Assignment 3

Advanced Operating Systems (CSC0016)

National Taiwan Normal University
Gábor Szolnok
2023.10.31.

1) Hardware and Software environment

The setup is the same as in the previous assignment. I use a virtualized environment, but do the compilation and coding on my host machine. The compiled code is shared with the guest machine via a shared folder. In the following description, whenever a command starts with “GUEST$” it implies that the command was initiated on the guest machine.

Since the code base here was significantly longer, I also collected the modified code into a repository: https://github.com/szogabaha/linux_kernel_development/tree/main/process_info

2) Implementation steps

For the implementation I first checked the getpid definition (as per the task description), which is in the kernel/sys.c file in the linux 6.6.0 code. The code used the “current” value as a parameter but this wasn’t defined anywhere. First I thought it was a global variable but as it turned out it is a macro for the function get_current() which returns a pointer to the current task defined as “task_struct”. The task struct definition is located in the include/linux/sched.h file and it contains most of the information that needs to be wrapped in the exercise's struct. **By importing the linux/sched.h header file in the system call’s file, it is possible to use both “current” and the “task_struct” struct.**

The steps for creating the system call are the same as in the previous exercise, the only difference is that the function’s name and assigned id are different. I put the prinfo struct in a separate header file (appendix A). The rest of the implementation can be found in appendix B.

1. cd ~/src/ntnu/advanced_os/1assignment/guest/linux-6.6.0
2. mkdir get_process_info
3. vim get_process_info/process_statistics.c
   // Create the file. The source code can be found in appendix B.
4. vim get_process_info/prinfo.h
   // Create the file. The source code can be found in appendix A.
5. vim get_process_info/Makefile
   // Create the Makefile. The content is one line: “obj-y := process_statistics.o”

6. vim Kbuild

```
                                                    //From the 97th line of the file
obj-$(CONFIG_NET)           += net/
obj-y                       += virt/
obj-y                       += $(ARCH_DRIVERS)
obj-y                       += hello/
obj-y                       += get_process_info/
```

7. vim arch/x86/entry/syscalls/syscall_64.tbl

```
                                                    //From the 416th line of the file
543     x32     io_setup                compat_sys_io_setup
544     x32     io_submit               compat_sys_io_submit
545     x32     execveat                compat_sys_execveat
546     x32     preadv2                 compat_sys_preadv64v2
547     x32     pwritev2                compat_sys_pwritev64v2
548     common  hello                   sys_hello
549     common  get_process_statistics  sys_get_process_statistics
# This is the end of the legacy x32 range.  Numbers 548 and above are
# not special and are not to be used for x32-specific syscalls.
```

An extra step here compared to the previous task was that in this assignment we need to pass a parameter for the system call. Since the parameter is a struct, it needs to be declared so that the file compiles.

8. vim include/linux/syscalls.h

```
                                                    //From the 75th line of the file
struct cachestat_range;
struct cachestat;
struct prinfo;
....
                                                    //From the 1274th line of the file
asmlinkage long sys_hello(void);
asmlinkage long sys_get_process_statistics(struct prinfo *pinto);
#endif
```

Another difference I made here is that I wanted to add a different label to the new version, to see how it is possible. I used the defconfig for building the conf, but then I modified the .conf file (CONFIG_LOCALVERSION=”prinfo_imp”).

9. GUEST$ cd /media/sf_guest/linux-6.6.0
10. GUEST$ make defconfig
11. make -j10
12. GUEST$ make modules_install install
13. GUEST$ //Reboot and choose the new version

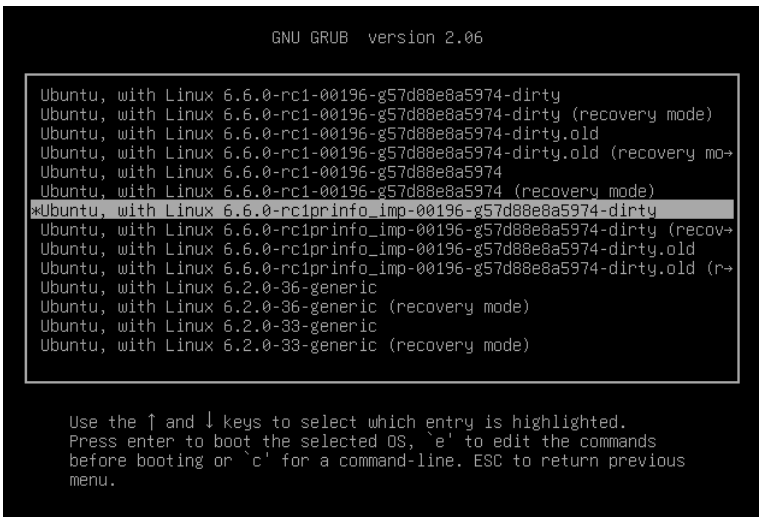

Figure 1: The kernels to choose from after compilation. As can be observed, the modified configuration caused the new version to have the "prinfo_imp" string in the name.

3) Demonstration

14. GUEST$ vim task2.c
    // Create the file, which tries the system call. Implementation is in Appendix C.
15. GUEST$ gcc task2.c
16. ./a.out

To test the separate values, I created a slightly more complicated demonstration code (Appendix C). The code creates a child process, then uses the system call and then waits until the child executes. The child does some calculation with the help of the loop, then calls the system call and then exits, signaling the parent process. The system call is wrapped in the processAndPrintInfo() function, because the result is printed to the terminal. The execution flow can be observed on Figure 2.

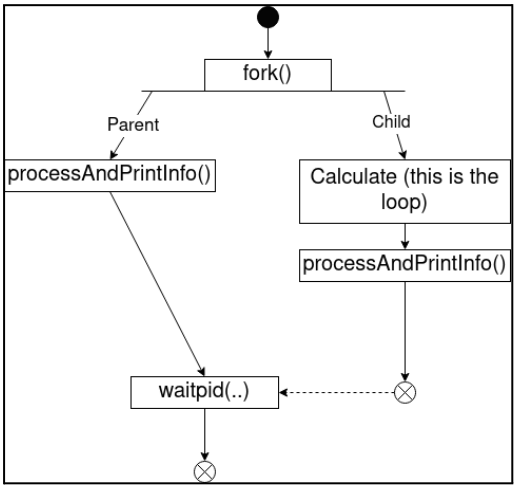

Figure 2: The execution of the demonstration code. It is not a complete execution sequence, the irrelevant steps are ignored and only the ones are kept that help the reader understand what is happening.

The following observations can be made:

- The parent process is the first to invoke the system call (the child does some computation first because of which the syscall invocation is delayed in it). Therefore **in the terminal the first part belongs to the parent, the second belongs to the child.**
- The parent process calls the syscall right after fork, so the stime and utime are 0. Even if we invoked it after the waitpid call, these values would still be 0, since the parent is sleeping while waiting and it doesn't use the CPU.
- The parent's first child PID and the child's PID are the same, as expected.
- The child's parent PID is the parent's PID as expected and **the parent's parent PID is the PID of the shell.** This is different in the two terminals, as expected.

The experiment was made using bash and zsh, two separate shells. The results can be observed on Figure 3.

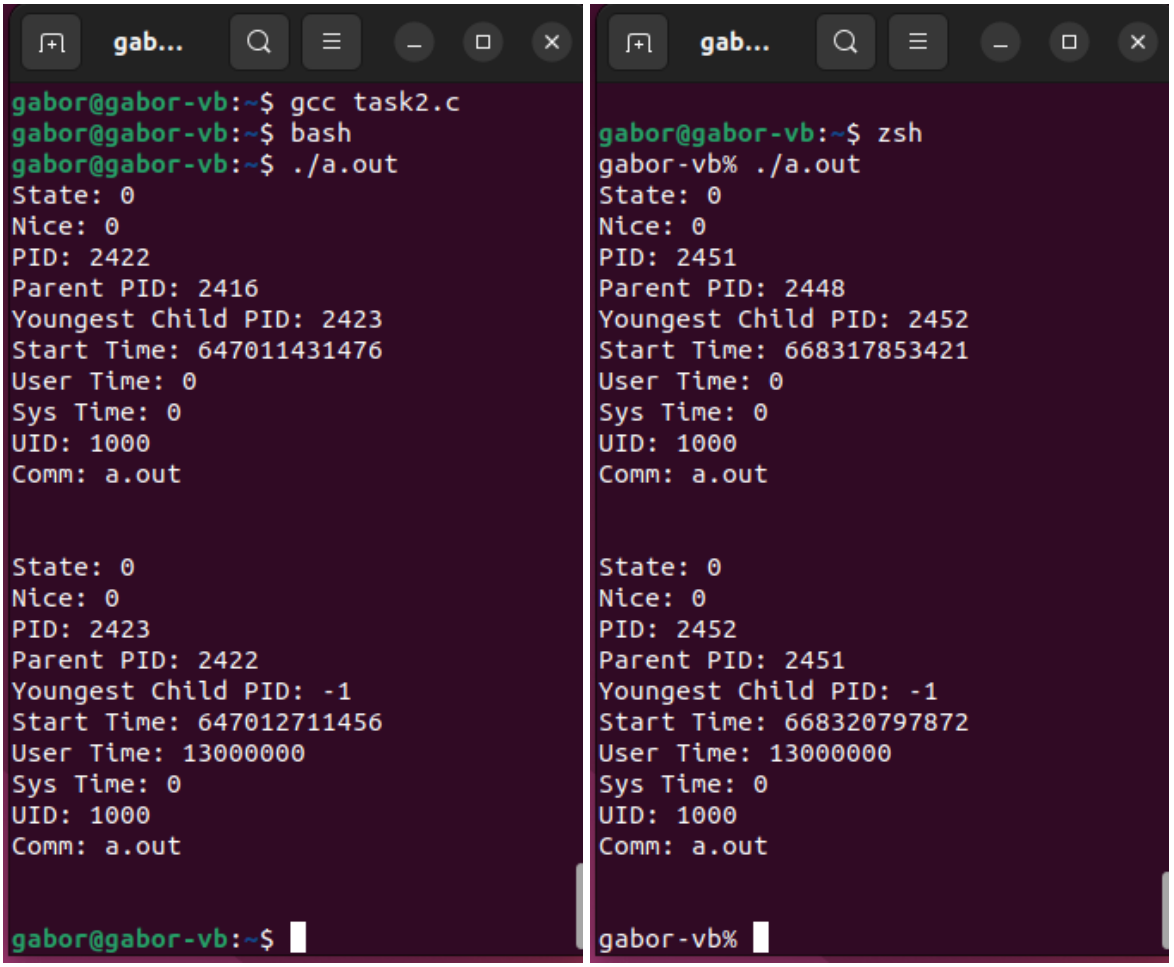

Figure 3: Compiling and running the demonstrating code. The first image shows the results using BASH, the second one shows the same file being executed using zsh.

4) Comments

I had 2 main issues with the exercise. The first one was to test if the system call works. I wanted to see if the utime and stime worked properly, so I put the task to sleep to spend some time waiting. However the received values didn't show that we spent time neither in user time, nor in systime. I spent over an hour debugging until I realized that these values actually report CPU time and if the task sleeps, it doesn't use CPU so these values remain 0. That was when I rewrote the demonstration code to do some computation instead of sleeping, and as it turned out, my core was right all along.

The other problem I had is that I needed to duplicate the prinfo struct definition since it needs to be used both in kernel space and user space and I didn't know how to export it from the kernel's source code for client usage. I spent a lot of time trying it. This is one of the reasons why the prinfo struct is put in a separate header file and encapsulated with macros. I was hoping this header file could have been accessed from outside but it didn't work. On some sites I was discouraged to do it at all, so I ended up duplicating it.

5) Appendix

```
#include <linux/syscalls.h>

#ifndef PRINFO_H
#define PRINFO_H
struct prinfo {
    long state; /* current state of process */
    long nice; /* process nice value */
    pid_t pid; /* process id */
    pid_t parent_pid; /* process id of parent */
    pid_t youngest_child_pid; /* pid of youngest child */
    unsigned long start_time; /* process start time */
    long user_time; /* CPU time spent in user mode */
    long sys_time; /* CPU time spent in system mode */
    long uid; /* user id of process owner */
    char comm[16]; /* name of program executed */
};

#endif  //PRINFO_H
```

Appendix A: The header file that contains the prinfo struct.

```c
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/uaccess.h>
#include <linux/cred.h> //to use current_uid
#include <linux/sched.h> //to get the "current"
#include "prinfo.h" //struct defined by the exercise

SYSCALL_DEFINE1(get_process_statistics, struct prinfo*, info)  {
    struct prinfo process_info;

    // Fill the struct with process information
    process_info.state = current->__state;
    process_info.nice = task_nice(current);
    process_info.pid = current->pid;
    process_info.parent_pid = current->real_parent->pid;

    if (!list_empty(&current->children)) {
        process_info.youngest_child_pid = list_first_entry(&current->children,
struct task_struct, sibling)->pid;
    } else {
        process_info.youngest_child_pid = -1;
    }

    process_info.start_time = current->start_time;
    process_info.user_time = current->utime;
    process_info.sys_time = current->stime;
    process_info.uid = from_kuid(&init_user_ns, current_uid());
    get_task_comm(process_info.comm, current);

    // Copy the struct to user space
    if (copy_to_user(info, &process_info, sizeof(struct prinfo)) != 0) {
        return -EFAULT; // Error handling for copy_to_user failure
    }

    return 0;
}
```

Appendix B: The system call implementation. ALL the values are fetched from the “current” value (most of it directly, some with the help of some utility functions).

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/unistd.h> // For syscall numbers

struct prinfo {
    ... //SAME AS IN THE KERNEL LEVEL
};

void processAndPrintInfo() {
   struct prinfo process_info;
   long result = syscall(549, &process_info);

   if (result == 0) {
      // Syscall succeeded, print the process information
      printf("State: %ld\n", process_info.state);
      printf("Nice: %ld\n", process_info.nice);
      printf("PID: %d\n", process_info.pid);
      printf("Parent PID: %d\n", process_info.parent_pid);
      printf("Youngest Child PID: %d\n", process_info.youngest_child_pid);
      printf("Start Time: %lu\n", process_info.start_time);
      printf("User Time: %ld\n", process_info.user_time);
      printf("Sys Time: %ld\n", process_info.sys_time);
      printf("UID: %ld\n", process_info.uid);
      printf("Comm: %s\n", process_info.comm);
   } else {
      // Syscall failed, print an error message
      perror("Syscall failed");
   }
   printf("\n\n");
}

int main() {
   pid_t pid = fork();

   if (pid < 0) {
      perror("Fork failed");
      exit(EXIT_FAILURE);
   } else if (pid == 0) { // Child process
      int k = 0;
      for (int i = 0; i < 100000000; i++) {
         k = i / 2;
      }
      processAndPrintInfo();
      exit(EXIT_SUCCESS);
   } else {  // Parent process
      processAndPrintInfo();
      int status;
      waitpid(pid, &status, 0);
   }
   return 0;
}
```

Appendix C: The demonstration code.

6) References

[1] Blog post: “utime and stime returned values are still 0 after some time running the process?” Last accessed: 2023.10.31. Link: “https://stackoverflow.com/questions/46325693/utime-and-stime-returned-values-are-still-0-after-some-time-running-the-process”