UF UNIVERSITY of FLORIDA
Herbert Wertheim
College of Engineering

COP4600

P1: System Calls

Prologue

You work for a top-secret shadow government organization dedicated to the rise of the Silurian overlords. You, as a faithful member of the Lizard Legion, are part of the team charged with improving data storage and handling, particularly tracking ***metadata*** – that is, data about data – within the organization's computer systems. You have been tasked to build a coded message subsystem under the guise of process logging for kernels running in "Sky Skink", the cloud computing system. Naturally, the Legion uses the superior Reptilian operating system distribution.

Overview

In this project, you will implement a system call in Reptilian along with three static library functions that allow the system call to be invoked from a C API. These custom system calls will *get* and *set* a custom process log level that will sit atop the standard Linux kernel's diagnostic message logging system (**`dmesg`**) and allow processes to submit log entries along with a log level. If the log level for the message is more severe (lower than) the current log level, the message will be added to the kernel log. Log levels and names will correspond to those in the Linux kernel. We, as your benevolent lizard overlords, will provide a program that exercises and demonstrates the new calls. You create a short video to demonstrate your code. (Our masters will be most pleased.) You will submit the project via Canvas so as not to invite suspicion.

Table 1. Kernel Log Levels and Corresponding Process Log Levels

Kernel Level Name	Description	#	Process Level Name
`KERN_EMERG`	Emergency / Crash Imminent (no process logging)	`0`	`PROC_OVERRIDE`
`KERN_ALERT`	Immediate Action Required	`1`	`PROC_ALERT`
`KERN_CRIT`	Critical/Serious Failure Occurred	`2`	`PROC_CRITICAL`
`KERN_ERR`	Error Condition Occurred	`3`	`PROC_ERROR`
`KERN_WARNING`	Warning; recoverable, but may indicate problems	`4`	`PROC_WARNING`
`KERN_NOTICE`	Notable, but not serious (e.g., security events)	`5`	`PROC_NOTICE`
`KERN_INFO`	Informational (e.g. initialization / shutdown)	`6`	`PROC_INFO`
`KERN_DEBUG`	Debug messages	`7`	`PROC_DEBUG`

NOTE: Take snapshots of your VM! You will probably brick your machine at some point during this or other projects, and you will not want to start from scratch. <u>No, seriously – take snapshots!</u>

Structure

The project is broken into four main parts:

1) Create a kernel-wide *process log level* attribute.
2) Create system calls that allow a process to *get* or *set* the *process log level* of the system.
3) Create system call that allows a process to add a process log message at a defined log level.
4) Create static library functions that allow the system calls to be invoked via a C API.

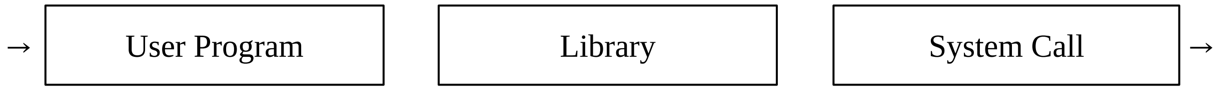


Figure 1: A system call invoked from a user program

While exact implementation may vary, the library functions must match the signatures laid out in this document, and the system calls must apply the security model properly. Logged messages have format "`$log_level_name [$executable, $pid]: $message`", e.g.:

```
PROC_ERROR [bacon_pancakes, 21]: Life is scary & dark. That is why we must find the light.
```

NOTE: Your output must match the format exactly this include whitespace and semicolon location, failure to do will result in point deductions.

System Call

The system will have a single, kernel-wide *process log level* which should initialize on boot in the kernel and must be stored persistently (until shutdown / reboot). The rules for logging are as follows:

1) Any process can read (get) the *process log level*.
2) Any process may send a process log to the kernel.
3) Only a process running as the *superuser* may write (set) the *process log level*.
4) If a message's log level is higher than the *process log level*, the message is ignored.
5) If a message's log level is lower than or equal to the *process log level*, the message will be logged.
6) The system-wide a *process log level* should be initialized to zero (0) – i.e., override logging only.
7) Log levels can have values between 0-7 (3-bit unsigned integer). Invalid level results in call failure.
8) Any successfully logged message should be logged with the corresponding kernel log level.
9) Messages will have a maximum length of 128 characters.

System calls are called via `syscall(call_num, param1, param2)`. To log a message, the call should be `syscall(PROC_LOG_CALL, msg, level).` ***Call parameters are limited to no more than two!***

Static Library

You will create a static library to invoke the system calls in a directory named **`process_log`**. This include a header, **`process_log.h`** (prototypes and level symbols), and static library file named **`libprocess_log.a`**. You will also need to provide a Makefile for the library. All sources must be contained within the **`process_log`** directory. Please note, *these filenames must match exactly!*

You will create a tarred gzip file of the **`process_log`** directory with name **`process_log.tar.gz`**. When testing, we will decompress the archive, enter the **`process_log`** directory, and build. All functions enumerated below must be made available by including **`" process_log.h"`**. See *Submission* for details.

Library Functions

`int get_proc_log_level()`

Invokes system call which reads system-wide process log level. Returns the process log level on success, and **`-1`** otherwise.

`int set_proc_log_level(int new_level)`

Invokes system call which attempts to change the system-wide process log level to **`new_level`**. Returns **`new_level`** on success, and **`-1`** otherwise. On failure, log level should be unchanged.

```
int proc_log_message(int level, char *message)
```

Invokes system call to log a message for this process. If logged, the message should appear in **`dmesg`** logs at the corresponding kernel level. Returns **`-1`** for invalid log level (or if message is too long), and **`level`** otherwise.

Harness Functions

In addition to the standard library functions, you will implement testing harness functions. The testing harness functions are used to verify security of the system calls from the system library (and are required for full credit on this assignment). We will call these functions to retrieve the information needed to make a system call. We will then call the system call within our own program. This ensures that no security checks are being done in the user-level library.

System call parameter retrieval data should be returned as a pointer to an **`int`** array of 2-4 values that can be used to make the system call (which can be cast from other types). It has this format:

```
{ call_number, num_parameters [, parameter1] [, parameter2] }
```

e.g.: `{ 42, 2, 867, 5309 }  syscall(42, 867, 5309)`

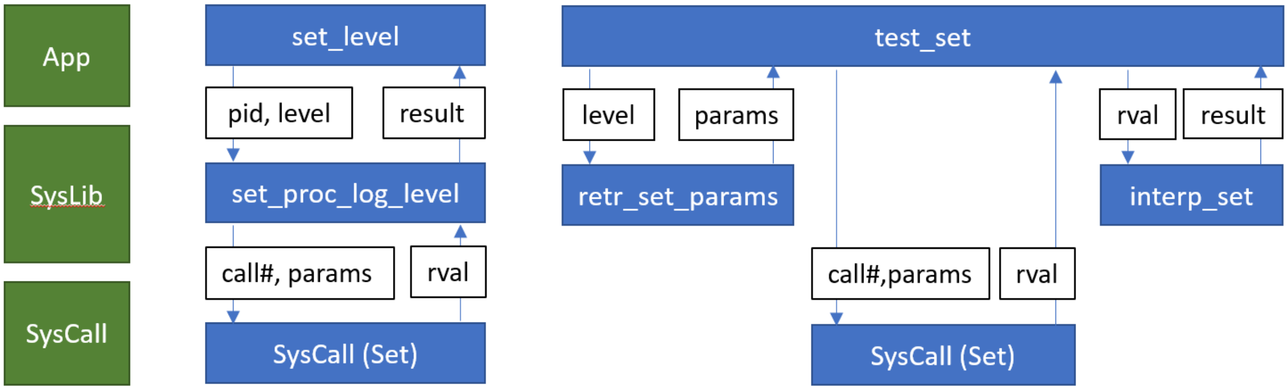


Figure 2: Harness functions can directly invoke system calls without the library functions.

These test harness elements must be implemented to test your security model:

```
#define PROC_LOG_CALL <number>
```

Definition for the system call number for **`proc_log_message`** (see ***System Call***); should be in header.

```
int* retrieve_set_level_params(int new_level)
```

Returns an **`int`** array of 2-4 values that can be used to make the set-process-log-level system call.

```
int* retrieve_get_level_params()
```

Returns an **`int`** array of 2-4 values that can be used to make the get-process-log-level system call.

```
int interpret_set_level_result(int ret_value)
```

After making the system call, we will pass the syscall return value to this function call. It should return **`set_proc_log_level`**'s interpretation of the system call completing with return value **`ret_value`**.

```
int interpret_get_level_result(int ret_value)
```

After making the system call, we will pass the syscall return value to this function call. It should return **`get_proc_log_level`**'s interpretation of the system call completing with return value **`ret_value`**.

`int interpret_log_message_result(int ret_value)`
After making the system call, we will pass the syscall return value to this function call. It should return **`proc_log_message`**'s interpretation of the system call completing with return value **`ret_value`**.

Note that there is no **`retrieve`** function for **`log_message`** as its system call format is defined above.

Submissions

You will submit the following at the end of this project (3 separate files):

- Report (`p1.txt`) in man page format on Canvas, including link to unlisted screencast video
- Kernel Patch File (`p1.diff`) on Canvas
- Compressed tar archive (`process_log.tar.gz`) for **`process_log`** library on Canvas

Report

Your report will explain how you implemented the new system call in the kernel, including what changes were made to which files and why for each. It will describe how testing was performed and any known bugs. The report should be created using `man` format saved as a .txt. The report should be no more than 500 words (about two pages in man format), cover all relevant aspects of the project, and be organized and formatted professionally – ***this is not a memo!***

Screencast

Students should submit a screencast (with audio) walking through the changes made to the operating system to enable the system calls. It should include showing/demoing your changes in action (no more than 5 minutes). Audio speed-up is prohibited. Video cannot have watermarks and must be an unlisted YouTube video!

Patch File

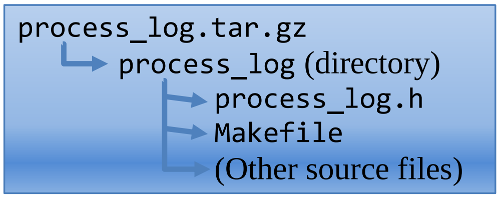
The patch file will include all changes to all files in a single patch. Applying the patches and remaking the necessary parts of Reptilian, then rebooting and then building the test code (which we will also copy over) should compile the test program.

Your project will be tested by applying the patch while in **`/usr/rep/src/reptilian-kernel`**:

```
$ git apply p1.diff
$ make && sudo make install && sudo make modules_install
```

Compressed Archive (process_log.tar.gz)

Your compressed tar file should have the following directory/file structure:



```
process_log.tar.gz
  └→ process_log (directory)
       ├→ process_log.h
       ├→ Makefile
       └→ (Other source files)
```

To build the library, we will execute these commands (**from a non-kernel-source directory**):

```
$ tar zxvf process_log.tar.gz
$ cd process_log
$ make
$ cd ..
```

To link against the library, we will execute this command:

```
$ cc -o program_name sourcefile.c -L ./process_log -lprocess_log
```

Please test your library build and linking before submission! If your library does not compile it will result in **zero credit** (0, none, goose-egg) for the library portion of the project.