CMSC 421
Final Project Design

[Evan Hart]
[12/8/2019]

1. Introduction
   1.1.System Description

The system described by the assignment was to replicate a method of process isolation known as sandboxing as its main functionality. To be more specific, the system implements two main system calls that manage a list of blocked processes from associated system call numbers. Additionally, the system should have a call that returns the number of times a blocked process has tried to access their blocked system call as well as run an application with correct permissions.

Give a brief (one paragraph) description of the project as it was assigned. You should not be describing your individual implementation in this first paragraph.

1.2.Kernel Modifications

Arch/x86/entry/common.c do_syscall_64(nr, regs)
Modified the conditional at the end of the function to block the syscall and return the correct error if a search function returns true or false.

Proj2/kernel/sbx421.c create_bst(void)
This function initializes the binary search tree array before other memory.

Proj2/kernel/sbx421.c search(proc)
This function returns a true or false based on if the existence of the process id in the data structure

List the files (and functions within files) that you modified as part of your implementation. Be sure to include the list of files that you added to the kernel as part of this list as well. For modified files, include a very brief description (single sentence per function modified) of what was modified in the file. You will be describing your changes more in-depth in a future section of this document. This section should essentially be formatted as a list.

2. Design Considerations
   2.1.System Calls and Data Structures Used

At first, I was going to implement a linked list to store all the data, as I thought that would be simple enough and I presumed it would be the easiest to handle when I would be modifying the kernel code. I then tried to keep track of it with a file table, but that ended up being very complicated, and I/O within the kernel is frowned upon. I did waste a lot of time on that without just googling it. I went back to the linked list, but I realized that the number of system calls was limited and that there would be a lot of duplicates. It would also be difficult to keep track of system calls that attempt to access their process.

My first successful data structure was a binary search tree where the system call number was stored in the root but was unable to associate the

system call numbers in a list. I first made an array of binary search trees with this same implementation of the root being the system call number but then found myself writing a large amount of code to get around this. Instead, I associate the index of the array with the system call number tied to its respective tree. This was after a lot of trial and error and is what I spent a large amount of unnecessary time on.

In this section, describe at a high level what your approach was to implementing the system calls required for the system and the data structures that they use. Basically, describe your thought process here. This is not meant to be an in-depth description, but rather just laying the groundwork for what you did in implementing the system, and why you chose to do things in the way that you did. This section should be a few (two to four) paragraphs long.

2.2.User-space Programs

The two wrapper programs I implemented check for command line errors and execute the system call. The wrapper functions are in C++ instead of C. This allows for better type handling of the pid.

I did not have time to fully implement count or sbx421_run.

This section is to be of the same format as the previous section, but about three user-space programs. You may split this into two different sub-sections – one for the three simple programs and one for the sbx421_run program. Once again, this section should be about two to four paragraphs long.

3. System Design

3.1.System Calls and Data Structures Used

To start, the main idea of the data structure I used was an array of binary search trees. I used the index of the array to reference the system call number parameter, nr, by making the size of the array the macro for the maximum system call number in the system. This way, if one chooses to add new system calls, this part of the code will still work fine. I found this from looking at kernel code.

The actual functionality of the binary search tree data structure includes search, insert, delete, minimum, get, and clear functions. I used previous information from data structures to implement these functions. These functions are defined in a separate C file in the same directory and called in the system call file. The function declarations, node, and new node definitions are in the same directory in a header file. Each node holds a process id, pointers to left and right nodes, a count variable, and a rw_lock pointer.

The search function is a very simple implementation of binary search. The parameters passed in are the root node of the tree and the process id of the node to be searched. Since binary trees are ordered during the insertion process, we can search in that same order. The function consists of two if statements, the first being if we have arrived at the correct location, or the binary tree doesn’t exist. In both instances, we

return the root. After this first conditional, we compare if the node's process id is less than the id we're searching for. If so, we recursively call search on the right node, if not, we recursively call search on the left.

The insert function takes in two parameters, the process id to be inserted and a pointer for the node to start from. Insert traverses in the same way as search, however, the sentinel if statement that precedes the traversal section is looking for the current node to be null. Once this is found, it calls the struct n_node, which takes in an argument of a pid and allocates memory for a new node to be inserted. The recursive call links the new node to the end of the path that was traversed.

For the delete function, I utilize the minimum function for rebalancing. This function uses a while loop to find the smallest value within the tree by traversing left. The delete function traverses the same as search and insert to find the value and checks if the root is null. Once it finds the parent of the node, it frees the correct child and reorders/ swaps parent and child nodes based on the minimum value using temporary nodes. The get function returns the pid of the node. I used this for testing in userspace.

The block system call adds a process id to the tree located at the index of the system call number. The unblock system call removes a process id from its tree. I implemented the block system call by calling the search function first. If it is a duplicate, we exit. If it is not, we add it to the tree located at the nr index. This system call also checks for errors in the process id values, and checks if the user has correct permissions and returns the corresponding error code. The unblock system call error checks in the same way, and conversely removes a process from its tree by searching for it first and then removing it. If it does not find the node, we return an error.

The count system call counts the number of times a process tried to access its blocked system call number. I did not have time to fully implement this into my binary search tree, but I wrote the system call definition for it. It is essentially the same as insert but instead, count is called. I planned to implement this by having a count variable within the function, as well as an external search function in the syscall file that adds a count every time the search is called and returns true. This function was also intended to be used by my kernel modifications. It would have taken in a system call number and used task_pid_nr(current) to search. I was unable to get this to work and it was returning errors during compilation.

Each system call uses pthreads for locking. I referenced a few pthread guides written by Greg Ippolito and ended up starting to implement a list of rwlocks to block incorrect permissions. I am not sure if this initial implementation was on the right path, as I was unable to get close to testing locks. However, I call two pthread functions before searching to determine an insert, delete, or count and afterwards. These are rw locks and unlocks. Read for count and write for delete and insert.

Within the kernel code, I modified the function do_syscall_64 within the file arch/x86/entry/common.c. The comments in the code refer to the conditional in the bottom as the area from which system calls are dispatched. I added a variable that acts as a boolean. This boolean is set by the external search function and searches for the nr and task_pid_nr(current), which I explained earlier. If this boolean returns true, the regs->ax is set to -EPERM, so that the system call will interrupt and a correct error code will be passed before execution.

Describe your implementation of the kernel portion of the assignment in-depth. The goal here is to give the reader enough information that he or she could implement the same system without having looked at your code. Here, you should dive down deeper into how the code is implemented, going as far as to describe exactly what changes you made to functions within the kernel and where those changes were made (and why they were made at the location you made them. This section should be of significant length, as this is where you will be describing the bulk of your design. Essentially, we're looking for at least a page or so (single-spaced) description of how everything works in this section of the document.

3.2.User-space Programs

As I explained earlier, I only implemented two wrapper functions that I am not even sure work. These functions check the command line and returns an error if the wrong number of arguments are given. If the correct number of arguments are given, the function attempts to call the system call it is wrapped around. The two userland functions are for block and unblock. They are identical except for the system call they utilize.

This section is to be of the same format as the previous section, but about the user-space code. It will probably be somewhat shorter than the previous section, as the user-space code is relatively simple in comparison.

4. References

Ippolito, Greg. "POSIX Thread (Pthread) Libraries." *Linux Tutorial: POSIX Threads*, https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html.

Ippolito, Greg. "Home." *Linux Tutorial: POSIX Threads*, http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#CREATIONTERMINATION.

List any references you used in your assignment here. You do not have to list the Linux kernel source code files here (or this template), but you should list any web references or any other such things that you used in developing your assignment here. Please use a standard citation format (MLA or APA are fine) for your list of references.