13 Honors topics

> If I have seen further it is by standing on the sholders [sic] of Giants
>
> Sir Isaac Newton

This chapter contains the contents of some of the honors lectures (CS 296-41). These topics are aimed at students who want to dive deeper into the topics of CS 241.

The Linux Kernel

Throughout the course of CS 241, you become familiar with system calls - the userspace interface to interacting with the kernel. How does this kernel actually work? What is a kernel? In this section, we will explore these questions in more detail and shed some light on various black boxes that you have encountered in this course. We will mostly be focusing on the Linux kernel in this chapter, so please assume that all examples pertain to the Linux kernel unless otherwise specified.

What kinds of kernels are there?

As it stands, most of you are probably familiar with the Linux kernel, at least in terms of interacting with it via system calls. Some of you may also have explored the windows kernel (which we won't talk about too much in this chapter) or `Darwin`, the UNIX-like kernel for macOS (a derivative of BSD). Those of you who might have done a bit more digging might have also encountered projects such a `GNU HURD` or `zircon`.

Kernels can generally be classified into one of two categories, a monolithic kernel or a micro-kernel. A monolithic kernel is essentially a kernel and all of it's associated services as a single program. A micro-kernel on the other hand is designed to have a *main* component which provides the bare-minimum functionality that a kernel needs. This involves setting up important device drivers, the root filesystem, paging or other functionality that is imperative for other higher-level features to be implemented. The higher-level features (such as a networking stack, other filesystems, and non-critical device drivers) are then implemented as separate programs that can interact with the kernel by some form of IPC, typically RPC. As a result of this design, micro-kernels have traditionally been slower than monolithic kernels due to the IPC overhead.

We will devote our discussion from here onwards to focusing on monolithic kernels and unless specified otherwise, **specifically** the Linux kernel.

System Calls Demystified

By now, you probably know that system calls are an instruction that can be run by a program operating in userspace that *traps* to the kernel (by use of a signal) to perform behavior that the user is not allowed to do directly. This includes actions such as writing data to disk, interacting directly with hardware in general or operations related to gaining or relinquishing privileges (e.g. becoming the root user and gaining all capabilities).

In order to fulfill a user's request, the kernel will rely on `kernel calls`. Kernel calls are essentially the "public" functions of the kernel - functions implemented by other developers for use in other parts of the kernel. Here is a snippet for a kernel call's man page:

```
Name
```

```
kmalloc âĂŤ allocate memory
Synopsis
void * kmalloc (  size_t size,
    gfp_t flags);

Arguments

size_t size

    how many bytes of memory are required.
gfp_t flags

    the type of memory to allocate.

Description

kmalloc is the normal method of allocating memory for objects smaller
    than page size in the kernel.

The flags argument may be one of:

GFP_USER - Allocate memory on behalf of user. May sleep.

GFP_KERNEL - Allocate normal kernel ram. May sleep.

GFP_ATOMIC - Allocation will not sleep. May use emergency pools. For
    example, use this inside interrupt handlers.
```

You'll note that some flags are marked as potentially causing sleeps. This tells us whether or not we can use those flags in special scenarios, like interrupt contexts, where speed is of the essence, and operations that may block or wait for another process may never complete.

Containerization

As we enter an era of unprecedented scale with around 20 billion devices connected to the internet in 2018, we need technologies that help us develop and maintain software capable of scaling upwards. Additionally, as software increases in complexity, and designing secure software becomes harder, we find that we have new constraints imposed on us as we develop applications. As if that wasn't enough, efforts to simplify software distribution and development, like package manager systems can often lead to headaches of their own, leading to broken packages, dependencies that are impossible to resolve and other such environmental nightmares that have become all to common today. While these seem like disjoint problems at first, all of these and more can be solved by throwing `containerization` at the problem.

What is a container?

You can think of a container as something that is almost like a virtual machine. In some senses, containers are to virtual machines as threads are to processes. A container is a lightweight environment that shares resources and a kernel with a host machine, while isolating itself from other containers or processes on the host. You may have encountered containers while working with technologies such as `Docker`, perhaps the most well-known implementation of containers out there.

Linux Namespaces

Building a container from scratch

Containers in the wild: Software distribution is a Snap

Bibliography