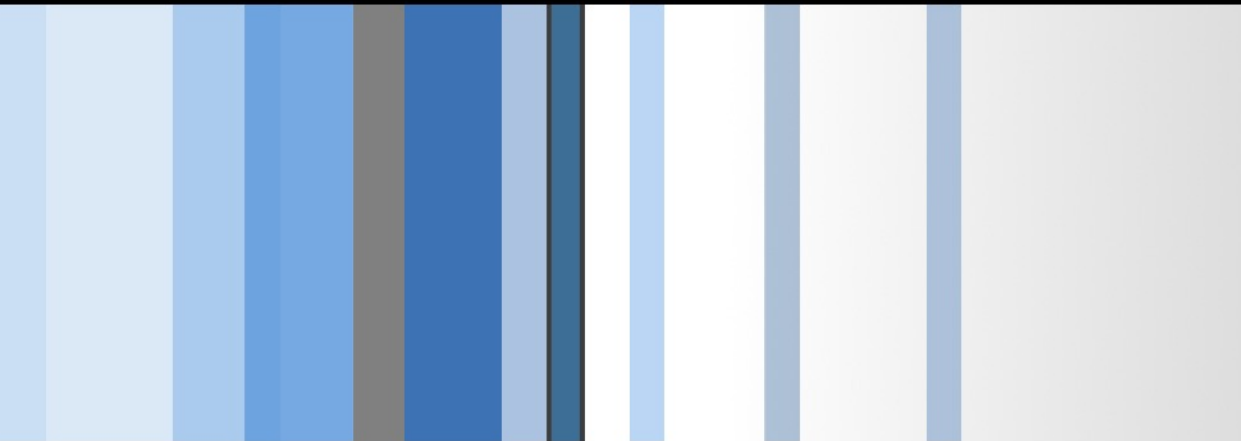


CSC 472 Software Security

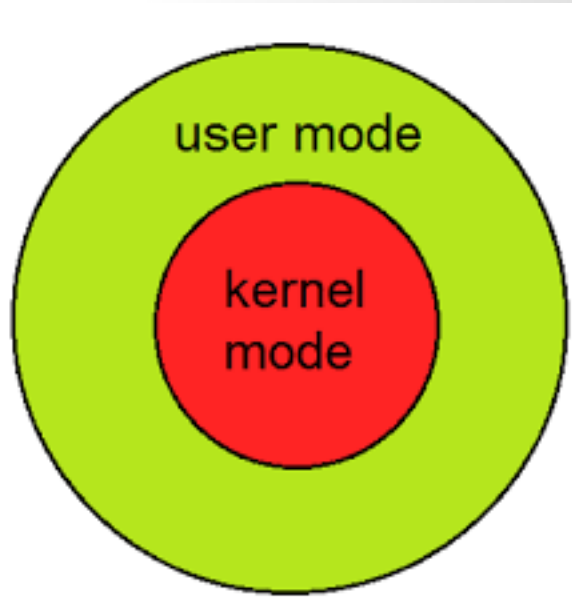
System Call, Shellcode

Calling Convention

Dr. Si Chen (schen@wcupa.edu)

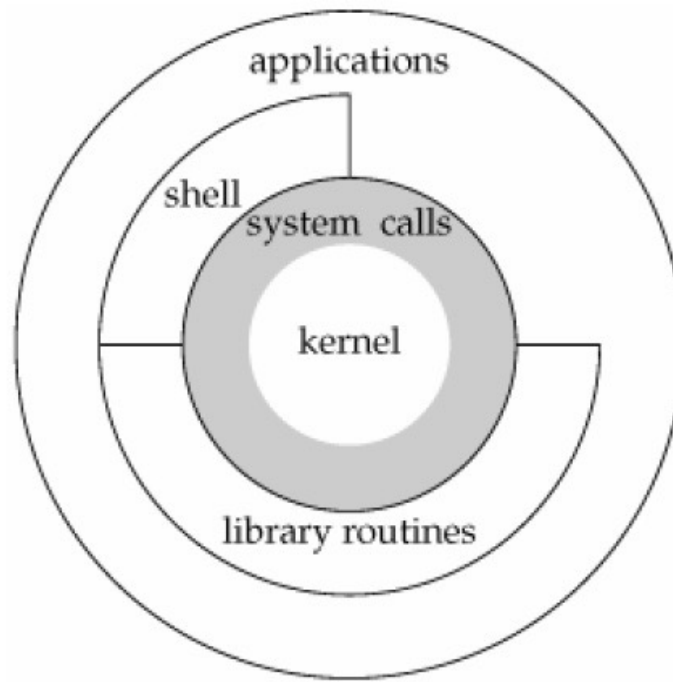


System Call



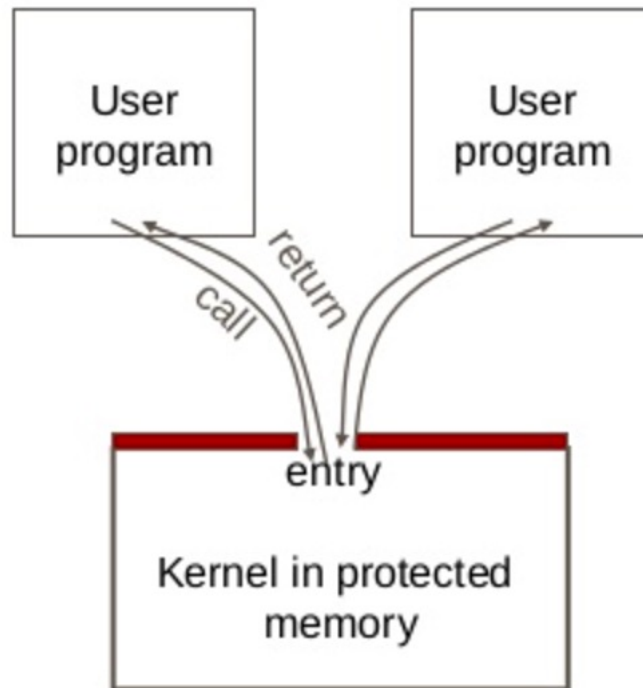
System Call

- A system call, sometimes referred to as a kernel call, is a request in a Unix-like operating system made via a software interrupt by an active process for a service performed by the kernel.



System Call

- User code can be arbitrary
- User code cannot modify kernel memory
- The call mechanism switches code to kernel mode



What is System Call?

- System resources (file, network, IO, device) may be accessed by multiple applications at the same time, can cause confliction.
- Modern OS protect these resources.
- E.g. How to let a program to wait for a while?

```
1 int i;  
2 for(int = 0; i < 100000; ++i);
```



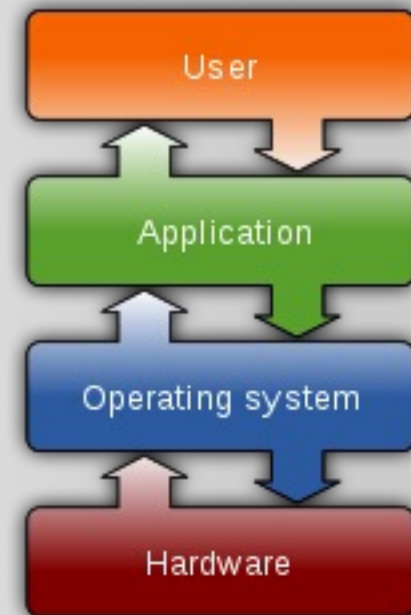
100Mhz CPU -> 1s
1000Mhz CPU -> 0.1s

Use OS provide Timer

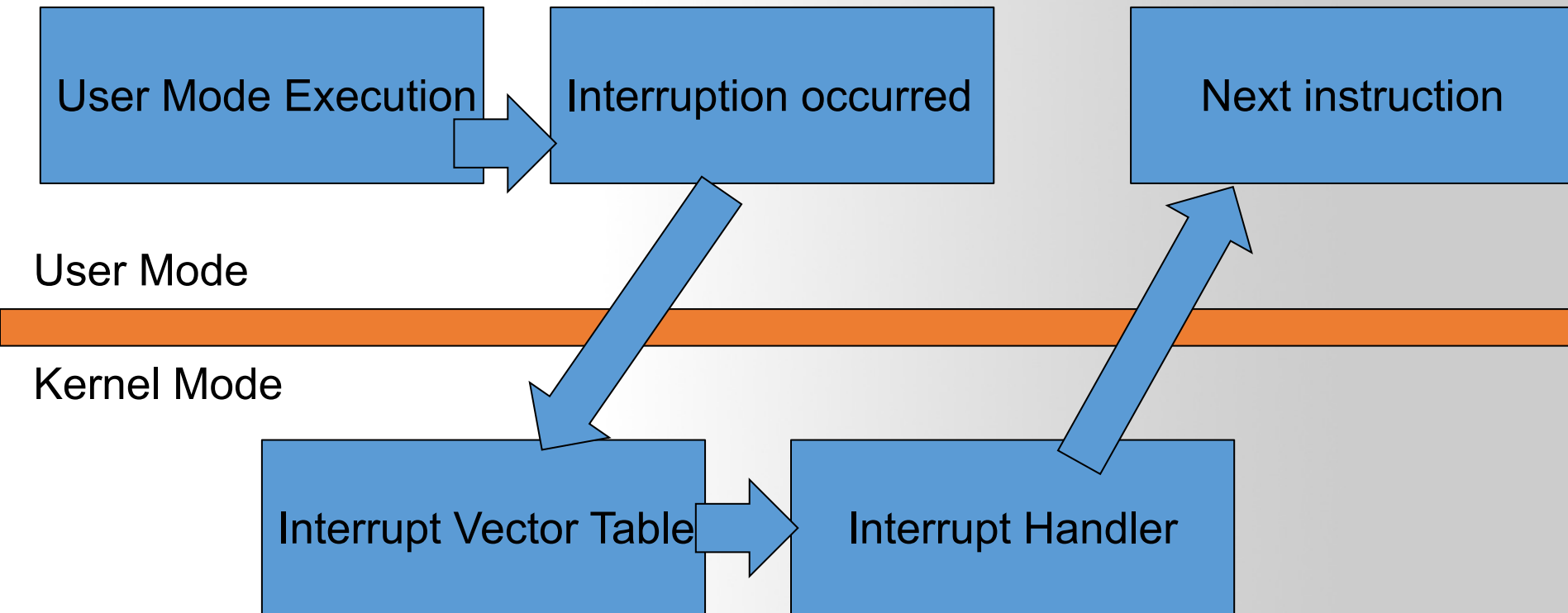
What System Call?

- Let an application to access system resources.
- OS provide an interface (**System call**) for the application
- It usually use the technique called “interrupt vector”
 - Linux use 0x80
 - Windows use 0x2E

In [system programming](#), an **interrupt** is a signal to the [processor](#) emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing. The processor responds by suspending its current activities, saving its [state](#), and executing a [function](#) called an [interrupt handler](#) (or an interrupt service routine, ISR) to deal with the event. This interruption is temporary, and, after the interrupt handler finishes, the processor resumes normal activities.^[1] There are two types of interrupts: hardware interrupts and software interrupts. – From Wikipedia



CPU Interrupt



fwrite() path in both Linux and Windows

Application

fwrite()

./program

fwrite()

program.exe

C Run Time Library

write()

libc.a
libc.so

write()

Libcmt.lib
msvcr90.dll

interrupt 0x80

libc.a
libc.so

API (Windows)

NtWriteFile()

Kernel32.dll

Interrupt 0x2e

NTDLL.dll

Kernel

sys_write()
Kernel

./vlinuxz

IoWriteFile()
Kernel

NtosKrnI.exe

Linux System Call

Linux Syscall Reference

<http://syscalls.kernelgrok.com>

Show <input type="button" value="All"/> <input type="button" value="entries"/> Search: <input type="text"/>								
#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
0	sys_restart_syscall	0x00	-	-	-	-	-	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	-	-	-	-	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *	-	-	-	-	arch/alpha/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	fs/read_write.c:391
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	fs/read_write.c:408
5	sys_open	0x05	const char __user *filename	int flags	int mode	-	-	fs/open.c:900
6	sys_close	0x06	unsigned int fd	-	-	-	-	fs/open.c:969
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	-	-	kernel/exit.c:1771
8	sys_creat	0x08	const char __user *pathname	int mode	-	-	-	fs/open.c:933
9	sys_link	0x09	const char __user *oldname	const char __user *newname	-	-	-	fs/namei.c:2520
10	sys_unlink	0x0a	const char __user *pathname	-	-	-	-	fs/namei.c:2352
11	sys_execve	0x0b	char __user *	char __user * __user *	char __user * __user *	struct pt_regs *	-	arch/alpha/kernel/entry.S:925
12	sys_chdir	0x0c	const char __user *filename	-	-	-	-	fs/open.c:361
13	sys_time	0x0d	time_t __user *tloc	-	-	-	-	kernel/posix-timers.c:855
14	sys_mknod	0x0e	const char __user *filename	int mode	unsigned dev	-	-	fs/namei.c:2067
15	sys_chmod	0x0f	const char __user *filename	mode_t mode	-	-	-	fs/open.c:507
16	sys_lchown16	0x10	const char __user *filename	old_uid_t user	old_gid_t group	-	-	kernel/uid16.c:27
17	not implemented	0x11	-	-	-	-	-	
18	sys_stat	0x12	char __user *filename	struct __old_kernel_stat __user *statbuf	-	-	-	fs/stat.c:150
19	sys_lseek	0x13	unsigned int fd	off_t offset	unsigned int origin	-	-	fs/read_write.c:167
20	sys_getpid	0x14	-	-	-	-	-	kernel/timer.c:1337
21	sys_mount	0x15	char __user *dev_name	char __user *dir_name	char __user *type	unsigned long flags	void __user *data	fs/namespace.c:2118
22	sys_oldumount	0x16	char __user *name	-	-	-	-	fs/namespace.c:1171

Show All entries		Search:	
#	Name	Registers	Definition
		eax ebx ecx edx esi edi	
0	sys_restart_syscall	0x00 - - - - -	kernel/signal.c:2058
1	sys_exit	0x01 int error_code - - - - -	kernel/exit.c:1046
2	sys_fork	0x02 struct pt_regs * - - - - -	arch/alpha/kernel/entry.S:716
3	sys_read	0x03 unsigned int fd char __user *buf size_t count - -	fs/read_write.c:391
4	sys_write	0x04 unsigned int fd const char __user *buf size_t count - -	fs/read_write.c:408
5	sys_open	0x05 const char __user *filename int flags int mode - -	fs/open.c:900
6	sys_close	0x06 unsigned int fd - - - - -	fs/open.c:969
7	sys_waitpid	0x07 pid_t pid int __user *stat_addr int options - -	kernel/exit.c:1771
8	sys_creat	0x08 const char __user *pathname int mode - - - -	fs/open.c:933
9	sys_link	0x09 const char __user *oldname const char __user *newname - - -	fs/namei.c:2520
10	sys_unlink	0x0a const char __user *pathname - - - - -	fs/namei.c:2352
11	sys_execve	0x0b char __user * char __user * __user char __user * __user struct pt_regs * -	arch/alpha/kernel/entry.S:925
12	sys_chdir	0x0c const char __user *filename - - - - -	fs/open.c:361
13	sys_time	0x0d time_t __user *tloc - - - - -	kernel/posix-timers.c:855
14	sys_mknod	0x0e const char __user *filename int mode unsigned dev - -	fs/namei.c:2067
15	sys_chmod	0x0f const char __user *filename mode_t mode - - -	fs/open.c:507
16	sys_lchown16	0x10 const char __user *filename old_uid_t user old_gid_t group - -	kernel/uid16.c:27
17	not implemented	0x11 - - - - -	-
18	sys_stat	0x12 char __user *filename struct __old_kernel_stat __user *statbuf - - -	fs/stat.c:150
19	sys_lseek	0x13 unsigned int fd off_t offset unsigned int origin - -	fs/read_write.c:167
20	sys_getpid	0x14 - - - - -	kernel/timer.c:1337
21	sys_mount	0x15 char __user *dev_name char __user *dir_name char __user *type unsigned long flags void __user *data	fs/namespace.c:2118
22	sys_oldumount	0x16 char __user *name - - - - -	fs/namespace.c:1171

```

836         const struct itimerspec __user *utmr,
837         struct itimerspec __user *otmr);
838
839 asmlinkage long sys_timerfd_gettime(int ufd, struct itimerspec __user *otmr);
840
841 asmlinkage long sys_eventfd(unsigned int count, int flags);
842
843 asmlinkage long sys_fallocate(int fd, int mode, loff_t offset, loff_t len);
844
845 asmlinkage long sys_old_readdir(unsigned int, struct old_linux_dirent __user *, unsigned int, fd_set __user *, fd_set __user *, fd_set __user *, struct timespec __user *, void __user *);
846
847 asmlinkage long sys_ppoll(struct pollfd __user *, unsigned int, struct timespec __user *, const sigset_t __user *, size_t);
848
849 asmlinkage long sys_fanotify_init(unsigned int flags, unsigned int event_f_flags);
850
851 asmlinkage long sys_fanotify_mark(int fanotify_fd, unsigned int flags, u64 mask, int fd, const char __user *pathname);
852
853 asmlinkage long sys_syncfs(int fd);
854
855 asmlinkage long sys_fork(void);
856
857 asmlinkage long sys_vfork(void);
858
859 #ifdef CONFIG_CLONE_BACKWARDS
860 asmlinkage long sys_clone(unsigned long, unsigned long, int __user *, int, int __user *);
861
862 #else
863 asmlinkage long sys_clone(unsigned long, unsigned long, int __user *, int __user *, int);
864
865 #endif
866
867 asmlinkage long sys_execve(const char __user *filename, const char __user *const __user *argv, const char __user *const __user *envp);
868
869 asmlinkage long sys_perf_event_open(struct perf_event_attr __user *attr_uptr, pid_t pid, int cpu, int group_fd, unsigned long flags);
870
871 asmlinkage long sys_mmap_pgoff(unsigned long addr, unsigned long len, unsigned long prot, unsigned long flags, unsigned long fd, unsigned long pgoff);
872
873 asmlinkage long sys_old_mmap(struct mmap_arg_struct __user *arg);
874
875 asmlinkage long sys_name_to_handle_at(int dfd, const char __user *name, struct file_handle __user *handle, int __user *mnt_id, int flag);
876
877 asmlinkage long sys_open_by_handle_at(int mountdirfd, struct file_handle __user *handle, int flags);
878
879 asmlinkage long sys_setns(int fd, int nstype);
880
881 asmlinkage long sys_process_vm_readv(pid_t pid, const struct iovec __user *lvec, unsigned long liovcnt, const struct iovec __user *rvec, unsigned long riovcnt, unsigned long flags);
882
883 asmlinkage long sys_process_vm_writev(pid_t pid, const struct iovec __user *lvec, unsigned long liovcnt, const struct iovec __user *rvec, unsigned long riovcnt, unsigned long flags);
884
885 asmlinkage long sys_kcmp(pid_t pid1, pid_t pid2, int type, unsigned long idx1, unsigned long idx2);
886
887 asmlinkage long sys_finit_module(int fd, const char __user *uargs, int flags);
888
889 #endif
890

```

Trace by strace (linux)

■ strace /bin/echo AAAAA

```
root@8539c0f89c7c:/workdir # strace /bin/echo AAAAA
```

```
execve("/bin/echo", ["/bin/echo", "AAAAA"], 0x7ffe50bb87b8 /* 17 vars */) = 0
brk(NULL)                               = 0x558df6174000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2663a9b000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=64842, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 64842, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f2663a8b000
close(3)                                 = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220x\2\0\0\0\0"... , 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@0\0\0\0\0\0\0@0\0\0\0\0\0\0@0\0\0\0\0\0\0"... , 784, 64) = 784
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1926256, ...}, AT_EMPTY_PATH) = 0
pread64(3, "\6\0\0\0\4\0\0\0@0\0\0\0\0\0\0@0\0\0\0\0\0\0@0\0\0\0\0\0\0"... , 784, 64) = 784
mmap(NULL, 1974096, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f26638a9000
mmap(0x7f26638cf000, 1396736, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7f26638cf000
mmap(0x7f2663a24000, 344064, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x17b000) = 0x7f2663a24000
mmap(0x7f2663a78000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1cf000) = 0x7f2663a78000
mmap(0x7f2663a7e000, 53072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f2663a7e000
close(3)                                 = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f26638a6000
arch_prctl(ARCH_SET_FS, 0x7f26638a6740) = 0
set_tid_address(0x7f26638a6a10)         = 47
set_robust_list(0x7f26638a6a20, 24)     = 0
rseq(0x7f26638a7060, 0x20, 0, 0x53053053) = 0
mprotect(0x7f2663a78000, 16384, PROT_READ) = 0
mprotect(0x558df52f1000, 4096, PROT_READ) = 0
mprotect(0x7f2663acd000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f2663a8b000, 64842)           = 0
getrandom("\x8d\x8d\x70\x85\x3a\x3d\x59\xc4", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x558df6174000
brk(0x558df6195000)                     = 0x558df6195000
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=3052896, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 3052896, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f26635bc000
close(3)                                 = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
write(1, "AAAAA\n", 6AAAAA
)                                         = 5
close(1)                               = 0
close(2)                               = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

system call

Example: Hello World

Quick review:

- DB** - Define Byte. 8 bits
- DW** - Define Word. Generally 2 bytes on a typical x86 32-bit system
- DD** - Define double word. Generally 4 bytes on a typical x86 32-bit system

From [x86 assembly tutorial](#),

```
section .text
global _start
_start:

    mov eax, 4        ; sys_write
    mov ebx, 1        ; fd
    mov ecx, msg      ; buf
    mov edx, 13       ; size
    int 0x80          ; write(1, "Hello world!\n", 13)

    mov eax, 1        ; sys_exit
    mov ebx, 0        ; status
    int 0x80          ; exit(0)

section .data
msg:

    db 'Hello world!', 0xA
```

helloworld.asm

```
[quake0day@quake0day-pc ~]$ nasm -felf32 helloworld.asm -o helloworld.o && ld helloworld.o -melf_i386 -o helloworld
[quake0day@quake0day-pc ~]$ ./helloworld
Hello world!
```

Example: launch a shell

```
section .text
global _start
_start:
    mov eax, 0x0b
    mov ebx, sh
    mov ecx, argv
    mov edx, envp
    int 0x80

section .data
argv:
    dd sh, 0

envp:
    dd 0

sh:
    db "/bin/sh", 0
~
```

shell.asm

```
[quake0day@quake0day-pc ~]$ nasm -felf32 shell.asm -o shell.o && ld shell.o -melf_i386 -o shell
[quake0day@quake0day-pc ~]$ ./shell
sh-4.4$
```

Some Useful System Call

▪ open/read/write

```
eax ebx ecx edx
0x05 path  0   0 open(path, O_RDONLY)
0x03  fd buf size read(fd, buf, size)
0x04  fd buf size write(fd, buf, size)
```

▪ mmap/mprotect

- mmap: use to allocate an executable area
- mprotect: disable data executable prevention

▪ execve

- execve(char* path, char* argv[], char* envp[]);
- path: path to the executable file
- argv: arguments (char* pointer array)
- envp: environment variable (char* pointer array)

Libc Wraps Syscalls

- **What is Libc?:** Libc is the **C Standard Library** as a library of routines that software applications commonly use.
 - Many of its functions are wrappers for system calls.
- **Why Wrappers?**
 1. **Ease of Use:** Syscalls often have a lower-level interface that is not convenient to use directly.
 2. **Portability:** Using Libc makes it easier to write portable code, as the library handles the platform-specific details.
 3. **Error Handling:** Libc functions usually provide higher-level error handling compared to raw syscall interfaces.

Examples

1. File Operations:

1. `fopen()` vs `open()`
2. `fclose()` vs `close()`

2. Memory Allocation:

1. `malloc()` vs `mmap()` or `sbrk()`

3. Process Creation:

1. `fork()` in Libc vs `clone()` syscall in Linux

4. Time and Sleep:

1. `sleep()` in Libc vs `nanosleep()` syscall

```
1  void main()  
2  {  
3      exit(0);  
4  }
```

Syscall Summary

- Linux Syscall use fastcall
 - specific syscall # is loaded into eax
 - arguments for call are placed in different registers
 - **int 0x80** executes call to syscall()
 - CPU switches to kernel mode
 - each syscall has a unique, static number

Shellcode

Shellcode is defined as a set of instructions injected and then executed by an exploited program. **Shellcode** is used to directly manipulate registers and the functionality of a exploited program.

Crafting Shellcode (the small program)

Example: Hello World

```
1  hello.asm
2  [SECTION .text]
3
4  global _start
5
6
7  _start:
8
9      jmp short ender
10
11     starter:
12
13     xor eax, eax    ;clean up the registers
14     xor ebx, ebx
15     xor edx, edx
16     xor ecx, ecx
17
18     mov al, 4        ;syscall write
19     mov bl, 1        ;stdout is 1
20     pop ecx          ;get the address of the string from the stack
21     mov dl, 5        ;length of the string
22     int 0x80
23
24     xor eax, eax
25     mov al, 1        ;exit the shellcode
26     xor ebx, ebx
27     int 0x80
28
29     ender:
30     call starter    ;put the address of the string on the stack
31     db 'hello'
```

hello.asm

Crafting Shellcode (the small program)

Example: Hello (hello.asm)

To compile it use nasm:

```
→ ~ nasm -f elf hello.asm
```

Use objdump to get the shellcode bytes:

```
[csc495@csc495-pc ~]$ objdump -d -M intel hello.o
;hello.asm
[SECTION .text]
hello.o:      file format elf32-i386
global _start

Disassembly of section .text:

_start:
00000000<_start>:
0:  eb 19                jmp     1b <call_shellcode>
   starter:

00000002<shellcode>:
2:  31 c0                xor     eax,eax
4:  b0 04                mov     al,0x4
6:  31 db                xor     ebx,ebx
8:  b3 01                mov     bl,0x1
a:  59                  ;syscall write
   ;stdout is 1
   pop     ecx
b:  d2                  ;get the address of the string from the stack
   mov     dl,edx
d:  b2 0d                ;length of the string
   int     0x80
f:  cd 80                int     0x80
11:  31 c0                xor     eax,eax
13:  b0 01                mov     al,0x1
   ;exit the shellcode
   xor     ebx,ebx
15:  31 db                xor     ebx,ebx
17:  b3 05                mov     bl,0x5
19:  cd 80                int     0x80
   ;put the address of the string on the stack
   call starter
```

Crafting Shellcode (the small program)

Disassembly of section .text:

```
00000000 <start>:
0:  eb 19          jmp     1b <ender>
00000002 <starter>:
2:  31 c0          xor     eax,eax
4:  31 db          xor     ebx,ebx
6:  31 d2          xor     edx,edx
8:  31 c9          xor     ecx,ecx
a:  b0 04          mov     al,0x4
c:  b3 01          mov     bl,0x1
e:  59             pop     ecx
f:  b2 05          mov     dl,0x5
11: cd 80          int     0x80
13: 31 c0          xor     eax,eax
15: b0 01          mov     al,0x1
17: 31 db          xor     ebx,ebx
19: cd 80          int     0x80
0000001b <ender>:
1b: e8 e2 ff ff ff call    2 <starter>
20: 68 65 6c 6c 6f push    0x6f6c6c65
```

Extracting the bytes gives us the shellcode:

```
\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\x
b2\x05\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\x
f\x68\x65\x6c\x6c\x6f
```

Test Shellcode (helloworld_shellcode.c)

```
# include <stdio.h>
# include <string.h>
# include <unistd.h>
# include <sys/mman.h>

# define EXEC_MEM ((void *) 0x80000000)

char shellcode[] = "\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x05\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x66";

int main() {
    mmap(EXEC_MEM, 0x1000, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_ANONYMOUS | MAP_FIXED | MAP_PRIVATE, -1, 0);
    memcpy(EXEC_MEM, (void *)shellcode, strlen(shellcode)+1);
    (*(int (*)())EXEC_MEM)();
    return 0;
}
```

```
gcc helloworld_shellcode.c -m32 -o helloworld_shellcode
```

```
~/ss2023/class6 >>> ./helloworld_shellcode
hello%
```

- Taking some shellcode from Aleph One's 'Smashing the Stack for Fun and Profit'

```
shellcode =  
("\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b" +  
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40xcd" +  
"\x80\xe8\xdc\xff\xff\xff/bin/sh")
```

Q & A

Calling Convention

Two Questions

- Q: When a function finished, how to handle the parameter left in the stack.

0012FF70	00000002	
0012FF74	00000001	
0012FF78	0012FFC0	
0012FF7C	00401250	RETURN to StackFra.00401250 from StackFra.00
0012FF80	00000001	
0012FF84	00342EE0	
0012FF88	00342F40	
0012FF8C	6945F8F1	
0012FF90	7C910228	ntdll.7C910228

A: We don't care...

- Q: When a function finished, how change the ESP value?

A: ESP should be restored to the previous value

Standard C Calling Conventions

- **Calling conventions** are a standardized method for functions to be implemented and called by the machine.
- A calling convention specifies the method that a compiler sets up to access a subroutine.
- There are three major calling conventions that are used with the C language on 32-bit x86 processors:
 - CDECL
 - STDCALL,
 - FASTCALL.

- The C language, by default, uses the CDECL calling convention
- In the CDECL calling convention the following holds:
 - Arguments are passed on the stack in Right-to-Left order, and return values are passed in eax.
 - The **calling function cleans the stack**. This allows CDECL functions to have *variable-length argument lists*.

STDCALL

```
_cdecl int MyFunction1(int a, int b)
{
    return a + b;
}
```

and the following function call:

```
x = MyFunction1(2, 3);
```

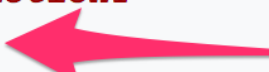
- The C language, by default
- In the CDECL calling convention
 - Arguments are passed on the stack and are passed in `eax`.
 - The **calling function cleans up** the stack after the function call.

These would produce the following assembly listings, respectively:

```
_MyFunction1:
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
pop ebp
ret
```

and

```
push 3
push 2
call _MyFunction1
add esp, 8
```



- STDCALL, also known as "WINAPI" (and a few other names, depending on where you are reading it) is used almost exclusively by Microsoft as the standard calling convention for the Win32 API.
 - STDCALL passes arguments right-to-left, and returns the value in eax.
 - The **called function cleans the stack**, unlike CDECL. This means that STDCALL doesn't allow variable-length argument lists.

STDCALL

- STDCALL, also known as "WINAPI" (on where you are reading it) is used as the standard calling convention for Windows API functions.
 - STDCALL passes arguments right-to-left.
 - The **called function cleans the stack** after it returns. STDCALL doesn't allow variable-length arguments.

RET 8 → RET + POP 8 Byte

Consider the following C function:


```
_stdcall int MyFunction2(int a, int b)
{
    return a + b;
}
```

and the calling instruction:

```
x = MyFunction2(2, 3);
```

These will produce the following respective assembly code fragments:

```
:_MyFunction2@8
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
pop ebp
ret 8
```



and

```
push 3
push 2
call _MyFunction2@8
```

FASTCALL

- The FASTCALL calling convention **is not completely standard** across all compilers, so it should be used with caution.
- The calling function most frequently is responsible for cleaning the stack, if needed.