OPERATING SYSTEMS PROJECT

NAME: Sushmitha S SRN: PES2UG22CS607

SECTION : J

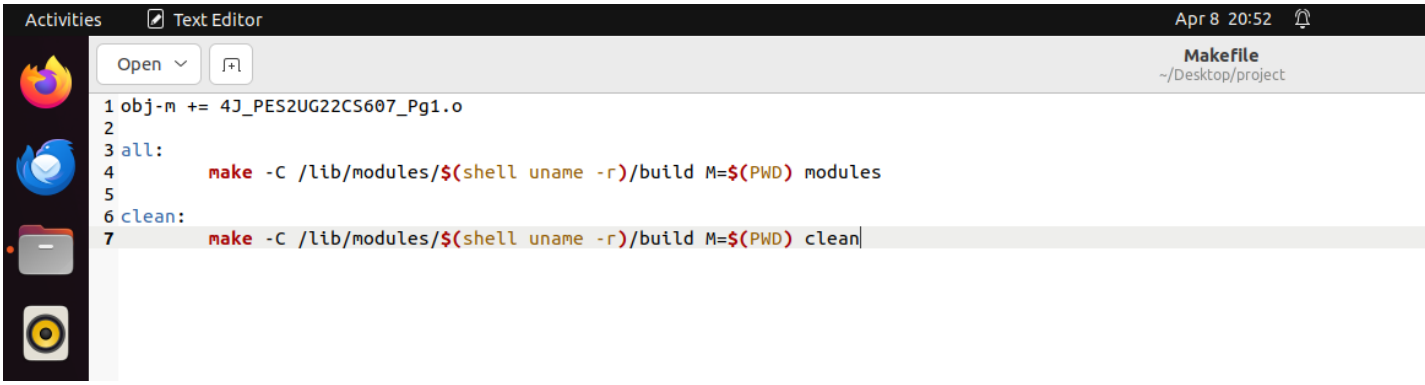
QUESTION :

Execute a program that will create multiple processes/threads (children and siblings). While this task is executing, output the task name (known as executable name), state and process id of each thread created by the process in a tree structure.

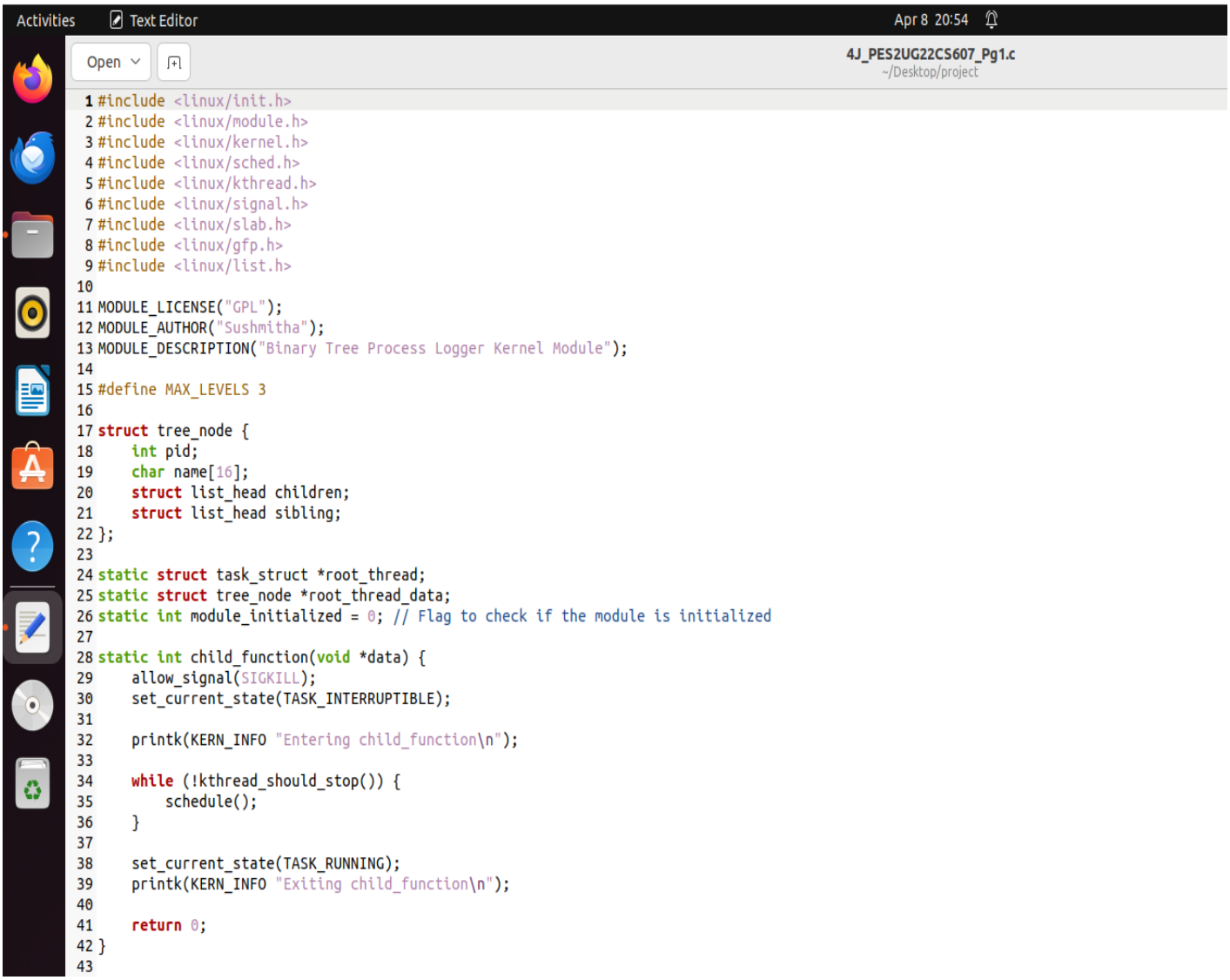
Example: my_kernel_module <process id of the program executing>

Code:

Makefile



Program file




```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/kthread.h>
#include <linux/signal.h>
#include <linux/slab.h>
#include <linux/gfp.h>
#include <linux/list.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sushmitha");
MODULE_DESCRIPTION("Binary Tree Process Logger Kernel Module");

#define MAX_LEVELS 3

struct tree_node {
    int pid;
    char name[16];
    struct list_head children;
    struct list_head sibling;
};

static struct task_struct *root_thread;
static struct tree_node *root_thread_data;
static int module_initialized = 0; // Flag to check if the module is initialized

static int child_function(void *data) {
    allow_signal(SIGKILL);
    set_current_state(TASK_INTERRUPTIBLE);

    printk(KERN_INFO "Entering child_function\n");

    while (!kthread_should_stop()) {
        schedule();
    }

    set_current_state(TASK_RUNNING);
    printk(KERN_INFO "Exiting child_function\n");

    return 0;
}

```

```c
static void print_tree(struct tree_node *root, int level) {
    struct tree_node *node;
    struct list_head *pos, *q;

    printk(KERN_INFO "%*s├── %s(%d)\n", level * 4, "", root->name, root->pid);

    list_for_each_safe(pos, q, &root->children) {
        node = list_entry(pos, struct tree_node, sibling);

        // Recursively print the tree structure
        print_tree(node, level + 1);

        // Remove the printed node from the list
        list_del(pos);
        kfree(node);
    }
}

static int create_binary_tree(int level, struct task_struct *parent, struct tree_node *parent_node) {
    int i;
    char thread_name[16];

    if (level >= MAX_LEVELS) {
        return 0;
    }

    for (i = 0; i < 2; ++i) {
        struct task_struct *thread;
        struct tree_node *thread_node;

        snprintf(thread_name, sizeof(thread_name), "thread_%d_%d", level, i);

        // Create a child thread
        thread = kthread_run(child_function, NULL, thread_name);
        if (IS_ERR(thread)) {
            printk(KERN_ERR "Failed to create child thread\n");
            return PTR_ERR(thread); // Return error code
        }

        // Log information about the created process/thread
        printk(KERN_INFO "Created thread: PID=%d, Parent PID=%d, Level=%d\n", thread->pid, parent->pid, level);

        // Create a tree node for the child
        thread_node = kmalloc(sizeof(struct tree_node), GFP_KERNEL);
        if (!thread_node) {
            // Handle memory allocation failure
            return -ENOMEM;
        }
        thread_node->pid = thread->pid;
        snprintf(thread_node->name, sizeof(thread_node->name), "%s", thread_name);
        INIT_LIST_HEAD(&thread_node->children);
```

```
95
96          // Add the child to the parent's list
97          list_add_tail(&thread_node->sibling, &parent_node->children);
98
99          // Recursively create the thread tree
100         create_binary_tree(level + 1, thread, thread_node);
101     }
102
103     return 0;
104 }
105
106 static int __init binary_tree_logger_init(void) {
107     int ret;
108
109     if (module_initialized) {
110         printk(KERN_INFO "Module already initialized\n");
111         return 0;
112     }
113
114     printk(KERN_INFO "Binary Tree Logger Module: Initialization\n");
115
116     // Create a root process/thread for the binary tree
117     root_thread = kthread_run(child_function, NULL, "root_thread");
118     if (IS_ERR(root_thread)) {
119         printk(KERN_ERR "Failed to create root thread\n");
120         return PTR_ERR(root_thread);
121     }
122
123     // Create a tree node for the root
124     root_thread_data = kmalloc(sizeof(struct tree_node), GFP_KERNEL);
125     if (!root_thread_data) {
126         // Handle memory allocation failure
127         kthread_stop(root_thread);
128         return -ENOMEM;
129     }
130     root_thread_data->pid = root_thread->pid;
131     snprintf(root_thread_data->name, sizeof(root_thread_data->name), "root_thread");
132     INIT_LIST_HEAD(&root_thread_data->children);
133
134     // Log information about the root thread
135     printk(KERN_INFO "Created root thread: PID=%d\n", root_thread->pid);
136
137     // Create a binary tree
138     ret = create_binary_tree(1, root_thread, root_thread_data);
139     if (ret) {
140         // Stop the root thread and free allocated memory in case of error
141         kthread_stop(root_thread);
142         kfree(root_thread_data);
143         return ret;
144     }
```

```
145
146     // Print the binary tree structure
147     printk(KERN_INFO "Process Tree Structure:\n");
148     print_tree(root_thread_data, 0);
149
150     module_initialized = 1;
151
152     return 0;
153 }
154
155 static void __exit binary_tree_logger_exit(void) {
156     // Stop the entire process tree starting from the root thread
157     kthread_stop(root_thread);
158
159     printk(KERN_INFO "Binary Tree Logger Module: Cleanup\n");
160 }
161
162 module_init(binary_tree_logger_init);
163 module_exit(binary_tree_logger_exit);
```

Commands:

1. make

The make command is a build automation tool used in Unix-like operating systems to compile and build software projects efficiently. It reads instructions from a file called Makefile to determine how to build the target(s) specified on the command line.

- Purpose: make automates the process of compiling and linking software projects by executing the necessary compilation commands based on the dependencies and rules specified in a Makefile.

- Syntax: The basic syntax of the make command is:
  make [options] [target(s)]

- Options: make supports various options to control its behavior, such as specifying the makefile to use (-f), setting the maximum number of parallel jobs (-j), changing to a different directory (-C), and more.

- Targets: Targets are components of the project that need to be built, such as executable binaries, object files, or libraries. They are specified as arguments to make on the command line.

- Default Target: If no target is specified on the command line, make builds the default target specified in the Makefile, typically the first target encountered.

- Makefile: The Makefile is a text file that contains rules and instructions for make on how to build the project. It specifies dependencies between files and the commands to execute to build the targets.

- Operation: make reads the Makefile and determines the dependencies and build commands for the specified target(s). It checks the timestamps of the target and its dependencies to decide whether the target needs to be rebuilt. If so, it executes the commands associated with the target to rebuild it.

  Overall, the make command is a versatile and widely used tool for automating the build process of software projects, providing developers with a standardized and efficient way to manage complex build tasks.

2. sudo insmod 4J_PES2UG22CS607_Pg1.ko

The sudo insmod 4J_PES2UG22CS607_Pg1.ko command is used to insert a kernel module (4J_PES2UG22CS607_Pg1.ko) into the running Linux kernel.

- sudo: This part of the command invokes the sudo (SuperUser Do) command, which allows the user to execute the subsequent command with superuser (root) privileges. Kernel module insertion typically requires root privileges because it involves modifying the kernel, which is a critical operation.

- insmod: This is the command used to insert a kernel module into the kernel. It stands for "insert module". It is part of the Linux kernel module utilities and is responsible for loading modules into the kernel's address space.

- 4J_PES2UG22CS607_Pg1.ko: This is the filename of the kernel module that you want to insert into the kernel. Kernel modules are typically compiled into files with a .ko extension.

3. sudo dmesg

The sudo dmesg command is a utility used in Unix-like operating systems to display the kernel's message buffer, which contains system log messages generated by the kernel during boot-up and while the system is running.

- sudo: This part of the command invokes the sudo (SuperUser Do) command, which allows the user to execute the subsequent command with elevated privileges, typically root privileges. Running dmesg usually requires root privileges to access system logs.

- dmesg: This is the actual command being executed. It stands for "diagnostic message" and is used to print the kernel ring buffer or system log messages. These messages can include information about hardware detection, device driver initialization, kernel errors, warnings, and other system-related events.

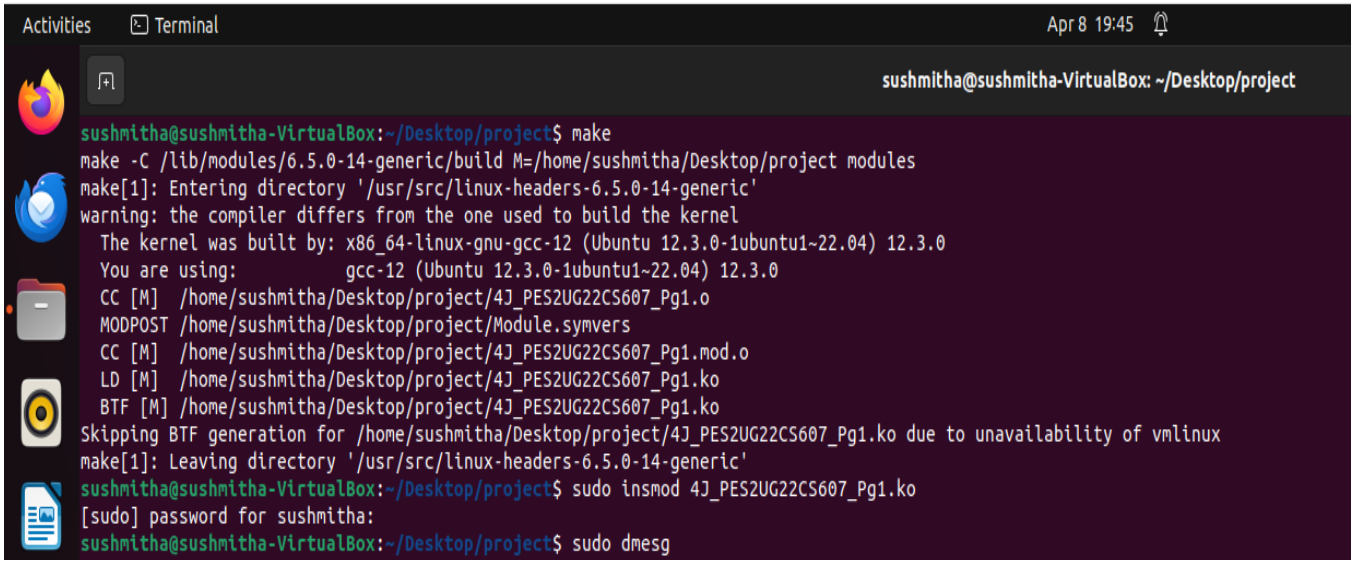
4. sudo rmmod 4J_PES2UG22CS607_Pg1

The command sudo rmmod 4J_PES2UG22CS607_Pg1 is used to remove a loaded kernel module from the running Linux kernel. Here's an explanation of each part of the command:

- sudo: This part of the command invokes the sudo (SuperUser Do) command, which allows the subsequent command (rmmod) to be executed with elevated privileges. Removing a kernel module typically requires root privileges because it involves modifying the kernel.

- rmmod: This is the command used to remove a loaded kernel module. It stands for "remove module". It is part of the Linux kernel module utilities and is responsible for unloading modules from the kernel's address space.

- 4J_PES2UG22CS607_Pg1: This is the name of the kernel module that you want to remove. It should be the name of the module without the .ko extension. When you run this command, rmmod will attempt to unload the specified module from the running kernel.



Output:

```
[ 2194.282055] Binary Tree Logger Module: Initialization
[ 2194.289821] Created root thread: PID=3016
[ 2194.289867] Entering child_function
[ 2194.289872] Created thread: PID=3017, Parent PID=3016, Level=1
[ 2194.289892] Entering child_function
[ 2194.289895] Created thread: PID=3018, Parent PID=3017, Level=2
[ 2194.289912] Entering child_function
[ 2194.289916] Created thread: PID=3019, Parent PID=3017, Level=2
[ 2194.289932] Entering child_function
[ 2194.289936] Created thread: PID=3020, Parent PID=3016, Level=1
[ 2194.289970] Entering child_function
[ 2194.293596] Created thread: PID=3021, Parent PID=3020, Level=2
[ 2194.293660] Entering child_function
[ 2194.301300] Created thread: PID=3022, Parent PID=3020, Level=2
[ 2194.301306] Process Tree Structure:
[ 2194.301307] ├── root_thread(3016)
[ 2194.301348]     ├── thread_1_0(3017)
[ 2194.301350]         ├── thread_2_0(3018)
[ 2194.301352]         ├── thread_2_1(3019)
[ 2194.301353]     ├── thread_1_1(3020)
[ 2194.301354]         ├── thread_2_0(3021)
[ 2194.301355]         ├── thread_2_1(3022)
[ 2194.310106] Entering child_function
sushmitha@sushmitha-VirtualBox:~/Desktop/project$
```