

# Lab3: Adding a Reptilian System Call

## COP4600 – Operating Systems

### 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 will write a README.md file to explain 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**

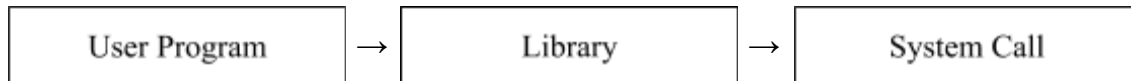
Kernel Level Name	Description	#	Process Level Name
KERN_EMERG	Emergency / Crash Imminent (no process logging)	0	PROC_OVERRIDE
KERN_ALERT	Immediate Action Required	1	PROC_ALERT
KERN_CRIT	Critical/Serious Failure Occurred	2	PROC_CRITICAL
KERN_ERR	Error Condition Occurred	3	PROC_ERROR
KERN_WARNING	Warning; 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. No, seriously – take snapshots!**

# 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\_ERR [bacon\_pancakes, 21]: Life is scary & dark. That is why we must find the light.**

## 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 *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.

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 will 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 file names 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, and **level** otherwise.

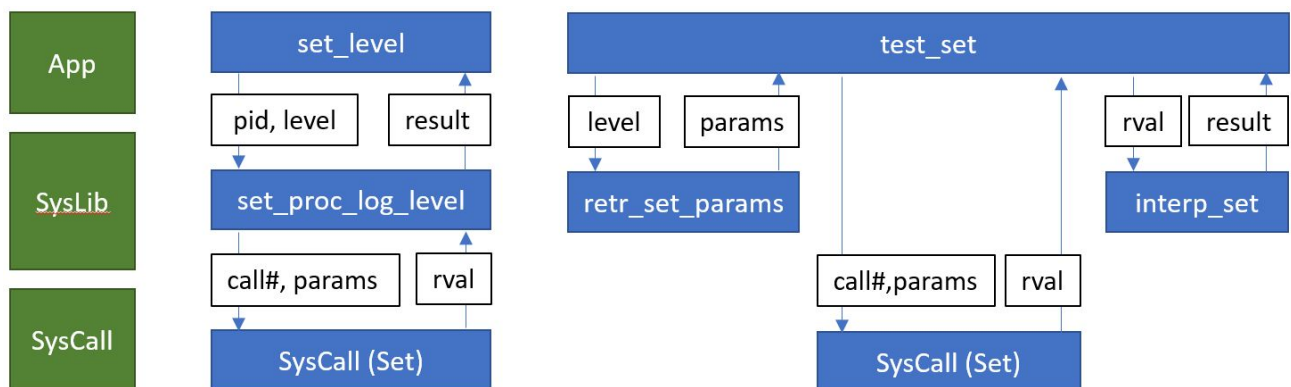
## Harness Functions

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):

- README ([README.md](#)) in Markdown file format on Canvas
- Kernel Patch File ([lab3.diff](#)) on Canvas
- Compressed tar archive ([process\\_log.tar.gz](#)) for `process_log` library on Canvas

## README

Your README 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 README should be aimed at a technical reader with little knowledge of operating systems concepts, and should be saved as a `.md`. The README should be no more than 700 words, cover all relevant aspects of the project, and be organized and formatted professionally – *this is not a memo!*

You may include more sections as you see fit, but we require that at least the following three sections be included in your README:

- Description: a specification of what your project accomplishes once installed
- Design: explanations of non-trivial design decisions you made
- Verification: an explanation of how to verify the project's functionality after installation

## Patch File

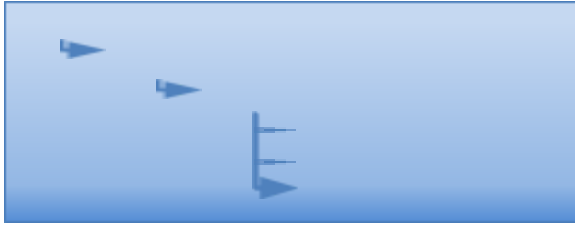
The patch file will include all changes to all files in a single patch. Applying the patches and remaking the necessary parts of Reptilian, then rebooting and then building the test code (which we will also copy over) should compile the test program.

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

```
$ git apply lab3.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.tar
    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.