Linux API

Tristan Miller, Simon Kadesh

2023

What is a kernel?

- Runs in a higher privilege level than user processes (Ring 0 on x86)
- Manages system resources
- Isolates processes from each other and from hardware details

What is a syscall?

- CPU instruction that tells the kernel to do a thing
- Linux provides a C function for each syscall
- `man 2 syscalls`

File descriptors

- ”Everything is a file”
- File descriptor: integer that refers to a file
- Standard streams: stdin (0), stdout (1), stderr (2)

Types of files

- `S_IFREG`: regular file
- `S_IFDIR`: directory
- `S_IFBLK`: block device (hard disk, SSD, etc.)
- `S_IFCHR`: character device (serial ports, TTYs, etc.)
- `S_IFIFO`: FIFO/named pipe
- `S_IFSOCK`: socket (TCP, UDP, Unix, etc.)

open(2)

```c
int open(char *pathname, int flags,
    ... /* mode_t mode */ );
```

- Open the file at the specified path
- Returns a new file descriptor, or -1 on failure
- `O_RDONLY`, `O_WRONLY`, `O_RDWR`: read or write to the file
- `O_APPEND`: append instead of overwriting
- `O_CREAT`: create the file if it doesn't exist
- `O_NONBLOCK`: reading/writing won't block
- `O_ASYNC`: async I/O - send a signal when ready to read/write again
- etc.

close(2)

```c
int close(int fd);
```

- Close the specified file
- Return 0 on success, -1 on failure

read(2)

```c
ssize_t read(int fd, void buf[], size_t count);
```

- Read at most `count` bytes from the file into the buffer
- May read fewer than `count` bytes
- Returns the number of bytes read, or -1 on failure. 0 *usually* indicates EOF.

write(2)

```c
ssize_t write(int fd, void buf[], size_t count);
```

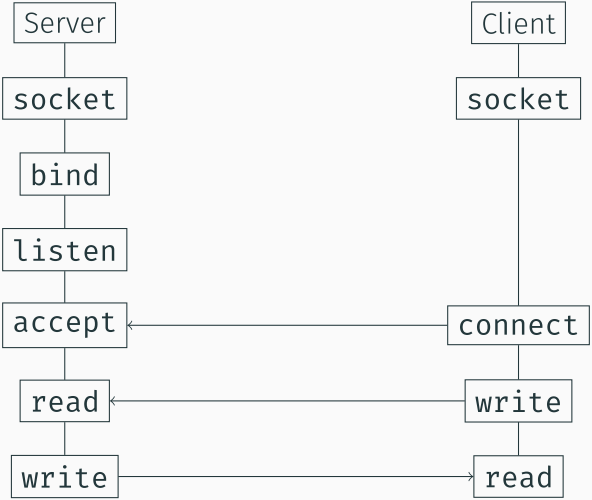
- Write at most `count` bytes from the buffer to the file
- May write fewer than `count` bytes
- Returns the number of bytes written, or -1 on failure.

stat(2), fstat(2), lstat(2)

```c
int stat(char *pathname, struct stat *statbuf);
```

- Get information about a file by its path (you don't need to `open` it first)
- `fstat`: same as `stat` but takes a file descriptor
- `lstat`: same as `stat` but doesn't follow symlinks
- Use `stat(1)` from the command line for human-readable info

Sockets


Server
socket
bind
listen
accept
read
write
Client
socket
connect
write
read

socket(2)

```
int socket(int domain, int type, int protocol);
```

- Open a socket and return its file descriptor
- Domains: `AF_INET`, `AF_INET6`, `AF_UNIX`, etc.
- Types: `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_SEQPACKET`, `SOCK_RAW`, etc.
- Protocol: usually 0

connect(2)

```c
int connect(int sockfd, struct sockaddr *addr,
    socklen_t addrlen);
```

- Connect socket to the given address
- Type of `addr` depends on socket type

bind(2)

```
int bind(int sockfd, struct sockaddr *addr,
    socklen_t addrlen);
```

- Bind a socket to an address so it listens for incoming data
- Type of `addr` depends on socket type

listen(2)

```
int listen(int sockfd, int backlog);
```

- Mark a socket as accepting connections

accept(2)

```c
int accept(int sockfd, struct sockaddr *addr,
    socklen_t *addrlen);
```

- Wait for a new incoming connection
- Returns a new file descriptor, address of new connection stored to `addr`

fcntl(2), ioctl(2), setsockopt(2)

```
int fcntl(int fd, int cmd, ... /* arg */ );
int ioctl(int fd, unsigned long request, ...);
int setsockopt(int socket, int level,
    int option_name, void *option_value,
    socklen_t option_len);
```

- Used to get information about and change settings on file descriptors
- `fcntl`: change file descriptor and status flags (see `open(2)`)
- `ioctl`: used to configure devices/drivers represented by a file
- `setsockopt`: used to change socket-specific options

poll(2)

```
int poll(struct pollfd *fds, nfds_t nfds, int timeo
```

- Wait on multiple file descriptors at once
  - `struct pollfd`: contains file descriptor, events you want to wait for, and events that occured
  - POLLIN, POLLOUT, POLLHUP, POLLERR
  - `timeout`: optionally specify a timeout, -1 to disable‘
  - After poll returns, check each `struct pollfd` to see if any events occurred
- There's also `select(2)` and io$_u$*ring*(7)

sysfs(5), procfs(5)

- Virtual filesystems which expose various information and control mechanisms to userspace
- `sysfs` exposes the kernel and the hardware
    - `devices` contains representations of the kernel device tree
    - `firmware` contains firmware-specific variables and objects
    - `fs` contains filesystem information
    - `kernel` contains kernel information and settings
    - `module` contains information about kernel modules
    - `power` contains power information

procfs(5)

- `procfs` exposes processes
  - Directory structure of the form `/proc/<PID>/`
  - Directories contain files that provide information about that process
  - `cmdline` the command that started the process
  - `cwd` the current working directory of the process
  - `environ` the environment variables
  - `exe` the actual executable of the process
  - `fd` a directory of all open file descriptors
  - `fdinfo` a directory containing information about open file descriptors
  - `maps` information about mapped memory
  - `root` the process's root (usually `/`)
  - `status` basic status of the process
  - `task` a directory of started tasks from this process

cgroups(7)

- Problem: We've got a bunch of processes and we want to be able to know and control what they're doing
  - Not really any good way to do this in POSIX
  - Process groups and sessions exist, but they're too easy to escape (either accidentally or on purpose)
  - This is the "double fork" that some processes use to daemonize
- Solution: We need a way to reliably group processes
- cgroups (control groups) provide this functionality
  - We can interact with cgroups through the cgroup virtual filesystem
  - We can control how processes can move between cgroups
  - We can impose restrictions on which resources processes can use

cgroups continued

- cgroups are controlled through a virtual filesystem
  - `sys/fs/cgroup`
- The cgroup filesystem has a tree structure
- Each directory defines a group
- Groups are defined as follows
  - Each group has `cgroup.controllers` and `cgroup.subtree_control`
  - Each group has a `cgroup.events` and `cgroup.stat`
  - Groups can either contain processes or subgroups, but not both
- Processes are assigned cgroups using the procfs filesystem
- cgroups are often utilized together with namespaces to build containers

- The `evdev` virtual filesystem is how the Linux kernel exposes raw input events from device drivers to the kernel
- `/dev/input` contains files corresponding to character devices
- Input events are written to the files in the form of a struct containing the timestamp, the event type, the event code, and the event's value

drm(7)/kms

- Graphical interfaces are great, but they come with one big problem
    - Only one program can control the actual physical GPU at a time
    - Thus a system to manage (direct) rendering was necessary
- DRM exposes graphics devices under the `/dev/dri` file hierarchy
    - `/dev/dri/card*` files are full device nodes that implement both priveleged and rendering functionality
    - `/dev/dri/renderD*` files are render nodes which only allow rendering
- Almost all of the functionality in DRM is controlled through `ioctl` syscalls on these device files

ALSA

- The Linux sound stack (usually) consists of a kernel-space component (ALSA) and a userspace component (usually pulseaudio, pipewire, JACK, etc.)
- ALSA is responsible for managing audio devices and their drivers
- The only sensible way to interface with ALSA seems to be the `alsa-lib` C library, also provided by the ALSA project