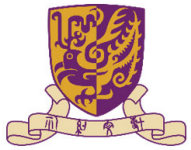


Operating System (CSC 3150)

Tutorial 3

YUANG CHEN

E-MAIL: YUANGCHEN@LINK.CUHK.EDU.CN

Target

In this tutorial, we will create process to execute user program through kernel mode.
(Referenced version: 4.10.14)

- Problem discussion for compiling kernel
- Process creation (_do_fork)
- Program execution (do_execve)
- Wait for signal (do_wait)
- Handle signal (k_sigaction)
- Export symbol

Problem discussion for compiling kernel

- Command “make menuconfig” does not working
  - Use command “sudo apt-get install libncurses5-dev libssl-dev” to install the tool

```
scripts/Makefile.host:124: recipe for target 'scripts/kconfig/mco
nf.o' failed
make[1]: *** [scripts/kconfig/mconf.o] Error 1
Makefile:546: recipe for target 'menuconfig' failed
make: *** [menuconfig] Error 2
root@VM:/usr/src/linux-4.10.14#
```

Problem discussion for compiling kernel

- Fatal error (No such file or directory)
  - Ensure your source package is extracted within Linux, not in Windows or Mac.
  - Filename is Linux is case sensitive, but it is case non-sensitive in Windows or Mac. It may rename some files when extracting them, so that these files cannot be found when compiling.

```
net/ipv4/netfilter/ipt_ECN.c:20:42: fatal error: linux/netfilter_ipv4/ipt_ECN.h:
 No such file or directory
compilation terminated.
scripts/Makefile.build:300: recipe for target 'net/ipv4/netfilter/ipt_ECN.o' fai
led
make[3]: *** [net/ipv4/netfilter/ipt_ECN.o] Error 1
scripts/Makefile.build:553: recipe for target 'net/ipv4/netfilter' failed
make[2]: *** [net/ipv4/netfilter] Error 2
scripts/Makefile.build:553: recipe for target 'net/ipv4' failed
make[1]: *** [net/ipv4] Error 2
Makefile:988: recipe for target 'net' failed
make: *** [net] Error 2
```

Problem discussion for compiling kernel

- Space is not enough
  - Copy the source file from original space(e.g., /usr/src) to extended space(e.g., /home/seed/work)
  - Remove the source file from original space(e.g., /usr/src)
  - In source file (stored in extended space), continue to installation.
    (Do not start from ‘make clean’, which will remove all the previous built, start from previous interrupted step)

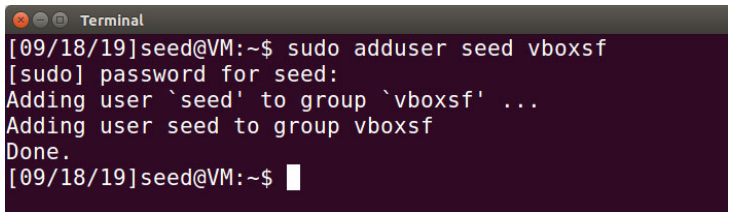
```
depmod: WARNING: /lib/modules/99.98.4.18.9/kernel/sound/isa/sb/snd-emu8000-synt
.ko needs unknown symbol snd_emux_new
depmod: ERROR: Could not create index 'modules.dep'. Output is truncated: No sp
ce left on device
Makefile:1248: recipe for target '_modinst_post' failed
make: *** [_modinst_post] Error 1
```

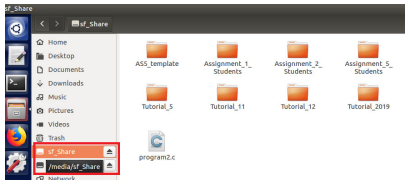
Problem discussion for compiling kernel

- Do not compile kernel source code under shared folder.
  - Copy the source code to extended space.
  - Then start to compile.
  - If you compiling it under shared folder, some files might be created failed, and it would lead to error when executing install step.

Problem discussion for compiling kernel

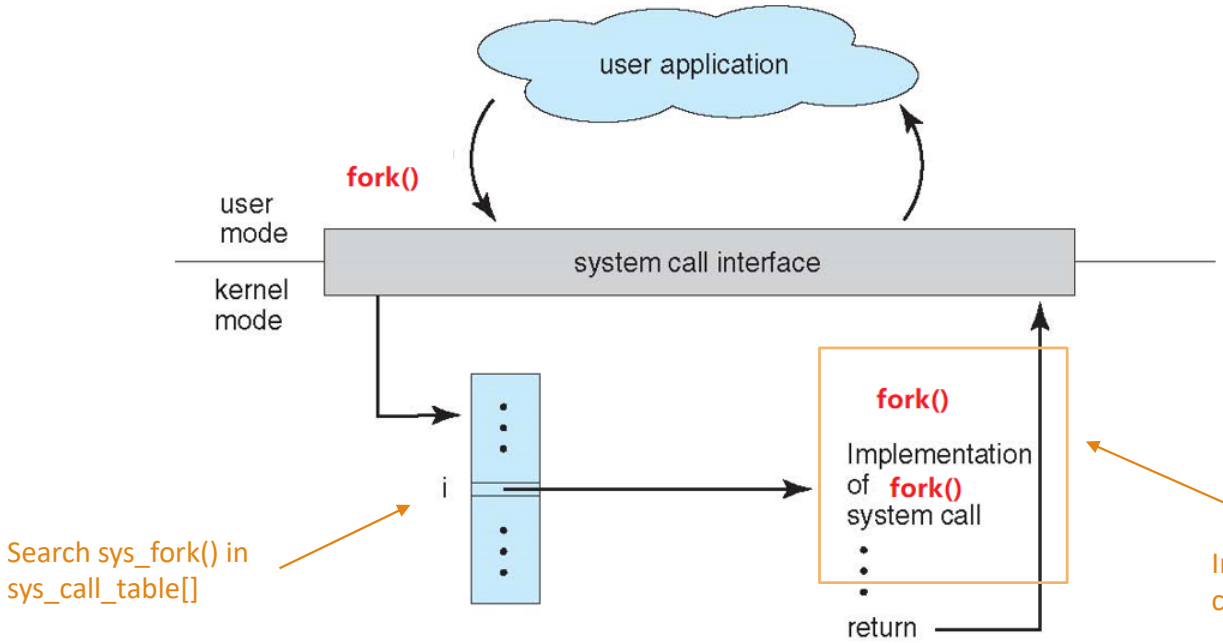
- Share folder mounting does not work
  - After creating a Share folder in Shared folder setting, open terminal and type command 'sudo adduser seed vboxsf'





  - Reboot the machine
  - You will see /media/sf_Share will auto mount to your share folder
  - You could access this share folder via 'cd /media/sf_Share'.

Process creation (_do_fork / do_fork)



Process creation (_do_fork / do_fork)

- In kernel mode, when system call _do_fork() / do_fork() is executed, it is loading fork.ko module.
- Its implementation is defined in '/kernel/fork.c':
  - Call dup_task_struct(), which creates a new kernel stack, thread_info structure, and task_struct for the new process.
  - Call get_pid() to assign an available PID to the new task.
  - Call copy_process() then either duplicates or shares open files, file system information, signal handlers, process address space, and namespace.
- For more details
  - https://elixir.bootlin.com/linux/v4.10.14/source/kernel/fork.c(_do_fork)
  - You should check basing on your own version

Process creation (_do_fork / do_fork)

- _do_fork:
  - long _do_fork (unsigned long clone_flags,
    unsigned long stack_start,
    unsigned long stack_size,
    int __user *parent_tidptr,
    int __user *child_tidptr,
    unsigned long tls);
- Arguments:
  - clone_flags: How to clone the child process. When executing fork(), it is set as SIGCHILD.
  - stack_start: Specifies the location of the stack used by the child process.
  - stack_size: Normally set as 0 because it is unused.
  - parent_tidptr: Used for clone() to point to user space memory in parent process address space. It is set as NULL when executing fork();
  - child_tidptr: Used for clone() to point to user space memory in child process address space. It is set as NULL when executing fork();
  - tls: Set thread local storage.

Process creation (_do_fork / do_fork)

- Return value:
  - Fork successfully: pid of child process
  - Failed: Err

Specify function's address as child process' stack pointer.

```
//fork process
pid=_do_fork(SIGCHLD,(unsigned long)&my_exec,0,NULL,NULL,0);


printk("[Do_Fork] : The child process has pid = %ld\n",pid);
printk("[Do_Fork] : This is the parent process,pid = %d\n",(int)current->pid);
```

```
[  334.769352] [Do_Fork] : The child process has pid = 2780
[  334.769353] [Do_Fork] : This is the parent process,pid = 2778
```

Program execution (do_execve)

- In kernel mode, when system call do_execve() is executed, it is loading exec.ko module.
- Its implementation is defined in '/fs/exec.c':
  - struct linux_binprm: This structure is used to hold the arguments that are used when loading binaries. It is defined in 'include/linux/binfmts.h'
  - Use prepare_bprm_creds API to initialize the struct
  - Open your program file and assign corresponding values into struct linux_binprm
  - Use the bprm_mm_init and prepare_binprm to prepare execute your binary program
  - Read the file content and copy corresponding information into your struct linux_binprm
  - Use search_binary_handler to execute your binary program
  - After execve succeeded, it calls acct_update_integrals and free the binary program by calling free_brpm
- For more details:
  - https://elixir.bootlin.com/linux/v4.10.14/source/include/linux/binfmts.h(binprm)
  - https://elixir.bootlin.com/linux/v4.10.14/source/fs/exec.c(do_execve)

Program execution (do_execve)

- do_execve:
  - int do_execve (struct filename *filename,
    const char __user *const __user *__argv,
    const char __user *const __user *__envp);
- Parameters:
  - filename: Filename of the executable file.
  - argv: The arguments for executing the file.
  - envp: System environment variable set.

Program execution (do_execve)

- Return value:
  - Execute successfully: 0
  - Failed: Err

```
//implement exec function
int my_exec(void){
        int result;
        const char path[]="/home/seed/Documents/Tutorial_2019/Tutorial_3/Do_Fork/test";
        const char *const argv[]={path,NULL,NULL};
        const char *const envp[]={"HOME=/","PATH=/sbin:/user/sbin:/bin:/usr/bin",NULL};

        struct filename * my_filename = getname(path);


        result=do_execve(my_filename,argv,envp);

        //if exec success
        if(!result)
                return 0;

        //if exec failed
        do_exit(result);
}
```

Path of executed file

Get filename from path

Wait for signal (do_wait)

- In kernel mode, when system call do_wait() is executed, it is loading exit.ko module.
- Its implementation is defined in '/kernel/exit.c':
  - Create wait_opts structure and add it into wait queue by calling add_wait_queue.
  - Use set_current_state to update state as TASK_INTERRUPTIBLE or TASK_RUNNING.
  - When child process terminates, it calls wake_up_parent. Then parent process will go to repeat scanning, so that it can get return of child process' termination.
- For more details:
  - https://elixir.bootlin.com/linux/v4.10.10/source/kernel/exit.c(do_wait)

Wait for signal (do_wait)

- do_wait:
  - long do_wait (struct wait_opts *wo);
- struct wait_opts:
  - struct wait_opts { enum pid_type wo_type; //It is defined in '/include/linux/pid.h'.
    int wo_flags; //Wait options. (0, WNOHANG, WEXITED, etc.)
    struct pid *wo_pid; //Kernel's internal notion of a process identifier. "Find_get_pid()"
    struct siginfo __user *wo_info; //Singal information.
    int __user *wo_stat; // Child process's termination status
    struct rusage __user *wo_rusage; //Resource usage
    wait_queue_t child_wait; //Task wait queue
    int notask_error ;};

Wait for signal (do_wait)

```
//implement wait function
void my_wait(pid_t pid){

        int status;
        struct wait_opts wo;
        struct pid *wo_pid=NULL;
        enum pid_type type;
        type=PIDTYPE_PID;
        wo_pid=find_get_pid(pid);

        wo.wo_type=type;
        wo.wo_pid=wo_pid;
        wo.wo_flags=WEXITED;
        wo.wo_info=NULL;
        wo.wo_stat=(int __user*)&status;
        wo.wo_rusage=NULL;


        int a;
        a=do_wait(&wo);
        printk("do_wait return value is %d\n",&a);


        // output child process exit status
        printk("[Do_Fork] : The return signal is %d\n",*wo.wo_stat);

        put_pid(wo_pid);

        return;
}
```

Look up a PID from hash table and return with it's count evaluated.

Decrease the count and free memory

Handle signal (k_sigaction)

- For each process in the system, the kernel must keep track of what signals are currently pending or masked. The kernel must keep track of how every thread group is supposed to handle every signal.
- To do this, the kernel uses several data structures accessible from the process descriptor. The most significant data structures related to signal handling：

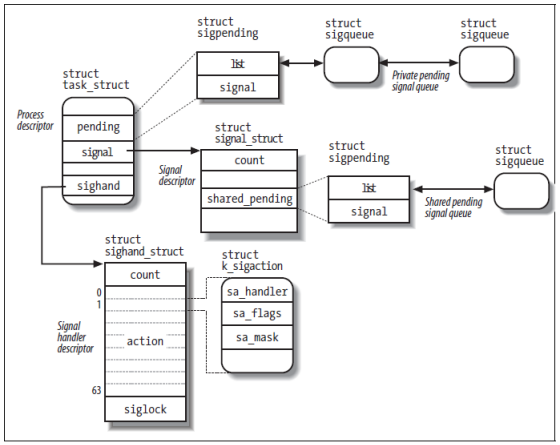


Figure 11-1. The most significant data structures related to signal handling

Handle signal (k_sigaction)

- The signal field of the process descriptor points to a *signal descriptor*, a signal_struct structure that keeps track of the shared pending signals.
- The of properties of a signal are stored in a k_sigaction structure, which contains the signal properties.
- The kernel noticed the arrival of a signal and invoked function to prepare the process descriptor of the process that is supposed to receive the signal. (do_signal())
- If a handler has been extablished for the signal, the do_signal() function must enforce its execution. (handle_signal())

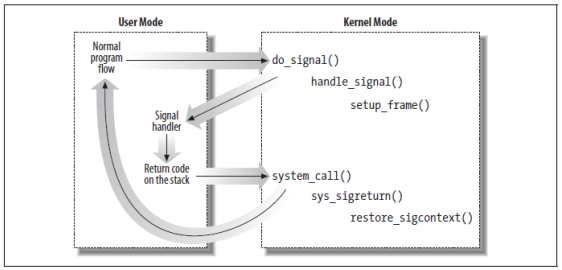


Figure 11-2. Catching a signal

Export Symbol

- EXPORT_SYMBOL() provides API to be used in other module.
- If the function in one module is non-static, we could use EXPORT_SYMBOL() to export it. In that case, this function could be used another module.
- Before using this symbol in another kernel module, should use 'extern' to clarify it.

```
static inline void put_signal_struct(struct signal_struct *sig)
{
        if (atomic_dec_and_test(&sig->sigcnt))
                free_signal_struct(sig);
}

void __put_task_struct(struct task_struct *tsk)
{
        WARN_ON(!tsk->exit_state);
        WARN_ON(atomic_read(&tsk->usage));
        WARN_ON(tsk == current);

        cgroup_free(tsk);
        task_numa_free(tsk);
        security_task_free(tsk);
        exit_creds(tsk);
        delayacct_tsk_free(tsk);
        put_signal_struct(tsk->signal);

        if (!profile_handoff_task(tsk))
                free_task(tsk);
}
EXPORT_SYMBOL_GPL(__put_task_struct);
```

```
extern void __put_task_struct(struct task_struct *t);

static inline void put_task_struct(struct task_struct *t)
{
        if (atomic_dec_and_test(&t->usage))
                __put_task_struct(t);
}
```

Export Symbol

- If you modify the kernel source code, you should rebuilt the kernel module and install the updated kernel. Then it takes effect.

- To save your time, when you rebuilt the kernel, start from ‘make bzImage’. It will only rebuild the updated modules.
  (Do not start from ‘make clean’, which will clean all previous built, and it takes hours to rebuild the kernel modules)

- Once the symbol is exported and re-built the kernel module, you will find it’s defined in “Module.symdvers” (auto generated under source code after run “make bzImage”)

```
14887 0x0dd599df      __wait_on_bit_lock      vmlinux EXPORT_SYMBOL
14888 0xeac5b58b      _do_fork        vmlinux EXPORT_SYMBOL
14889 0xb3253ed9      hpet_rtc_timer_init     vmlinux EXPORT_SYMBOL_GPL
```

Recompile your kernel

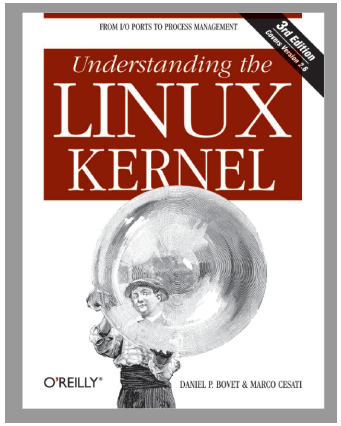
- If you've already compiled and installed the new kernel, then modify the source code and need to make the changes effect. Please start compile from command 'make bzImage'. (Follow compile steps from Tutorial 2)

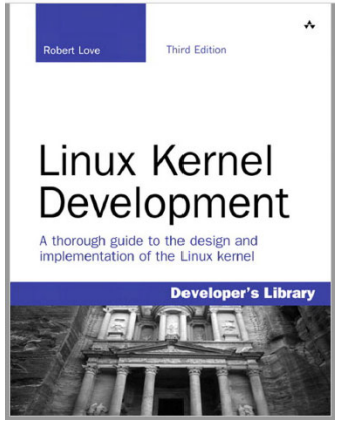
- To save your time, do not start from 'make clean' again, which will remove all your previous built.

- Starting from 'make bzImage', it will only build the changes.

References

- Understanding the Linux KERNEL (3rd edition)
  - do_fork() (Page 117 – 130)
  - sigaction (Page 420 – 455)
- Linux Kernel Development (3rd edition)
  - Process descriptor and task structure (Page 24 – 27)
  - Current task_struct (Page 29 – 30)





Thank you