<Term Project1>

2016311600

Kim Kyung Min

<Implementation>

: My description about the implementation will be classified to 5 topics as the assignment pdf notifies the five outputs. **Duplicated explanations will be omitted**. I will explain about the variables and some useful structure first and then explain about the 5 outputs.

Before implementing, I've enrolled by syskkm.c(located in ${kernel_src_dir}/kernel) C file to syscall_64.tbl(located in ${kernel_src_dir}/arch/x86/entry/syscalls) and the object file to Makefile(located in ${kernel_src_dir}/kernel) following the assignment lecture slide.

1. Variables & parameters

   1) int option, int pid (line 11)

      : These two variables are indicating the parameters that sysps.c passes. In sysps.c source code, we can see that it is invoking the system call by calling syscall(SYSCALL_NUMBER,OPTION,pid). It is stating the number of sysnum, given option and give process ID. I used 'option' to divide 5 outputs and 'pid' to print the children and sibling of process as output 3 and 4.

   2) struct task_struct *task (line 12)

      : This struct pointer is pointing the place which contains the most important information of the given process. Passing it to 'for_each_task' macro, we can earn the task_struct structure space for each process.

   3) char const state_array[10] (line 14)

      : In fact, it is defined in other space of kernel already. I just bring it to my system call. Same applies on number 4.

   4) unsigned int state, char chstate (line 22 and 23)

      : These two are used to represent the state of the process. I bring a function defined in some other kernel space, then get the state of task by referencing the state_array.

   5) int cpu (line 32)

      : It is similar with for_each_process usage. I'd pass the task pointer to the previous macro and now, I pass the integer value. There is a pre-defined macro for_each_possible_cpu in linux. It traverses all possible CPUs and can access various information using other functions. The number of cpu can be simply represented with integer type.

   6) int parent (line 45)

      : This is used to store the parent process ID when finding siblings of given process.

   7) struct task_struct *temp (line 46)

      : It works same with *task mentioned above.

8) struct mm_struct *mm (line 89)

   : Each process has each mm_struct and this structure contains the information of VMA such as address of the corresponding physical memory space. It is used to access those data to print the 5th output.

2. Useful structures

   1) task_sruct

      - Declared in ${kernel_src_dir}/include/linux/sched.h
      - #include <linux/sched.h>
      - This structure contains several useful information in doing our assignment such as the state, pid, uid of given process.

   2) rq

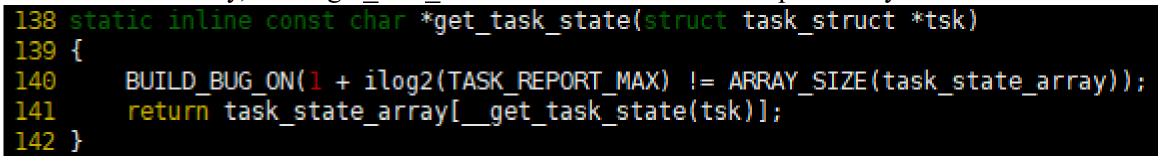
      - Declared in ${kernel_src_dir}/kernel/sched/sched.h
      - #include "./sched/sched.h"
      - Runqueue stores the processes to execute in linked list form and every CPU has its own runqueue.

   3) mm_struct

      - Declared in ${kernel_src_dir}/include/linux/mm_types.h
      - #include <linux/mm_types.h>
      - This structure is used to manage the memory space. Each mm_struct has the vmarea which corresponds to the physical memory. Those vmarea is linked using single linked list and the final pointer points to NULL space.

3. './sysps'

: In the first output, I should print user ID, process ID, PPID and so on. All information is about the process, so I made the code to traverse all processes. A pre-defined macro named 'for_each_process' provides traveling all processes. Before accessing the specific information, I used some trick to represent the task's state. Because I couldn't find the macro which returns the state of task directly, I use 'get_task_state' function defined in 'fs/proc/array.c'.



```
static inline const char *get_task_state(struct task_struct *tsk)
{
    BUILD_BUG_ON(1 + ilog2(TASK_REPORT_MAX) != ARRAY_SIZE(task_state_array));
    return task_state_array[__get_task_state(tsk)];
}
```

There, it defines the state as power of 2 to enable the bitwise operation(It gets the final state with bitwise operations). Here, the character of each state is stored in task_state_array, so I just bring it to my code. By using this function with TASK_REPORT macro, I could print the state of the given task. Next is printing the information of process. I can search the useful variables easily because they are explained well with several comments.

1) task->real_cred->uid: There was uid variable in the task_struct structure field in the past, but it is now moved to cred.h for credibility. Now, I should first access the credential field of task and then access to uid structure to earn the uid value of process.

2) task->pid: This pid value is declared as pid_t pid and returns the process id in an integer type.

3) task->real_parent->pid: Earning PPID is similar with earning PID. Access the parent task_struct first and get the pid of process. 'real_parent' can be found in task_struct field. The comment kindly explained that we should access to real_parent first, unlike the past when we can just access as task->father.

4) task_nice(): There is no exact variable that specifies 'it is the nice value of process!'. I used another function named 'task_nice' to get the nice value of process.

5) task_cpu(task): First, I thought I could get the cpu# by using 'int on_cpu' declared on task_struct. However, all values were zero, so I looked up for another way. In the task_struct structure field, there is a function task_cpu. By tracking the function definition in depth, I could find that it returns the cpu number where the given process is allocated.

4. './sysps -C'

: Second output is about the CPU information. Values needed here locate in 'runqueue' structure, so I will explain about the related function and macros.

1) cpu_rq(cpu)->nr_switches & cpu_rq(cpu)->nr_runnings

: 'rq' structure contains the variable nr_switches and nr_running. To reference those variables, I first should access the runqueue structure with given CPU number. In 'kernel/sched/sched.h' field, there is a defined function which passes the cpu num as parameter and get the rq of given cpu. After accessing the runqueue structure, I just simply print out the nr_something value.

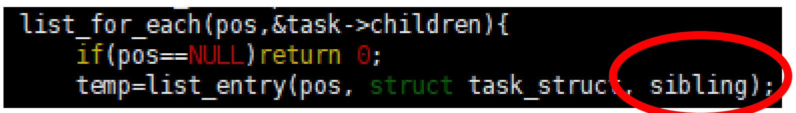
5. './sysps -s -p'

: This is the part where I most like. I can feel the sense of linux developers and beauty of nice data structure. In task_struct field, there is a struct list_head sibling and the comment says that it is the linkage of siblings. With double and circular linked list, I can traverse the siblings of given process. I first found the task of given pid with for_each_process macro and then use the list_for_each macro to access the siblings' task_struct structure. The list_entry function indicates that I'm storing the pointer of task_struct to temp variable. I inserted the if statement in order to catch the illegal accessing issue. Get the state of that task and print out the information just like the way I did above.

6. './sysps -c -p'

: All thing except the circled part below is same with printing the siblings of given pid.

When getting the siblings' information, I should access the siblings of the given task's



```
list_for_each(pos,&task->children){
    if(pos==NULL)return 0;
    temp=list_entry(pos, struct task_struct, sibling);
```

sibling(the same hierarchy in process tree). Here, I should first get the children list of given process and then access the siblings of children list. That is, ‘Original task-> children list -> sibling of the children list of original task’.

7. ‘./sysps -v’

: I concerned much because there were two ways to get the VMA information of the task. By the assignment pdf, accessing the given task’s mm_struct, access the VMA with pointer *mmap, then traverse VMAs to get the address in physical memory method was introduced. However, I found that in mm_struct, there already exist many variables that contain the address of stack, heap, text and data region in physical memory. Just accessing those variables leads me to print out the information of VMA.

1) start_code, end_code: Start and end address of text space.
2) start_data, end_data: Start and end address of data space.
3) start_brk, brk: Start and end address of heap space.
4) start_stack: Start address of stack space.

<ftrace>

: ftrace module enables users to trace the system call path easily. In this part, I have to print out the path of functions which involved in creating processes. Before that, I’ll briefly explain about ‘how to use ftrace’ first.

1. Move to the directory /sys/kernel/debug/tracing/ as a root user.
2. You can designate the filters of which function you would like to trace.

   “echo [plugin] >> set_ftrace_filter”

   If you want to see the filters you can use, just use cat command with this file.
3. You can also designate the method of tracing by typing

   “echo [plugin] >> current_tracer”

   Many options are provided such as printing the function graph or scheduling order, etc.
4. Type

   “echo 1 >> trace”

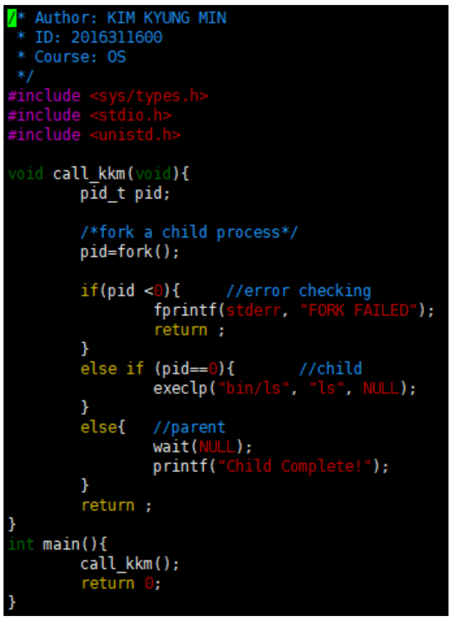
   to activate the tracing. You can replace 1 to 0 to deactivate the tracing.
5. If you want to print out the tracing result, type

   “cat trace”.

   You can add the –head option to specify the line numbers you want to see.

   Also, you can erase the result of tracing by “echo > trace”.
6. To sum up, activate the tracing, run your program, deactivate tracing and print out the result.

➔ To check the functions related in process creating, I made some simple C code which involves fork() system call.



```c
/* Author: KIM KYUNG MIN
 * ID: 2016311600
 * Course: OS
 */
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

void call_kkm(void){
        pid_t pid;

        /*fork a child process*/
        pid=fork();

        if(pid <0){     //error checking
                fprintf(stderr, "FORK FAILED");
                return ;
        }
        else if (pid==0){       //child
                execlp("bin/ls", "ls", NULL);
        }
        else{   //parent
                wait(NULL);
                printf("Child Complete!");
        }
        return ;
}
int main(){
        call_kkm();
        return 0;
}
```

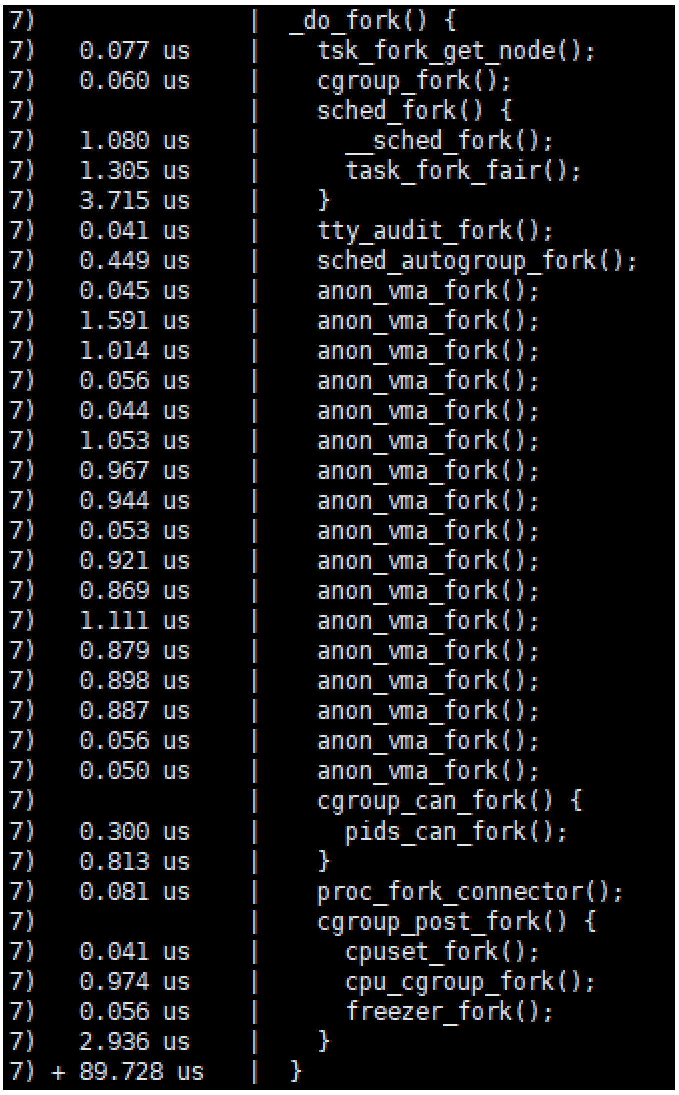
To trace the fork() call path, I found the related function filters in available_filter_functions. In order to insert options related to fork() call, I type

"echo *fork* >> set_ftrace_filter".

Many functions such as _do_fork, sys_fork, etc. are inserted. Also, I type

"echo function_graph >> current_tracer"

to trace the call path relations. I activate the tracing, run my program and print out the result. The result is as below.



```
7)               |  _do_fork() {
7)   0.077 us    |    tsk_fork_get_node();
7)   0.060 us    |    cgroup_fork();
7)               |    sched_fork() {
7)   1.080 us    |      __sched_fork();
7)   1.305 us    |      task_fork_fair();
7)   3.715 us    |    }
7)   0.041 us    |    tty_audit_fork();
7)   0.449 us    |    sched_autogroup_fork();
7)   0.045 us    |    anon_vma_fork();
7)   1.591 us    |    anon_vma_fork();
7)   1.014 us    |    anon_vma_fork();
7)   0.056 us    |    anon_vma_fork();
7)   0.044 us    |    anon_vma_fork();
7)   1.053 us    |    anon_vma_fork();
7)   0.967 us    |    anon_vma_fork();
7)   0.944 us    |    anon_vma_fork();
7)   0.053 us    |    anon_vma_fork();
7)   0.921 us    |    anon_vma_fork();
7)   0.869 us    |    anon_vma_fork();
7)   1.111 us    |    anon_vma_fork();
7)   0.879 us    |    anon_vma_fork();
7)   0.898 us    |    anon_vma_fork();
7)   0.887 us    |    anon_vma_fork();
7)   0.056 us    |    anon_vma_fork();
7)   0.050 us    |    anon_vma_fork();
7)               |    cgroup_can_fork() {
7)   0.300 us    |      pids_can_fork();
7)   0.813 us    |    }
7)   0.081 us    |    proc_fork_connector();
7)               |    cgroup_post_fork() {
7)   0.041 us    |      cpuset_fork();
7)   0.974 us    |      cpu_cgroup_fork();
7)   0.056 us    |      freezer_fork();
7)   2.936 us    |    }
7) + 89.728 us   |  }
```

1. _do_fork()

: It is directly related to fork() call in my program. It call other necessary functions successively.

2. tsk_fork_get_node()

: Defined in "kermel/kthread.c" and takes task_struct pointer as a prarmeter. It returns the node value.

3. cgroup_fork()

: Defined in "kernel/cgroup.c" and it also takes the task_struct pointer. I don't know exactly, but it process some jobs related to child's group.

4. sched_fork()

: Defined in "kernel/sched/core.c" and it takes some flag and pointer as parameters. Inside this function, I can see that __sched_fork() function is called and, I can also check in the tracing result. This function marks the process as running in given CPU and do some many other jobs.

5. anon_vma_fork()

: Defined in "mm/ramp.c" and takes vm_area_struct pointer as parameters. I can guess from comments that it allocates some new or existing VMA from parent process.