Week 05MW: System Calls

Thuy Vu

Reminders

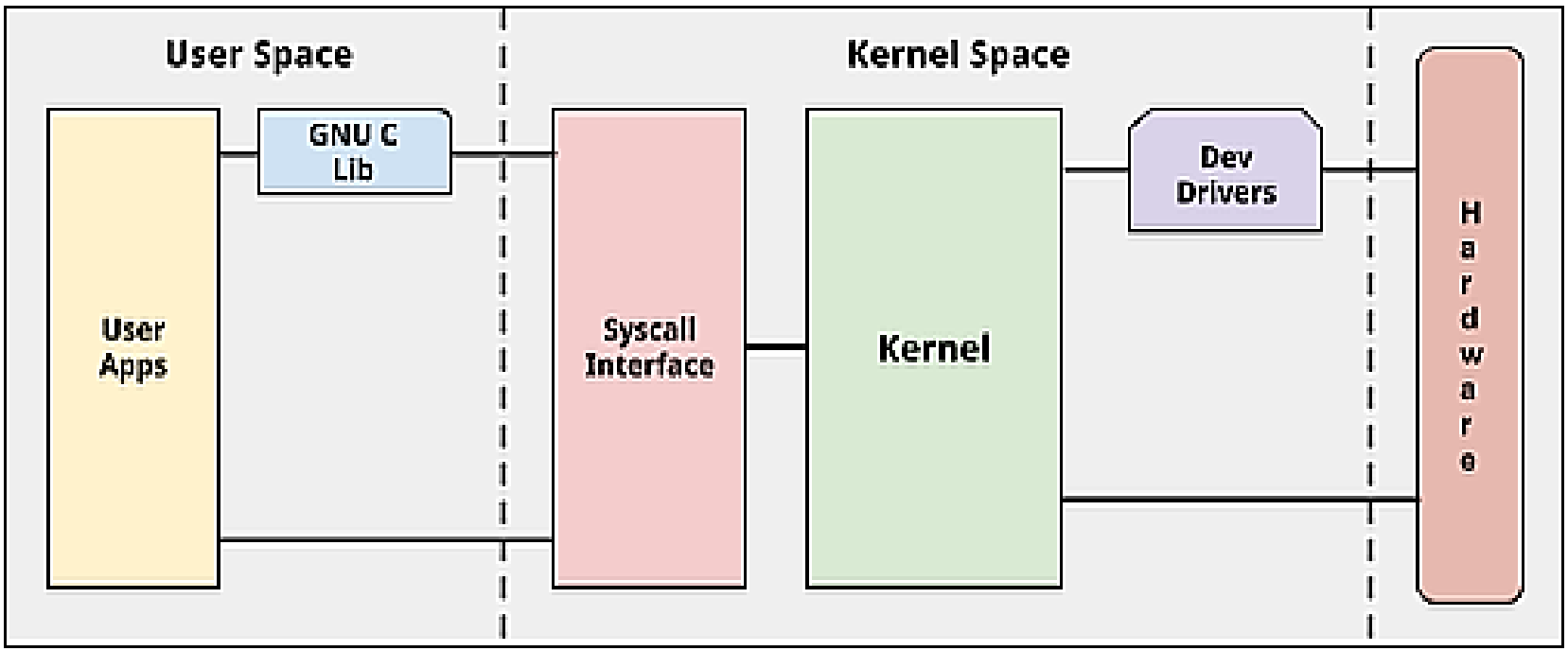
- Assignment 4
  - `web.cs.ucla.edu/classes/winter17/cs35L/assign/assign4.html`
  - Time due 23:55 this Friday, February 10 (tomorrow!)
- Assignment 5
  - `web.cs.ucla.edu/classes/winter17/cs35L/assign/assign5.html`
  - Time due 23:55 this Friday, February 17 (in 8 days!)

- Assignment 10 Sign-up
  - `https://goo.gl/794MQM`

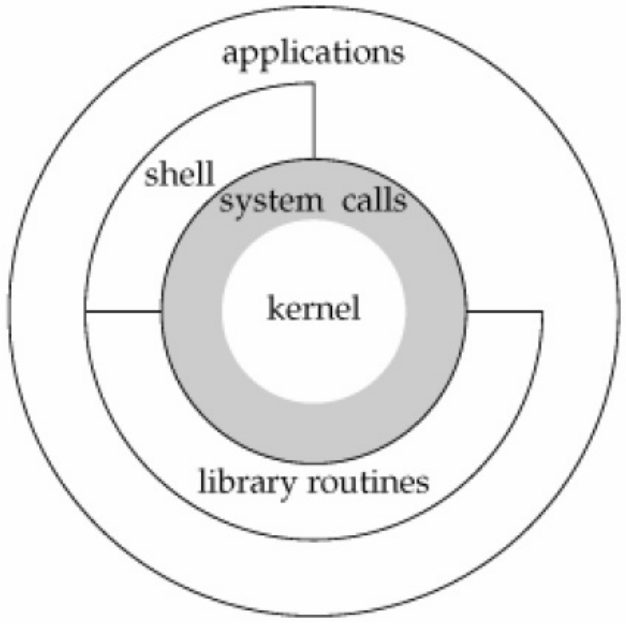
Kernel

- **kernel** is the core of the OS
  - interface between hardware and software
  - controls access to system resources: memory, I/O, CPU
  - ensure protection and fair allocations
- **user space** – where normal user processes run
  - limited access to system resources: memory, I/O, CPU

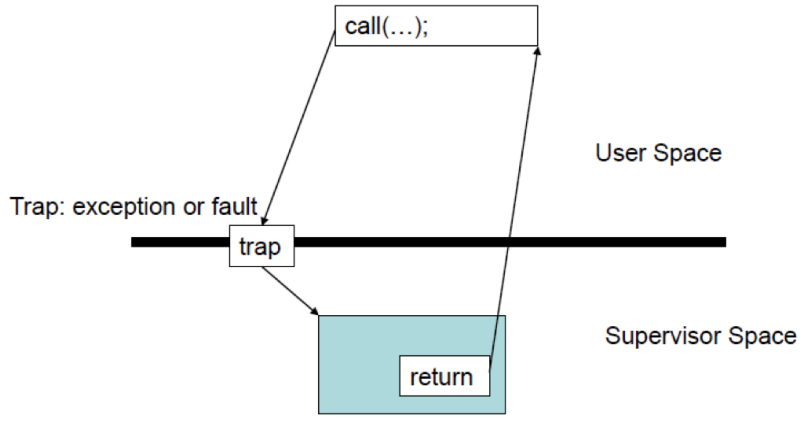
  → need a manager, the kernel
- **kernel space** – stores the code of the kernel, which manages processes
  - prevent processes messing with each other and the machine
  - only the kernel code is trusted



System Calls

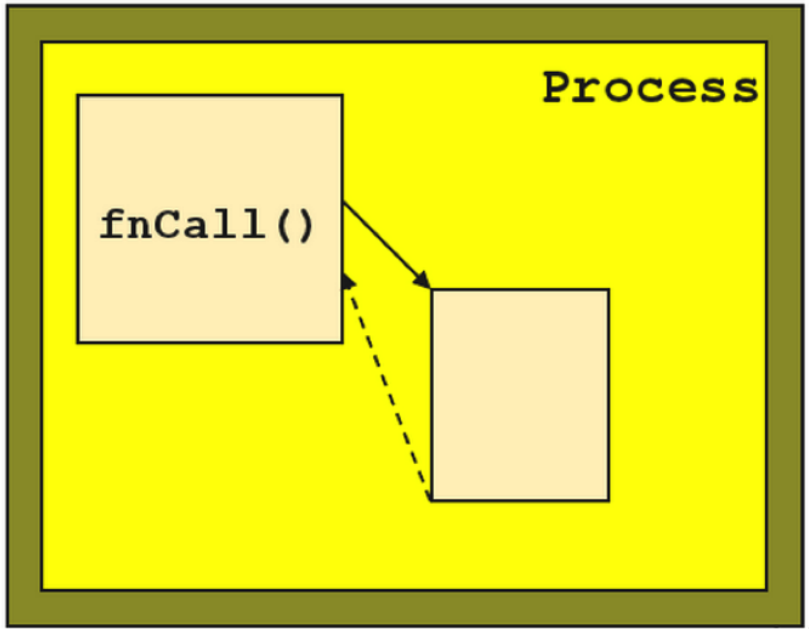


system calls – a part of kernel accessible from user space (applications)



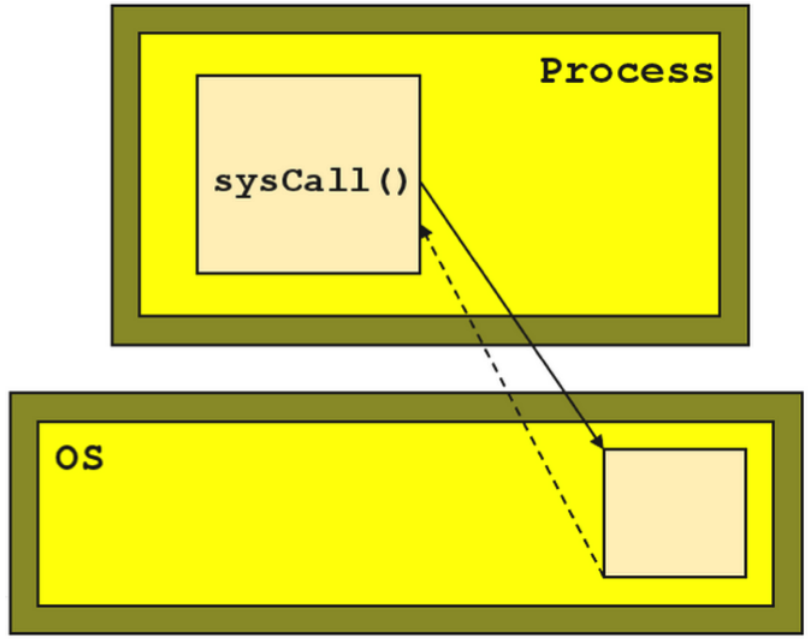
the executable is interrupted and its control is passed to the kernel to perform if valid

Function Call versus System Call



function call – caller and callee are of the same process, same "domain of trust"

`getchar`, `putchar`, `fopen`, `fclose`



system call – transfer control from untrusted user process to trusted OS thus, expensive performance-wise

`read`, `write`, `open`, `close`

1. interrupt process and saves the state
2. OS takes control of CPU and checks if valid
3. perform requested action
4. restore the state, switch back to user mode
5. give CPU control back to process

What Can Go Wrong?

- library functions invoke system calls
  - internally
  - *efficiently*, $\sim$ minimizing system calls
  - e.g. cat stdin

```
#include <stdio.h>
.  .  .  .  .
int byte = getchar();
while (byte != EOF) {
     putchar(byte);
     byte = getchar();
}
```

```
#include <unistd.h>
.  .  .  .  .
char buffer[1];
while (read(0,buffer,1)>0) {
     write(1,buffer,1);
     //nothing here
}
```

- we will empirically prove that in this assignment
  - transliteration `tr2b` and `tr2u`
  - sorting encrypted text `sfrob.c`

Buffered versus Unbuffered I/O

- tr2b – using `getchar` and `putchar`
- tr2u – using `read` and `write` (`nbyte` should be 1)

- *from* and *to* are byte strings of the same length
- copies standard input to standard output and transliterates bytes
- `[`, `-`, and `\` have no special meaning in the operands

Encrypted Sort Revisited

- `sfrob` → `sfrobu`: system-call version of `sfrob`
  - use system calls to read/write from/to stdin/stdout
- `sfrob` → `sfrobs`: shell-script version of `sfrob`
  - use `tr` and `sort` to sort encrypted files through pipeline
  - expected to be long
- "`-f`": ignore case while sorting, by using the standard `toupper`
- compare the performance

How to compare?

1. `strace` to trace system calls
   - `strace -o output -c cat note`
2. `time` to time a program
   - `time cat note`

What to check?

1. if code compiled
2. message for invalid arguments
3. exit *properly*
4. functionalities
5. memory allocation: malloc/realloc/free
6. EOF, input, output, null, empty, trailing newline, case-sensitivity
7. formatting