Umeå University
Department of Computing Science

Operating Systems 7.5 p
5DV171

Assignment 1

Submitted
2017-09-23

Author:
Niklas Königsson (`dv15nkn@cs.umu.se`)
Niclas Nyström (`c14nnm@cs.umu.se`)
Lorenz Gerber (`dv15lgr@cs.umu.se`)

Instructor:
Jan Erik Moström / Adam Dahlgren Lindström

1 Introduction

This assignment was about implementing a new system call in the Linux kernel. The system call should gather information about number of processes, observed file descriptors and pending signals for the current user. The information shall be obtained in kernel space and then sent back to user space.

2 Setup and Preparation

A recent version of raspbian was installed on the SD card. Here we used 'Raspian Stretch Lite', Version September 2017, release 2017-09-07 with initial Kernel version 4.9. For performance and convenience, we cross-compile the kernel from a Linux machine. Hence the tool-chain had to be set up for cross compilation. The tool-chain was obtained from `git clone https://github.com/raspberrypi/tools`. The Linux kernel source repository for Raspberry Pi Raspian was forked form `https://github.com/raspberrypi/linux`. For convenience, scripts for clean up, configuration, build and deployment were setup (`https://github.com/lorenzgerber/kernel_build_scripts.git`).

3 Implementation Syscall

3.1 Summary of Implementation

All data needed to implement the required syscall is available in the `struct task_struct` that is defined in the `sched.h` file. We did a step wise implementation where we first wrote a simple 'Helloworld' syscall that would print to the kernel message buffer. In a second iteration, we implemented the data gathering which will be described in more detail below. Finally, we coded the data transfer between user and kernel space. During all steps, we tried to use test and verification procedures to make sure that the obtained data is correct.

3.2 Used Data structures

To get an idea of the involved data structures, initially chapter 3 of [1] was consulted and compared with the data structures found in the code. Using Eclipse as IDE proved to be of great use for getting pop-up descriptions of macros and functions. Further, it greatly simplified navigating the code by the possibility to link from implementations to definitions etc.

As mentioned earlier, the main data container to be used was the `struct task_struct` that was obtained as a double linked list and could be traversed to calculate the number of processes with the current uid.

Within the iteration loop, the number of watched file descriptors was also obtained from the `files` struct that contained another struct `fdtab` with the field 'open'.

The pending signals were extracted by bit shifting from the 'personal' and 'shared' pending signal data structures (`signal->shared_pending.signal.sig`, `pending.signal.sig`). A good description of the signal data structures was found in [2].

3.3 Files changed and added

Below follows a list of files that were modified. The code is available on `https://github.com/lorenzgerber/linux_kernel/tree/submission_1`.

1. *kernel/Makefile*, the new source file was included
2. *include/uapi/asm-generic/unistd.h*, a define in the arch generic unistd header
3. *arch/arm/include/uapi/unistd.h*, a define in the arch specific unistd header
4. *arch/arm/kernel/calls.S*, the call was added in arch specific the syscall table
5. *include/linux/syscalls.h*, a prototype was added in the syscalls header file
6. *init/Kconfig*, a reference to the configuration file was added

The new files added were:

1. *kernel/Kconfig.processInfo*, the configuration file
2. *kernel/processInfo.c*, the source file
3. *include/linux/processInfo.h*, the headers

3.4 Syscall Return values and error codes

All return values were verified with separate methods. Number of processes was compared to using `ps -u uid`. Starting and shutting down processes was tested to check that the syscall's return value represented the changes. Open file descriptors were verified using `lsof -u uid | wc -l`. Also here it was checked during run-time that opening/closing files was represented in the result values. Finally, for verifying the pending signals, two small c programs were written, where one was only looping over a sleep(5) instruction for a number of times **??**. The other program was a wrapper that would start a program with blocked `SIGINT` signal handler. Invoking the looper with the wrapper, then pressing `Ctrl-x` resulted in pending signals, which was first checked from the console using `cat /proc/pid/status`. Then the corresponding procedure was run while invoking the syscall to verify the correct values.

3.5 Data transfer Kernel to User Space

Here we only used data transport from kernel to user space. This was implemented using the `put_user` functions. The variables in user space were provided to kernel space as arguments in the syscall. Hence our syscall was implemented using the `SYSCALL_DEFINE3` macro.

3.6 User space code

The user space stub does nothing else than setting up the variables to receive data from kernel space, invoke the system call and finally print the obtained information to stdout **??**.

References

[1] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel.* Oreilly & Associates Inc, 3rd edition, 2005.

[2] Robert Love. *Linux Kernel Development.* Addison-Wesley Professional, 3rd edition, 2010.

```
/*
 * looper.c
 */
#include <stdio.h>
#include <unistd.h>

int main(void){
  for (int i = 1; i < 10; i++){
    sleep(5);
  }
  printf("finished \n");
  return 0;
}


/*
 * wrapper.c
 */
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  sigset_t sigs;

  sigemptyset(&sigs);
  sigaddset(&sigs, SIGINT);
  sigprocmask(SIG_BLOCK, &sigs, 0);

  if (argc > 1) {
    execvp(argv[1], argv + 1);
    perror("execv");
  } else {
    fprintf(stderr, "Usage: %s <command> [args...]\n", argv[0]);
  }
  return 1;
}
```

Figure 1: *Code listing of looper.c and wrapper.c programs that were used to verify the pending signals. looper.c was started as argument of wrapper.c. wrapper.c blocks SIGINT, hence invocation of Ctrl-x allowed then to verify in the kernel for pending SIGINT signals during run-time.*

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>

long processinfo_sys(int *procs, int* fds, int* sigpends){
return syscall(398);
}

int main (int argc, char *argv[]){

int procs = 0;
int fds = 0;
int sigpends = 0;

long int a = processinfo_sys(&procs, &fds, &sigpends);

printf("Processes: %d\n", procs);
printf("Watched fds: %d\n", fds);
printf("Pending Signals: %d\n", sigpends);
printf("System call returned %ld\n", a);
return 0;
}
```

Figure 2: *Code listing processInfo.c, the user space stub for invoking the system call. After setting up the variables where the data can be received, the program invokes the syscall by its number and finally prints out the obtained data to stdout.*