CMPSC473, Fall 2019
Concurrency Lab: Implement Your Device Driver
Assigned: October 16, 2019, Due: November 19, 2019
Expected Programming Time: 30 hours +

Introduction

A hardware driver is a computer program that controls a piece of hardware. A driver typically provides 2 main APIs for clients: schedule (queues a job) and handle (do a job). Notice that clients could be human being and other programs. Schedule and handle operations could be synchronous or asynchronous.

In synchronous mode, handler always blocks until there is job to handle, whereas in asynchronous mode, they simply return. Similarly, with schedule, in synchronous mode, if the job queue is full, schedulers wait until a handler has processed a job and there is available space in the job queue whereas in asynchronous mode, they simply leave without scheduling.

Another variation would be if a driver is queued (that is, if the driver has a job queue helping it to manage jobs). In the queued case, the scheduler blocks **only** until the job has been scheduled to the queue. On the other hand, if the driver is unqueued, the scheduler blocks until a handler has handled the job.

In this lab you will be writing your own version of a driver which will be used to communicate among multiple clients, both synchronously and asynchronously. A client can either schedule a job or handle a job from it. Keep in mind that multiple clients can schedule and handle simultaneously from the driver. You are encouraged to explore the design space creatively and implement a driver that is correct, efficient, and fast.

The only files you will be modifying and handing in are *driver.c* and *driver.h* and optionally *linked_list.c* and *linked_list.h*. Consider the job queue to be of finite size, configurable during creation. Buffered drivers will have a positive size, whereas unqueued drivers will have a 0 size of job queue. You will be implementing the following functions:

- driver_t* driver_create(size_t size)
- enum driver_status driver_schedule(driver _t* driver, void* job)
- enum driver_status driver_handle (driver_t* driver, void** job)
- enum driver_status driver_non_blocking_schedule(driver_t* driver, void* job)
- enum driver_status driver_non_blocking_handle(driver_t* driver, void** job)
- enum driver_status driver_close(driver_t* driver)
- enum driver_status driver_destroy(driver_t* driver)
- enum driver_status driver_select(select_t* driver_list, size_t driver_count, size_t* selected_index)

You are encouraged to define other helper functions, structures, etc. to help model the code in a better way.

Description of the driver related functions

- driver_create: Creates a new driver with the provided job queue size and returns it to the caller function. A 0 size indicates an unqueued driver, whereas a positive size indicates a queued driver.

- driver_schedule: Checks if the given driver has space to accommodate the new job and schedule it. This is a **blocking** call, i.e., the function only returns on a successful completion of schedule. In case the queue is full, the function waits till the queue is available to take in new job. The return type is enum driver_status as defined in driver.h. Return

    - SUCCESS for successful queuing of job,
    - DRIVER_CLOSED_ERROR when the queue is closed, and
    - DRIVER_GEN_ERROR on encountering any other generic error of any sort.

- driver_handle: Picks up data from the given driver and stores it in the function's input parameter, job (Note that it is a double pointer). This is a **blocking** call, i.e., the function only returns on a successful job completion. In case the queue is empty, the function waits until the queue has some jobs to pick up. The return type is enum driver_status as defined in driver.h. Return

    - SUCCESS for successful retrieval of job,
    - DRIVER_CLOSED_ERROR when the driver is closed, and
    - DRIVER_GEN_ERROR on encountering any other generic error of any sort.

- driver_non_blocking_schedule: Checks if the given driver has space to accommodate the new job and populates it. This is a **non-blocking** call, i.e., the function simply returns if the queue is full. The return type is enum driver_status as defined in driver.h. Return

    - SUCCESS for successful queuing of job,
    - DRIVER_FULL if the queue is full and the data was not added to the queue,
    - DRIVER_CLOSED_ERROR when the driver is closed, and
    - DRIVER_GEN_ERROR on encountering any other generic error of any sort.

- driver_non_blocking_handle: Picks up data from the given driver and stores it in the function's input parameter job (Note that it is a double pointer). This is a **non-blocking** call, i.e., the function simply returns if the driver is empty. The return type is enum driver_status as defined in driver.h. Return

    - SUCCESS for successful retrieval of job,
    - DRIVER_EMPTY if the driver is empty and nothing was stored in job
    - DRIVER_CLOSED_ERROR when the driver is closed, and
    - DRIVER_GEN_ERROR on encountering any other generic error of any sort.

- driver_close: Closes the driver and informs all the blocking schedule/handle/select calls to return with DRIVER_CLOSED_ERROR. Once the driver is closed, schedule/handle/select operations will cease to function and return

- SUCCESS if close is successful,
- DRIVER_GEN_ERROR in any other error case.

- driver_destroy: Free all the memory allocated to the driver. The caller is responsible for calling driver_close and waiting for all threads to finish their tasks before calling driver_destroy. Return
  - SUCCESS if destroy is successful,
  - DRIVER_DESTROY_ERROR if driver_destroy is called on an open driver, and
  - DRIVER_GEN_ERROR in any other error case.

- driver_select: Takes an array of drivers, driver_list, of type select_t and the array length, driver_count, as inputs. This API iterates over the provided list and finds the set of possible drivers which can be used to invoke the required operation (schedule or handle) specified in select_t. If multiple options are available, it selects the first option and performs its corresponding action. If no driver is available, the call is blocked and waits until it finds a driver which supports its required operation. Once an operation has been successfully performed, select should
  - set selected_index to the index of the driver that performed the operation and then return SUCCESS.
  - In the event that a driver is closed or encounters an error such as DRIVER_GEN_ERROR, you should propagate the error and return the error through select. Additionally, set selected_index to the index of the driver that generated the error.

  Notice: for select function, you can safely assume that all drivers in the driver_list are buffered.

  select_t: This struct has following parameters:
  - driver_t* driver: Driver on which we want to perform operation
  - enum operation op: Specifies whether we want to handle (HANDLE) or schedule (SCHDLE) on the driver.
  - void* data: If op is HANDLE, then the job handled from the driver is stored as an output in this parameter, job. If op is SCHDLE, then the job that needs to be scheduled is given as input in this parameter, job.

Support Routines

The queue.c file contains the helper constructs for you to create and manage a queued driver. These functions will help you separate the queue management from the concurrency issues in your driver code. Please note that these functions are **NOT** thread-safe. You are welcome to use any of these functions, but you should not change them.

- queue_t* queue_create(size_t capacity): Creates a queue with the given capacity.

- enum queue_status queue_add(queue_t* queue, void* job): Adds the job into the queue. This function returns QUEUE_SUCCESS if the queue is not full. Otherwise, it returns QUEUE_ERROR.

- enum queue_status queue_remove(queue_t* queue, void** job): Removes the job from the queue in FIFO order and stores it in job. This function returns QUEUE_SUCCESS if the queue is non-empty. Otherwise, it returns QUEUE_ERROR.

- void queue_free(queue_t* queue): Frees the memory allocated to the queue.

- size_t queue_capacity(queue_t* queue): Returns the total capacity of the queue.

- size_t queue_current_size(queue_t* queue): Returns the current number of jobs in the queue.

We have also provided the **optional** interface for a linked list in linked_list.c and linked_list.h. You are welcome to implement and use this interface in your code, but you are **not required** to implement it if you don't want to use it. It is primarily provided to help you structure your code in a clean fashion if you want to use linked lists in your code. You can add/change/remove any of the functions in linked_list.c and linked_list.h as you see fit.

Programming rules

You are not allowed to take any of the following approaches to complete the assignment:

➢ Spinning with or without a polling loop to implement blocking calls (it will fail the test anyway 😊)

➢ Sleep for an arbitrary amount of time

➢ Try to change the timing in 'test.c' to hide bugs such as race conditions. (We will run your driver with our test.c file, so no need to change code in test cases 😊)

You are only allowed to use the pthread library, the semaphore library, basic standard C library functions (e.g., malloc/free), and the provided code in the assignment for completing your implementation. If you think you need some other library function, please contact the course staff to determine the eligibility.

Here are a bunch of functions that maybe helpful:

- pthread_mutex_init
- pthread_mutex_destroy
- pthread_mutex_lock
- pthread_mutex_unlock
- pthread_cond_wait
- pthread_cond_signal
- pthread_cond_broadcast

- sem_init
- sem_destroy
- sem_wait
- sem_trywait (be cautious if you need to use this function)
- sem_post

Look for manual page if you don't know how to use them.

Testing your code

To run the supplied test cases (including the ones listed below) simply run the following command in the project folder:

```
make test
```

Your code will be tested in the following areas:

- On running the make command in your project, two executable files will be created. The default executable, driver, is used to run specific test cases on your code. Check for the name of the test case you want to run in the file test.c and run the following command, replacing <test_case_name> with the name of the test:

  ```
  ./driver <test_case name>
  ```

- The other executable, driver_sanitize, will be used to help detect data races in your code. It can be used with any of the test cases in test.c. To run a specific test case, you can run the following command, replacing <test_case_name> with the name of the test:

  ```
  ./driver_sanitize <test_case_name>
  ```

  Any detected data races will be output to the terminal. You should produce code that does not generate any errors or warnings from the data race detector.

- Valgrind is being used to check for memory leaks, report uses of uninitialised values, and detect memory errors such as freeing a memory space more than once. To run a valgrind check by yourself, use the command:

  ```
  valgrind --leak-check=full ./driver
  ```

  Note that driver_sanitize should ***not*** be run with valgrind. Only driver should be used with valgrind. Valgrind will issue messages about memory errors and leaks that it detects for you to rectify them. You should produce code that does not generate any valgrind errors or warnings.

Hints

To compile your code in debug mode (to make it easier to debug with gdb), you can simply run:

```
make debug
```

It is important to realize that when trying to find race conditions, the reproducibility of the race condition often depends on the timing of events. As a result, sometimes, your race condition may only show up in non-debug (i.e., release) mode and may disappear when you run it in debug mode. Bugs may sometimes also disappear when running with gdb or if you add print statements. **Bugs that only show up some of the time are still bugs, and you should fix these. Do not try to change the timing to hide the bugs.**

A reasonable approach to debugging these race condition bugs is to try to identify the symptoms of the bug and then read your code to see if you can figure out the sequence of events that caused the bug based on the symptoms. If your bug only shows up outside of gdb, one useful approach is to look at the core dump (if it crashes). Here's a link to how to get and use core dump files: http://yusufonlinux.blogspot.com/2010/11/debugging-core-using-gdb.html If your bug only shows up outside of gdb and causes a deadlock (i.e., hangs forever), one useful approach is to attach gdb to the program after the fact. To do this, first run your program. Then in another command prompt terminal run:

```
ps aux
```

This will give you a listing of your running programs. Find your program and look at the PID column. Then run:

```
gdb
```

Within gdb, then run:

```
attach <PID>
```

where you replace <PID> with the PID number that you got from ps aux. This will give you a gdb debugging session just like if you had started the program with gdb.

Evaluation

You will receive zero points if:

- ➢ you break any of the rules
- ➢ your code does not compile/build
- ➢ you do not follow the hand in instruction (see **Handin** section)

Your code will be evaluated for correctness, properly handling synchronization, and ensuring it does not violate any of the programming rules (e.g., do not spin or sleep for any period of time). We have provided our auto grader program. To use it, simply run

```
make test
```

or

`python grade.py` after compilation

In terms of a grade breakdown, we will assign:

- 50% for basic functionality of buffered channels (e.g., blocking and non-blocking send/receive, create and destroy)
- 15% for the closing of channels

- 25% for select
- 10% for a proper submission (builds and tests automatically)

Handin

To handin your code, first change the environment variable 'SNUM' in your Makefile to your student ID, and then run the following command:

```
make handin
```

You will see a *handin-YOUR_STUDENT_ID.tar.gz* file created. Submit this tgz file on Canvas.

Bonus

We also provide bonus points in this project. The task is to improve your driver_select() function such that it also works with unbuffered drivers.

To test your bonus code, run

```
make bonus
```

Or

`python grade.py bonus` after compilation.

Warning: the bonus is considered to be very hard to implement. It is fairly possible to restructure all your schedule and handle functions just to make the unbuffered driver_select function work. Also, you will need to come to professor's or TA's office hour to defend your code.