Update: 11/10, added libtsan0 to the list of packages

Main Project Code

Please get the code and create your group, similar to Projects 1-2 by following this link:
https://classroom.github.com/a/hu3o-99D
Note that groups for Project 3 are different, you must specify tgroup rather than group/sgroup when creating or joining github groups to properly distinguish with Project 1-2 groups. This is very important to avoid any confusion with Project 1-2 groups! Any repositories that do not conform to this naming convention will be deleted without any prior warning.

Extra Credit (Only)

Please use the following link to create **a separate repository for extra credit only**. **Do not create this repository unless you are ready to work on the extra credit part (i.e., to be done after the main part is complete).** Note that it is NOT for the main part of the project even if you are working alone, you still must create and use the link for one-person groups above! Only extra credit part will be evaluated and graded for the link below. **Each partner must work individually for the extra credit part! Only additional points (i.e., on top of standard 200 points will be considered for correct implementations).**
https://classroom.github.com/a/tQm4A9NI

ATTN: Lab Machines and Personal VMs

Similarly to project 1, **we will grade project 3 on lab machines**. That **means you must make sure that your code compiles and runs on the lab machines.** (Lab Machine 01 to Lab Machine 28, i.e., <your_user_name>@e5-cse-135-01.cse.psu.edu to <your_user_name>@e5-cse-135-28.cse.psu.edu – see Programming Assignment Setup for more details. If any of these lab machines is slow, just try another one.)

For the **development purposes only**, you can use the same VMs that you used in project 2 with Ubuntu 22.04 for x86-64 (Windows, Linux, older Macs with Intel Silicon) or Ubuntu 22.04 for ARM64 (newer Macs with Apple Silicon). We have tested the provided code on both x84-64 and ARM64. You just need to make sure that you install build-essential, gdb, valgrind as in the following command (run it on your personal VM only, on the lab machines all packages are already installed) – the same packages are required for both x86-64 and ARM64: **sudo apt install build-essential gdb valgrind libtsan0**

As with project 2, you will use VMware. Other hypervisors such as VirtualBox and/or other Linux distributions may work but we have not tested them and will not provide any support if any issue arises. Note that for this project, we will enforce the **ZERO tolerance policy** for any "small" mistakes, e.g., when the code does not compile or runs on lab machines. **You must confirm that everything works on the lab machines for a specific git SHA that you are submitting before the deadline!!!** Please make periodic checks that your code still compiles and runs on lab machines during the development cycle even if you decide to use your personal VMs.

Introduction

Note that the concurrent programming is hard – start early.

A channel is a model for synchronization via message passing. Messages may be sent over a channel by a thread (*Sender*) and other threads which have a reference to this channel can receive them (*Receivers*). A channel can have multiple senders and receivers referencing it at any point of time.
Channels are used as a primitive to implement various other concurrent programming constructs. For example, channels are heavily used in Google's *Go* programming language and are very useful frameworks for high-level concurrent programming. In this lab you will be writing your own version of a channel which will be used to communicate among multiple clients. A client can either write onto the channel or read from it. Keep in mind that multiple clients can read and write simultaneously from the channel. You are encouraged to explore the design space creatively and implement a channel that is correct and not exceptionally slow or inefficient. Performance is not the main concern in this assignment (functionality is the main concern), but your implementation should avoid inefficient designs that sleep for any fixed time or unnecessarily waste CPU time.

There are multiple variations to channels, such as whether the send/receive is blocking or non-blocking. In blocking mode, receivers always block until there is data to receive, whereas in non-blocking mode, they simply return. Similarly, with senders, in blocking mode, if the buffer is full, senders wait until some receiver has retrieved a value and there is available space in the buffer whereas in non-blocking mode, they simply leave without sending. In this lab, you will support both blocking and non-blocking send/receive functions.

Another variation to channels would be if a channel is buffered (i.e., channel buffer size > 0) or unbuffered (i.e., channel buffer size = 0). In the buffered case, the sender blocks **only** until the value has been copied to the buffer. On the other hand, if the channel is unbuffered, the sender blocks until the receiver has received the value. In this lab, you will only be responsible for supporting buffered channels. Supporting unbuffered channels is extra credit and is especially difficult when implementing select. The amount of extra credit is really small for the amount and difficulty of work involved, and correctly implementing the unbuffered version for select is probably 2-3 times the work of the entire buffered assignment. **The extra credit part will have to be done in a separate repository individually.**
The only files you will be modifying are *channel.c* and *channel.h* and optionally *linked_list.c* and *linked_list.h*. **You should NOT make any changes in any file besides these four files.** You will be implementing the following functions, which are described in channel.c and channel.h:

- chan_t* channel_create(size_t size)
- enum chan_status channel_send(chan_t* channel, void* data, bool blocking)
- enum chan_status channel_receive(chan_t* channel, void** data, bool blocking)
- enum chan_status channel_close(chan_t* channel)
- enum chan_status channel_destroy(chan_t* channel)
- enum chan_status channel_select(size_t channel_count, select_t* channel_list, size_t* selected_index)

The enum chan_status is a named enumeration type that is defined in channel.h. Rather than using an int, which can be any number, enumerations are integers that should match one of the defined values. For example, if you want to return that the function succeeded, you would just return SUCCESS.

You are encouraged to define other (static) helper functions, structures, etc. to help structure the code in a better way.

Support routines

The buffer.c and buffer.h files contain the helper constructs for you to create and manage a buffered channel. These functions will help you separate the buffer management from the concurrency issues in your channel 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.

- buffer_t* buffer_create(size_t capacity)
- Creates a buffer with the given capacity.
- bool buffer_add(void* data, buffer_t* buffer)
  Adds the value into the buffer. Returns 'true' if the buffer is not full and value was added. Returns 'false' otherwise.
- void* buffer_remove(buffer_t* buffer)
  Removes the value from the buffer in FIFO order and returns it. Otherwise, returns BUFFER_EMPTY (a special pointer value) if the buffer is empty and a value was not removed.
- void buffer_free(buffer_t* buffer)
  Frees the memory allocated to the buffer.
- size_t buffer_capacity(buffer_t* buffer)
- Returns the total capacity of the buffer.
- size_t buffer_current_size(buffer_t* buffer)
  Returns the current number of elements in the buffer.

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. **Linked lists may NOT be needed depending on your design, so do not try to force it into your solution.** 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 in a polling loop to implement blocking calls
- Sleeping for any fixed amount of time; instead, use cond_wait or sem_wait
- Trying to change the timing of your code to hide bugs such as race conditions
- Using global variables in your code

You are only allowed to use the pthread library, the POSIX 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 library function, please contact the course staff to determine the eligibility. You can find a reference/tutorial for the pthread library at:
https://computing.llnl.gov/tutorials/pthreads/
You can find a reference/tutorial for the POSIX semaphore library at:
http://www.csc.villanova.edu/~mdamian/threads/posixsem.html
Please also see this reference for sem_trywait and sem_wait:
https://www.man7.org/linux/man-pages/man3/sem_trywait.3p.html
Looking at documentation (e.g., for these libraries, for tools, etc.) is fine, but as stated in the academic integrity policy, **looking online for any hints about implementing channels is disallowed**.

Evaluation and testing your code

You will receive zero points if:

- You violate the academic integrity policy (sanctions can be greater than just a 0 for the project)
- You break any of the programming rules
- Your code does not compile/build
- Your code crashes the grading script
- The GIT history (in your github repo) is not present or simply shows one or multiple large commits; it must contain the *entire* history which shows the actual *incremental* work for **you** as a member of your team.

Please commit your changes relatively frequently so that we can see your work clearly. **Do not forget to push changes to the remote repository in github!** Each member has to commit their changes ***separately*** to reflect ***their*** work, i.e., do not commit the code on behalf of your partner since we will not be able to tell who did that part of the work! You can get as low as ***zero points*** if we do not see any confirmation of your work. Please split the work equally! **Do not attempt to falsify time stamps, this may be viewed as an academic integrity violation!**

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 many tests, **but we reserve the right to add additional tests during the final grading**, so you are responsible for ensuring your code is correct, where a large part of correctness is ensuring you don't have race conditions, deadlocks, or other synchronization bugs. To run the supplied test cases, simply run the following command in the project folder:

```
make test
```

make test will compile your code in release mode and run the grade.py script, which runs a combination of the following tests to autograde your assignment:

- After running the make command in your project, two executable files will be created. The default executable, channel, is used to run test cases in the normal mode. The test cases are located in test.c, and you can find the list of tests at the bottom of the file. If you want to run a single test, run the following:

  ```
  ./channel [test_case_name] [iters]
  ```

  where [test_case_name] is the test name and [iters] is the number of times to run the test. If you do not provide a test name, all tests will be run. The default number of iterations is 1.

- The other executable, channel_sanitize, will be used to help detect data races in your code. It can be used with any of the test cases by replacing ./channel with ./channel_sanitize.

  ```
  ./channel_sanitize [test_case_name] [iters]
  ```

  Any detected data races will be output to the terminal. You should implement 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 uninitialized 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 -v --leak-check=full ./channel [test_case_name] [iters]
  ```

Note that channel_sanitize should ***not*** be run with valgrind as the tools do not behave well together. Only the channel executable should be used with valgrind. Valgrind will issue messages about memory errors and leaks that it detects for you to fix them. You should implement code that does not generate any valgrind errors or warnings.

IMPORTANT: Note that any test FAILURE may result in the sanitizer or valgrind reporting thread leaks or memory leaks. This is expected since test failures will cause the test to prematurely end without cleaning up any threads or memory. Thus, you should first fix the test failure.

Grades will be assigned by compiling and running your code on W135 machines. So, please check and evaluate your code on those machines. It is NOT sufficient to check your code on your personal computer only irrespective of what system you use for the development. (You can develop on your personal computer, but please also check your code from time to time on the lab machines. Testing on different machines may also uncover hidden bugs in your code, so it is a good idea anyway.)

Handin

Similar to the previous projects, we will be using GitHub for managing submissions, and **you must show your partial work by periodically adding, committing, and pushing your code to GitHub.** This helps us see your code if you ask any questions on Piazza (please include your group number) and also helps deter academic integrity violations.

Please see below for the submission format. You will indicate the GIT commit number corresponding to your submission. Your submission can use the late policy, and we will calculate and use the better of the normal score and the late score with the late penalty.

Please note that ***each*** member needs to submit the GIT commit number (must be the same that your partner submits!) and the group name that corresponds to the project (e.g., for the p3-tgroup1 repository, you should specify `tgroup1`). You should also specify **your** github username. That should be your github username, not the name of your partner!

Be sure to update README.md to specify which part was implemented by each partner (similar to Projects 1-2).

The submission format in plain text

<group>:<your_github_username>:GIT commit

For example,

tgroup512:runikola:936c332e7eb7feb5cc751d5966a1f67e8089d331

Extra Credit Instructions

The extra credit part MUST be performed individually in a separate repository (created from the link above) and must be submitted separately under separate Project 3 Extra Credit category. Please relocate your Project 3 code to that separate repository. **No extra credit will be given for non-individual submissions or those that do not conform to any of these submission requirements.**

The submission format FOR EXTRA CREDIT

<your_github_username>:GIT commit

For example,

runikola:936c332e7eb7feb5cc751d5966a1f67e8089d331

Hints

- Carefully read the output from the sanitizer and valgrind tools and think about what they're trying to say. Usually, they're printing call stacks to tell you which locations have race conditions, or which locations allocate some memory that was being accessed in the race condition, or which locations allocate some memory that is being leaked, etc. These tools are tremendously useful, which is why we've set them up for you for this assignment.
- While the tools are very useful, they are not perfect. Some race conditions are rare and don't show up all the time. 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.
- Debugging with gdb is a useful way of getting information about what's going on in your code. 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.**
- 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 a tutorial on 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 within gdb (may require sudo) 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.