Syscalls

Written by Alan Ton.

Also known as system calls, syscalls serve two primary purposes:

1. **Provide a layer of abstraction between hardware and user-level processes**. For instance, when reading or writing to a file, processes don't need to be concerned about the type of disk, media, or filesystem.
2. **Ensure system security and stability**. Since the kernel is the middle-man between system resources and user-space, the kernel can arbitrate access based on permissions, users, etc.

Section 2 of the *man pages* defines the system call interface (e.g. write, sbrk) and Section 3 defines general-purpose library functions (e.g. printf, malloc). All of these functions are written in C. To read a function's documentation, type `man <section-number> <function-name>` in your terminal.

Ex 1) `man 2 read`

Ex 2) `man 3 scanf`

The functions in Section 3 may invoke one or more of the functions in Section 2. For instance, `printf` uses the `write` system call to output a string. Functions in Section 2 (i.e. syscalls) are entry points into the kernel.

When a syscall is invoked in user-space, the function does the following three things:

1. **Tells the kernel which syscall to execute.** Each system call is registered with a unique identifying number called the syscall number. The syscall number is pushed to the `eax` register.
2. **Loads syscall parameters**. The parameters of the syscall are pushed into registers. In x86, `ebx, ecx, edx, esi`, and `edi` contain the first five arguments.
3. **Generates a software interrupt.** In x86, this is `int $0x80`.
   a. The `int` instruction raises an interrupt that triggers the switch into kernel-mode.
   b. `$0x80` (or 128 in decimal) indicates the interrupt handler for syscalls (i.e. system call handler)

After switching to kernel-mode, the system call handler is executed and will do three tasks:

1. **Verify the parameters**. It would be troublesome if users can pass invalid input into the kernel without restraint. An important check is to validate pointers passed in as arguments. This includes:
   a. Pointer is not NULL.
   b. Pointer points to memory in user-space.
   c. Pointer points to a region in the calling process's address space.
2. **Execute the syscall**. The kernel will invoke the function corresponding to the value in the `eax` register. The result of the syscall is pushed to the `eax` register. If anything fails, then a value indicating failure is pushed instead.
3. **Return to user-space**. This is performed via the `iret` instruction in x86.

You will be writing the system call handler for PintOS in Project 1. In PintOS, the syscall handler is registered as interrupt vector `$0x30` (or 48 in decimal). Additionally, the parameters for syscalls are pushed onto the stack (`esp` register) of the user process, so you don't need to look at the `ebx, ecx, edx, esi`, or `edi registers.` Take a look at `./userprog/syscall.c`, `./threads/interrupt.c`, and `./threads/intr_stubs.S` to dive deeper.

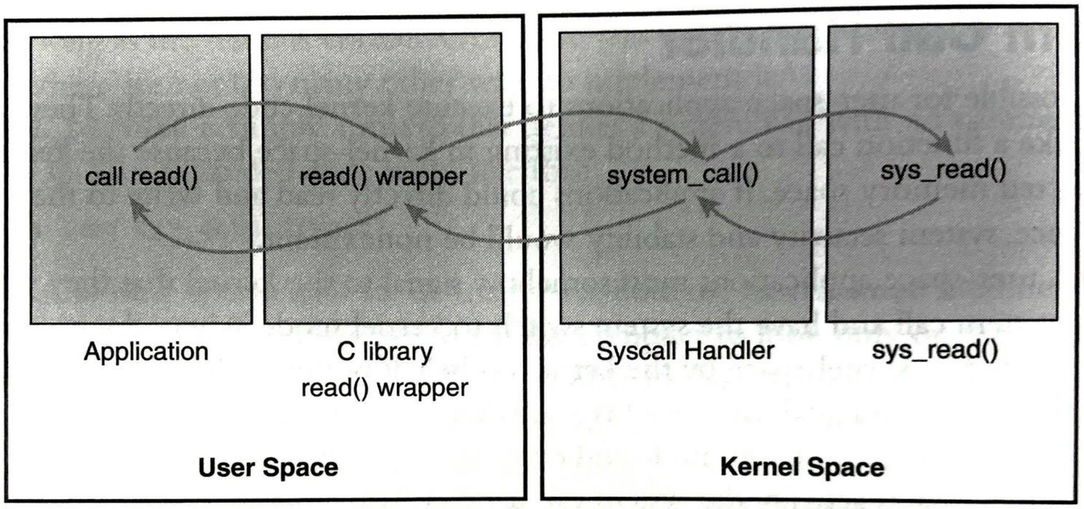

Figure 5.2 Invoking the system call handler and executing a system call.