CSE 306 Operating Systems
System Calls

YoungMin Kwon

System Calls

- A way to communicate with the kernel
- Interfaces, provided by the kernel, through which user-space processes can interact with the system
  - Controlled access to hardware
  - Create new processes
  - Communicate with other processes

System Calls

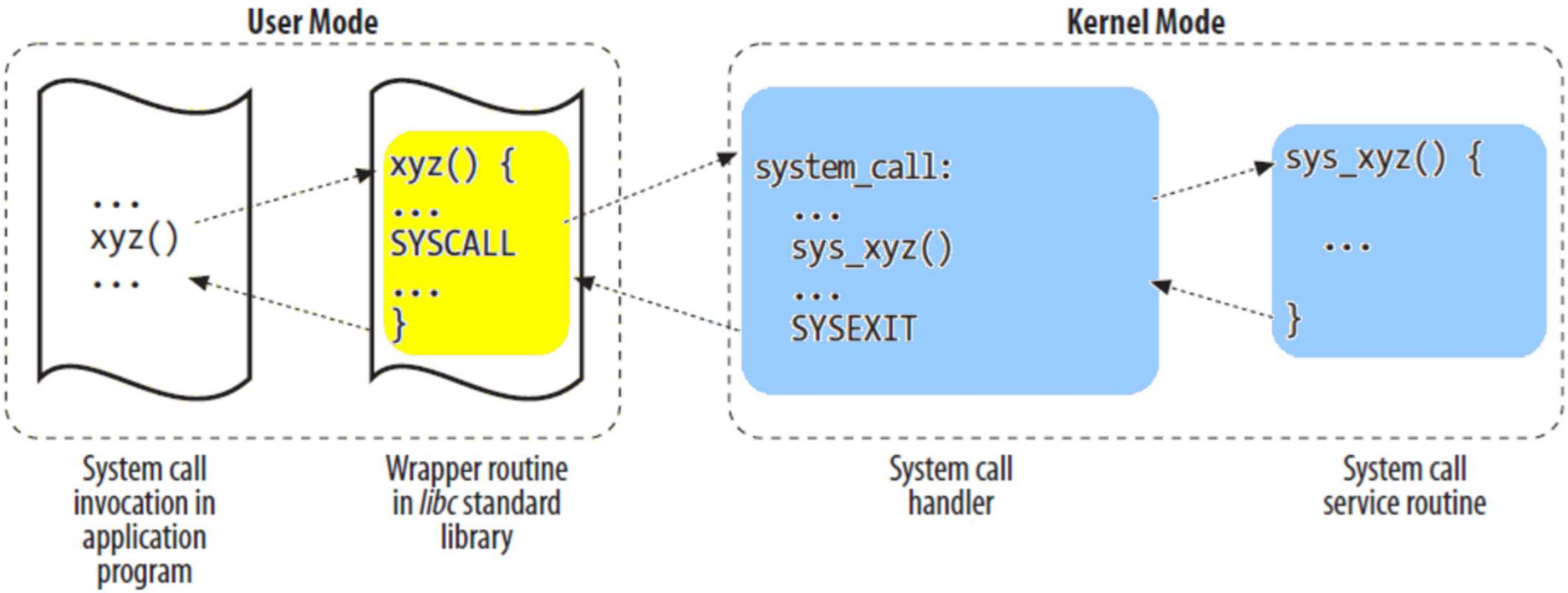
- Provides an abstracted hardware interfaces for user-space
  - Hide hardware specific details from the programmers
- Provides a security and stability
  - Arbitrate accesses to system resources base on permissions, users, and other criteria

System Calls

- Provides a virtualized system to processes
  - A single common layer between user-space and the rest of the system
  - If applications can freely access resources without the kernel's knowledge, it's nearly impossible to implement:
    - Multitasking, virtual memory, stability, security, …

System Calls

- An overview of a control flow



APIs and System Calls

- Usually, each system call has a wrapper routine that defines an API

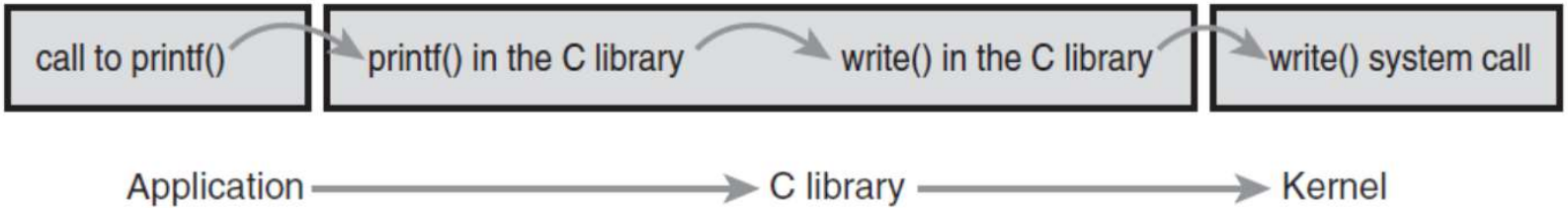
- Converse is not true
  - An API can be implemented totally in user mode
  - An API can call multiple system calls
  - Several APIs can make the same system calls
    - malloc(), calloc(), and free() can call brk()

APIs and System Calls

- From a programmer'spoint of view
  - Distinction between APIs and system calls are irrelevant
  - Only function names, parameters, and return types are important
- From a kernel designer's point of view
  - APIs and system calls are different
  - System calls belong to the kernel
  - API belongs to a user mode library

APIs

- C Library
  - Applications are programmed against user-space APIs, not directly to system calls
  - Portability: same source code can be complied to work on various OS with different sets of system calls



APIs

- POSIX (Portable Operating System Interface)
  - A series of IEEE standards
  - Referring to APIs, not system calls
  - One of the most common APIs in Unix, Linux
  - Microsoft Windows offers POSIX-compatible libraries

Syscalls: System Calls in Linux

- Accessed via function calls defined in the C library
  - Take zero, one, or more arguments
  - Might result in one or more side effects
    - E.g. writing to a file
    - getpid() doesn't have a side effect
  - Return a long type value
    - Zero usually means a success
    - Negative values usually mean an error
      - errno is set
      - perror() returns a human-readable error message

Syscalls

- Example

```
asmlinkage long sys_getpid(void);
in include/linux/syscalls.h
```

- `asmlinkage` compiler directive
  - Tells that compiler that all arguments should be in the stack
  - Required for all syscalls

- `sys_` prefix
  - A naming convention for Linux syscalls
  - getpid() is implemented as sys_getpid() in the kernel

Syscalls

- Syscall number:
  - In Linux, each system call is assigned with a unique syscall number
  - User-space processes refer to syscall numbers, not the syscall names (Application Binary Interface)
- syscall_table
  - Kernel keeps a list of all registered system calls in syscall_table
  - Sys_call_table keeps track of the syscall numbers
  - arch/x86/entry/syscalls/syscall_64.tbl

Syscalls

- Syscall number should not be changed
  - Otherwise, compiled apps will break
  - Even after removing a system call, its syscall number should not be recycled
    - Otherwise, existing apps may invoke the incorrect syscall
    - *Not Implemented* syscall handler (`sys_ni_syscall()`) should be used for those removed syscalls

System Call Handler

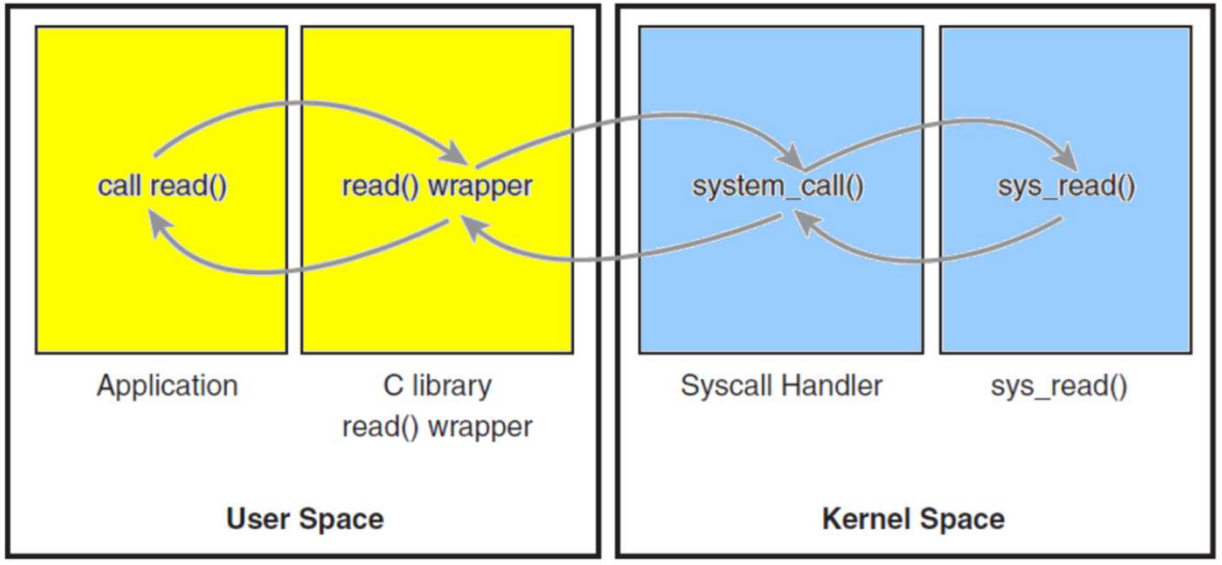
- How to invoke a kernel code from a user-mode process
  - User mode processes cannot execute kernel code directly
    - Function calls do not work
    - Needs a way to signal the kernel to switch mode and execute kernel code
  - Linux system calls use a software interrupt
    - Incur an exception (int $128)

System Call Handler

- Syscall number
  - After int $128, system switches to the kernel mode and execute the exception handler `system_call()`
  - `system_call` calls the system call function
    - Corresponding to the syscall number stored in `%rax`
  - `call *sys_call_table(, %rax, 8)`
    - call absolute address in `sys_call_table + %rax * 8 + 0`
    - arch/x86/entry/entry_64.S

System Call Handler

- Syscall parameters
  - ebx, ecx, edx, esi, and edi registers are used for the first 5 parameters (x86-32)
  - From the 6th onward, a single register is used to point to the user-space, where all the parameters are stored



Invoking the system call handler and executing a system call.

System Call Implementation

- A syscall should have exactly one purpose
  - Bad example: ioctl multiplexes multiple system calls

```
int ioctl(int fd, unsigned long request, ...);
```

- Syscall must carefully verify all parameters
  - Pointers should point to user-space
  - Pointers should point to the process's address space
  - For reading, the memory should be marked readable; for writing, the memory should be marked writable

System Call Implementation

- Use `copy_to_user()` and `copy_from_user()` to validate the parameters

- Use `capable()` to validate the caller's capabilities
  - e.g. `capable(CAP_SYS_BOOT);`

```
#include <linux/uaccess.h>
//asmlinkage long sys_silly_copy(
//  unsigned long *src, unsigned long *dst, unsigned long len)
SYSCALL_DEFINE3(silly_copy,
    unsigned long*, src, unsigned long*, dst, unsigned long, len)
{
    unsigned long buf;
    // copy *src in user-space into buf
    if(copy_from_user(&buf, src, len))
        return -EFAULT;
    // copy buf to *src in user-space
    if (copy_to_user(dst, &buf, len))
        return -EFAULT;
    return 0;
}
```

Writing a System Call (Hello World)

- Create a directory you will work on
  - `mkdir linux-5.4.49/cse306`

- Edit `linux-5.4.49/Makefile`

```
...
ifeq ($(KBUILD_EXTMOD),)
core-y      += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/

#cse306
core-y      += cse306/

vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
...
```

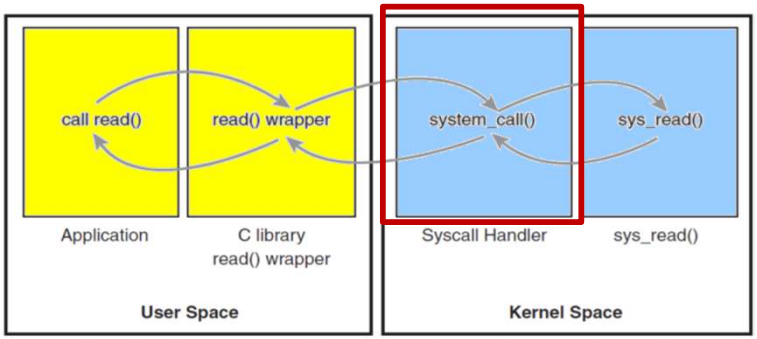
Writing a System Call (Hello World)

- Edit `linux-5.4.49/arch/x86/entry/syscalls/syscall_64.tbl`

```
...
435 common  clone3       __x64_sys_clone3/ptregs

#cse306
450 64  hello            __x64_sys_hello

# x32-specific system call numbers start
at 512 to avoid cache impact ...
```



Invoking the system call handler and executing a system call.

- Edit `linux-5.4.49/include/linux/syscalls.h`

```
...
asmlinkage long sys_ni_syscall(void);

/*cse306*/
asmlinkage long sys_hello(void);

#endif
```

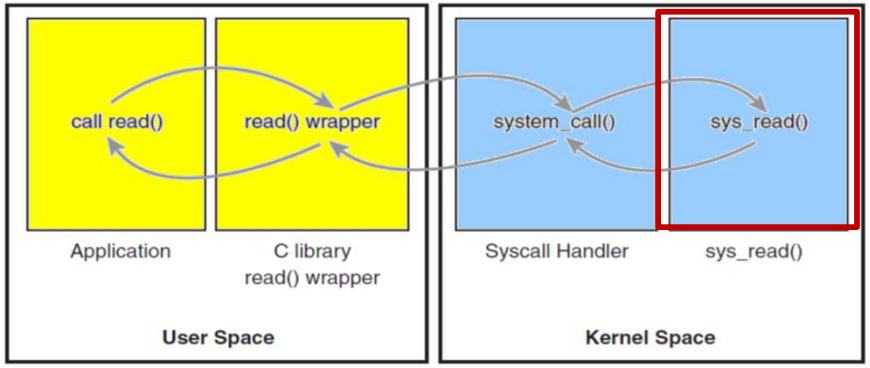
Writing a System Call (Hello World)

- Create `linux-5.4.49/cse306/Makefile` with

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

- Create `linux-5.4.49/cse306/hello.c`

```
#include <linux/syscalls.h>
#include <linux/kernel.h>
SYSCALL_DEFINE0(hello)
{
    printk("Hello world\n");
    return 0;
}
```



Invoking the system call handler and executing a system call.

- Run `make cse306` from `linux-5.4.49/` to check if your code compiles
- Run `make` from `linux-5.4.49/`

Writing a System Call (Hello World)

- Launch the test machine with the compiled kernel

```
qemu-system-x86_64 -nographic -serial mon:stdio -kernel
linux-5.4.49/arch/x86/boot/bzImage -hda ubuntu.img -
append "root=/dev/sda5 console=ttyS0 init=/sbin/init" -
enable-kvm -m 4096
```

- Alternatively, you can create a shell script `boot.sh`

```
#!/bin/bash
qemu-system-x86_64 -nographic -serial mon:stdio -kernel
linux-5.4.49/arch/x86/boot/bzImage -hda ubuntu.img -append
"root=/dev/sda5 console=ttyS0 init=/sbin/init" -enable-kvm
-m 4096
```

  - `chmod u+x boot.sh`
  - Run `boot.sh`

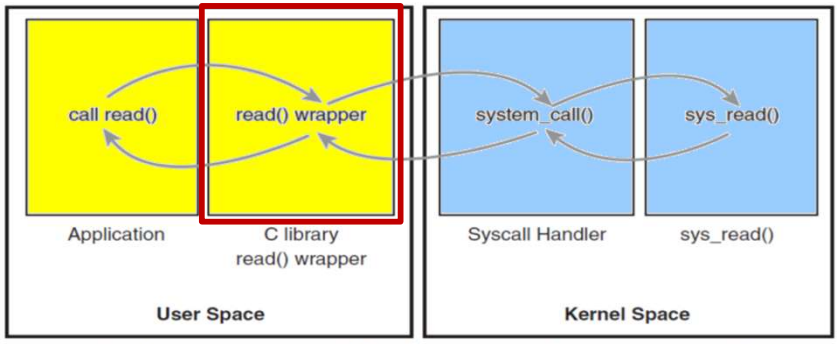
Writing a System Call (Hello World)

- In your test machine
  - Create `wrapper.c`

```
#include <unistd.h>
#define __NR_hello 450
long hello() {
        return syscall(__NR_hello);
}
```

  - Create `wrapper.h`

```
#ifndef __WRAPPER__
#define __WRAPPER__
extern long hello();
#endif
```

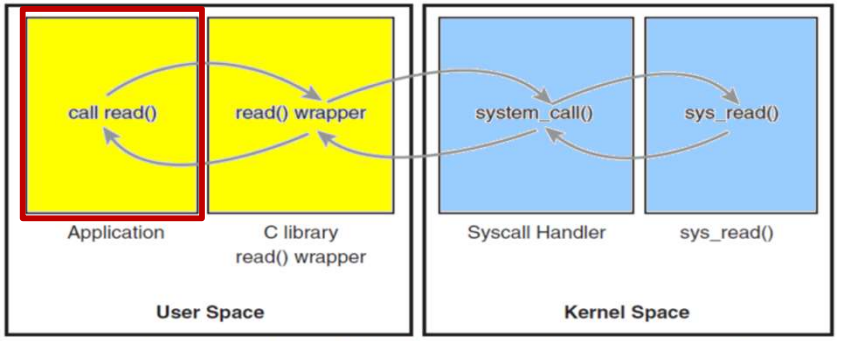


Invoking the system call handler and executing a system call.

Writing a System Call (Hello World)

- In your test machine
  - Create `hello.c`

```
#include <stdio.h>
#include "wrapper.h"
int main() {
        printf("result %ld\n",
                hello());
        return 0;
}
```



Invoking the system call handler and executing a system call.

  - `gcc hello.c wrapper.c`
  - `./a.out`
  - `dmesg` (to check the result)
  - `sudo shutdown now` (to shutdown)

Assignment 1

- Create hello_name.c and add a system call

```
SYSCALL_DEFINE4(hello_name, char*, name, int, namelen,
                char*, msg, int, msglen/*max msg*/)
```

  - It takes a string `name` and updates `msg` with Hello name
  - Use `copy_from_user` and `copy_to_user` for the parameter check
  - Check if `namelen` and `msglen` are valid
  - You can use `sprintf` to construct a message like
    - "Hello <name>"
    - sprintf returns the number of bytes written as well.

Assignment 1

- In your test machine, update wrapper.c, wrapper.h and write a user-space program that invokes the system call `hello_name`

```
#define NAME "YoungMin" //use your name
...
    char msg[100];
    long res = hello_name(NAME, sizeof(NAME), msg, sizeof(msg));
...
```

- Submit the files you modified or added in a single zip file

- Due date 3/21/2024

Creating a Patch File (Optional)

- `git status`
  - List the modified files
  - `git diff` will show you the changes
  - `git diff > patch_file` will create a patch file
  - `git apply patch_file` will apply the changes in the patch_file
  - `git checkout file` will revert the changes in the file

- `git add` *`the_files_you_changed`*
  - Move the files to the staging area (don't forget the new files)
  - `git diff --cached` will show you the changes
  - `git diff --cached > patch_file` will create a patch file
  - `git apply patch_file` will apply the changes in the patch_file
  - `git reset HEAD file` will unstage the file

Creating a Patch File (Optional)

- `git commit -m "description for the change"`
  - Commit the changes in the staging area
- `git format-patch -1`
  - Create a patch file from the last committed change
- `cat patch_file | colordiff`
  - To see the difference