Principles of Operating Systems

CMSC 421 - Spring 2018

Project 1

Due by 11:59PM EDT on March 26, 2018

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 Inter-Process Communication. This assignment is designed to allow you to create new functionality in the kernel through the use of system calls, and to learn how to interact with the kernel's handling of memory management.

Before you begin, be sure to create your new GitHub repository for Project 1 by using the link posted on the course Piazza page. Then, follow the same steps you did in Project 0 to clone this new repository (obviously, substitute `project1` for `project0` from the earlier instructions). Also, you may remove the `/usr/src/vanilla-project0` directory and create a new `/usr/src/vanilla-project1` directory with the newly checked out code.

As a first step, change the version string of the new kernel to reflect that it is for Project 1. That is, make the version string read `4.15.0-cmsc421project1-USERNAME` (substituting your UMBC username where appropriate).

Incremental Development

One of the nice things about using GitHub for submitting assignments is that it lends itself nicely to an incremental development process. As they say, Rome wasn't built overnight — nor is most software. Part of our goal in using GitHub for assignment submission is to give all of the students in the class experience with using an source control system for incremental development.

You are encouraged 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. For instance, the locking/thread safety portions of the assignment can be easily added after the main functionality is implemented, in most cases. You are also encouraged to seek out the review of your TAs to determine whether an approach might be feasible.

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 doing either of these tends to lead 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 during the first week that the assignment is assigned.

A non-trivial commit is defined for this assignment as one that meets all of these 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.

Failure to adhere to these requirments 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 assigment with a late penalty.

Skip Lists

In this assignment, you will be working with a data structure known as a Skip List. This data structure combines the amortized performance of binary search trees (that is to say O(log n) search, insertion, deletion, etc) with the relative simplicity of singly-linked lists. As this data structure is probably not one that you have studied in the past (for instance in CMSC 341), the following resources will be helpful to you in figuring out how to implement it in code:

- Skip Lists: A Probabilistic Alternative to Balanced Trees - the original research paper outlining how a skip list works
- Concurrent Maintenance of Skip Lists
- Skip list Wikipedia article

In this assignment, you will be implementing a probailistic skip list. You need not worry about rebalancing the skip list or resizing the number of pointers in the list after it is created, regardless of how many nodes are added to the list.

As probabilistic skip lists require some sort of random number generation in order to determine when to add an additional level of pointers to a given node, you will need to have some code responsible for generating random numbers. For simplicity, you may use the segment of code defined below for this purpose (based on code from the C11 standard):

```
static unsigned int next_random = 9001;
```

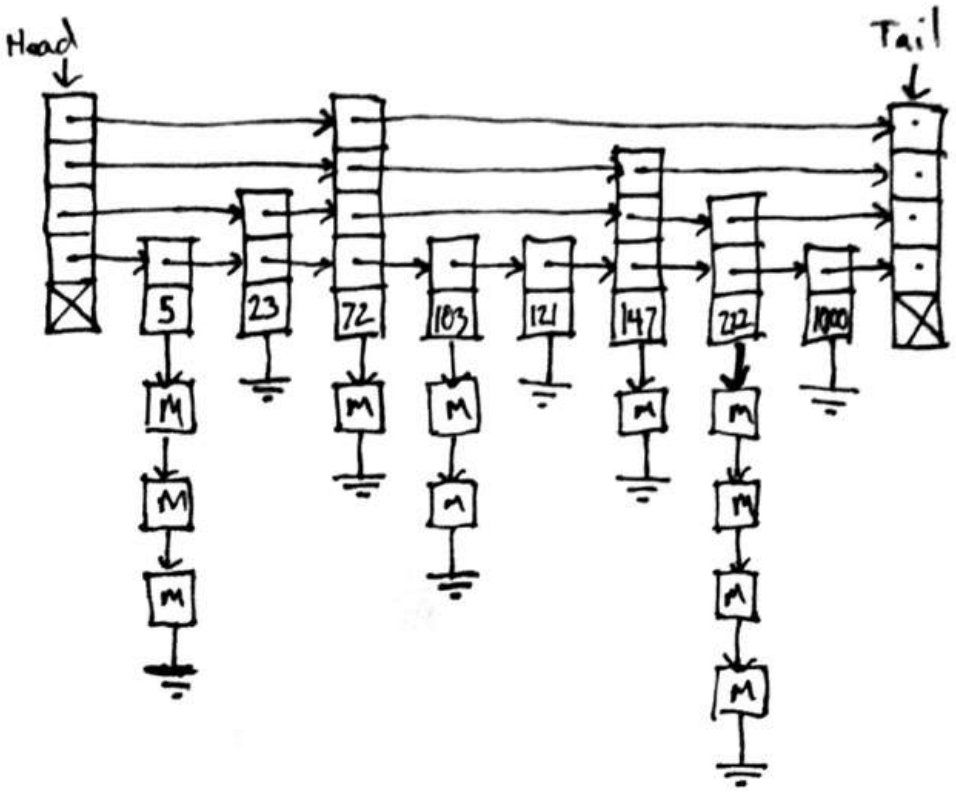
```
static unsigned int generate_random_int(void) {
    next_random = next_random * 1103515245 + 12345;
    return (next_random / 65536) % 32768;
}

static void seed_random(unsigned int seed) {
    next_random = seed;
}
```

Please note that this segment of code will generate a random integer between 0 and 32767. You can then use this to help you determine how to proceed in creating your list (for instance, if on a given list, the probability of making a node have an additional level of pointers is 1/2, then you could say that values from 0-16383 promote the node, while anything beyond that will not). **It is extremely important that you do not do any floating-point arithmetic inside the kernel, so be very careful while writing the code to determine if a node gets an additional level of pointers to not do any floating-point arithmetic!**

Each node in your skip list will be given an ID, which should be used as the key with which you will sort the list. You must ensure that all IDs are unique. Nodes within your skip list 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 in it, and you may have an "unlimited" number of total mailboxes as well.

Message Queues

As mentioned above, each node in your skip list is to act as a mailbox, each containing a FIFO queue of messages. In this way, you can sort of think of the assignment as creating a 2D linked list (but each dimension has its own requirements). The following image may help you to visualize what we are asking you to do in this assignment:



In this image, boxes containing an 'M' are messages added to a queue. The data structure above the chains of messages is the skip list, sorted by the ID of each queue. Each box above the ones with numbers are a level of pointers within the list. The ground symbol (three lines shaped like a triangle) represents the end of a message queue.

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. You may choose to reuse part of your skip list code for implementing the queues themselves, or you may use some other functionality, such as the kernel's built-in linked list code.

New System Calls

You will be adding a few new system calls to the Linux kernel in order to manage the skip list of message queues and the messages within each queue. For some system calls, you will be required to check the user id of the process calling the system call to check for whether that process has permission to run the system call.

Please keep in mind that these functions may well be called from multiple different processes simultaneously.**If you provide any**

state information in kernel-space for these functions you must provide 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 is valid. If in doubt, assume that the data passed in is invalid. Users tend to do a lot of really stupid things, after all. 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 your skip lists and queues, if you so desire (but you are not required to do so in either or both cases).

The signature and semantics for your system calls must be as follows (and they must be added to the system call table in the following order):

- long slmbx_init(unsigned int ptrs, unsigned int prob) : Initializes the mailbox system, setting up the initial state of the skip list. The ptrs parameter specifies the maximum number of pointers any node in the list will be allowed to have. The prob parameter specifies the inverse of the probability that a node will be promoted to having an additional pointer in it (that is to say that if the function is called with prob = 2, then the probability that the node will have 2 pointers is 1 / 2 and the probability that it will have 3 pointers is 1 / 4, and so on). The only valid values for the prob parameter are 2, 4, 8, and 16 — any other value should result in an error being returned. Additionally, the ptrs parameter must be non-zero — a zero value should result in an error being returned. Returns 0 on success. Only the root user (the user with a uid of 0) should be allowed to call this function.
- long slmbx_shutdown(void): Shuts down the mailbox system, deleting all existing mailboxes and any messages contained therein. Returns 0 on success. Only the root user should be allowed to call this function.
- long slmbx_create(unsigned int id, int protected) : Creates a new mailbox with the given id if it does not already exist (no duplicates are allowed). If the protected parameter is non-zero, the created mailbox will only be accessible to processes owned by the same user that owns the process that creates the mailbox (that is to say that if user bob runs a program to create the mailbox, only other programs run by bob should be able to send messages to or read messages from that mailbox if the protected parameter is non-zero). Returns 0 on success or an appropriate error on failure. If an id of 0 or (2^{32} - 1) is passed, this is considered an invalid ID and an appropriate error should be returned.
- long slmbx_destroy(unsigned int id): Deletes the mailbox identified by id if it exists and the user has permission to do so. 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 slmbx_count(unsigned int id): Returns the number of messages in the mailbox identified by id if it exists and the user has permission to access it. Returns an appropriate error code on failure.
- long slmbx_send(unsigned int id, const unsigned char *msg, unsigned int len): Sends a new message to the mailbox identified by id if it exists and the user has access to it. 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 slmbx_recv(unsigned int id, unsigned char *msg, unsigned int len): Reads the first message that is in the mailbox identified by id if it exists and the user has access to it, 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 slmbx_length(unsigned int id): Retrieves the length (in bytes) of the first message pending in the mailbox identified by id, if it exists and the user has access to it. Returns the number of bytes in the first pending message in the mailbox on success, or an appropriate error code on failure.

Each system call returns an appropriate non-negative integer on success, and a negative integer on failure which 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 a 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).

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 kernel tree called proj1tests 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 1 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 master` 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 `proj1tests` directory, in the root of the kernel 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 kernel `Makefile`. Also, include a `README` file in this 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.proj1 file in the root directory of the kernel 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 4.15 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 - 2008, 2016 Edition/POSIX.1-2008
- The Unreliable Guide to Locking [in the Linux Kernel]
- Kernel Korner: System Calls — old and outdated, but still demonstrates the concepts used for system calls

If in doubt, the Kernel API and Linux Cross Reference should be your ultimate guides.

What to do if you want to get a 0 on this project

Any of the following will cause a significant point loss on this project. Items marked with a * will result in a 0.

- 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.
- Hello World system call 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.

Please do not make us take off points for any of these things!

Last modified Tuesday, 06-Mar-2018 13:54:38 EST