Section 1: Lab introduction

CSE 451 20wi

C Language Review

Please look at “Throwbacks to C”

Lab overview

1. OSV is a new operating system for teaching
2. We use `qemu` (quick emulator) to simulate a computer
3. We will have 4 labs, with the last one as an open ended lab

Administrative

- Lab1 was released yesterday
- Design doc already provided: no due date (usually one week for other labs)
- The complete lab1 will be due 2 weeks from now

Setting Up Environment

We suggest you to use attu.

See lab1.md

1. Clone the project from gitlab
2. Create a new private repo on gitlab
3. Push to the new repo
4. Add all staff as developer (Settings >> Members)

Test your environment

1. Remember to use `export PATH=/cse/courses/cse451/17au/bin/x86_64-softmmu:$PATH` each time, or add it to ‘.bashrc’ to automate this process
2. Run `make qemu`

How to Submit Your Lab

1. Run `python3 ./test.py 1` to test your code
2. Add a tag on the version you want to submit: git tag end_lab1
    - You can safely work on later labs because new commits won’t affect this tag
3. When pushing your work, add the option ` --tags`
4. Check that all TAs are added as developers
5. Check on Gitlab that the tag is uploaded

Introduction to GDB

- In one terminal: make qemu-gdb
- In the other: gdb
- Copy the arch/x86_64/gdbinit as your `~/.gdbinit` as prompted

See the cheat sheet for details

```
Break main
Continue
Control-x 1
Focus cmd
Focus src
Next
Print main
Break kernel/main.c:42
Break *0xffffffff80108cb5
Continue
x/10i main
x/10x main
```

Lecture review

- When should the CPU go to kernel mode?

- Interrupt
    - Timer / Disk / Network / User Input
- System call
    - Can the user pass in the address to kernel function?
- Exceptions
    - Unknown instruction / page fault / privileged instruction / divide by zero

How do we make sure that the user is unaware of interrupt / switch between processes?

In other words, how do we enter/exit trap?

- The trap/syscall will automatically save/restore important registers (e.g. rip) from/to stack so that the trap handler can safely run
- The trap handler is responsible for saving/restoring other registers if needed

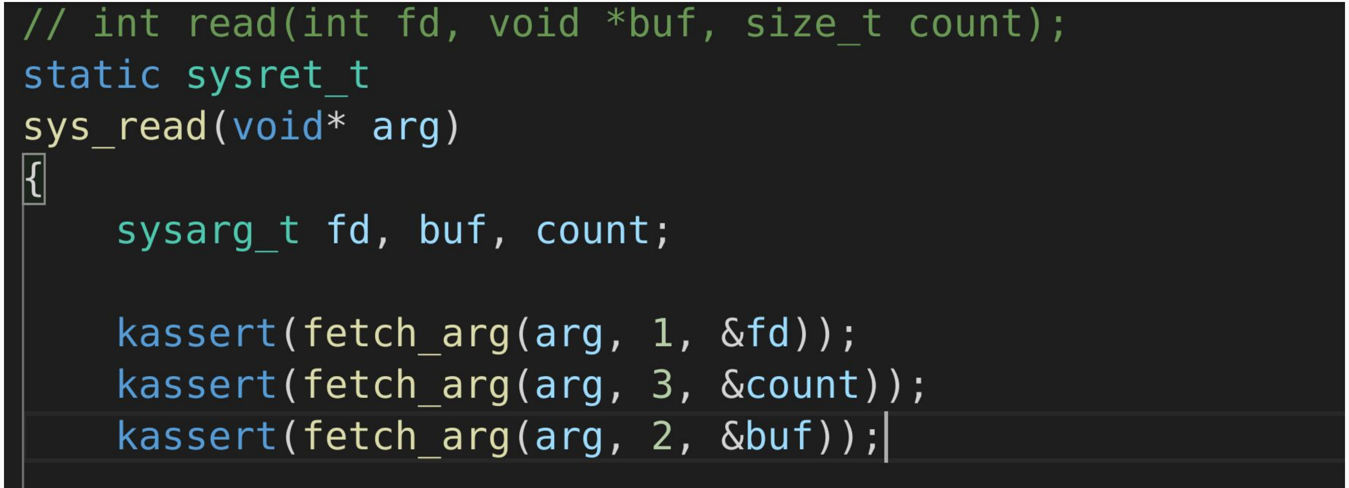
System Call

How does the user pass in arguments to the kernel?

How does the user pass in arguments to the kernel?

By convention, through registers (rdx, rsi ...)

We provided a helper function to extract the arumgents.

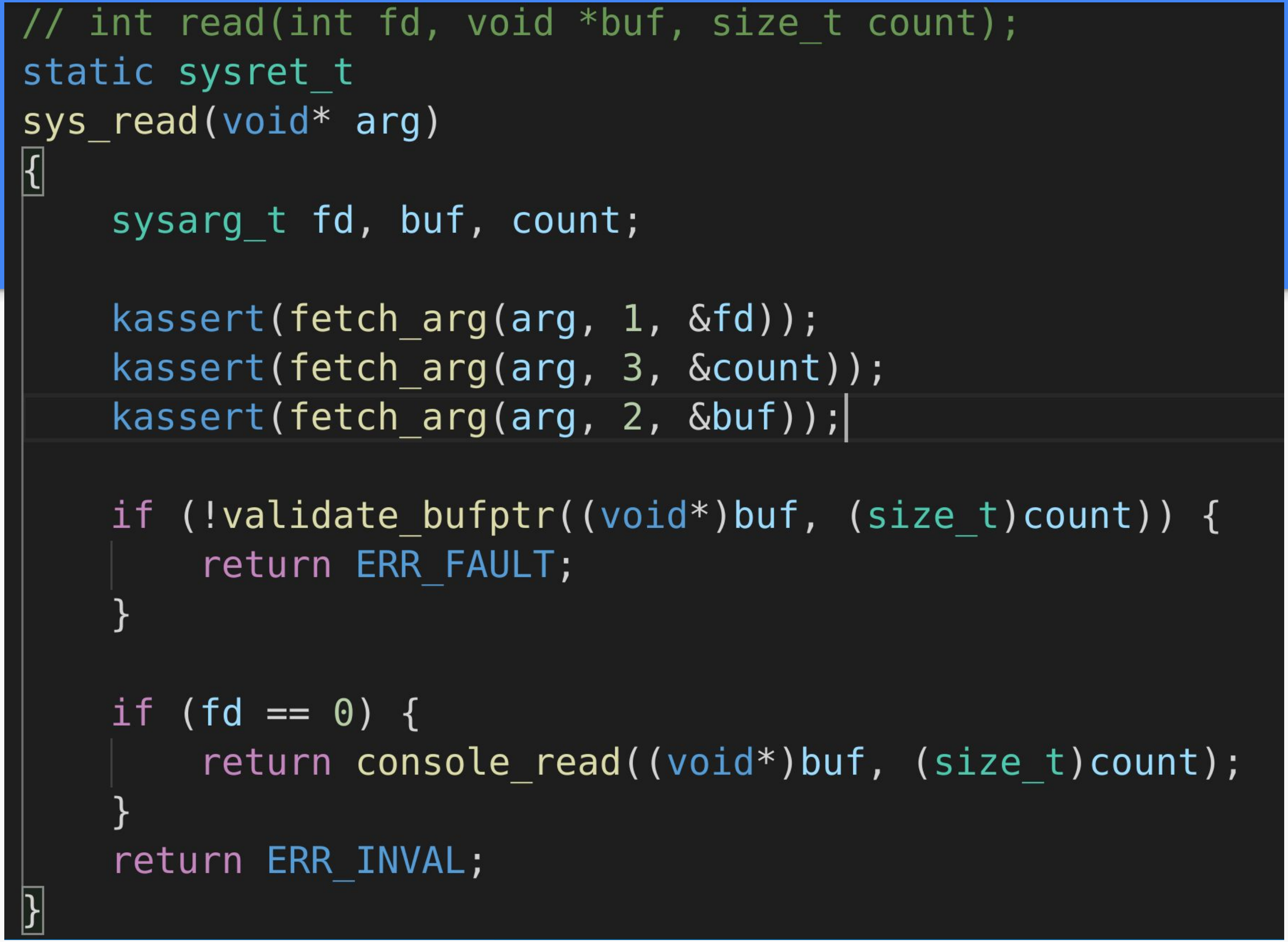


```
// int read(int fd, void *buf, size_t count);
static sysret_t
sys_read(void* arg)
{
    sysarg_t fd, buf, count;

    kassert(fetch_arg(arg, 1, &fd));
    kassert(fetch_arg(arg, 3, &count));
    kassert(fetch_arg(arg, 2, &buf));
```

Can we directly use the arguments?

No! The arguments might be invalid and possibly malicious!



```
// int read(int fd, void *buf, size_t count);
static sysret_t
sys_read(void* arg)
{
    sysarg_t fd, buf, count;

    kassert(fetch_arg(arg, 1, &fd));
    kassert(fetch_arg(arg, 3, &count));
    kassert(fetch_arg(arg, 2, &buf));

    if (!validate_bufptr((void*)buf, (size_t)count)) {
        return ERR_FAULT;
    }

    if (fd == 0) {
        return console_read((void*)buf, (size_t)count);
    }
    return ERR_INVAL;
}
```