

## principles of Operating Systems

### Introduction/Objectives

In this project, you will create a new version of the Linux kernel that adds various system calls to implement a new data structure inside the kernel for use in Interprocess Communication (IPC). This assignment is designed to allow you to create new functionality in the kernel through the use of system calls, learn how to interact with the kernel's handling of memory management, and learn how the kernel handles locking and synchronization.

For the purposes of this project, we will be using the amd64 (64-bit) version of the Debian 11 Linux distribution. This kernel, like the one in Project 0 will be built on the Long-term support version 5.15.67 of the Linux kernel. Also, like Project 0, you are expected to develop this system on the same VirtualBox based virtual machine. You should not need to reinstall the VM from scratch — just use the same one you used for Project 0. You may remove the `/usr/src/project0` directory from the VM to reclaim the disk space it takes up as long as you are satisfied with the grade you received on Project 0.

To start, click the link on Teams to create your repository for Project 2 on GitHub, then run the following commands in your VM to set up your working copy of the kernel for this project (replacing yourusername as appropriate with your GitHub username):

```
cd /usr/src
git clone git@github.com:UMBC-CMSC421-FA2022/linux.git project2
cd project2
git remote remove origin
git remote add origin git@github.com:UMBC-CMSC421-FA2022/project2-yourusername.git
git push -u origin main
```

We have prepared a kernel configuration file for you for this project that you may use if you would like to. This is completely optional — you may continue to follow the same procedure from Project 0 to set up the kernel configuration with `make xconfig`, or you may use the one we provide here. If you would like to use this file, download it from this link, copy the file into your `/usr/src/project2` directory, renaming it to `".config"` (note the fact that this filename starts with a dot) and do not run `make mrproper` or `make xconfig` at all while building. This kernel configuration disables many unnecessary modules that you should not need on our VM setup (as well as disabling the build of the `-dbg` package that takes a lot of time and space to build. Incremental Development

You are required in this project to plan out an incremental development process for yourself — one that works for you. There is no one-size-fits-all approach here. One suggested option is to break the assignment down into steps and implement things as you go.

You should not attempt to complete this entire project in one sitting. Also, we don't want you all waiting until the last minute to even start on the assignment. Students who do either of these

tend to get rather poor grades on the assignment. To this end, we are requiring you to make at least 3 non-trivial commits to your GitHub repository for the assignment. These three commits must be made on different dates and at least one must be done before October 30th 2022. You may make more than three commits during the timeline of the project — three is simply a minimum number required for full credit. In addition, you must have made a reasonable attempt at implementing the basic operations on the data structure described later in this document by October 30th. That is to say that we expect to see some version of code that is capable of initializing the Binary Search Tree data structure, inserting items into it, deleting items from it, and searching for existing items by October 30th. This may be in the form of kernel-space code or in the form of a user-space prototype (as detailed below).

In addition, as it is significantly easier to build and test code in user-space than it is in the kernel, you are encouraged to build a user-space prototype of at least part of your system before attempting to implement it in the kernel. Please note that we are not suggesting you implement the entire project in user-space, but rather just some of the basic functionality. One approach that many previous students have found that works for this is to implement a prototype version of the data structure that you will be using in user-space. This will allow you to ensure that you have the basic algorithms for implementing insertion, deletion, search, etc. working without having to spend a long time waiting on kernel builds. If you implement a user-space prototype of your code, please place it in a directory called `proj2proto` in the root of the Linux kernel source code and be sure to tell us about it in your `README.proj2` file (let us know what you did in the prototype, why you chose to do so, how things changed when you ported it over to the kernel, etc).

A non-trivial commit is defined in this assignment as one that meets ALL of the following requirements:

- Does not contain only documentation (i.e, just committing a README file does not count).

- Does not contain only Makefile modifications or creation.

- Modifies/creates at least 10 lines of code in a combination of existing or newly created `.c` or `.h` files. That is to say, creating a new file with 10 lines of code counts, but creating a new file with a 10-line comment does not.

- Code modifications/creation must be relevant to the project. Creating a bunch of useless files/functions that are unrelated or otherwise superfluous to the assignment does not count. It is ok to reorganize your code after you have started and remove pieces of code, of course, but if you are obviously only adding code to the repository early on that you completely delete later (or that has nothing to do with the assignment), then that commit will not count toward the requirements herein.

- Code implementing a prototype version of the assignment (for instance, a user-space version) does count, but any example code or anything else of that nature that you use in that prototype (for instance, a user-space version of the `<linux/list.h>` header file) does not count.

Be sure to also adequately comment your code wherever applicable. Use your own judgment when commenting: if you feel that you are writing some code block which implements a crucial functionality then make a comment explaining what that code block does. If you feel that a line or block of code is syntactically and/or logically complex, make a comment on what is being done in that code segment. The same goes for all the system calls; comment above each

system call function header briefly explaining its purpose and functionality. The better you are at following commenting, the more likely you are to get partial credit even if your implementation is flawed. However, once again, do not excessively comment in the hopes of getting partial credit through a hit-or-miss fashion. Commenting too much or wherever unnecessary may incur penalties if it is in excess, although the bar for such a penalty is rather high to be met. If in doubt, as long as the comment is relevant, it is almost certainly fine.

Failure to adhere to these requirements will result in a significant deduction in your score for the assignment. This deduction will be applied after the rest of your score is calculated, much like a deduction for turning in the assignment with a late penalty.

### Binary Search Trees

In this assignment you will be implementing a Binary Search Tree (BST) data structure to act as a repository of mailboxes. Each node in your BST will be given an ID at creation, which should be used as the key with which you will create the BST. You must ensure that all node IDs are unique. Nodes within your BST are to be used as a mailbox type data structure to hold a FIFO queue of messages. Each mailbox may have an "unlimited" number of messages within it, and you may have an "unlimited" number of total mailboxes as well ("unlimited" is in quotes here because there is still the constraint of the total memory in your system, but the idea is to not put a limit on the number of messages that can be within a mailbox; it will be variable).

### Implementation and Academic Integrity

The kernel has libraries for implementing a red-black tree; however, we recommend that you create your own simple BST implementation as it is quite complex to use the kernel's red-black tree. You should already be familiar with how a BST data structure works and how to sort and iterate over a BST from CMSC 341, but feel free to refer to online tutorials and knowledge references if needed. **DO NOT ATTEMPT TO COPY/PASTE ANY CODE FROM ONLINE RESOURCES, EVEN IF IT IS FOR SAMPLE TESTING PURPOSES.** We have already encountered a few students in office hours who have either knowingly or unknowingly done this in the previous project; regardless of intent, under no circumstances should any code be directly copied. Copying any code and reformatting or refactoring is still plagiarism and strictly prohibited. We have identified, for the last project, and will identify any such cases of plagiarism if they occur again without the leniency or warnings offered for Project 1.

### Message Queues

As mentioned above, each node in your Binary Search Tree is to act as a mailbox, each containing a FIFO queue of messages. The following image may help you to visualize what we are asking you to do in this assignment:

### Binary Search Tree Message Queue

In this image, boxes containing an 'M' are messages added to a queue. The data structure above the chains of messages is the Binary Search Tree, sorted by the ID of each queue. The ground symbol (three lines shaped like a triangle) represents the end of a message queue. We have shown message queues on a few of the nodes of Binary Search Tree as an example (in the above image), but you will have message queues for each node in the BST (which may or may not be empty at any point in time).

We make no requirements on how the queues themselves are implemented, other than requiring that they be implemented in a First-in, First-out (FIFO) manner. The kernel has a built-in linked list structure (see the `include/linux/list.h` header file) that you may find useful.

## New System Calls

You will be adding a few new system calls to the Linux kernel in order to manage the Binary Search Tree of message queues and the messages within each queue. The mailboxes and their contents all exist only in the kernel's address space (all data must be stored in buffers allocated in the kernel — do not attempt to store any user-space pointers in the data structure you create). You will develop the system calls specified below in order to access the mailboxes and their contents.

Each mailbox should be identified by an unsigned long ID value, which is passed in at creation time (as well as into all functions that access the mailbox later). Each mailbox must be able to store an "unlimited" number of messages, each of which is of "unlimited" length (with "unlimited" having the same meaning as earlier in this document). Messages must be treated as binary data and not as ASCII/text strings — this means that messages may have embedded NULL (0-value) bytes. To this end, ensure that you do not attempt to use any string functions such as `strcpy()` or `strlen()` to perform the functionality we've described here. Lengths will be provided with the messages when sent and you must store these in your message queue to ensure that you access the data properly later.

Each mailbox will store its messages in FIFO order. That is to say that each mailbox may be seen as a queue.

Please keep in mind that these functions may well be called from multiple different processes simultaneously. Any accesses to state information in kernel-space for these functions must use appropriate locking in order to ensure their correct operation!

As this code will be part of the kernel itself, correctness and efficiency should be of primary concern to you in the implementation. Particularly inefficient (memory-wise, algorithmic, or other poor coding choices) solutions to the problem at hand may be penalized in grading. In regard to correctness, you will probably find that the majority of your code for this assignment will be spent in ensuring that arguments and other such information passed in from user-space are valid. If in doubt, assume that the data passed in is invalid. Users tend to do a lot of really silly things, after all. Also, remember that malicious users do exist! Crashing the kernel because a NULL pointer is passed in will result in a significant deduction of points.

Finally, you are to implement this system on your own — no group work is allowed on this assignment. You are welcome to use the kernel's basic linked list functionality in assisting you to create the message queues, if you so desire (but you are not required to do so).

`long mailbox_init(void)`: Initializes the mailbox system, setting up the initial state of the Binary Search Tree. You may initialize the Binary Search Tree by adding the root node (which may have an ID of 0 if it is applicable to your implementation).

`long mailbox_shutdown(void)`: Shuts down the mailbox system, deleting all existing mailboxes and any messages contained therein. Returns 0 on success.

`long mailbox_create(unsigned long id)`: Creates a new mailbox with the given id if it does not already exist (no duplicates are allowed). Returns 0 on success or an appropriate error on failure. If an id of 0 or (264 - 1) is passed, this is considered an invalid ID and an appropriate error shall be returned.

`long mailbox_destroy(unsigned long id)`: Deletes the mailbox identified by id if it exists. If the mailbox has any messages stored in it, these messages should be deleted. Returns 0 on success or an appropriate error code on failure.

`long mailbox_count(unsigned long id)`: Returns the number of messages in the mailbox identified by id if it exists. Returns an appropriate error code on failure.

`long mailbox_send(unsigned long id, const unsigned char __user *msg, long len)`: Sends a new message to the mailbox identified by id if it exists. The message shall be read from the user-space pointer msg and shall be len bytes long. Returns 0 on success or an appropriate error code on failure.

`long mailbox_recv(unsigned long id, unsigned char __user *msg, long len)`: Reads the first message that is in the mailbox identified by id if it exists, storing either the entire length of the message or len bytes to the user-space pointer msg, whichever is less. The entire message is then removed from the mailbox (even if len was less than the total length of the message). Returns the number of bytes copied to the user space pointer on success or an appropriate error code on failure.

`long message_delete(unsigned long id)`: Delete the oldest added message in the mailbox identified by id if it exists. This is a FIFO data structure — hence why we are deleting the oldest message. HINT: Keep in mind all the things you have to do for this one — it's not just a matter of just deleting a node in the queue. What other data within your mailbox data structure (not the BST data structure, just one single mailbox) gets affected by the deletion other than just the message node in the FIFO queue?

`long mailbox_length(unsigned long id)`: Retrieves the length (in bytes) of the oldest message pending in the mailbox identified by id, if it exists. Returns the number of bytes in the oldest pending message in the mailbox on success, or an appropriate error code on failure.

Once again, remember that you will be graded for the efficiency of your code as well. There are efficient ways to implement these functions and inefficient ways. Keep asymptotic runtime efficiency in mind when implementing your system call functions.

Remember from Project 0 that system calls are defined in the kernel by way of using a `SYSCALL_DEFINE`n macro, where n is the number of arguments that the system call takes in. So, for instance, the `mailbox_send` syscall from the list above would be defined like shown below:

```
SYSCALL_DEFINE3(mailbox_send, unsigned long, id, const unsigned char __user *, msg,
long, len) {
    /* Code goes here. */
}
```

Each system call returns an appropriate non-negative integer on success, and a negative integer on failure which is indicative of the error that occurred. See the `<errno.h>` header file for a list of error codes. The following error codes would be sensibly used by your code (this may not be an exhaustive list):

- EINVAL: invalid value passed (that doesn't fit into one of the below cases)
- ENODEV: mailbox system not initialized (or has been shut down and has not subsequently been re-initialized)
- EEXIST: mailbox already exists
- ENOENT: mailbox doesn't exist
- EPERM: permission denied
- EFAULT: bad pointer passed or couldn't read/write pointer
- ESRCH: no messages in mailbox
- ENOMEM: memory allocation failure

As this system is designed as an IPC system, messages must be copied into properly allocated kernel memory when sent and copied back into user-space memory when received. Also, please note that there is no requirement that messages be textual strings (so do not assume that messages will be NUL terminated), or that they have any content (zero-length messages are to be considered valid, however messages with negative lengths are not valid and shall generate an error if one is attempted to be sent).

#### Locking Resources for Shared Access

Your BST and its contents must be a shared resource, meaning that two or more processes can access and manipulate this data structure. Coordinating access when different processes are trying to compete for access to the same resource is crucial to avoid race conditions. For example, suppose that two processes are trying to delete the same mailbox: if there is no access orchestration in place, what will end up happening? One process accesses the mailbox and deletes it... but what about the other process' delete request? It will try to delete something that's not present, which will result in a nightmare that you are very familiar with: segmentation faults. However, within the Kernel, you probably won't even get an error message of any sort and will instead face a kernel panic. Building a kernel image takes long enough as is, and the last thing you want to encounter is the nightmare of a kernel panic.

To this end, it is imperative to develop some sort of locking mechanism to regulate access to certain functional code segments. You must employ locking mechanisms to protect and coordinate access to your resource. The kernel provides various locking mechanisms for your use. You should choose the most appropriate strategy using these mechanisms for full credit on this assignment. You must justify how and why you chose to use the mechanism that you choose. Keep in mind that different locking mechanisms exist for different purposes and may not be ideal in certain use cases in comparison to other mechanisms. You will be graded based on your design choices for this project as well, so be sure to do your research and review the learning content on locking mechanisms to give yourself the best chance of making the best design choices. Make sure to properly understand the functionality and the relevant use cases of all the options that you consider and pick the one you think is optimal for the purposes of this project.

#### User-space Driver Program(s)

You must adequately test your kernel changes to ensure that they work under all sorts of situations (including in error cases). You should build one or more testing drivers and include them in your sources submitted. Create a new directory in the Linux source tree called

proj2tests to include your test case program(s). Be sure to include a Makefile to build them and instructions on how to run them in a README file within this directory. Your README for the test programs should also describe your general strategy for testing the system calls. Remember that testing is one of the primary jobs of a developer in the real world!

It is strongly suggested that you additionally build a separate program for each system call to be implemented to simply call that system call with user-provided arguments. For the data to be sent as a message, you might consider allowing the user to specify a file of data to send or a string on the command line. These programs will likely prove to be invaluable in debugging.

#### Submission Instructions

You should follow the same basic set of instructions for submitting Project 2 that you did for Project 0. That is to say, you should do a git status to ensure that any files you modified are detected as such, then do a git add and a git commit to add each modified/newly created file or directory to the local git repository. Then do a git push origin main to push the changes up to your GitHub account.

Be sure to include not only your modified kernel files, but also your driver program files. The driver should go in a proj2tests directory, in the root of the Linux source tree. You must include a Makefile that can build your test program(s) in this directory as well. You should not attempt to add your test directory to the main Linux Makefile. Also, include a README file in the proj2tests directory describing your approach to testing this project. Tell us what your testcases actually test, and why you chose to test those things. If your testcases are supposed to fail at any point, make sure to tell us that in the README (after all, you should not only test your code with good inputs, but with bad ones too — we'll do just that in our testcases).

You must also include a README.proj2 file in the root directory of the Linux source code that describes anything you might want us to know when we're grading your assignment. This can include an outline of how you implemented the requirements of the project, for instance. This is also where you should cite any references you have used for the assignment other than those given in this assignment description.

You should also verify that your changes are reflected in the GitHub repository by viewing your repository in your web browser.

#### References

Below is a list of references that you may find useful in your quest to complete this project:

The Linux Kernel API — documentation of the internal API for programming in the Linux kernel (please note that in the User Space Memory Access chapter, only certain functions are covered (which do "less checking"), the versions that do "more checking" are named the same, but without leading underscore characters)

The Linux Cross-Reference (for version 5.15.67 of the kernel) — a cross-referenced copy of the Linux kernel source code for relatively easy searching

The Open Group Base Specifications Issue 7/IEEE Std. 1003.1 - 2017, 2018 Edition/POSIX.1-2017

The Unreliable Guide to Locking [in the Linux Kernel] — a decent resource, despite its name

(just ignore any sections on the speed of operations)

If in doubt, the Kernel API and Linux Cross Reference should be your ultimate guides.  
What to do if you want to lose points on this project

- Not pushing changes to your GitHub repository by the project due date.

- Excessive unnecessary changes made to the kernel sources.

- Extraneous files are included.

- Files are missing that needed to be modified.

- Project 0 system calls included, or the system calls required are otherwise out of the order specified.

- Failure to follow the requirements in the "Incremental Development" section of the assignment.