**CS492 Assignment 2**

Accessing the Task struct using a Character Device Driver

We updated the required format to print the linked list, so as to increase consistency and avoid some formatting/printing problems that some students have encountered. If you have submitted your code following the old format, no worries as we will keep an eye on that.

# Task Description

The objective of this assignment is to extend a character device driver and get familiar with accessing kernel data structures. Specifically, you will:

* Extend a character device driver as a kernel module
* Implement an IOCTL interface in the character device driver, following the specifications which we will describe later
* Extend a user space program to interact with the character device driver, following the specifications which we will describe later
* Test the code in the VM we distributed for the class, using the default kernel

# Project Steps

1. Fork the repository including a basic character driver and a user-space program interacting with it: <https://classroom.github.com/a/MrmI8h10> (you will have a private repo containing the driver after you click the link and accept the assignment)
2. Study about characters device drivers: <https://docs.google.com/document/d/13mEewYtOjMOSL113oNcOJG0UDpqFYcn8DQ8jvZ570QA/edit#>
3. Extend the character device driver (under driver/) and user-space program (under src/) to satisfy the specifications of the assignment. DO NOT include new files. Modify existing files to introduce your functionality.
   1. (30pts) Extend the scull device driver by adding support for a new IOCTL command:
      1. We have prepared several example commands in the “scull\_ioctl" function in the “scull.c” file; You may refer to those examples when adding a new command.
      2. This command should return (copy) a task\_info structure (shown below) to the issuer of the IOCTL in userspace. The struct should be populated with the corresponding fields in task\_struct of the task corresponding to the issuer of the IOCTL, which can be accessed using the “current” macro in the IOCTL handler in the driver. Note: You need to define this structure as shown below.
      3. The command should be defined with the appropriate IOCTL MACRO using ‘k’ and 7 as code and sequence numbers, and struct task\_info as the type of the data being transferred.

struct task\_info {

long state;

void \*stack;

unsigned int cpu;

int prio;

int static\_prio;

int normal\_prio;

unsigned int rt\_priority;

pid\_t pid;

pid\_t tgid;

unsigned long nvcsw;

unsigned long nivcsw;

};

We have prepared examples of copying information from the kernel space to the user space program (e.g, handling of the SCULL\_IOCGQUANTUM command in the “scull\_ioctl" function)

* 1. (30pts) Extend the scull device driver to log information about every task that issues the new command you add.
     1. Initialize a linked list with each node recording the “pid” and “tgid” of a thread
        1. You can design the data structure for nodes as you see proper; The requirement here is each node should keep at least the above mentioned “pid” and “tgid”
        2. You may want to initialize the linked list when initializing the device driver
     2. Whenever the new command is requested by a task and, if the task is not already in the linked list, create a new node recording the “pid” and “tgid” of the task and insert the node to the linked list
        1. You can create your own functions to operate on the linked list or you can use built-in kernel functions to help you do that (Reference: <https://medium.com/@414apache/kernel-data-structures-linkedlist-b13e4f8de4bf>)
        2. When you create a new node, you may want to use “kmalloc” to allocate memory for the node (Reference: <http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch11lev1sec4.html>)
        3. Because multiple tasks may issue the new command concurrently, that means there may be multiple concurrent operations to the linked list. You may want to use a lock when you operate on the linked list. One way of using lock is to use the kernel version of “mutex” (you must have become familiar with user-space “mutex” from your CS-392 course) (Reference: example code of using mutex with a linked list: <https://www.kernel.org/doc/htmldocs/kernel-locking/Examples.html>)
     3. When you “get rid of the char dev entry” (note: you can find this as a comment in the code we give):
        1. You will need to print the items in the linked list to “dmesg”, in the format of:

Linked List to be deleted:

Task 1: PID: XX; TGID: XX ->

Task 2: PID: XX; TGID: XX ->

…

(please replace XX with the exact PID/TGID, and please follow the above format as we will rely on that when we check the execution results; you can skip the “->” for the last element. if you have submitted your code following the previous format, please no worries as we will pay attention to that)

* + - 1. You will need to destroy the linked list. When you unlink a node, you may want to recycle the memory of the node using “kfree” (Reference: <http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch11lev1sec4.html>)

1. Extend the userspace program (under src/) to utilize the new IOCTL by adding the following commands, triggered from the command line (help on currently supported commands can be seen by running ./scull h):
   * 1. (20pts) Add ./scull p <n>

The userspace program should use n **processes** (0 < n < 11) to concurrently issue the IOCTL after opening the device (please only open and close the device only once from the parent process). Each process should print out the information received by the driver.

Example:

$ ./scull p 2

Device (/dev/scull) opened

state 0, stack 0000000060ab50dc, cpu 1, prio 120, sprio 120, nprio 120, rtprio 0, pid 47924, tgid 47924, nv 2, niv 5

state 0, stack 000000008f001547, cpu 1, prio 120, sprio 120, nprio 120, rtprio 0, pid 47925, tgid 47925, nv 0, niv 3

Device (/dev/scull) closed

* + 1. (20pts) Add ./scull t <n>

The userspace program should use n **threads** (0 < n < 11) to concurrently issue the IOCTL after opening the device. Each thread should print out the information received by the driver, as above.

1. Update README.md with your details, including a link to the github classroom repo for the assignment.
2. Create a tar archive of your repo including only .c, .h, and Makefile files. Do not include the .git directories. Only the following files should be included:

scull

scull/README.md

scull/driver

scull/src

scull/src/Makefile

scull/src/scull.c

scull/driver/debug.h

scull/driver/scull.init

scull/driver/Makefile

scull/driver/scull.h

scull/driver/access\_ok\_version.h

scull/driver/scull.c

1. Submit the archive using gradescope: <https://www.gradescope.com/courses/241080/assignments/1106595>

# Grading Policy

* This is an individual assignment. Individual assignments, as the words indicate, are to be done INDIVIDUALLY. Any form of plagiarism (from each other or the Internet) will result in a 0 and you will be reported to the Honor Board.
* Any late submission without pre-approval will result in a 0.
* The assignment will be graded based on correctness and style (well formed code with comments). Code not compiling will result in a 0 in the corresponding assignment task.