Operating Systems 2019 Spring - HW2 Report

StudentID: 1062022S
Name: 廖以諾

Objectives

Design kernel modules that traverse all tasks in the Linux system. Firstly, traverse system tasks in a linear manner and the expected list of tasks are ordered by pid. In addition, design the DFS algorithm to traverse system tasks in a parent-children manner and the expected list of tasks are ordered according to the relationship of inheritance.

There are two missions in the assignment:

1. Iterate Tasks Linearly
2. Iterate Tasks in the DFS Manner

1. Iterate Tasks Linearly

1.1 Initiate and Terminate the Kernel

To initiate and terminate kernel, we need `module_init()` and `module_exit()` API. Therefore, we have to include the related libraries as shown below:

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
```

Then, we create `task_iterator_init()` and `task_iterator_exit()` functions to accomplish our tasks during kernel initiation and termination, respectively. Please check code for details.

1.2 Iterate Tasks Linearly

To iterate kernel tasks linearly, we can use `for_each_process()` function defined in `<linux/sched/signal.h>`. `for_each_process()` will iterate over all tasks and its input is in the type of `struct task_struct`. The only thing we have to do is print out the pid, process name, and state while traversing.

Before jump into using `for_each_process()`, we have to understand how `struct task_struct` looks like and how we retrieve the tasks' pid, process name, and state. Check `sched.h` carefully, we understand that to get pid, process name, and state, we call task->pid, task->comm, and task->state, respectively. To use `task_struct`, remember to include `<linux/sched/signal.h>`.

```
#include <linux/sched/signal.h>
```

Now that we know how to retrieve information from `task_struct`, we call `for_each_process()` and print the information to the kernel log while iterating as shown in the following code.

```
// load kernel module
int task_iterator_init(void) {
    printk(KERN_INFO "----- Loading Module -----\n");

    // iterate tasks linearly
    struct task_struct *task;
    for_each_process(task) {
        printk(KERN_INFO "pid: %d | process name: %s | state: %d\n", \
               task->pid, task->comm, task->state);
    }

    return 0;
}
```

1.3 Final Results

To check results, we initiate/terminate the kernel module and check kernel log:

- `sudo insmod hw2_linear.ko`
- `sudo rmmod hw2_linear.ko`
- `dmesg`

The kernel log (partial) is shown as:

```
[ 1636.885194] ----- Loading Module -----                                    [147/1982]
[ 1636.885197] pid: 1 | process name: systemd | state: 1
[ 1636.885198] pid: 2 | process name: kthreadd | state: 1
[ 1636.885199] pid: 4 | process name: kworker/0:0H | state: 1026
[ 1636.885200] pid: 5 | process name: kworker/u8:0 | state: 1026
[ 1636.885201] pid: 6 | process name: mm_percpu_wq | state: 1026
[ 1636.885202] pid: 7 | process name: ksoftirqd/0 | state: 1
[ 1636.885203] pid: 8 | process name: rcu_sched | state: 1026
[ 1636.885204] pid: 9 | process name: rcu_bh | state: 1026
[ 1636.885205] pid: 10 | process name: migration/0 | state: 1
[ 1636.885206] pid: 11 | process name: watchdog/0 | state: 1
[ 1636.885207] pid: 12 | process name: cpuhp/0 | state: 1
[ 1636.885208] pid: 13 | process name: cpuhp/1 | state: 1
[ 1636.885209] pid: 14 | process name: watchdog/1 | state: 1
[ 1636.885210] pid: 15 | process name: migration/1 | state: 1
[ 1636.885211] pid: 16 | process name: ksoftirqd/1 | state: 1
[ 1636.885212] pid: 18 | process name: kworker/1:0H | state: 1026
[ 1636.885213] pid: 19 | process name: cpuhp/2 | state: 1
[ 1636.885214] pid: 20 | process name: watchdog/2 | state: 1
[ 1636.885215] pid: 21 | process name: migration/2 | state: 1
[ 1636.885215] pid: 22 | process name: ksoftirqd/2 | state: 1
[ 1636.885216] pid: 24 | process name: kworker/2:0H | state: 1026
[ 1636.885217] pid: 25 | process name: cpuhp/3 | state: 1
[ 1636.885218] pid: 26 | process name: watchdog/3 | state: 1
[ 1636.885219] pid: 27 | process name: migration/3 | state: 1
[ 1636.885220] pid: 28 | process name: ksoftirqd/3 | state: 1
[ 1636.885221] pid: 30 | process name: kworker/3:0H | state: 1026
[ 1636.885222] pid: 31 | process name: kdevtmpfs | state: 1
[ 1636.885224] pid: 32 | process name: netns | state: 1026
[ 1636.885225] pid: 33 | process name: rcu_tasks_kthre | state: 1
[ 1636.885225] pid: 34 | process name: kauditd | state: 1
[ 1636.885226] pid: 35 | process name: kworker/0:1 | state: 1026
[ 1636.885227] pid: 36 | process name: khungtaskd | state: 1
[ 1636.885228] pid: 37 | process name: oom_reaper | state: 1
[ 1636.885229] pid: 38 | process name: writeback | state: 1026
[ 1636.885230] pid: 39 | process name: kcompactd0 | state: 1
[ 1636.885231] pid: 40 | process name: ksmd | state: 1
[ 1636.885232] pid: 41 | process name: khugepaged | state: 1
[ 1636.885233] pid: 42 | process name: crypto | state: 1026
[ 1636.885234] pid: 43 | process name: kintegrityd | state: 1026
[ 1636.885235] pid: 44 | process name: kblockd | state: 1026
[ 1636.885236] pid: 45 | process name: ata_sff | state: 1026
[ 1636.885237] pid: 46 | process name: md | state: 1026
[ 1636.885238] pid: 47 | process name: edac-poller | state: 1026
[ 1636.885239] pid: 48 | process name: devfreq_wq | state: 1026
[ 1636.885240] pid: 49 | process name: watchdogd | state: 1026
[ 1636.885241] pid: 50 | process name: kworker/u8:1 | state: 1026
```

To verify our result, use `ps -el` to compare. The `ps -el` result (partial) is shown as:

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD   [145/384]
4 S     0     1     0  0  80   0 - 29922 -      ?        00:00:01 systemd
1 S     0     2     0  0  80   0 -     0 -      ?        00:00:00 kthreadd
1 I     0     4     2  0  60 -20 -     0 -      ?        00:00:00 kworker/0:0H
1 I     0     5     2  0  80   0 -     0 -      ?        00:00:00 kworker/u8:0
1 I     0     6     2  0  60 -20 -     0 -      ?        00:00:00 mm_percpu_wq
1 S     0     7     2  0  80   0 -     0 -      ?        00:00:00 ksoftirqd/0
1 I     0     8     2  0  80   0 -     0 -      ?        00:00:00 rcu_sched
1 I     0     9     2  0  80   0 -     0 -      ?        00:00:00 rcu_bh
1 S     0    10     2  0 -40   - -     0 -      ?        00:00:00 migration/0
5 S     0    11     2  0 -40   - -     0 -      ?        00:00:00 watchdog/0
1 S     0    12     2  0  80   0 -     0 -      ?        00:00:00 cpuhp/0
1 S     0    13     2  0  80   0 -     0 -      ?        00:00:00 cpuhp/1
5 S     0    14     2  0 -40   - -     0 -      ?        00:00:00 watchdog/1
1 S     0    15     2  0 -40   - -     0 -      ?        00:00:00 migration/1
1 S     0    16     2  0  80   0 -     0 -      ?        00:00:00 ksoftirqd/1
1 I     0    18     2  0  60 -20 -     0 -      ?        00:00:00 kworker/1:0H
1 S     0    19     2  0  80   0 -     0 -      ?        00:00:00 cpuhp/2
5 S     0    20     2  0 -40   - -     0 -      ?        00:00:00 watchdog/2
1 S     0    21     2  0 -40   - -     0 -      ?        00:00:00 migration/2
1 S     0    22     2  0  80   0 -     0 -      ?        00:00:00 ksoftirqd/2
1 I     0    24     2  0  60 -20 -     0 -      ?        00:00:00 kworker/2:0H
1 S     0    25     2  0  80   0 -     0 -      ?        00:00:00 cpuhp/3
5 S     0    26     2  0 -40   - -     0 -      ?        00:00:00 watchdog/3
1 S     0    27     2  0 -40   - -     0 -      ?        00:00:00 migration/3
1 S     0    28     2  0  80   0 -     0 -      ?        00:00:00 ksoftirqd/3
1 I     0    30     2  0  60 -20 -     0 -      ?        00:00:00 kworker/3:0H
5 S     0    31     2  0  80   0 -     0 -      ?        00:00:00 kdevtmpfs
1 I     0    32     2  0  60 -20 -     0 -      ?        00:00:00 netns
1 S     0    33     2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks_kthre
1 S     0    34     2  0  80   0 -     0 -      ?        00:00:00 kauditd
1 I     0    35     2  0  80   0 -     0 -      ?        00:00:00 kworker/0:1
1 S     0    36     2  0  80   0 -     0 -      ?        00:00:00 khungtaskd
1 S     0    37     2  0  80   0 -     0 -      ?        00:00:00 oom_reaper
1 I     0    38     2  0  60 -20 -     0 -      ?        00:00:00 writeback
1 S     0    39     2  0  80   0 -     0 -      ?        00:00:00 kcompactd0
1 S     0    40     2  0  85   5 -     0 -      ?        00:00:00 ksmd
1 S     0    41     2  0  99  19 -     0 -      ?        00:00:00 khugepaged
1 I     0    42     2  0  60 -20 -     0 -      ?        00:00:00 crypto
1 I     0    43     2  0  60 -20 -     0 -      ?        00:00:00 kintegrityd
1 I     0    44     2  0  60 -20 -     0 -      ?        00:00:00 kblockd
1 I     0    45     2  0  60 -20 -     0 -      ?        00:00:00 ata_sff
1 I     0    46     2  0  60 -20 -     0 -      ?        00:00:00 md
1 I     0    47     2  0  60 -20 -     0 -      ?        00:00:00 edac-poller
1 I     0    48     2  0  60 -20 -     0 -      ?        00:00:00 devfreq_wq
1 I     0    49     2  0  60 -20 -     0 -      ?        00:00:00 watchdogd
1 R     0    50     2  0  80   0 -     0 -      ?        00:00:00 kworker/u8:1
```

If we check the pid and process name carefully, we find that our kernel module correctly iterates over all tasks, proving that our program functions properly.

2. Iterate Tasks in the DFS Manner

2.1 Depth-First Search

DFS is a simple and powerful algorithm. To traverse all tasks in the DFS manner, we can follow the below pseudo code. Hence, our task is, in fact, to figure out how to access the children list and apply the DFS function.

```
// pseudo code of depth-first search
void dfs(task) {

    // manipulation for current task
    ...

    // visit neighbors of current task
    for(all neighbors of current task) {
        // dfs
        dfs(neighbor);
    }

    return;
}
```

2.2 Linux Task's Children Structure

A task's `children` and `sibling` are defined by `list_head` type as shown in `<linux/sched.h>`. `children` is the root of the task's children list and `sibling` is the entry of the children list. We can imagine that `children` is the pointer pointing to the location of children list from parent and `sibling` is the list that contains all current level children.

Now that we understand the children list is implemented by `list_head`, it is intuitive that we can exploit `list_for_each()` to traverse all children of the current task. We get each child one-by-one and pass to `dfs(child)` recursively. To get the child, we also need the help of `list_entry()` function. It is defined as:

```
/**
 * list_entry - get the struct for this entry
 * @ptr:    the &struct list_head pointer.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 */
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

Note: reference of traversing children list of `task_struct` is available here [here.]

2.3 Traverse Tasks in DFS

Finally, we use the DFS algorithm explained in section 2.1 along with the understanding of how we access a task's children to reach the final code.

```
// dfs
void dfs(struct task_struct *task) {
    struct list_head *list;
    struct task_struct *child;

    // manipulation for current task: print task
    printk(KERN_INFO "pid: %d | process name: %s | state: %d\n", \
           task->pid, task->comm, task->state);

    // visit children of current task
    list_for_each(list, &task->children) {
        child = list_entry(list, struct task_struct, sibling);

        // dfs
        dfs(child);
    }

    return;
}
```

Notes:

- Current task uses `&task->children` to point the root of children list

2.4 Final Results

Since the traversal is in the DFS manner, we expect to see processes being printed in the manner of the parent, children, grandchildren, and so on. As shown in `ps -eLf` result, we see that **pid_748** is a child of **pid_713**. We can also see that our kernel module prints **pid_713** first and then **pid_748** right after **pid_713**. This proves that our DFS traversal algorithm is correct.

- partial log from implemeneted kernel

```
[ 4781.074021] pid: 713 | process name: avahi-daemon | state: 1
[ 4781.074022] pid: 748 | process name: avahi-daemon | state: 1
```

- partial log from `ps -eLf` (columns: UID | PID | PPID | ... | TIME | CMD)

```
avahi      713     1   713  0    1 11:44 ?        00:00:00 avahi-daemon: running [i
avahi      748   713   748  0    1 11:44 ?        00:00:00 avahi-daemon: chroot hel
```

Let's take a look at another similar example as shown in the following two figures. From `ps -eLf`, **pid_2176** is a child of **pid_2175**. Our kernel module prints **pid_2175** first and then **pid_2176** right after **pid_2175**. This, again, proves that our DFS traversal algorithm is correct.

- partial log from implemeneted kernel

```
[ 4781.074027] pid: 2175 | process name: VBoxClient | state: 1
[ 4781.074028] pid: 2176 | process name: VBoxClient | state: 1
```

- partial log from `ps -eLf` (columns: UID | PID | PPID | ... | TIME | CMD)

```
root      2175     1  2175  0    1 11:44 ?        00:00:00 VBoxClient --vmsvga
root      2176  2175  2176  0    1 11:44 ?        00:00:00 VBoxClient --vmsvga
```