System Calls

W4118 Operating Systems I

columbia-os.github.io

Credits to Jae

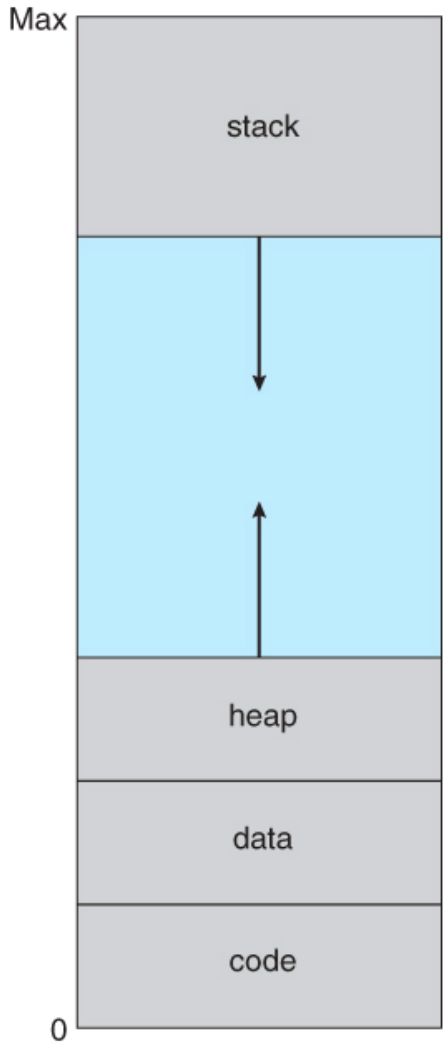
Allocating Memory

Run **`malloc.c`**, malloc() does not appear in the strace, why?

`brk()`, changes the location of the program break, which define the end of the process's data segment (i.e., the top of the heap)

`brk(NULL)` gets the current process break

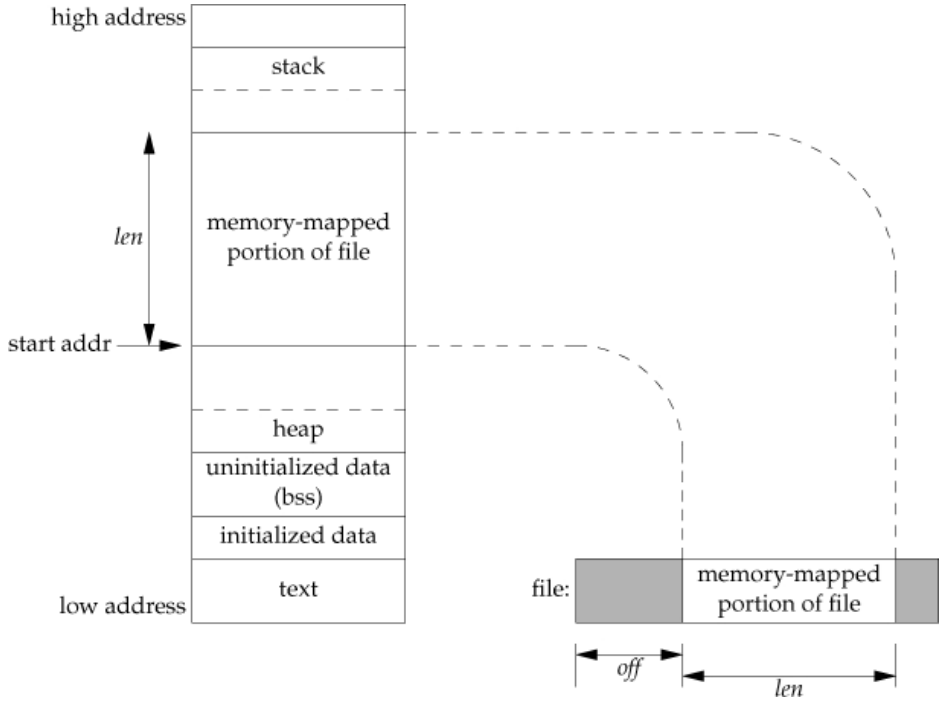
`brk(addr)` sets the break to **`addr`**



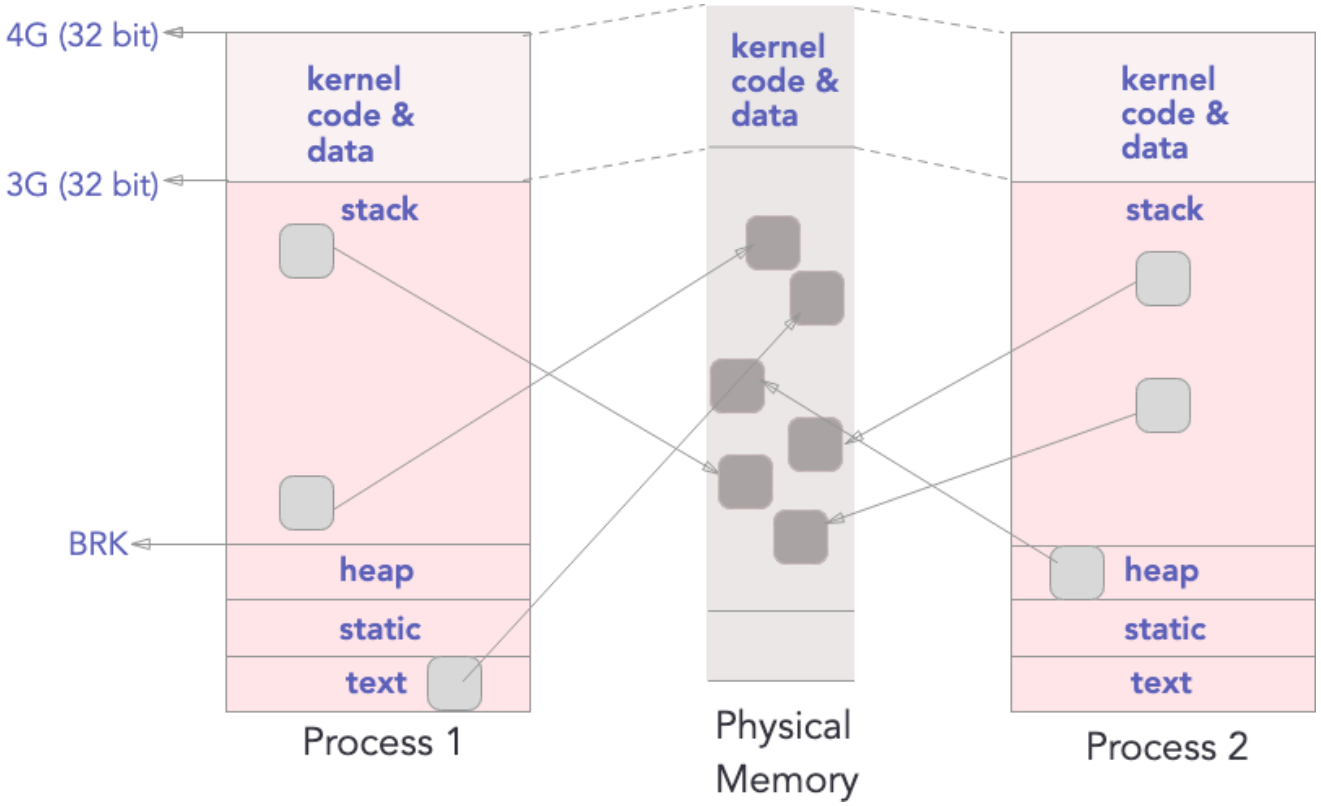
File-backed mappings

Program setup involved mapping in the C standard library:

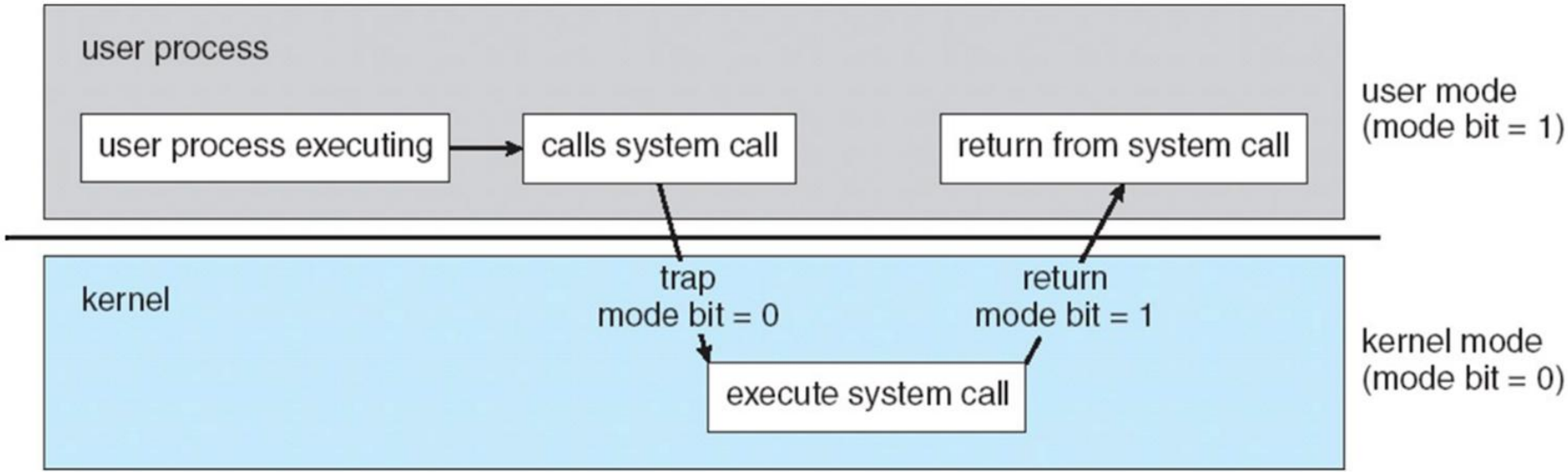
```
openat() = fd

mmap(..., fd, …)

close(fd)
```



Full Virtual Address Space



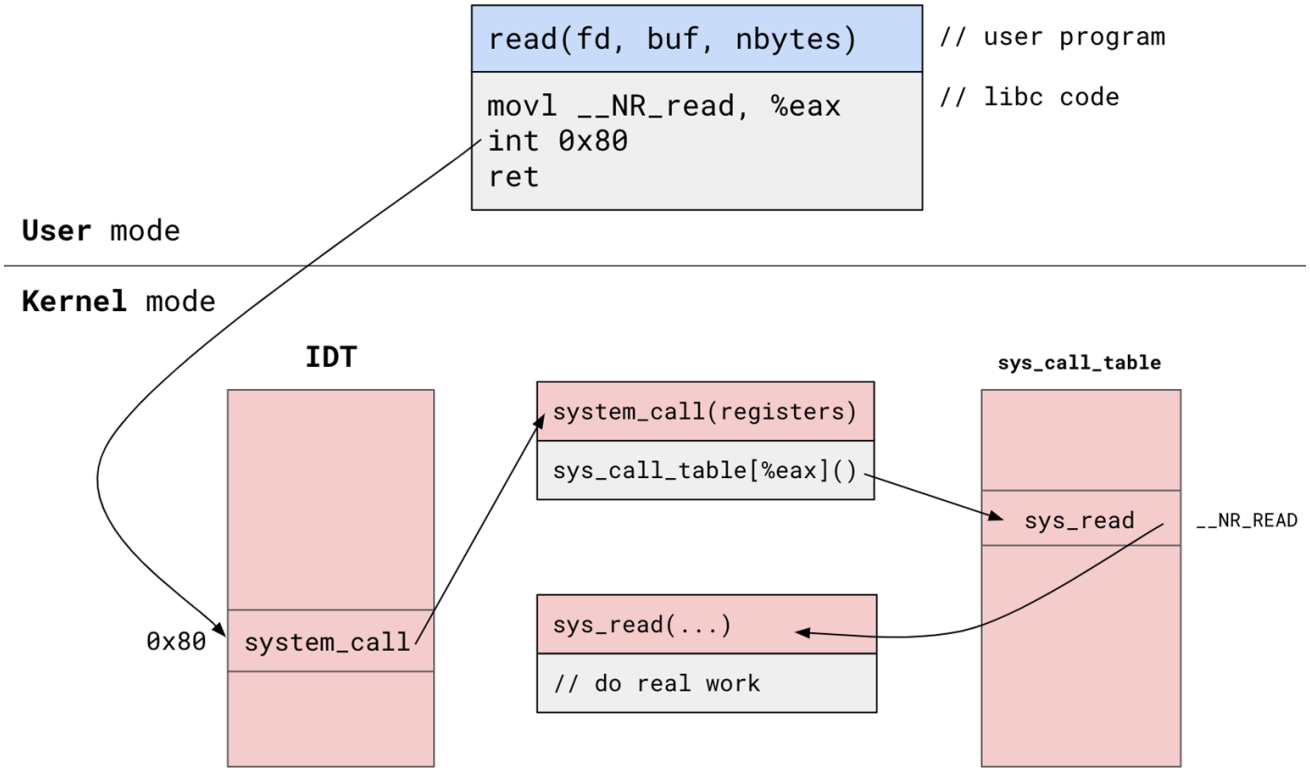
Processor Modes



Interrupts

- Hardware interrupts
  - asynchronous
  - e.g. network packet arrival, timer, key press, mouse click
- Exceptions/Faults
  - synchronous
  - e.g. dividing by zero, page fault
- Software interrupts
  - synchronous
  - x86 assembly int: raise software interrupt
  - e.g. syscall (int 0x80), debugger

Linux System Call Dispatch



Linux System Call Dispatch Notes

- `int 0x80` is how syscalls were invoked in 32-bit x86, e.g., x86-64 has a `syscall` instruction
- See system call handler and syscall dispatch under /arch/x86/entry
- See system call table here
- See `sys_read()` implementation in /fs/read_write.c
- Check vdso for even faster “system calls”

System Call Parameters

- Syscall parameters are passed via registers
  - Max arg size is the register size
  - Use struct pointer to pass in more/larger arguments (e.g. struct sigaction)

Need to validate memory! Why?

How are syscalls implemented ?

- First, one has to understand how arguments in any regular function call are passed.
- For this, a calling code convention is defined.
- Typically, arguments are passed through registers
  (sometimes as offsets on the stack)
- Those registers can be modified by the function called, any other registers most be saved and restored by the callee function
  - Volatile register (args,stackptr) and non-volative registers (callee must save and restore)
- Generally referred to as ABI: Application Binary Interface
- Syscalls are simply an extension on this. All compilers need to agree on ABI or code will no cooperate/work.

arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7
arm/OABI	a1	a2	a3	a4	v1	v2	v3
arm/EABI	r0	r1	r2	r3	r4	r5	r6
arm64	x0	x1	x2	x3	x4	x5	-
blackfin	R0	R1	R2	R3	R4	R5	-
i386	ebx	ecx	edx	esi	edi	ebp	-
ia64	out0	out1	out2	out3	out4	out5	-
mips/o32	a0	a1	a2	a3	-	-	-
mips/n32,64	a0	a1	a2	a3	a4	a5	-
parisc	r26	r25	r24	r23	r22	r21	-
s390	r2	r3	r4	r5	r6	r7	-
s390x	r2	r3	r4	r5	r6	r7	-
sparc/32	o0	o1	o2	o3	o4	o5	-
sparc/64	o0	o1	o2	o3	o4	o5	-
x86_64	rdi	rsi	rdx	r10	r8	r9	-
x32	rdi	rsi	rdx	r10	r8	r9	-

arch/ABI	instruction	syscall #	retval	Notes
arm/OABI	swi NR	-	a1	NR is syscall #
arm/EABI	swi 0x0	r7	r0	
arm64	svc #0	x8	x0	
blackfin	excpt 0x0	P0	R0	
i386	int $0x80	eax	eax	
ia64	break 0x100000	r15	r8	See below
mips	syscall	v0	v0	See below
parisc	ble 0x100(%sr2, %r0)	r20	r28	
s390	svc 0	r1	r2	See below
s390x	svc 0	r1	r2	See below
sparc/32	t 0x10	g1	o0	
sparc/64	t 0x6d	g1	o0	
x86_64	syscall	rax	rax	See below
x32	syscall	rax	rax	See below

Trap/SC instruction

Excellent writeup: https://lwn.net/Articles/604287/

syscall implementation (user side)

/usr/include/asm-generic/unistd.h
unistd.h:extern ssize_t pread64 (int __fd, void *__buf, size_t __nbytes)

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <unistd.h>
#include <sys/syscall.h>   /* For SYS_xxx definitions */

long syscall(long number, ...);
```

Definition agreed upon by libc and kernel
➔ ABI is known. Compiler assembles args

```
0000000000400596 <main>:
  400596:       55                      push   %rbp
  400597:       48 89 e5                mov    %rsp,%rbp
  40059a:       48 83 ec 70             sub    $0x70,%rsp
  40059e:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  4005a5:       00 00
  4005a7:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  4005ab:       31 c0                   xor    %eax,%eax
  4005ad:       48 8d 45 a0             lea    -0x60(%rbp),%rax
  4005b1:       ba 50 00 00 00          mov    $0x50,%edx
  4005b6:       48 89 c6                mov    %rax,%rsi
  4005b9:       bf 00 00 00 00          mov    $0x0,%edi
  4005be:       e8 ad fe ff ff          callq  400470 <read@plt>
  4005c3:       89 45 9c                mov    %eax,-0x64(%rbp)
  4005c6:       8b 45 9c                mov    -0x64(%rbp),%eax
  4005c9:       48 8b 4d f8             mov    -0x8(%rbp),%rcx
  4005cd:       64 48 33 0c 25 28 00    xor    %fs:0x28,%rcx
  4005d4:       00 00
  4005d6:       74 05                   je     4005dd <main+0x47>
  4005d8:       e8 83 fe ff ff          callq  400460 <__stack_chk_fail@plt>
  4005dd:       c9                      leaveq
  4005de:       c3                      retq
  4005df:       90                      nop
```

ABI: Application Binary Interface

```
/* fs/read_write.c */
#define __NR3264_lseek 62
__SC_3264(__NR3264_lseek, sys_llseek, sys_lseek)
#define __NR_read 63
__SYSCALL(__NR_read, sys_read)
#define __NR_write 64
__SYSCALL(__NR_write, sys_write)
#define __NR_readv 65
__SC_COMP(__NR_readv, sys_readv, compat_sys_readv)
#define __NR_writev 66
__SC_COMP(__NR_writev, sys_writev, compat_sys_writev)
#define __NR_pread64 67
__SC_COMP(__NR_pread64, sys_pread64, compat_sys_pread64)
#define __NR_pwrite64 68
__SC_COMP(__NR_pwrite64, sys_pwrite64, compat_sys_pwrite64)
#define __NR_preadv 69
__SC_COMP(__NR_preadv, sys_preadv, compat_sys_preadv)
#define __NR_pwritev 70
__SC_COMP(__NR_pwritev, sys_pwritev, compat_sys_pwritev)

/* fs/sendfile.c */
#define __NR3264_sendfile 71
__SYSCALL(__NR3264_sendfile, sys_sendfile64)
```

#define __SYSCALL(number)
syscall(number...)

syscall is implemented as assembler largely taking the arguments already in the right registers and TRAP-ing into the kernel.

syscall implementation (kernel side)

- Kernel defines a table (using the compiler help)

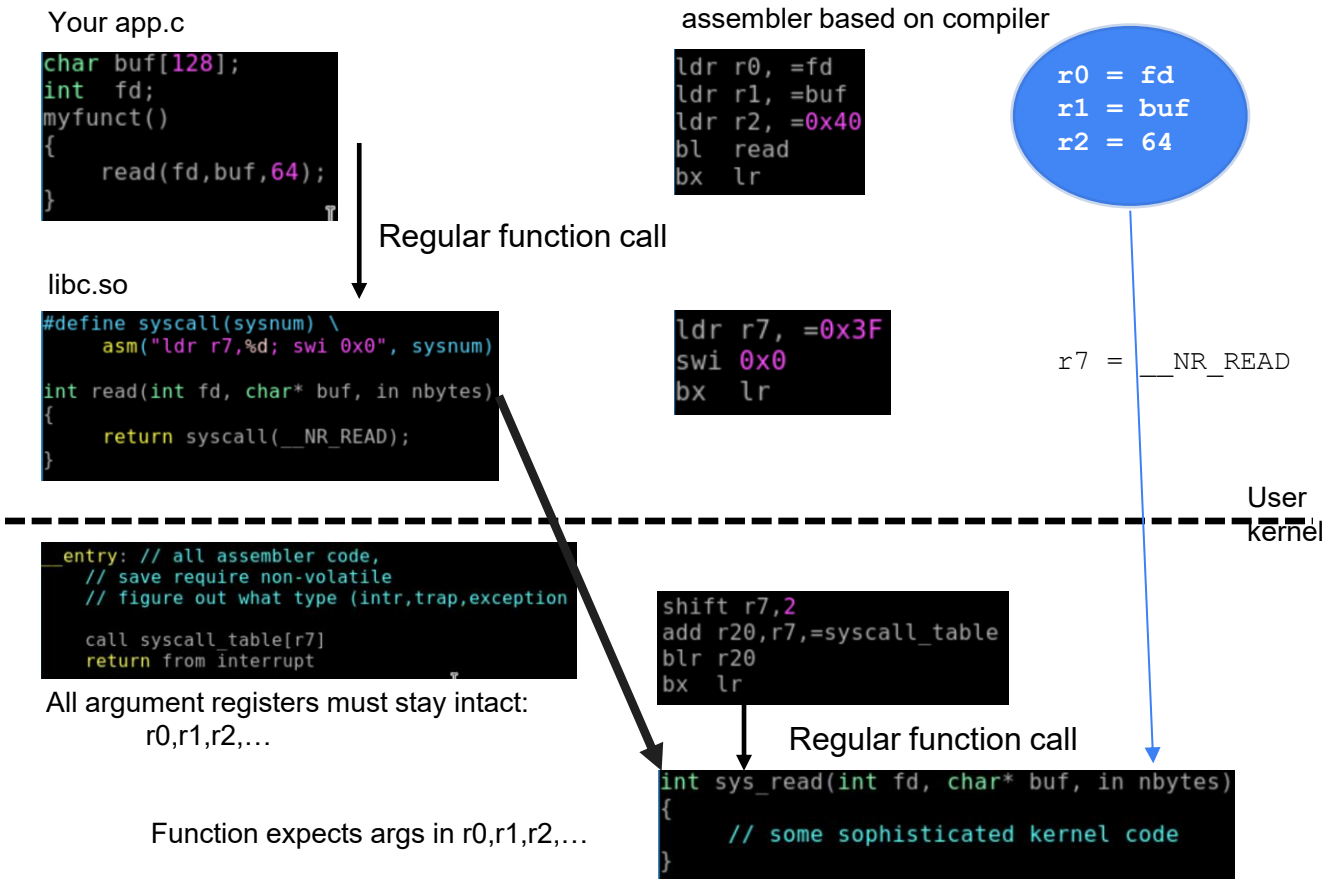
```
int syscall_table [ __NR_SYSCALL_MAX ] = {
      :
     [ __NR_READ]  = sys_read,
     [ __NR_WRITE] = sys_write,
      :
}
```

The compiler does the magic and associates the syscall number with the kernel internal function

- On system call trap, architecture automatically and immediately enters kernel mode and runs a small piece of assembler code that is stored at a machine register address set by the OS at boot time.

- Said trap assembler code (aka interrupt handler) does the following:
  - Checks the syscall number in well known register (see ABI) to be in range
  - Assembler equivalent:
    - Change stack to kernel (more on this in a bit)
    - All arguments are already in right place thanks to the ABI and the compiler's help
    - "Call/jmp" to syscall_table[registers.syscall_number]; // see ABI definition
    - After return from ^^^^, switch back from kernel stack to user stack and RFI (return from kernel mode).

Putting it together (based on ARM/EABI)



System Call Parameters

- Syscall parameters are passed via registers
  - Max arg size is the register size
  - Use struct pointer to pass in more/larger arguments (e.g. struct sigaction)

Need to validate memory!

Example, `read()/write()`: What if the buffer actually points to kernel memory?

- Pointer points to a region of memory in user-space, not kernel-space.
- If reading/writing/executing, memory must be marked readable/writable/executable accordingly

`copy_to_user()`
`copy_from_user()`

Why not to write a system call?

Why not to write a system call?

- You use a syscall number
- You need to maintain it forever
- You need to register and support it for each architecture
- Not easily usable from scripts or the filesystem
- You cannot maintain it easily outside the kernel tree
- It’s an overkill!

Alternatives:

- Use a special file and manipulate it instead