CS 3502
Operating Systems

System Call

Kun Suo

Computer Science, Kennesaw State University

https://kevinsuo.github.io/

Outline

- What is system call?
  - Kernel space vs user space
  - System call vs library call
  - What service can system call provide?
  - System call naming, input, output
- How to design a system call
  - Example
  - Project 1

User space vs. Kernel space

- **Kernel space** is strictly reserved for running a privileged operating system kernel, kernel extensions, and most device drivers.
- **User space** is the area where application software and some drivers execute.

User mode	**User applications**	For example, bash, LibreOffice, GIMP, Blender, 0 A.D., Mozilla Firefox, etc.				
	Low-level system components:	**System daemons**: *systemd*, *runit*, *logind*, *networkd*, *PulseAudio*, ...	**Windowing system**: *X11*, *Wayland*, *SurfaceFlinger* (Android)	**Other libraries**: *GTK+*, *Qt*, *EFL*, *SDL*, *SFML*, *FLTK*, *GNUstep*, etc.		**Graphics**: *Mesa*, *AMD Catalyst*, ...
	C standard library	`open()`, `exec()`, `sbrk()`, `socket()`, `fopen()`, `calloc()`, ... (up to 2000 subroutines) *glibc* aims to be POSIX/SUS-compatible, *musl* and *uClibc* target embedded systems, *bionic* written for Android, etc.				
Kernel mode	**Linux kernel**	`stat`, `splice`, `dup`, `read`, `open`, `ioctl`, `write`, `mmap`, `close`, `exit`, etc. (about 380 system calls) The Linux kernel System Call Interface (SCI, aims to be POSIX/SUS-compatible)				
		Process scheduling subsystem	IPC subsystem	Memory management subsystem	Virtual files subsystem	Network subsystem
		Other components: ALSA, DRI, evdev, LVM, device mapper, Linux Network Scheduler, Netfilter Linux Security Modules: *SELinux*, *TOMOYO*, *AppArmor*, *Smack*				
Hardware (CPU, main memory, data storage devices, etc.)						

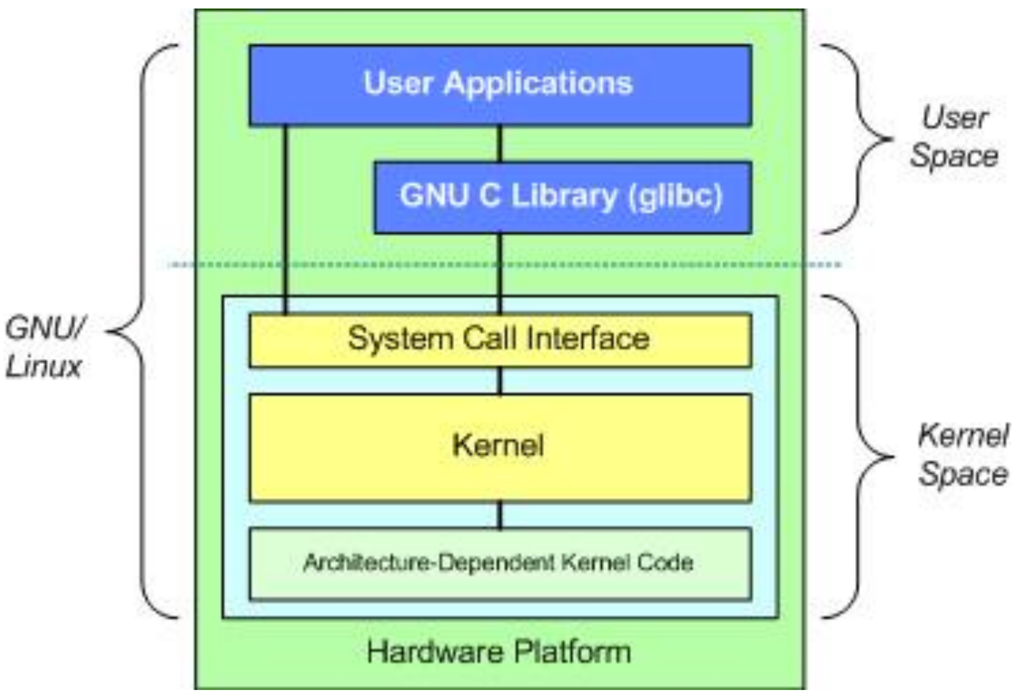
User mode vs. Kernel mode

- The difference between kernel and user mode?
  - The CPU can execute every instruction in its instruction set and use every feature of the hardware when executing in kernel mode.
  - However, it can execute only a subset of instructions and use only subset of features when executing in the user mode.

- The purpose of having these two modes
  - Purpose: protection – to protect critical resources (e.g., privileged instructions, memory, I/O devices) from being misused by user programs.

Interact between user space and kernel space

- For applications, in order to perform privileged operations, it must transit into OS through well defined interfaces
  - System call

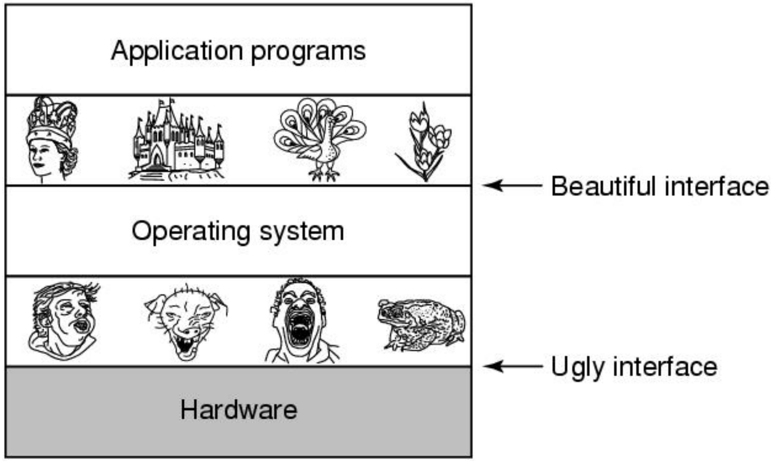



```
printf("%d", helloworld);
        ↓
write(buffer, count) ;
        ↓
os->write(buf, count, pos);
        ↓
```

System calls

- A type of special "protected procedure calls" allowing user-level processes request services from the kernel.

- System calls provide:
  - An abstraction layer between processes and hardware, allowing the kernel to provide access control
  - A virtualization of the underlying system
  - A well-defined interface for system services



System calls vs. Library functions

- What are the similarities and differences between ***system calls*** and ***library functions*** (e.g., libc functions)?

libc functions

https://www.gnu.org/software/libc/manual/html_node/Function-Index.html

system calls

https://filippo.io/linux-syscall-table/

System calls vs. Library functions

- Similarity
  - Both appear to be APIs that can be called by programs to obtain a given service
    - E.g., open,
    - https://elixir.bootlin.com/linux/latest/source/tools/include/nolibc/nolibc.h#L2038
    - E.g., strlen
    - https://www.gnu.org/software/libc/manual/html_node/String-Length.html#index-strlen

System calls vs. Library functions

```
/* strlen example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char szInput[256];
  printf ("Enter a sentence: ");
  gets (szInput);
  printf ("The sentence entered is %u characters long.\n",(unsigned)strlen(szInput))
  return 0;
}
```

Output:

```
Enter sentence: just testing
The sentence entered is 12 characters long.
```

libc functions:

<string.h> - - -> strlen() : all in user space

System calls vs. Library functions

```c
// C program to illustrate
// open system call
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
extern int errno;
int main()
{
    // if file does not have in directory
    // then file foo.txt is created.
    int fd = open("foo.txt", O_RDONLY | O_CREAT);

    printf("fd = %d/n", fd);

    if (fd ==-1)
    {
        // print which type of error have in a code
        printf("Error Number % d\n", errno);

        // print program detail "Success or failure"
        perror("Program");
    }
    return 0;
}
```

System calls:
<fcntl.h> - - -> open()
- - -> do_sys_open() // wrapper system call

https://elixir.bootlin.com/linux/latest/source/fs/open.c#L1074

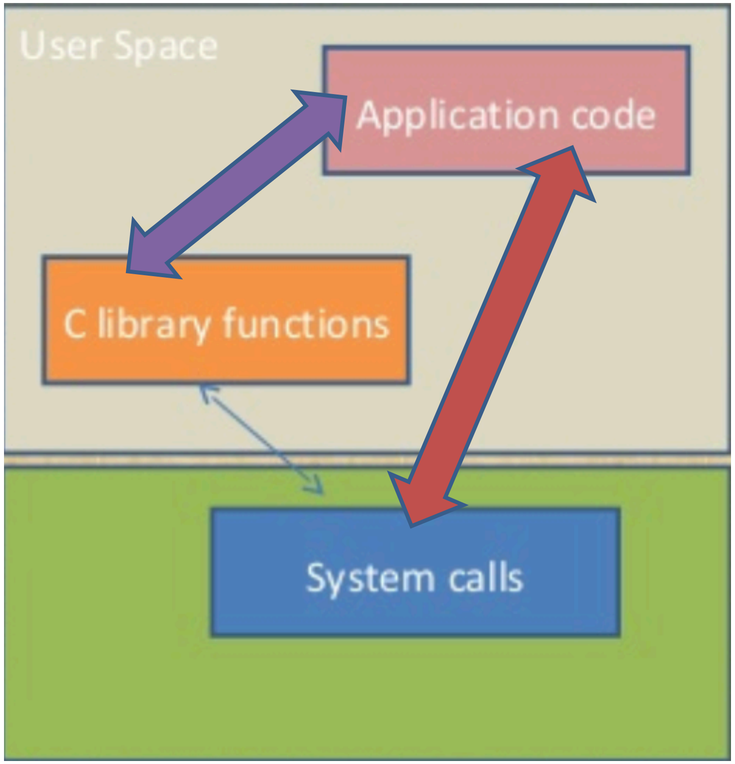
```c
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
        struct open_flags op;
        int fd = build_open_flags(flags, mode, &op);
        struct filename *tmp;

        if (fd)
                return fd;

        tmp = getname(filename);
        if (IS_ERR(tmp))
                return PTR_ERR(tmp);

        fd = get_unused_fd_flags(flags);
        if (fd >= 0) {
                struct file *f = do_filp_open(dfd, tmp, &op);
                if (IS_ERR(f)) {
                        put_unused_fd(fd);
                        fd = PTR_ERR(f);
                } else {
                        fsnotify_open(f);
                        fd_install(fd, f);
                }
        }
        putname(tmp);
        return fd;
}

SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
        if (force_o_largefile())
                flags |= O_LARGEFILE;

        return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

System calls vs. Library functions

- Difference
  - Library functions execute in the user space
  - System calls execute in the kernel space

strlen() (<string.h>) ? → all in user space

open() (<fcntl.h>)? → do_sys_open()

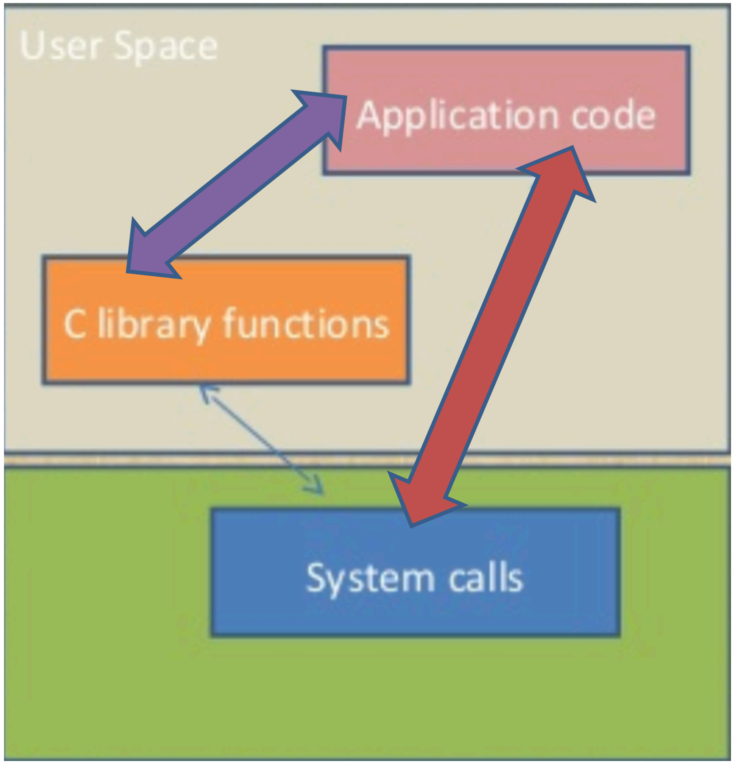


System calls vs. Library functions

- Difference
  - Fast, no context switch
  - Slow, high cost, kernel/user context switch

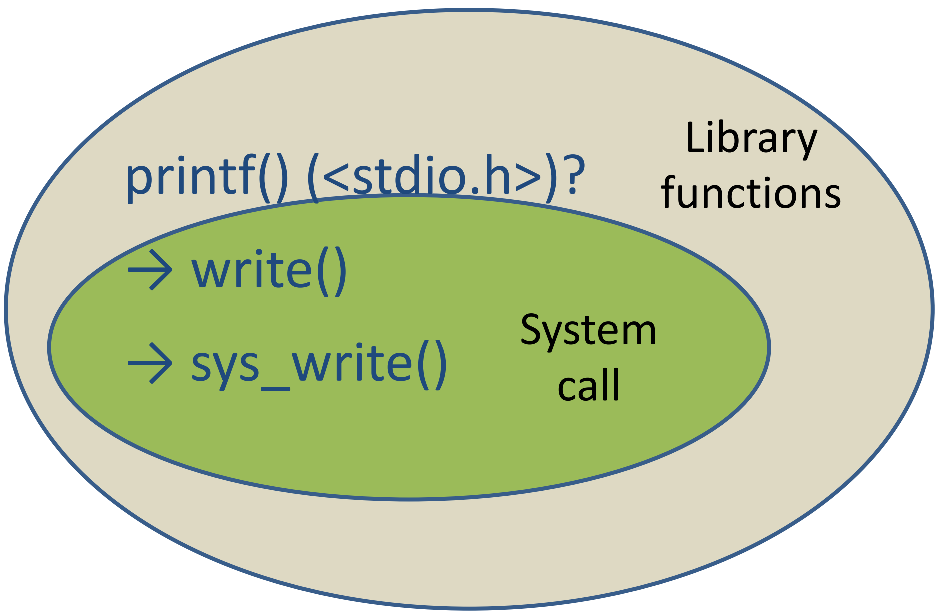
strlen() (<string.h>) ? → all in user space

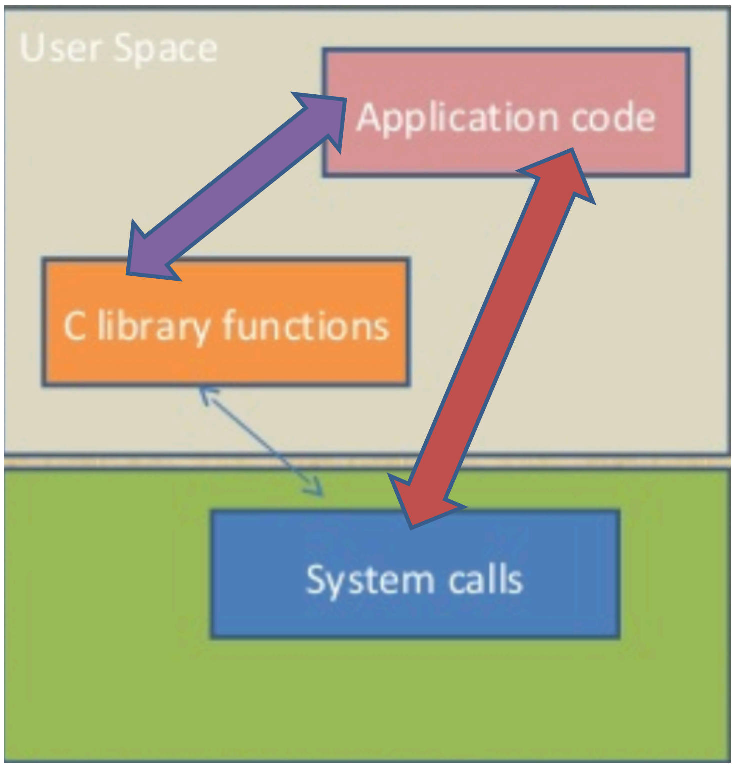
open() (<fcntl.h>)? → do_sys_open()



System calls vs. Library functions

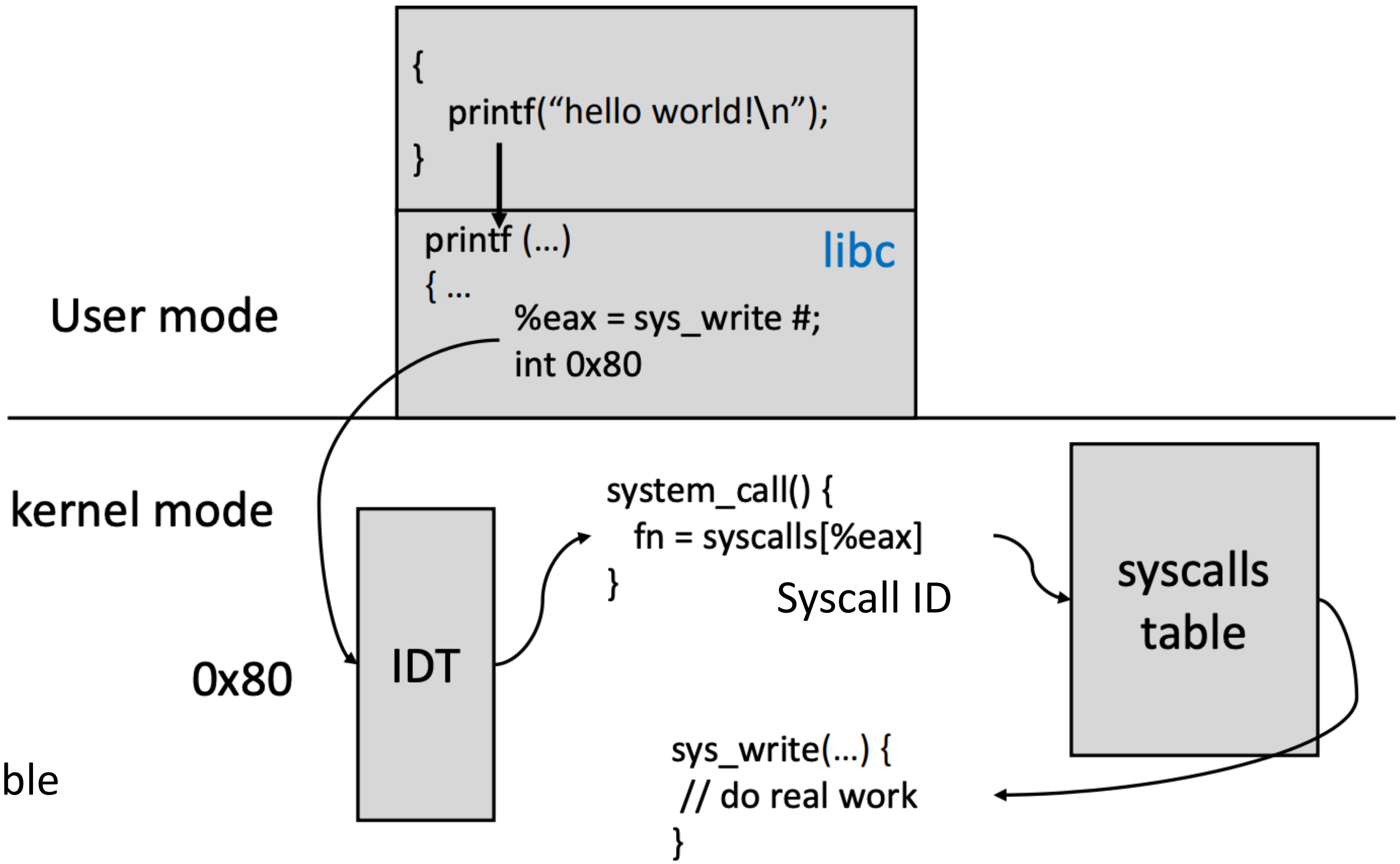
- Many system calls have a corresponding standard C library wrapper routines, which hide the details of system call entry/exit.





System calls vs. Library functions

Syscall Wrapper Macros



IDT: interrupt description table

System calls vs. Library functions

	System call	Library function
position	The functions which are a part of Kernel.	The functions which are a part of standard C library.
space	Get executed in kernel mode.	Get executed in user mode.
privilege	Runs in the supervisory mode so have every privileged.	Doesn't have much privileged.
performance	System calls are slow as there is context switch involved.	Faster execution as it doesn't involve context switch.

Services Provided by System Calls

- Process creation and management
- Main memory management
- File Access, Directory and File system management
- Device handling(I/O)
- Protection, e.g., encrypt
- Networking, etc.

Services Provided by System Calls

- Process related
  - end, abort
  - load, execute. E.g., exec
  - create process, terminate process. E.g., fork
  - get process attributes, set process attributes
  - wait for time. E.g., wait
  - wait event, signal event
  - allocate and free memory

https://filippo.io/linux-syscall-table/

fork

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

Output:

```
Hello world!
Hello world!
```

Services Provided by System Calls

- File related
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

https://filippo.io/linux-syscall-table/

open

```
#include <unistd.h>
#include <fcntl.h>

int main()
{
    int filedesc = open("testfile.txt", O_WRONLY | O_APPEND);
    if(filedesc < 0)
        return 1;

    if(write(filedesc,"This will be output to testfile.txt\n", 36) != 36)
    {
        write(2,"There was an error writing to testfile.txt\n");    // str
        return 1;
    }

    return 0;
}
```

Services Provided by System Calls

- Device related
  - register device, release device
  - read, write
  - get device attributes, set device attributes
  - logically attach or detach devices

https://filippo.io/linux-syscall-table/

ioctl

```
int main(void) {
  int fd;
  int i;
  int iomask;

  if ((fd = open("/dev/gpiog", O_RDWR))<0) {
    printf("Open error on /dev/gpiog\n");
    exit(0);
  }

  iomask=1<<25;

  for (i=0;i<10;i++) {
    printf("Led ON\n");
    ioctl(fd,_IO(ETRAXGPIO_IOCTYPE,IO_SETBITS),iomask);
    sleep(1);

    printf("Led OFF\n");
    ioctl(fd,_IO(ETRAXGPIO_IOCTYPE,IO_CLRBITS),iomask);
    sleep(1);
  }
  close(fd);
  exit(0);
}
```

Configure the device fd

Services Provided by System Calls

- Information related
  - Get time or date, set time or date
  - Number of current users
  - Amount of free memory or disk space

https://filippo.io/linux-syscall-table/

getpid

```c
#include<stdio.h>
#include<dos.h>
int main()
{
   struct date dt;

   getdate(&dt);

   printf("Operating system's current date is %d-%d-%d\n"
   ,dt.da_day,dt.da_mon,dt.da_year);

   return 0;
}
```

```c
#include <stdio.h>
#include <time.h>
int main ()
{
    time_t seconds;
    seconds = time (NULL);

    printf ("Number of hours since 1970 Jan 1st " \
            "is %ld \n", seconds/3600);
    return 0;
}
```

OUTPUT:

Operating system's current date is 12-01-2012

OUTPUT:

Number of hours since 1970 Jan 1st is 374528

Services Provided by System Calls

- Communication related
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach and detach remote devices

https://filippo.io/linux-syscall-table/

pipe

```c
int main()
{
    // We use two pipes
    // First pipe to send input string from pare
    // Second pipe to send concatenated string f

    int fd1[2];  // Used to store two ends of fi
    int fd2[2];  // Used to store two ends of se

    char fixed_str[] = "forgeeks.org";
    char input_str[100];
    pid_t p;

    if (pipe(fd1)==-1)
    {
        fprintf(stderr, "Pipe Failed" );
        return 1;
    }
    if (pipe(fd2)==-1)
    {
        fprintf(stderr, "Pipe Failed" );
        return 1;
    }

    scanf("%s", input_str);
    p = fork();

    if (p < 0)
    {
        fprintf(stderr, "fork Failed" );
        return 1;
    }
```

```c
    // child process
    else
    {
        close(fd1[1]);  // Close writing end of

        // Read a string using first pipe
        char concat_str[100];
        read(fd1[0], concat_str, 100);

        // Concatenate a fixed string with it
        int k = strlen(concat_str);
        int i;
        for (i=0; i<strlen(fixed_str); i++)
            concat_str[k++] = fixed_str[i];

        concat_str[k] = '\0';   // string ends w

        // Close both reading ends
        close(fd1[0]);
        close(fd2[0]);

        // Write concatenated string and close w
        write(fd2[1], concat_str, strlen(concat_
        close(fd2[1]);

        exit(0);
    }
}
```

chown

root : bash — Konsole

```
root@kali:~# ls -l file1.txt
-rw-r--r-- 1 root root 12 Feb  4 12:04 file1.txt
root@kali:~# chown master file1.txt
root@kali:~# ls -l file1.txt
-rw-r--r-- 1 master root 12 Feb  4 12:04 file1.txt
root@kali:~#
```

Examples of System Calls

Process Control	File Manipulation	Device Manipulation	Information Maintenance	Communication	Protection
fork() exit() wait()	open() read() write() close()	ioctl() read() write()	getpid() alarm() time()	pipe() send() recv() mmap()	chmod() umask() chown()

http://man7.org/linux/man-pages/man2/syscalls.2.html

https://filippo.io/linux-syscall-table/

Syscall interface

- Important to keep interface small, stable (for binary and backward compatibility). Every syscall does one thing.
- Early UNIXs had about 60 system calls, Linux 2.6 has about 300; Solaris more, Window more still
- Aside: Windows does not publicly document syscalls and only documents library wrapper routines (unlike UNIX/Linux)
- Syscall numbers cannot be reused (!)

Modern Linux system calls:

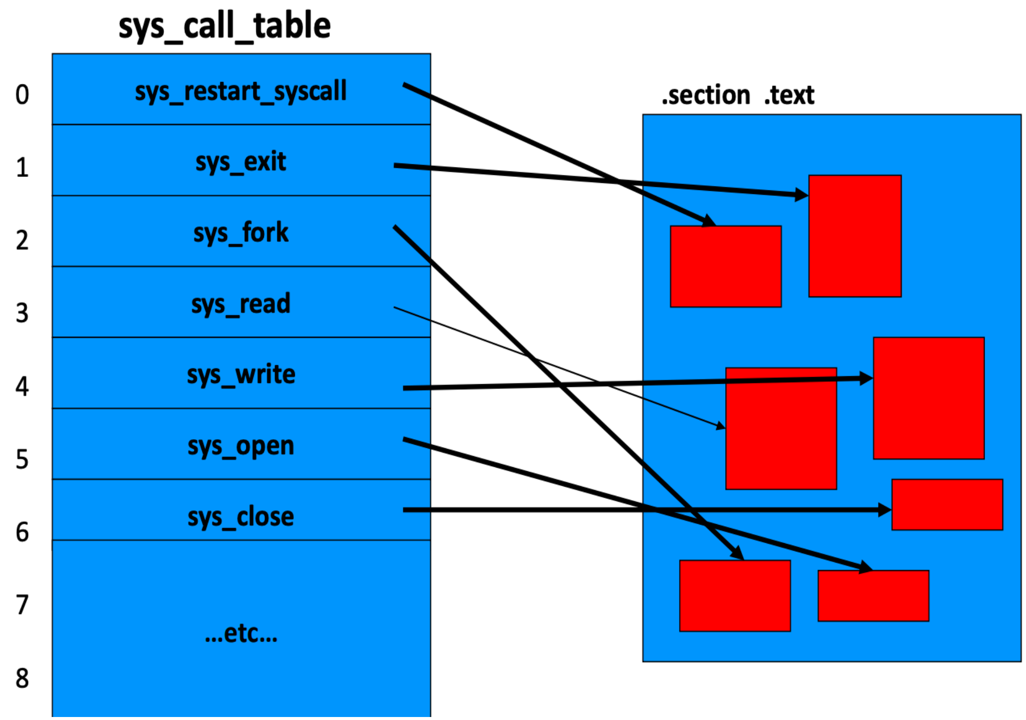
https://elixir.bootlin.com/linux/v5.0/source/arch/x86/entry/syscalls/syscall_64.tbl

Syscall interface APIs

- Three most common APIs in OSes are
  - Win32 API for Windows,
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X),
  - Java API for the Java virtual machine (JVM)

System call table

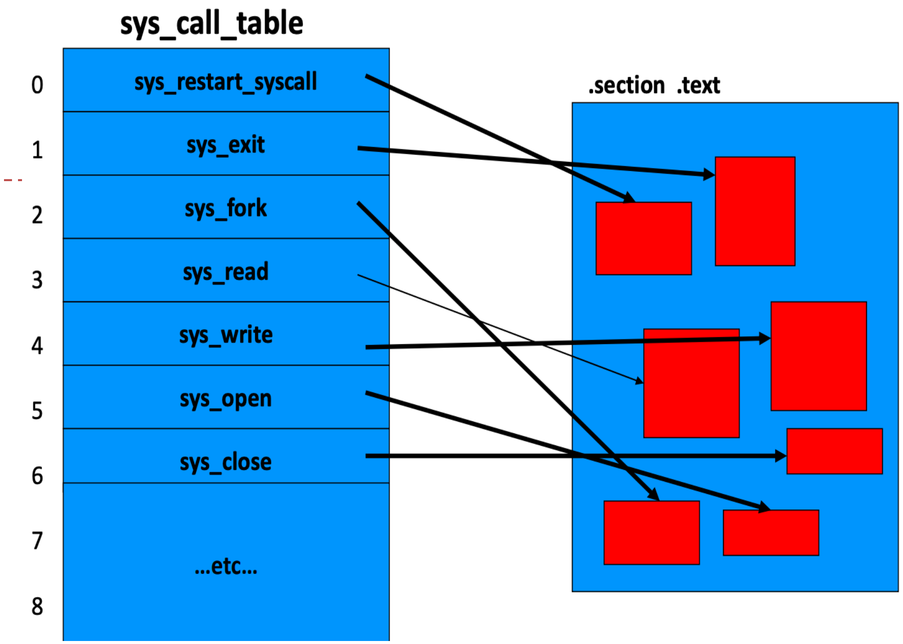
- There are approximately 300 system-calls in Linux 2.6.
- An array of function-pointers (identified by the ID number)
- This array is named 'sys_call_table[]' in Linux https://elixir.bootlin.com/linux/v5.0/source/arch/x86/entry/syscall_64.c

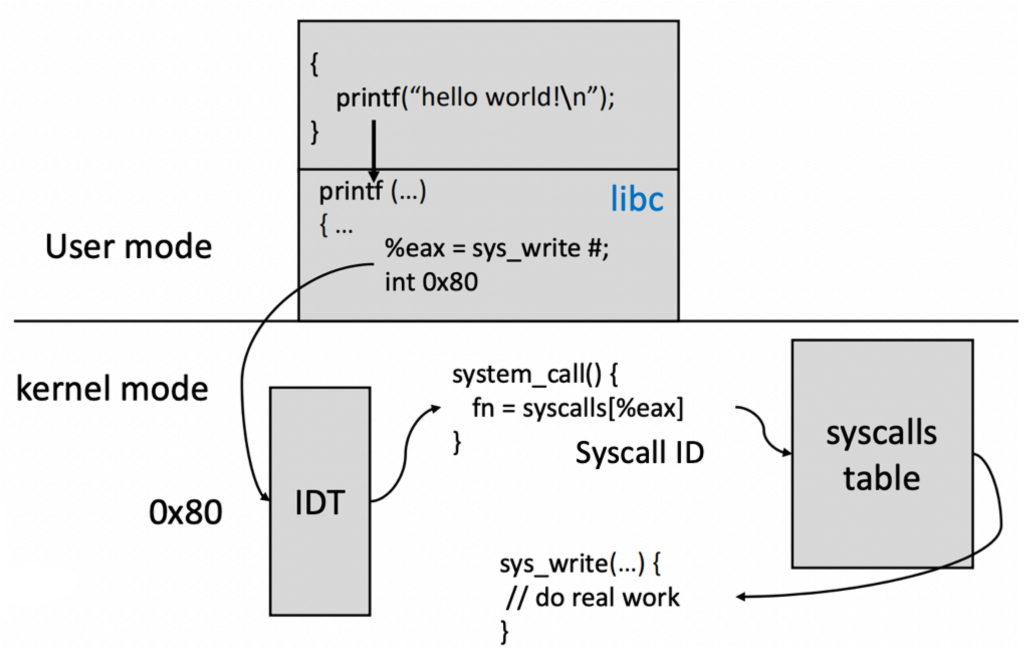


The 'jump-table' idea

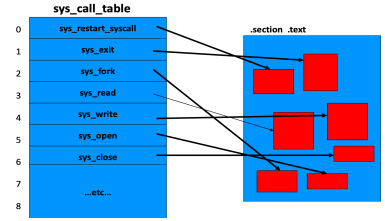
System call table

- Any specific system-call is selected by its ID-number (i.e., the system call number, which is placed into register %eax)

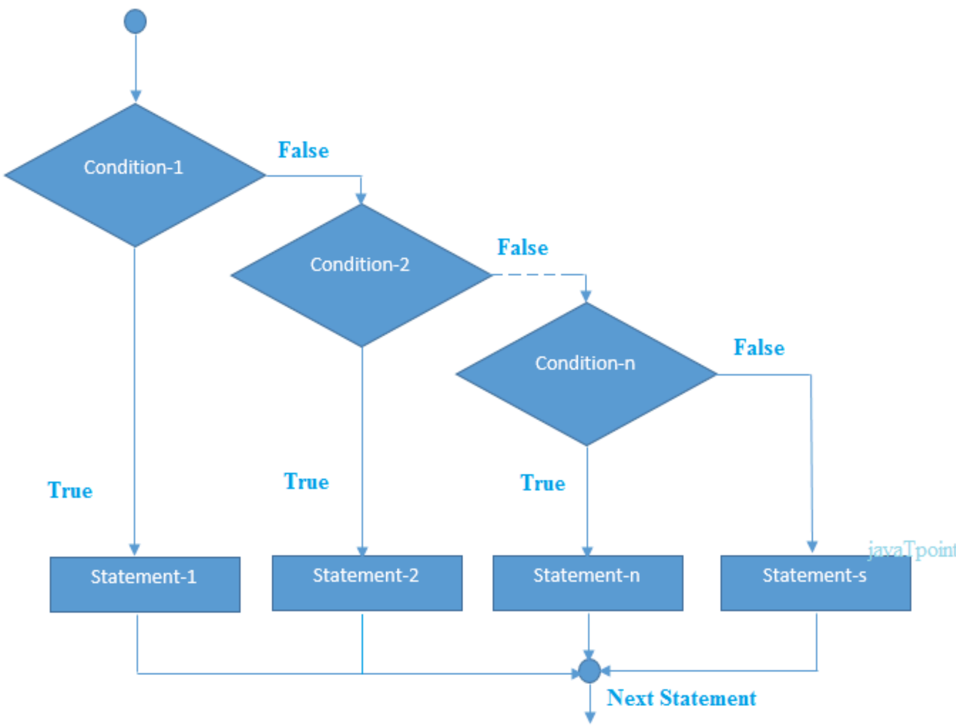




Discussion



- Instead of using the approach of system table, can we use if-else tests or switch statement to transfer to the service routine's entry point?



- Functionality wise, yes.
- But it would be extremely inefficient. O(n)
- System call invocations are synchronous, long system call execution is not desired.

Syscall Naming Convention

- Usually a library function “foo()” will do some work and then call a system call (“sys_foo()”)
- In Linux, all system calls begin with “sys_”
- Often “sys_abc()” just does some simple error checking and then calls a worker function named “do_abc()”

https://elixir.bootlin.com/linux/v4.14/source/arch/x86/entry/syscalls/syscall_64.tbl

open:

https://elixir.bootlin.com/linux/v4.14/source/fs/open.c#L1072

do_sys_open:

https://elixir.bootlin.com/linux/v4.14/source/fs/open.c#L1044

Syscall return values

- Recall that library calls return -1 on error, and place a specific error code in the global variable *errno*

- System calls return *specific negative values* to indicate an error
  - on x86, the return value is put into %eax, so that the library wrapper function can access.

library calls

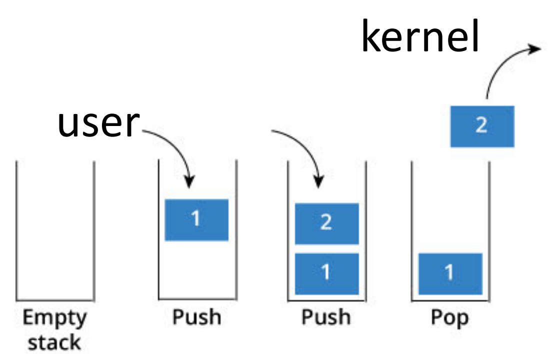
```
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>

…

int rc;

rc = syscall(SYS_chmod, "/etc/passwd", 0444);

if (rc == -1)
   fprintf(stderr, "chmod failed, errno = %d\n", e
```

System calls

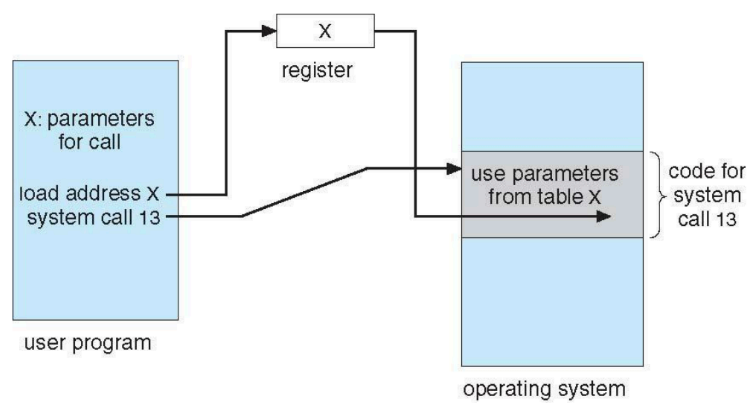
```
if (flags & __O_SYNC)
        flags |= O_DSYNC;

if (flags & __O_TMPFILE) {
        if ((flags & O_TMPFILE_MASK) != O_TMPFILE)
                return -EINVAL;
        if (!(acc_mode & MAY_WRITE))
                return -EINVAL;
} else if (flags & O_PATH) {
        /*
         * If we have O_PATH in the open flag. Then we
         * cannot have anything other than the below set of
         */
        flags &= O_DIRECTORY | O_NOFOLLOW | O_PATH;
        acc_mode = 0;
```

System call argument passing

- Three general methods used to pass arguments to the OS:
  - Method 1: pass the arguments in registers (simplest)
  - Method 2: arguments are placed, or pushed, onto the stack by the program and popped off the stack by the OS kernel code



  - Method 3: arguments are stored in a block, or table, in memory, and address of block passed as a parameter in a register, %eax
    - This approach taken by Linux and Solaris



System call argument passing discussion

- Consider a system call, zeroFill, which fills a user buffer with zeroes:

  zeroFill(char* buffer, int bufferSize);

- The following kernel implementation of zeroFill contains a security vulnerability. What is the vulnerability, and how would you fix it?

```
void sys_zeroFill(char* buffer, int bufferSize) {
        for (int i=0; i < bufferSize; i++) {
                buffer[i] = 0;
        }
}
```

System call argument passing discussion

```
void sys_zeroFill(char* buffer, int bufferSize) {
        for (int i=0; i < bufferSize; i++) {
                buffer[i] = 0;
        }
}
```

- The user buffer pointer is untrusted, and could point anywhere. In particular, it could point inside the kernel address space. This could lead to a system crash or security breakdown.
- Fix: verify the pointer is a valid user address

System call argument passing discussion

```
void sys_zeroFill(char* buffer, int bufferSize) {
	for (int i=0; i < bufferSize; i++) {
		buffer[i] = 0;
	}
}
```

- Is it a security risk to execute the zeroFill function in user-mode?

System call argument passing discussion

```
void sys_zeroFill(char* buffer, int bufferSize) {
    for (int i=0; i < bufferSize; i++) {
        buffer[i] = 0;
    }
}
```

- Is it a security risk to execute the zeroFill function in user-mode?
    - No. User-mode code does not have permission to access the kernel's address space. If it tries, the hardware raises an exception, which is safely handled by the OS.

Outline

- What is system call?
  - Kernel space vs user space
  - System call vs library call
  - What service can system call provide?
  - System call naming, input, output
- How to design a system call
  - Example
  - Project 1