at the implementation for the consumer. Why are the threads for consumers not exiting? Explain this in terms of the code given. Why might the code be written like this?

2. To help illustrate the termination problem, we have provided you another executable: `spooler2_killed`. This executable kills the program after all producers have finished. Will this method still meet our correctness constraint?

   Try running the following command multiple times:

   ```
   $ make
   $ ./spooler2_killed 100 100 10000 > killed_output.txt
   $ python3 consumer_counter.py killed_output.txt
   ```

   This runs the executable with 100 producers each producing 10,000 strings (or 1,000,000 total strings). Does the python script output indicate that all 1,000,000 strings are printing every time? Why or why not is this happening?

3. We want to determine how efficient our waiting on the empty and full condition is. To do so, we will profile your `spooler2_killed` executable using `gprof` (which should already be installed on your VM). `gprof` works by sampling the PC of the CPU and using that to determine what function in your executable is running. To setup `gprof`, follow these steps:

   (a) Edit the `CFLAGS` in the `Makefile` to include `-pg`. This is the compilation flag that tells `gprof` what functions are running. We did not include this by default to avoid having profiling run in every execution.

   (b) Rebuild your executables using `make clean && make`

   (c) Run your executable. This creates `gmon.out` which holds the sampling information.

   (d) Run `gprof EXECUTABLE_NAME`. This outputs a call stack of where the program was running during each sample.

   For `spooler2_killed` run the following command:

   ```
   $ ./spooler2_killed 1000 1000 10000 > /dev/null
   $ gprof spooler2_killed
   ```

   How many calls to `spool_full` and `spool_empty` were there? How many seconds did it take? What percentage of the total runtime was this?

   Note: To avoid having `gprof` sample in all future runs of executables, you may want to remove the `-pg` flag from the `Makefile` at this point.

4 Adding Condition Variables and Termination Detection

Now we want to make our implementation more efficient by removing the busy-waiting in part two and actually allow our threads to terminate. To do this, under the `PART 3` section in `spooler.h` we have added a single condition variable to our `spool_t` struct and additional fields to indicate whether all producers have finished.

Modify `spooler_one_cv.c` to add support for the following API:

- `void spool_init_cv1(spool_t *sp)`
- `void spool_shutdown(spool_t *sp)`
- `bool spool_insert_cv1(spool_t *sp, char *item)`
- `char *spool_remove_cv1(spool_t *sp)`

To verify that your solution is working properly, you can run the below command to see whether 1,000,000 strings are printed every time.

```
$ make
$ ./spooler3 100 100 10000 > output.txt
$ python3 consumer_counter.py output.txt
```

4.1 Assessing Condition Variable Efficiency

Answer the following questions in "hw3.txt"

4. Run `gprof` on `spooler3`. To do so refer to the `gprof` setup instructions in the last section and then run:

   ```
   $ ./spooler3 1000 1000 10000 > /dev/null
   $ gprof spooler3
   ```

   How many calls to `spool_full` and `spool_empty` were there? How many seconds did it take? What percentage of the total runtime was this?

5 Adding a Second Condition Variable

Now we want to compare our performance from the previous section with that of an implementation using **two** condition variables. We have added a second condition variable to our `spool_t` struct for just producers under the `PART 4` section in `spooler.h`

Implement the following API in `spooler_two_cv.c` (which should be the same as the previous section, except that producers now utilize a separate condition variable). You should not need to make any modifications for termination detection to work properly.

- `void spool_init_cv2(spool_t *sp)`
- `bool spool_insert_cv2(spool_t *sp, char *item)`
- `char *spool_release_cv2(spool_t *sp)`

To verify that your solution is working properly, you can run the below command to see whether 1,000,000 strings are printed every time.

```
$ make
$ ./spooler4 100 100 10000 > output.txt
$ python3 consumer_counter.py output.txt
```

5.1 Assessing 2 Condition Variables Efficiency

Answer the following questions in "hw3.txt":

5. Run `gprof` on `spooler4`. To do so, refer to the `gprof` setup instructions in section 4 and then run:

   ```
   $ ./spooler4 1000 1000 10000 > /dev/null
   $ gprof spooler4
   ```

   How many calls to `spool_full` and `spool_empty` were there? How many seconds did it take? What percentage of the total runtime was this?

6. Compare your results in questions 3-5. What do you think contributed to the output you saw? Did it match what you expected?

7. `gprof` works by sampling the running program and determining what function it is in. How might this impact your results? How might we want to restructure our testing to account for this?

8. Compare the time it takes to run the following two commands:

   ```
   time ./spooler3 1 500 100000 > /dev/null
   time ./spooler4 1 500 100000 > /dev/null
   ```

   What do you think influenced the results you saw? In lecture you learned that the default scheduling algorithm on your VM is the Linux CFS. How might this contribute to the results you saw?