SYSTEM SOFTWARE
SYSTEM CALL

Nguyen Huu Duc

School of Information and Communication Technology
Hanoi University of Science and Technology

Overview of System Call

- In a computer system, applications are normally not allowed to access system resources; instead, they tell the operating system to help them access neccessary resources.
  - System resources are balanced among applications
  - Protect critical data in the system from unwanted changes made by applications
  - Guarantee that an application only operates inside permitted space
- The operating system provides a set of system calls to satisfy the demands of the application that needs to access resources
  - Cover all types of resources
  - Constructed in a stabale and sophisticated manner

Execution Mode

- Modern processors support many different execution modes, two of which are used by the operating system
    - User mode: used when executing code in an application
        - only allowed to access the application's address space
        - only permited to execute a limited set of instructions
        - be forbidden from direct access to data structures that are under protection by the operating system
    - Kernel mode: used when executing the operating system's instructions (such as system calls)
        - Be allowed to execute certain special instructions
        - Be allowed to access protected address spaces
        - Be allowed to run system programs

System call and library functions

- System calls are performed in kernel mode, but library functions run in user mode (possibly require system calls)
- System calls that are not linked to applications
- Be limited in number (about 250)
- Library functions are rarely activated with call/return, while system calls make use of software interrupt
- For the ease of use, system calls are packed in a wrapper, which will then be used by applications as normal library functions

System calls (1)

- File and I/O operations
  - `open()`, `close()`: Open and close file
  - `creat()`, `unlink()`: Create and delete file
  - `read()`, `write()`: Read and write file
  - `lseek()`: Moving pointer on file
  - `chmod()`: Change file access permissions
  - `mkdir()`, `rmdir()`: Create and delete folders

System calls (2)

- File and I/O operations
  - `stat()`: Read information about file status
  - `access()`: Check file access permissions
  - `chmod()`: Change file access permissions
  - `chown()`: Change file owner
  - `ioctl()`: Control devices

System calls (3)

- Process management
  - `fork()`: Create a new process
  - `exec()`: Run an executable file
    - `execl(fn,arg0[,arg1,...,argn],NULL)`
    - `execle(fn,arg0[,arg1,...,argn],NULL,env)`
    - `execv(fn,argv)`
    - `execve(fn,argv,env)`
  - `exit()`: Terminate a process
  - `wait()`: Wait for child processes to finish

System calls (4)

- Inter-process communication
  - `kill()`, `sigsend()`: Send a signal to a process
  - `pause()`: Suspend a process until receiving a required signal
  - `sigaction()`: Set up procedure to handle signals

System calls (5)

- Inter-process communication
  - `pipe()`: Create a pipeline
  - `socket()`: Create a socket
  - `bind()`: Assign an ID to a socket
  - `connect()`: Initiate a socket
  - `listen()`: Listen for data from a socket
  - `accept()`: Accept connection
  - `send()`: Send data to a socket
  - `recv()`: Receive data from a socket

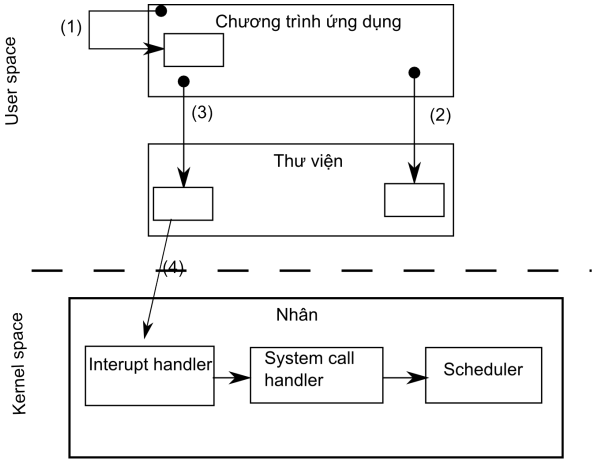
System calls (6)

- Memory management
  - `brk()`, `sbrk()`: Change heap size

Look up system calls

- `man 2`: look up a system call
- `man 3`: look up a library
- `man 4`: look up a device file
- `man 5`: look up file format
- `man 8`: look up command

The process of making system calls



The process of making system calls

1. Applications call routines inside their program
2. Application call functions in the library (without making system calls)
   - Example: `sscanf()`, `memset()`, `tolower()`,...
3. System programs call functions in the library (using system calls)
   - Example: `getchar()`, `printf()`,...
4. Library functions activate system calls

Activate system calls

- Library functions (user mode) activate system calls (kernel mode) through interrupts
  - `0x21`: MS-DOS
  - `0x80`: Linux
  - `0x81`: XEN
- A system call is determined with a number and is passed through register `eax`
- Parameters required by the system call are passed through general registers
  - In case the number of parameter is so large, these parameters are passed through a pointer
  - Pay attention to the parameters
- Variable `errno` stores the error code returned by the system call

Activate system calls

- The interrupt handlers (system_call) choose the system call from a table of system calls (sys_call_table)
  - Store general registers (also parameters of the system call) on kernel stack
  - Choose system call from sys_call_table and make a request for the corresponding handler
  - Return control to the library function through instruction iret

```
system_call:
  SAVE_ALL
  ...
  call *sys_call_table(0, %eax, 4)
  ...
ret_from_sys_call:
  ...
  iret
```

Scheduling

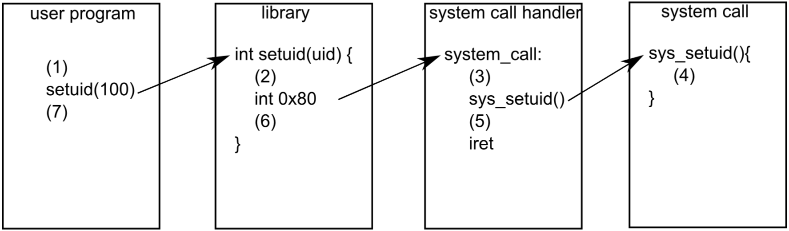
- Some system calls cannot terminate immediately
  - `nanosleep, read, write`
- The process need to be scheduled in order to be able to pause (and transfer system control to another process) and wait for success signals from hardware

Some system calls

```
int setuid (uid_t uid)
```

- Set valid UID for the current process
- Used frequently in server applications such as `httpd, bind,...`
- `uid`: user identifier
- Return 0 if succeed, -1 if fail(`errno=ERPERM`)

setuid



user program

(1)
setuid(100)
(7)

library

```
int setuid(uid) {
    (2)
    int 0x80
    (6)
}
```

system call handler

```
system_call:
    (3)
    sys_setuid()
    (5)
    iret
```

system call

```
sys_setuid(){
    (4)
}
```

setuid

1: An application makes system calls

- Call conventions `cdecl`
- Instruction usage `call`
- Return address on stack
- Transfer control to library functions

setuid

2: A library function prepares to make system call

- Check parameters
- Trigger 0x80 interrupt
    - Switch to kernel mode
    - `SS,ESP, EFLAGS, CS, EIP` is pushed on the kernel stack
    - Assign values `SS,ESP, EFLAGS, CS, EIP` to kernel

setuid

3: Prepare to make system calls

- The idetifying number of the system call
- Store values that are currently on general register (`es, ds, eax, ebp, edi, esi, edx, ecx, ebx`)
- Pass `es, ds` to kernel data space

setuid

```
<- Ret addr
24: * 0 (% esp) -% ebx first argument
25: * 4 (% esp) -% ecx second argument
26: * 8 (% esp) -% edx third argument
27: * C (% esp) -% esi fourth parameter
28: * 10 (% esp) -% edi fifth argument
29: * 14 (% esp) -% ebp
30: * 18 (% esp) -% eax
31: * 1C (% esp) -% ds
32: * 20 (% esp) -% es
33: * 24 (% esp) -% user_eax
34: * 28 (% esp) -% user_eip
35: * 2C (% esp) -% user_cs
36: * 30 (% esp) -% user_eflags
37: * 34 (% esp) -% user_esp
38: * 38 (% esp) -% user_ss
```

setuid

3: Prepare to make system calls

- Check the identifying number of the system calls
- Call `sys_setuid`
  - `call *sys_call_table (0,%eax, 4)`

setuid

4: Execute a system call

- Check parameters
  - EAGAIN: Resources are already in use
  - EPERM: Permission denied
- Execute effective uid

setuid

5: Prepare to return results to the library function
- Put the value of eax to the 24th position (%esp)
- Restore the previous values of general registeres
- Restore the value of eax
- iret
    - Restore `eip, cs, eflags, esp,ss`
    - Switch to user mode

setuid

6: Prepare to return results to the application
   - Check for errors
   - Set up errno
7: The application
   - Receive returned results through eax
   - Check for error (errno)

Steps in constructing a system call

- Prepare tools for re-compiling the kernel of the operating system
- Prepare source code of the kernel of the opersting system
- Change and add code of the new system call
- Recompile the kernel of the operating system
- Test the new system call

Construct a system call (1)

- Source code of the system call
  - `/usr/src/linux/mycall.c`

mycall.c

```
#include<linux/linkage.h>
asmlinkage long sys_mycall(int i)
{
  return i+10;
}
```

Construct a system call (2)

- Add the system call to syscall_table.S
  - /usr/src/linux/arch/i386/kernel/syscall_table.S

mycall.c

```
. . .
.long sys_mycall
```

Construct a system call (3)

- Add the system call to `unistd.h`
  - `/usr/src/linux/include/asm-i386/unistd.h`

unistd.h

```
...
#define __NR_vmsplice 316
#define __NR_mycall   317
...
#define NR_syscalls   318
```

Construct a system call (4)

- Add the system call to `syscalls.h`
  - `/usr/src/linux/include/linux/syscalls.h`

syscalls.h

```
...
asmlinkage long sys_mycall(int i);
```

Construct a system call (5)

- Change Makefile of the kernel
  - Create Makefile for the "mycall" folder
  - Change Makefile of the kernel
    - Insert “mycall/” to “core-y”

Makefile

```
obj-y := mycall.o
```