System calls

program can't do everything by itself
need OS help (read/write to disk, ...)

System calls vs. library calls

write(stdout, buf, len)

write(stdout, "a", 100)

- Bug??

write(stdout, "44.44", 5)

- Print up to two decimal places
- Easy to setup?

printf(manyFormats)

printf("a")

- Length is implicit

printf("%5.2f", float)

- Nice formatting supports

Other examples:

- Java OutputStream

Libraries user friendly

- ***Eventually will make system calls!***

internally makes system call

There are lots of system calls

each system fn is an exception, then need hella exceptions
→ but exception table is limited: 256
→ 2 level indirection

2 level indirection

① 1st level use only (syscall handler exception 0x80

② 2nd level uses syscall num & syscall arguments

With only 1 handler, how to route different system calls?

- e.g. open, read, write, exit, …

2nd level: Use syscall number and syscall arguments

- Different OSes (Linux, Mac, etc.) will have a different syscall table and syscall arguments (the compilers will take care of this)

In OS kernel:

```
syscall_handler(num, args) {
 if(num==0) sys_read(args);
  …
 if(num==1) sys_write(args);
  …
}
```

Number	Name	Description	Number	Name	Description
0	read	Read file	33	pause	Suspend process until signal arrives
1	write	Write file	37	alarm	Schedule delivery of alarm signal
2	open	Open file	39	getpid	Get process ID
3	close	Close file	57	fork	Create process
4	stat	Get info about file	59	execve	Execute a program
9	mmap	Map memory page to file	60	_exit	Terminate process
12	brk	Reset the top of the heap	61	wait4	Wait for a process to terminate
32	dup2	Copy file descriptor	62	kill	Send signal to a process

Figure 8.10 Examples of popular system calls in Linux x86-64 systems.

Making write system call

- How does a program make a system call?
- **Step 1: Set the arguments in registers (e.g. rax, rdi, ..)**
  - **rax:** the system call number (4 → write)
  - **The rest (rdi, rsi, rdx):** used by the specific system calls
    - write() accepts 3 arguments
- **Step 2: Make syscall (interrupt 0x80)**
  - *(The OS automatically copy eax-edx to "num" and "args")*

```
8   main:
      First, call write(1, "hello, world\n", 13)
9     movq $1, %rax          write is system call 1
10    movq $1, %rdi          Arg1: stdout has descript
11    movq $string, %rsi     Arg2: hello world string
12    movq $len, %rdx        Arg3: string length
13    syscall                Make the system call
```

```
// Inside OS:
syscall_handler(num, args) {
  if(num==0) sys_read(args);
  …
  if(num==1) sys_write(args);
  …
}
```

User & Kernel Mode

CPU runs User (my code) code → user mode ☆

CPU runs OS kernel code → kernel mode ☆

Youtube → both
infinite i++ →

browser + network
loop code by user

how does CPU know if in system or user mode?
doesn't

flip internal register whether to use syscall (int 0x80)

Why care?
kernal mode is hacker's dream → OS

less system calls → quicker → why we use library calls

want to reduce user/kernal mode crossing
jump between types of memory & register prepration

man 3 f → see if it's a sys fn

Processes

Program vs. Process

Motivational example

- "myprog" opens file.txt
- Two shells run the programs but in different directories

Which one works?

- Shell 1 or shell 2 ?
- Why?
- Yes, but specifically why ?
  - What is inside "sys_open()" that returns the error?

Program becomes process(es)

- Each process has its own process state

```
My files:
/code/myprog
/data/file.txt
```

```
myprog:
main() {
  fd = open("file.txt")
  if (fd == -1) ERROR;
}
```

```
Shell 1:
% cd /code
% ./myprog
...
Success/Fail?
```

→ can't see .txt file

```
Shell 2:
% cd /data
% /code/myprog
...
Success/Fail?
```

→ file is now visible

program → process

same program, runs under different states

instance of running a program → process

Program vs. Process

Program:

- A collection of static code and static data
- E.g. /code/myprog, /bin/ls

Process

- Instance of a computer program that is being executed
- Has its own context
  - Context: its code and data stored in memory (stack, registers, program counter, heap, etc.)
- Has private virtual address space (more later in VM)

Process != program

- In previous example: the two processes of the same program have different current working directories (CWD)
- CWD is inherited from the parent process (i.e. the shell in this example) – more when we discuss fork()

Shell 1:
% cd /code ? CWD
% ./myprog

Shell 2:
% cd /data ? CWD
% /code/myprog

OS creates Process Control Board (PCB) to hold state of every process

each process has its own stack