Introduction into OS internals/Distributed computing

Author: Jed Salazar
jsalazar@pivotal.io

Before you begin

1. Install brew
   a. `/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"`
2. Install binaries
   a. `brew install {htop,pstree,wget}`

Machine level

- What is Linux? Linux is a *kernel*. What is a kernel? Kernels most notably schedule running executable programs called *processes.* Everything that's running on your system is run as a process by the kernel. The executable file on disk (e.g., `/usr/bin/ssh`) is called a program.

Processes

You can observe a running process by running `ps` in the Terminal on a Mac

```
ps aux|grep ssh
```

The kernel keeps track of the amount of resources any process uses. For example, when a process executes in the CPU it's effectively "stealing" CPU resources from other processes in the system waiting in line to execute (the kernel calls this the *run queue*). While the process is executing, the kernel has counters that keep track of process statistics such as the amount of time the process has spent in the CPU since launch (called *Time+* seen below)

```
  PID USER      PRI  NI  VIRT   RES S CPU% MEM%   TIME+  Command
43087 jedsalaza  24   0 3864M  159M ? 22.3  1.0 51:21.62 /Applications/Google Chrome.app/C
```

The following is output from the `htop` command which shows the processes running with the most resources from the system.

Different commands such as `ps` show different values

```
USER               PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
jedsalazar        3352   0.0  0.0  2448908     16   ??  S    29Aug17   0:00.02
/usr/bin/ssh-agent
```

Throughout their lifetime, processes exist in different *states* on the system (seen as `STAT` above). Below is a table of process states on a system

PROCESS STATE CODES

R `running or runnable (on run queue)`

Process is either being executed by the kernel or is in the run queue. You can query for the number of processes in the run queue by observing the *load average* on a Linux system. top provides a load average as seen below

Processes: 385 total, 2 running

Load Avg: 1.97, 1.87, 1.92

On a single core system (which doesn't really exist anymore but for the basis of this illustration please entertain the idea) a load average of >1 indicates a slow performing system since there's only one CPU and a >1 amount of processes running (or being queued to run). This suggests that *each CPU core has its own scheduler*. So a 4 core system with a load average of 2 is 50% utilized. A load average provides a decent figure for the average load of a system but can hide issues such as slow I/O.

D `uninterruptible sleep (usually IO)`

Generally occurs when a process is waiting for an I/O event (called `IO_WAIT`). Tools such as `iostat` provide detailed I/O statistics for block devices

S `interruptible sleep (waiting for an event to complete)`

The same `IO_WAIT` condition, except it's interruptible (i.e., other processes can be scheduled and executed) since it's asynchronous IO.

Z `defunct/zombie, terminated but not reaped by its parent`

Zombie processes occur when a parent process doesn't call `wait()` and therefore cannot die. Explained in greater detail at some point.

T `stopped, either by a job control signal or because it is being traced`

Most probably someone ran Ctrl-Z and stopped a process, or it's being observed by `strace` (discussed later)

Each process has a process ID or *PID* (a number that begins with 2 and increases monotonically. PID 1 (and 0) is reserved which we'll discuss in a few). Every process is owned and executed with the privileges of a user. There are *real* and *effective* IDs that a process can run under. We'll discuss this concept in greater detail later.

- Processes are executed by the kernel in the CPU. the kernel also pulls reusable data storage (called *memory*). There are several forms of memory such as:
    - CPU registers (very small and extremely fast (runs at clock speed))
    - CPU caches (`0.5 nanoseconds` (`ns`) lookup time)
    - RAM (`100 ns` seek time depending on the memory type, e.g. SRAM, DRAM)
    - block I/O such as hard drives/solid state drives (large but extremely slow (`150,000 ns` seek time for SSD) (`10,000,000 ns` seek time for HDD))
    - network packet (speed depends on latency and bandwidth but also includes protocol latency such as TCP and UDP. Round trip packet Cali->Netherlands->Cali `150,000,000 ns = 150 ms`)

Only one process can execute in the CPU at a time (assuming 1 CPU core) so the kernel schedules the thousands of processes on the machine to make it seem like they're all running simultaneously.

The kernel is also responsible for providing an interface to all the machine's hardware (CPU, memory, network, hard drive, etc) which is why kernels must explicitly provide support for different chipsets (x86, ARM, PowerPC, etc)

- Special processes run on the system, such as PID 1, or *init*. Processes in Linux run in parent-child relationships. Every process has a parent process that watches over it and if it starts (`fork()`s) another process if watches over it. Processes look like trees with parents as the branches and children as the leafs. `init` (or `launchd` on a Mac) are special processes that "babysit" system processes (called `daemons`) to make sure they're running. They're also responsible for starting all processes at start time. An example of the process tree is seen below from the command `pstree`

```
-+= 00001 root /sbin/launchd
 |--= 00036 root /usr/libexec/UserEventAgent (System)
 |--= 00037 root /usr/sbin/syslogd
```

How does a process start?

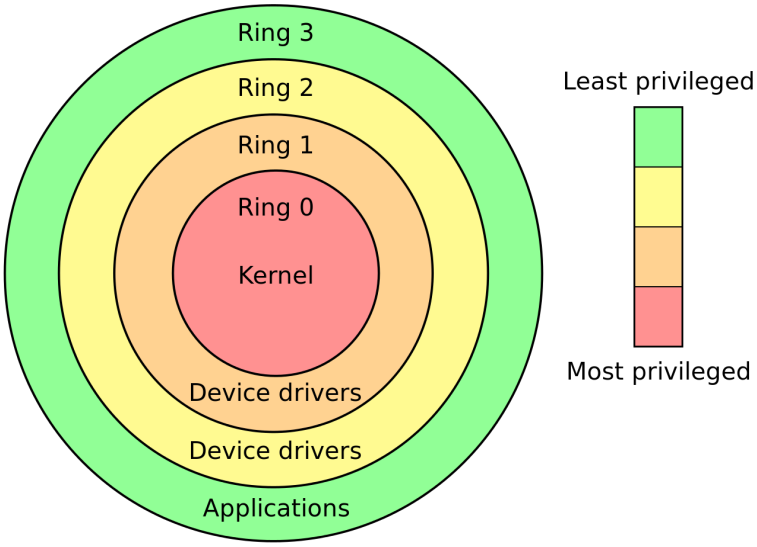
Processes begin their life as binaries on disk. When a user runs the command to start a process on a system the following takes place. We'll use the example of `ps aux`

- The command is tokenized into the command (`ps`) + args (`aux`)
- bash checks if `ps` is a builtin
- bash checks for the presence of `ps` in `$PATH`
- Once the binary is located, the shell (bash in this case) `fork()`s a child process
    - A fork() is a *system call* (more on this later) that creates a direct replica of the process, in this case bash. After calling `fork()`, the created *child* process is an exact copy of the parent except for the return value. This includes open files, register state, and all memory allocations, which includes the program's executable code.

    - The child switches to running another binary executable using the `exec()` system call
    - `exec()` includes the binary on disk to initialize into memory as well as any command line arguments
        - `execve("/bin/ps", ["ps", "aux"], [/* 11 vars */]) = 0`
- `fork()` returns the child's PID to the parent, and returns 0 to the child, to allow the two processes to distinguish one another
- The parent process can either continue execution or wait for the child process to complete
    - `strace -e waitpid ps aux 1>/dev/null`
    - `+++ exited with 0 +++`
- If the parent chooses to wait for the child to die, the parent will receive the exit status (`$?`) of the child. To prevent the child becoming a zombie the parent should call `waitpid()` on its children

System calls

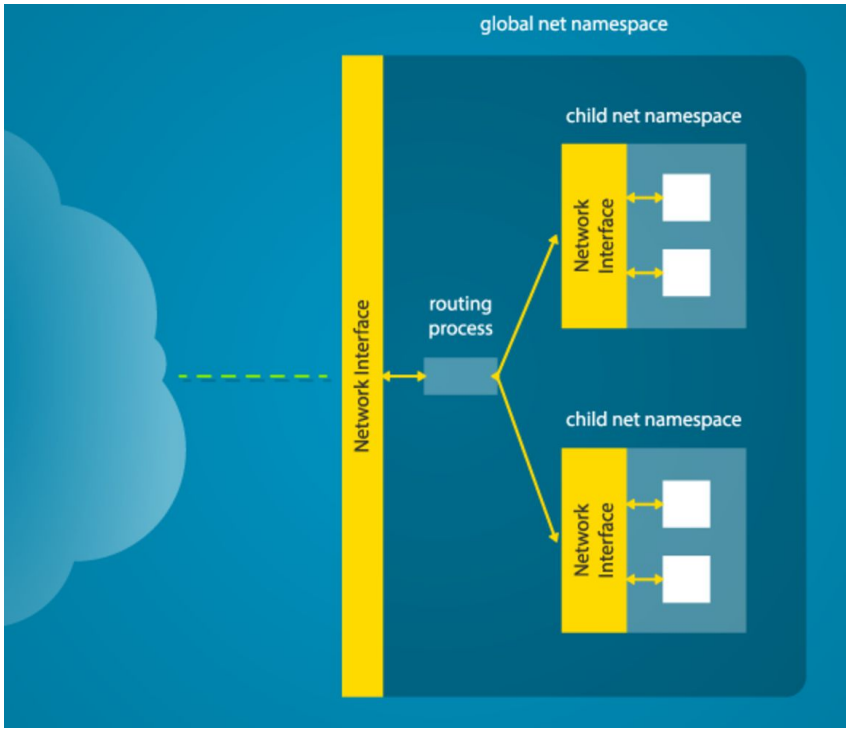
- Processes are executed by the kernel. A process requests resources (CPU, memory allocations, network sockets, block I/O, etc) via a kernel API called *system calls*. System calls are a uniform method to safely request resources from the system
- A process executes in *user space*, a limited environment where the process cannot access resources from other processes. For example, if a process in user space attempts to access memory from another process it will result in a segmentation fault (often called segfault)
    - Each process on a system believes it's the only process running and has full access over the memory of a system `0x0000000` - `0xFFFFFFF`
        - This is achieved via a system called *virtual memory* where a virtual memory address space (`0xdeadbeef`) translates to a physical memory address space (`0x123456`) via a hardware memory management unit (MMU)
- The kernel resides in *kernel space*, a privileged environment where direct access to all process memory is available. x86 provides an abstraction for protecting kernel space by creating a concentric ring of privileges from *ring0* to *ring3* where ring0 is the kernel space and ring3 is user space



- The kernel's virtual memory separation primitives of userspace is the basis of containers and obviates the need to have a dedicated kernel or hypervisor to isolate resources
- `strace` is a binary that traces the system calls a process makes. By default, `strace` logs to `stderr` so some plumbing is necessary

Kernel namespaces

- Linux *namespaces* allow aspects of the operating system to be independently isolated. This includes the process tree, networking interfaces, mount points, inter-process communication, etc
- Limiting resources is another key property of namespaces. Namespaces allow the admin of a system to provide precise limits and quotas on system resources, which is essential to a multitenancy environment
- `mnt` (dedicated mount points, filesystem)
- `pid` (dedicated process tree)
    - Implements multiple "nested" process trees. Each process tree can have an entirely isolated set of processes. This can ensure that processes belonging to one process tree cannot inspect or kill processes in other parent process trees.
    - A single process can now have multiple PIDs associated with it, one for each namespace it falls under
- `net` (dedicated network stack)
    - Network namespace allows each of these processes to see an entirely different set of networking interfaces
    - `veth`, `bridge`, `iptables`



- `ipc` (inter-process comms)
- `uts` (unique hostname)
- `user` (UIDs)
    - The user namespace allows a process to have root privileges within the namespace, without giving it root access to processes outside of the namespace
- `cgroups`
- `security`
    - `seccomp` allows a process to make a one-way transition into a "secure" state where it cannot make any system calls except `exit()`, `sigreturn()`, `read()` and `write()` to already-open file descriptors. Should it attempt any other system calls, the kernel will terminate the process with `SIGKILL`

Network level

- TCP/IP
- TCP and UDP
- DNS
- Routers and switches
- Routing algorithms (link state and metric)
- Load balancing algorithms and tradeoffs for each
- Proxies and reverse proxies

Distributed Computing

- Cloud provider infrastructure
    - Virtual private cloud networks (VPC)
    - Load balancers
    - Firewalls
    - Cloud SQL
    - Key value stores (such as `etcd`)
    - Blob storage
    - VM instances
    - Containers
- Kubernetes
- Cloud Foundry
- Scalability
- Microservices vs. monolithic