HO CHI MINH UNIVERSITY OF TECHNOLOGY
COMPUTER SCIENCE AND ENGINEERING

REPORT: OPERATING SYSTEMS

CHAPTER 6:

SYNCHRONIZATION TOOLS

Teacher:	Nguyen Hoai Nam	
Student:	Nguyen Tan Dat	1810700

Ho Chi Minh city, 04/2020

Contents

1 Compile Linux Kernel

1.1 Preparation

- At very first step, I create a virtual machine by Virtual Box and boot Ubuntu 18.04 in this machine.
- Install build package to get all necessary package for build a new kernel.
- Create /kernelbuild work space for build kernel package. Then download and unzip kernel version 5.6.8 in here.

QUESTION: Why we needs to install kernel-package?

Answer: We need to install kernel-package because kernel-package is a tool for building a kernel. This package will do some process automatically, some that allow you to custom your own kernel that fit with your real or virtual machine.

QUESTION: Why we have to use another kernel source from the server such as http://www.kernel.org, can we compile the original kernel (the local kernel on the running OS) directly?

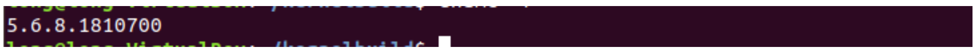
Answer: Yes, we can compile the original kernel directly. But we will work and change something in the kernel (add new system call in this assignment), so we have to re-compile a new kernel. Therefor, we have to use another source in kernel.org to get more benefits: optimize kernel by trim unnecessary process, handling special requirements of hardware...

1.2 Configuration

- Follow the guild line in assignment description.
- Add .1810700 in the version of kernel by the GUI.

1.3 Build the Configured Kernel

- Follow the guild line of assignment description. After reboot, version has change to 5.6.8.1810700 as below:



```
5.6.8.1810700
```

Hình 1: Kernel version after build successfully

QUESTION: What is the meaning of these two stages, namely "make" and "make modules"? What are created and what for?

Answer:

- **make**: compile and link the kernel package. The file vmlinuz is created from that stages.
- **make modules**: compiles individual files for each question you answered [M] during kernel config. The object code is linked against your freshly built kernel. (For questions answered [*], these are already part of vmlinuz, and for questions answered [ ] they are skipped).

1.4 Installing the new kernel

2 Trim The Kernel

Spending a long time for waiting for compiling in the first time make me feel worried about the deadline. Then I find that Trim is solution to decrease time-consuming of build kernel. I used **make localmodconfig** to get a auto config process for my virtual machine, and I try to trim manually some more process that are not necessary for my custom kernel. After all, I reduced the time to under 10 minutes per compile stage.

3 System call

3.1 The Role of The System Call

- System call is a service of kernel that allow process requests its execution.
- This may relate with hardware or affect to hardware process, creation and execution of new processes, and communication with integral kernel services such as process scheduling.

3.2 Prototype

- Read requirements of the assignment and try to understand what the defined function and struct are use for.

3.3 Implementation

- Try to add syscall that will print *Hello DAT* to screen if build successfully.
- There were some problems:
  - We have to add a definition of a syscall by SYSCALL_DEFINE2 which is a macro of linux/syscalls.h.

```
SYSCALL_DEFINE2 (sys_get_proc_info, pid_t, pid, struct procinfos*, info)
     {
  return sys_get_proc_info(pid, info);
}

```

    * The SYSCALL_DEFINE0 macro is defined in include/linux/syscall.h and is used by other tools to get metadata about the system call. It means that we are defining a new system call.
    * 2 means this sycall get 2 arguments.
    * The first parameter passed to the macro is the system call name. Here, we named it sys_get_proc_info.
  - Add syscall to syscall_x64.tbl as below:

```
438 common  pidfd_getfd    __x64_sys_pidfd_getfd
439 64   sys_get_proc_info __x64_sys_sys_get_proc_info

#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation. The __x32_compat_sys stubs are created
# on-the-fly for compat_sys_*() compatibility system calls if X86_X32
# is defined.
#
512 x32 rt_sigaction    __x32_compat_sys_rt_sigaction

```

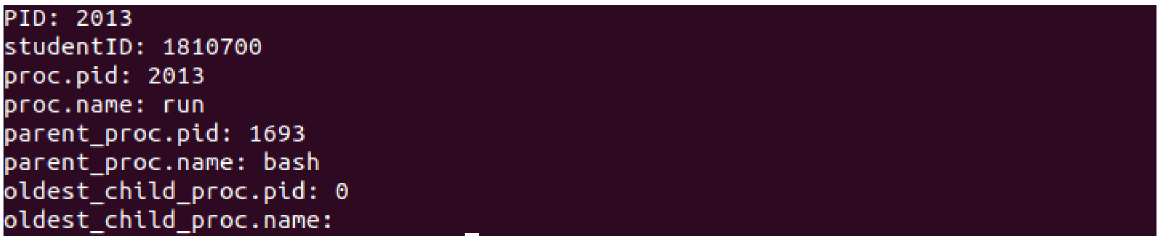
    * Change number identity to 439. It is just an unused number. But to be continuous, we choose 439 to add this syscall before the first x32 syscall.
    * Rename syscall to sys_get_proc_info.
    * Add __64_sys_ before name to get entry point of this syscall because of the affect of SYSCALL_DEFINED from the definition of the syscall.
  - Add definitions of structs and function that are implemented in sys_get_proc_info.c to the end syscalls.h (before the last #endif)

- When the syscall print ***Hello DAT*** successfully, I started to implement the required syscall. First problem was how to get process information by pid number.
- Discover the **sched.h** and find out data structure **task_struct**
- Research the usage of data structure **task_struct**.
- Implement syscall as below:

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

struct proc_info {
  pid_t pid;
  char name[16];
};

struct procinfos {
  long studentID;
  struct proc_info proc;
  struct proc_info parent_proc;
  struct proc_info oldest_child_proc;
};


asmlinkage long  sys_get_proc_info (pid_t pid, struct procinfos * info) {
  printk(KERN_INFO "Student ID: 1810700\n");
  info->studentID = 1810700;
  if (pid == -1) {
    pid = current->pid;
  }
  struct task_struct *process;
  for_each_process(process) {
    if (process->pid == pid) {
      info->proc.pid = process->pid;
      strcpy(info->proc.name, process->comm);
      if (process->real_parent == NULL) {
        info->parent_proc.pid = 0;
        strcpy(info->parent_proc.name, "\0");
      }
      else {
        info->parent_proc.pid = process->real_parent->pid;
        strcpy(info->parent_proc.name, process->real_parent->comm);
      }
      struct task_struct * cProc;
      cProc = list_first_entry_or_null(&process->children, struct task_struct
    , sibling);
      if (cProc == NULL) {
        info->oldest_child_proc.pid = 0;
        strcpy(info->oldest_child_proc.name, "\0");
      }
      else {
        info->oldest_child_proc.pid = cProc->pid;
        strcpy(info->oldest_child_proc.name, cProc->comm);
      }
      return 0;
    }
  }
  return EINVAL;
}



SYSCALL_DEFINE2 (sys_get_proc_info, pid_t, pid, struct procinfos*, info) {
  return sys_get_proc_info(pid, info);
}

```

I will explain how sys_get_proc_info work:

- In the case pid is -1, I assign the current−>pid to pid.
- Use a loop that for_each_process to travel all process to find out which process have the same pid equal to the required pid.
- If there exists a process that satisfy the condition (the same pid as requirement). We will explore more information of this process: parent process and oldest child process.



```
PID: 2013
studentID: 1810700
proc.pid: 2013
proc.name: run
parent_proc.pid: 1693
parent_proc.name: bash
oldest_child_proc.pid: 0
oldest_child_proc.name:
```

Hình 2: Result of new syscall

QUESTION: What is the meaning of fields of the line that just add to the system call table (548, 64, get proc info, i.e.)

Answer: Each system call is defined in a line in a system call table file with parameters:

- <**number**>: This is the unique identity number of a system call in x64 or x32. When system is called (by this number), the system will find the system call upon this number and table of system calls.
- <**abi**>: Application Binary Interface, it is "i386" in x32 system (in syscall_x32.tbl) or "common", "x64", "x32" in x64 system (in syscall_x63.tbl).

- <**name**>: name of this system call.
- <**entry point**>: The entry point of the function.
- <**compat entry point**>: (only appear in syscall_x32.tbl) this is entry point of function when user use x64 system.

QUESTION: What is the meaning of each line below?

```
struct proc_info; // line 1
struct procinfos; // line 2
asmlinkage long sys_get_proc_info(pid_t pid, struct procinfos * info);  // line 3
```

Answer:

- line 1: define a new structure name proc_info
- line 2: define a new structure name procinfos
- line 3: define function sys_get_proc_info()

QUESTION: Why this program could indicate whether our system call works or not?

```
#include <sys/syscall.h>
#include <stdio.h>
#include <unistd.h>
#define SIZE 200
int main(){
    long sys_return_value;
    unsigned long info[SIZE];
    sys_return_value = syscall(439, -1, &info);
    printf("My student ID: %lu\n", info[0]);
    return 0;
}
```

Answer: Because this test function was use try to execute new syscall (with new identity number 439). If this function can work well, it means that our new system call was added successfully and vice versa.

QUESTION: Why we have to redefine procinfos and proc info struct while we have already defined it inside the kernel?

Answer: Because we have just defined it **inside the kernel** which mean that this function only can be use in kernel space. We re-define this function in usepace to use it in userspace.

QUESTION: Why root privilege (e.g. adding sudo before the cp command) is required to copy the header file to /usr/include?

Answer: /usr directory can only be written by root user and sudo is to confirm this command have permit to execute.

QUESTION: Why we must put -shared and -fpic option into gcc command?

- shared: create a shared library
- fpic: create a PIC (Position independent code) to use shared library.

4 References

- How to build and install the latest Linux kernel from source
- kernel-package
- Adding a New System Call
- task_struct definition
- LAb on Hacking kernel