Everything's a File Descriptor

Josh Triplett
josh@joshtriplett.org

Linux Plumbers Conference 2015

“Everything’s a file”

- /home/josh/doc/presentations/lpc-2015/fd/fd.pdf

- /home/josh/doc/presentations/lpc-2015/fd/fd.pdf
- /etc/hostname

- /home/josh/doc/presentations/lpc-2015/fd/fd.pdf
- /etc/hostname
- /dev/null
- /dev/zero

- /home/josh/doc/presentations/lpc-2015/fd/fd.pdf
- /etc/hostname
- /dev/null
- /dev/zero
- /dev/ttyS0
- /dev/dri/card0
- /dev/cpu/0/cpuid

- /home/josh/doc/presentations/lpc-2015/fd/fd.pdf
- /etc/hostname
- /dev/null
- /dev/zero
- /dev/ttyS0
- /dev/dri/card0
- /dev/cpu/0/cpuid
- /tmp/.X11-unix/X0

- /home/josh/doc/presentations/lpc-2015/fd/fd.pdf
- /etc/hostname
- /dev/null
- /dev/zero
- /dev/ttyS0
- /dev/dri/card0
- /dev/cpu/0/cpuid
- /tmp/.X11-unix/X0
- /proc/1/environ

- /home/josh/doc/presentations/lpc-2015/fd/fd.pdf
- /etc/hostname
- /dev/null
- /dev/zero
- /dev/ttyS0
- /dev/dri/card0
- /dev/cpu/0/cpuid
- /tmp/.X11-unix/X0
- /proc/1/environ
- /proc/cmdline

- /home/josh/doc/presentations/lpc-2015/fd/fd.pdf
- /etc/hostname
- /dev/null
- /dev/zero
- /dev/ttyS0
- /dev/dri/card0
- /dev/cpu/0/cpuid
- /tmp/.X11-unix/X0
- /proc/1/environ
- /proc/cmdline
- /sys/class/block/sda/queue/rotational
- /sys/firmware/acpi/tables/DSDT

Everything has a filename?

~~Everything has a filename?~~

~~Everything has a filename?~~

- Pipes
- Sockets
- epoll
- memfd
- KVM virtual machines and CPUs
- ...

Everything's a file *descriptor*

- What is a file descriptor, really?
- What can you do with a file descriptor?
- What interesting file descriptors exist?
- How do you build a new type of file descriptors?
- What interesting file descriptors don't exist?

- What is a file descriptor, really?
- What can you do with a file descriptor?
- What interesting file descriptors exist?
- How do you build a new type of file descriptors?
- What interesting file descriptors don't exist yet?

What is a file descriptor, really?

- `struct fd`, `struct fdtable`
- `struct file`

struct fd versus struct file

testfile contains “0123456789”

```
x = open("testfile", O_RDONLY);
xdup = dup(x);
y = open("testfile", O_RDONLY);
```

struct fd versus struct file

testfile contains “0123456789”

```
x = open("testfile", O_RDONLY);
xdup = dup(x);
y = open("testfile", O_RDONLY);
read(x, &c, 1);
putchar(c);
read(xdup, &c, 1);
putchar(c);
read(y, &c, 1);
putchar(c);
```

struct fd versus struct file

testfile contains “0123456789”

```
x = open("testfile", O_RDONLY);
xdup = dup(x);
y = open("testfile", O_RDONLY);
read(x, &c, 1);
putchar(c); /* Prints '0' */
read(xdup, &c, 1);
putchar(c);
read(y, &c, 1);
putchar(c);
```

struct fd versus struct file

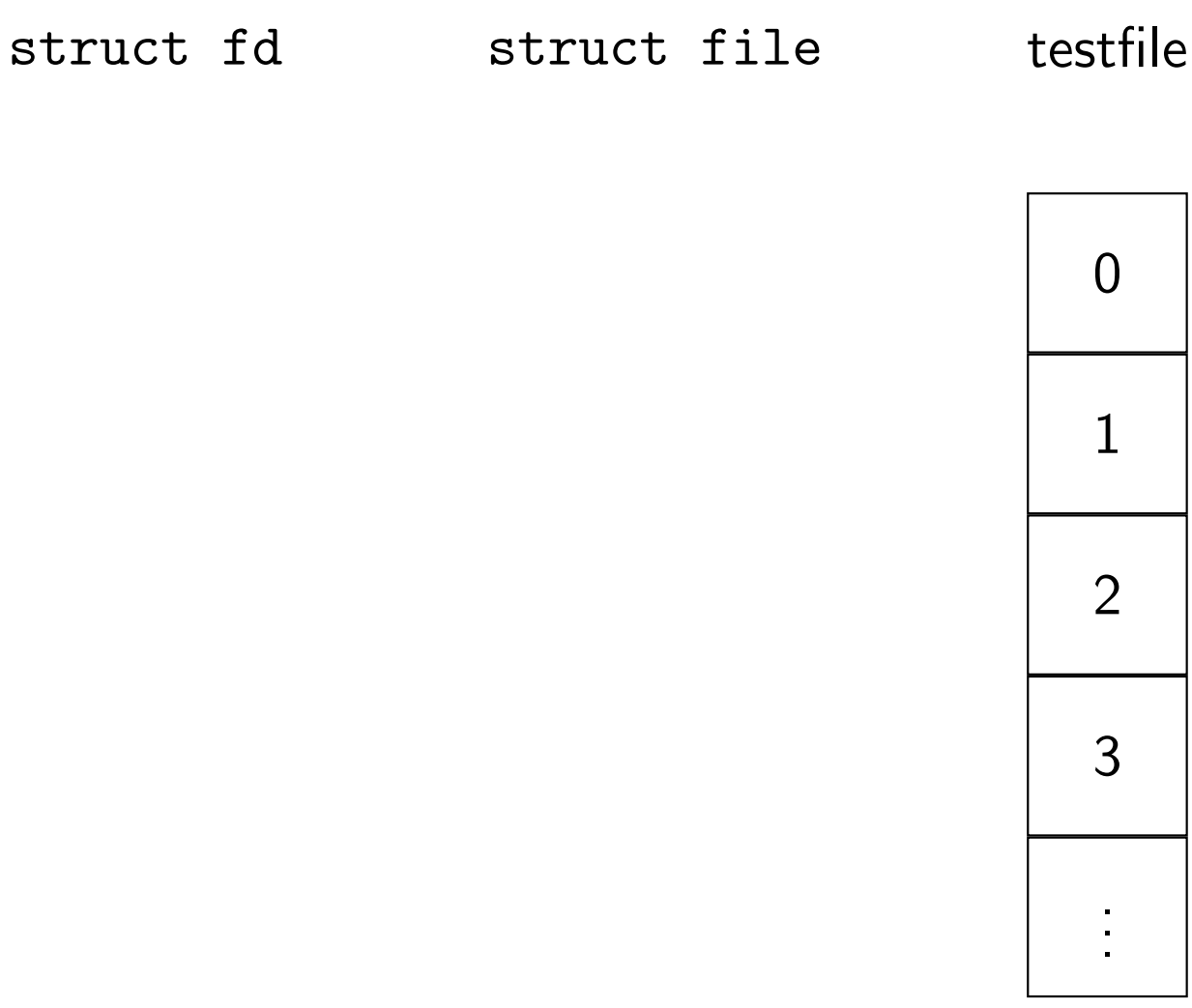
testfile contains “0123456789”

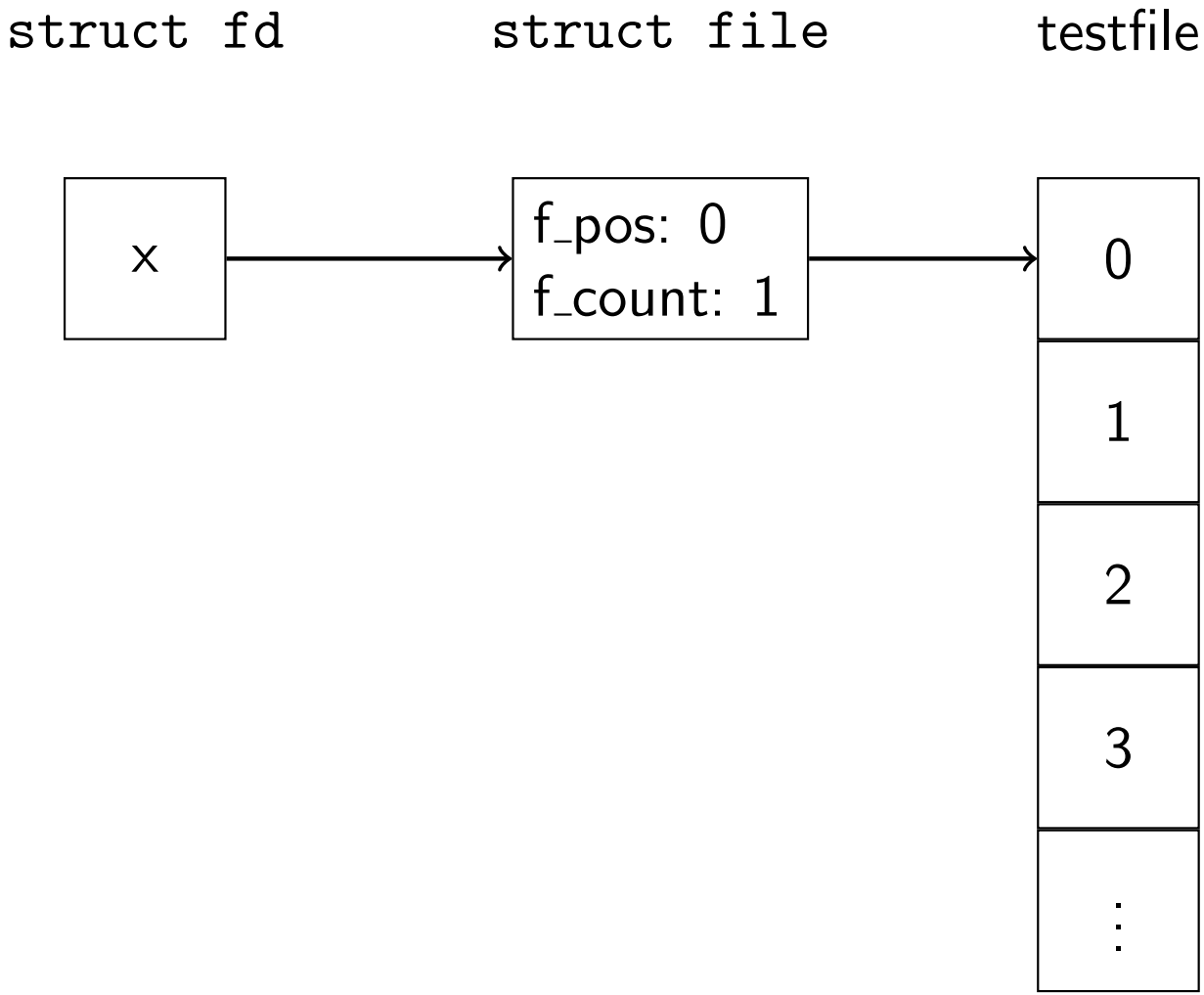
```
x = open("testfile", O_RDONLY);
xdup = dup(x);
y = open("testfile", O_RDONLY);
read(x, &c, 1);
putchar(c); /* Prints '0' */
read(xdup, &c, 1);
putchar(c); /* Prints '1' */
read(y, &c, 1);
putchar(c);
```

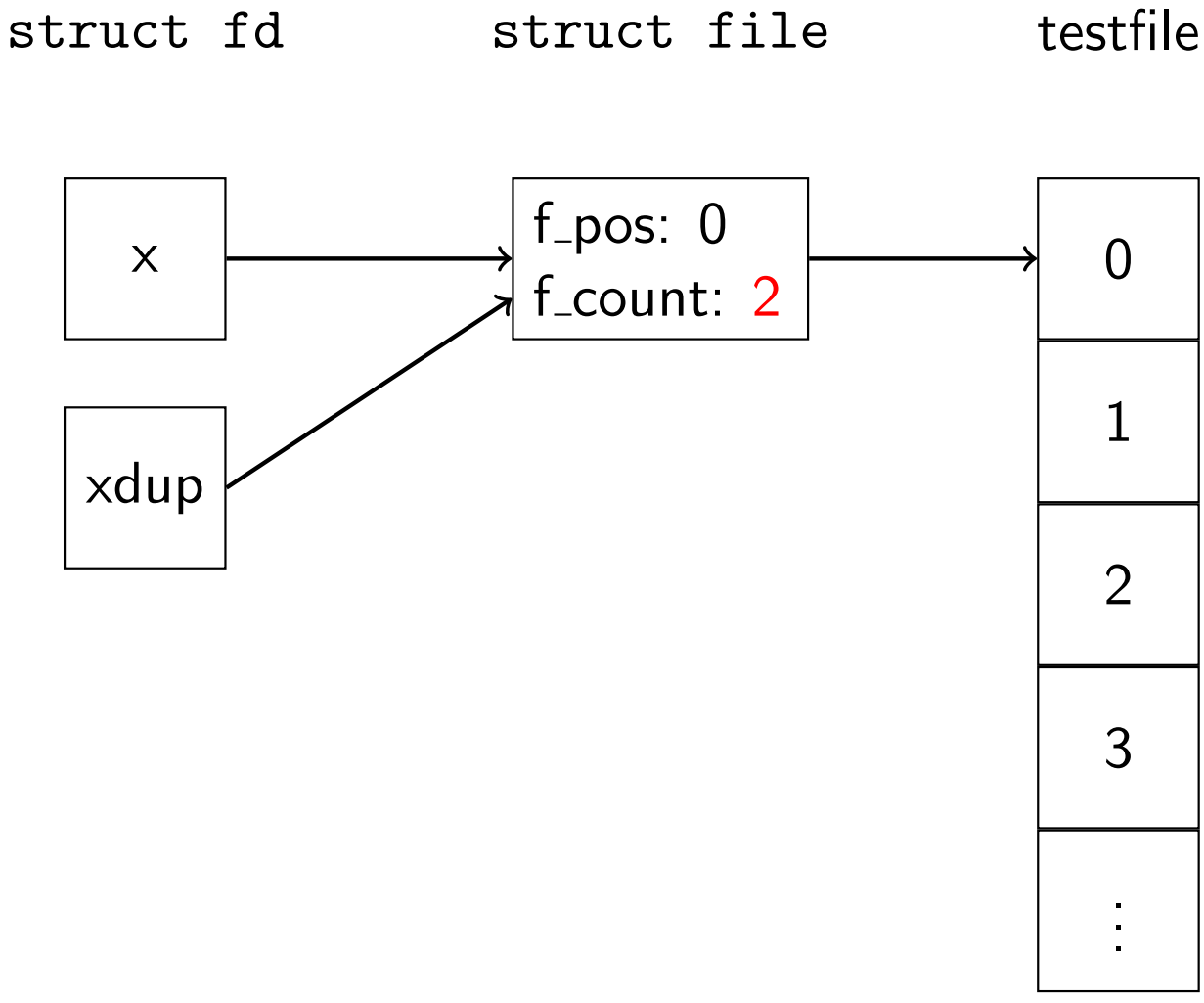
struct fd versus struct file

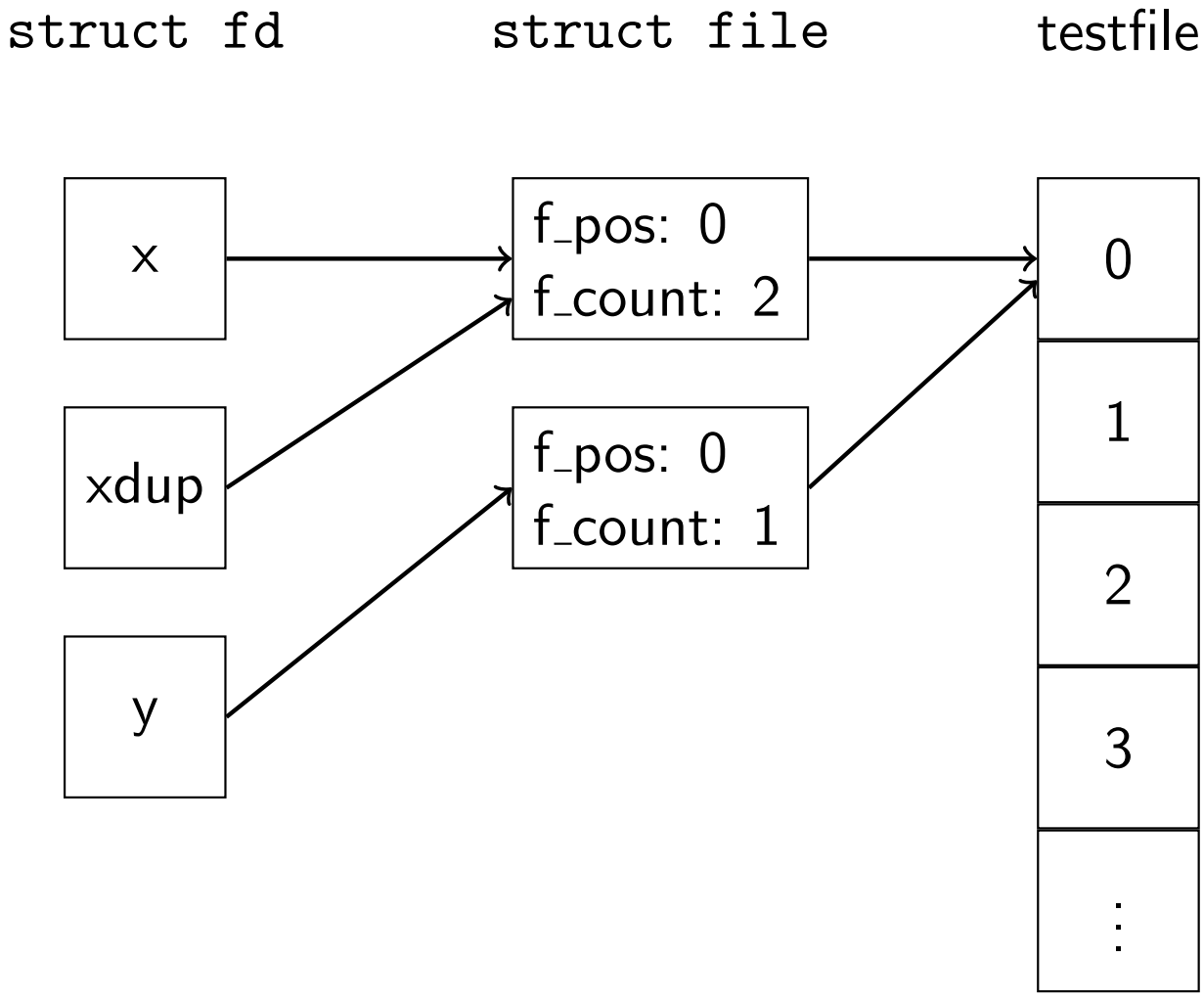
testfile contains “0123456789”

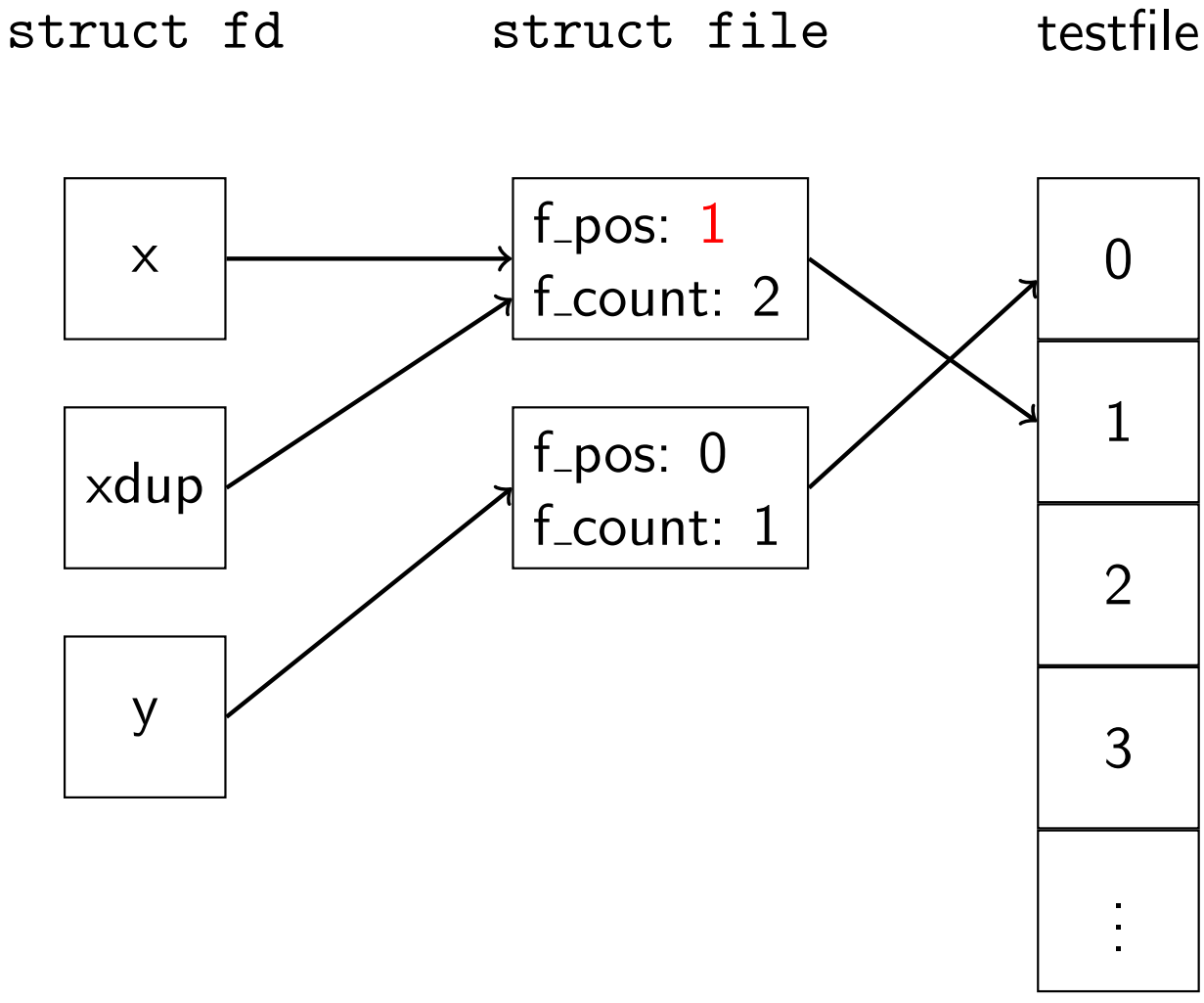
```
x = open("testfile", O_RDONLY);
xdup = dup(x);
y = open("testfile", O_RDONLY);
read(x, &c, 1);
putchar(c); /* Prints '0' */
read(xdup, &c, 1);
putchar(c); /* Prints '1' */
read(y, &c, 1);
putchar(c); /* Prints '0' */
```

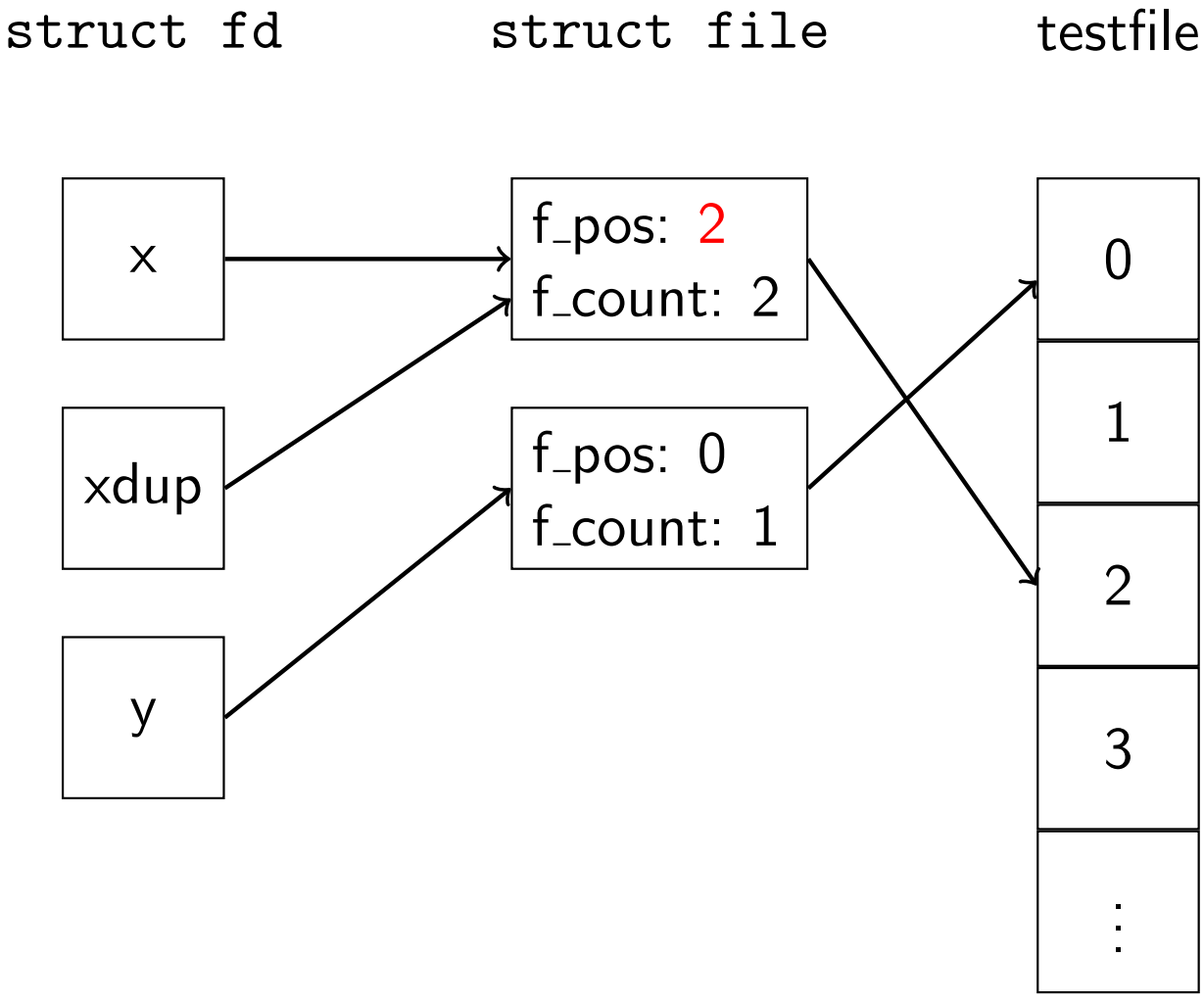

struct fd
struct file
testfile
0
1
2
3
⋮

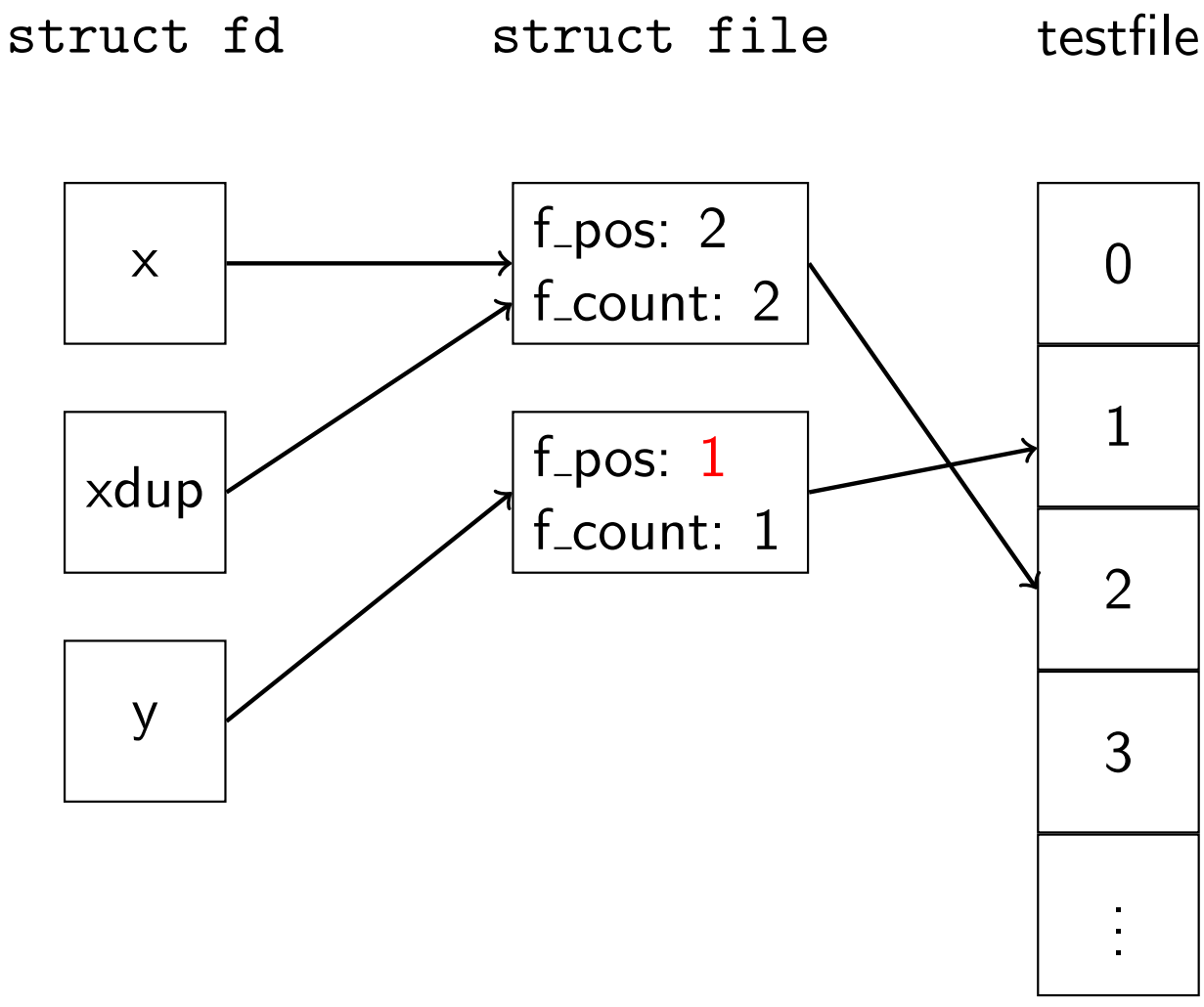

struct fd
struct file
testfile
x
f_pos: 0
f_count: 1
0
1
2
3
⋮

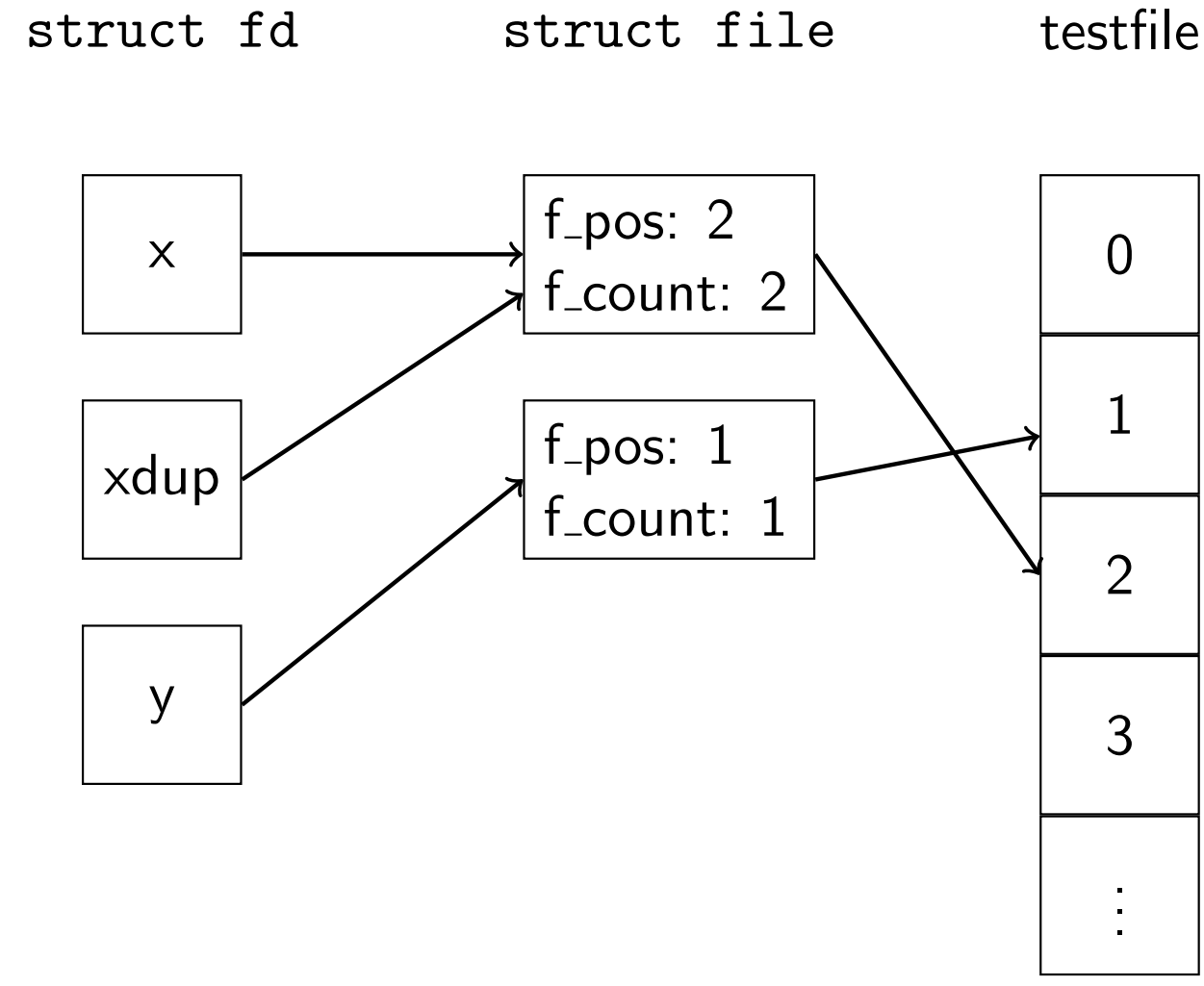

struct fd
struct file
testfile
x
xdup
f_pos: 0
f_count: 2
0
1
2
3
⋮

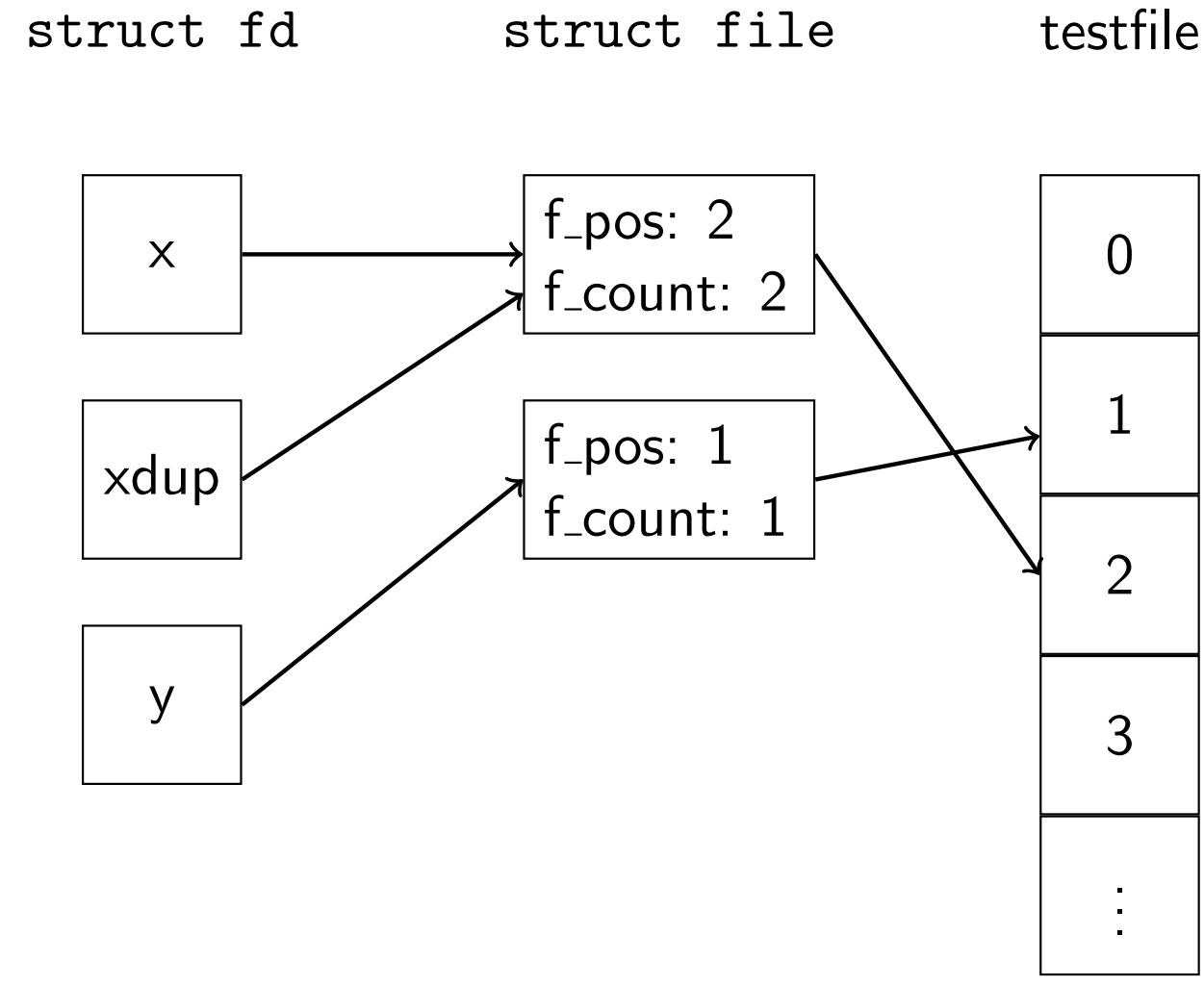

struct fd
struct file
testfile
x
xdup
y
f_pos: 0
f_count: 2
f_pos: 0
f_count: 1
0
1
2
3
⋮

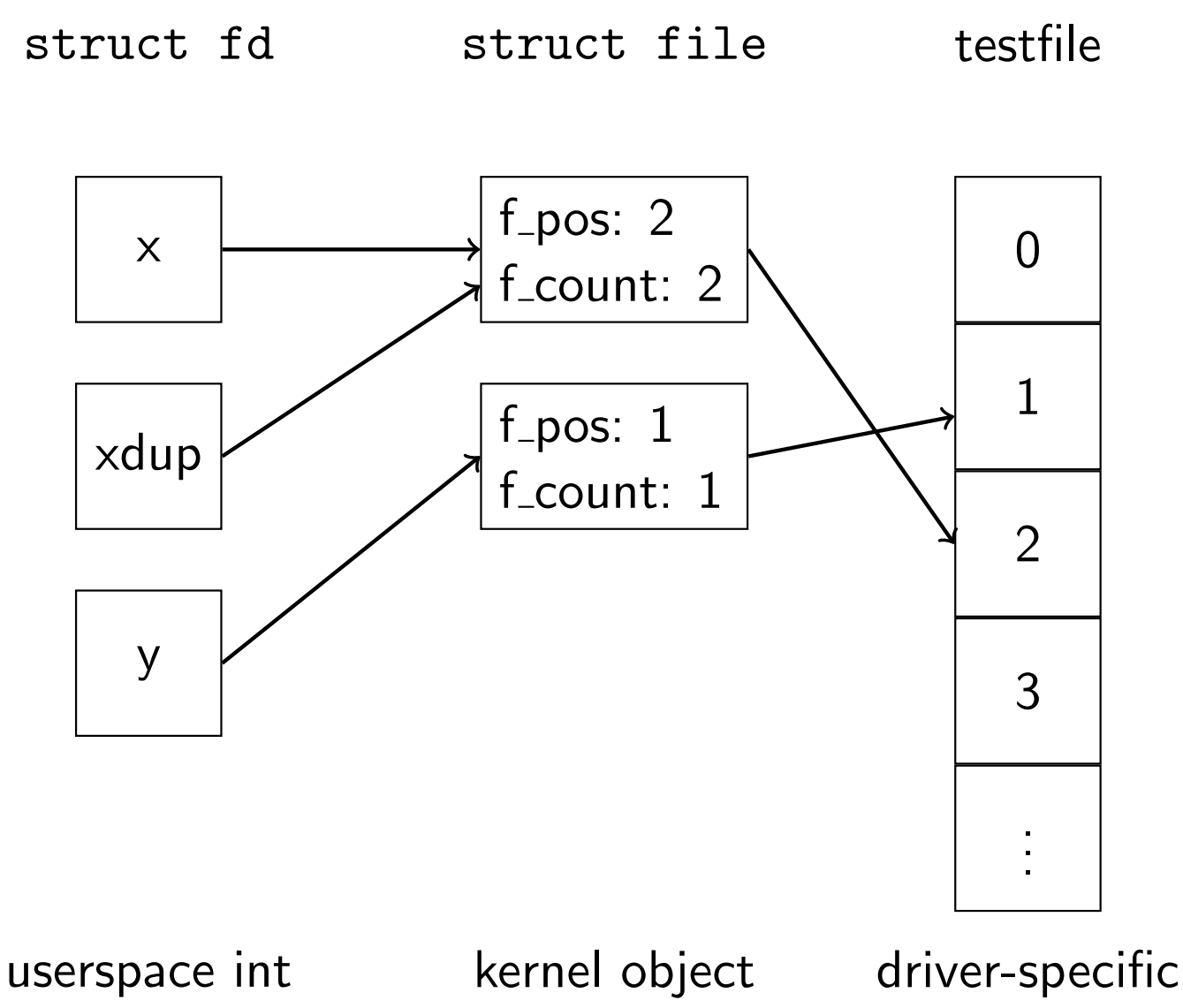

struct fd
struct file
testfile
x
xdup
y
f_pos: 1
f_count: 2
f_pos: 0
f_count: 1
0
1
2
3
⋮


struct fd
struct file
testfile
x
xdup
y
f_pos: 2
f_count: 2
f_pos: 0
f_count: 1
0
1
2
3
⋮


struct fd
struct file
testfile
x
xdup
y
f_pos: 2
f_count: 2
f_pos: 1
f_count: 1
0
1
2
3
⋮


struct fd
struct file
testfile
x
xdup
y
f_pos: 2
f_count: 2
f_pos: 1
f_count: 1
0
1
2
3
⋮
userspace int


struct fd
struct file
testfile
x
xdup
y
f_pos: 2
f_count: 2
f_pos: 1
f_count: 1
0
1
2
3
⋮
userspace int
kernel object


struct fd
struct file
testfile
x
xdup
y
f_pos: 2
f_count: 2
f_pos: 1
f_count: 1
0
1
2
3
⋮
userspace int
kernel object
driver-specific

File descriptor: Userspace reference to kernel object

What can you do with a file descriptor?

- `read`, `write`

- `read`, `write`
- `seek`

- `read`, `write`
- `seek`
- `preadv`, `pwritev`

- `read`, `write`
- `seek`
- `preadv`, `pwritev`
- `stat`

- `read`, `write`
- `seek`
- `preadv`, `pwritev`
- `stat`
- Blocking or non-blocking

- `read`, `write`
- `seek`
- `preadv`, `pwritev`
- `stat`
- Blocking or non-blocking
- `poll`, `select`, `epoll`

- `read`, `write`
- `seek`
- `preadv`, `pwritev`
- `stat`
- Blocking or non-blocking
- `poll`, `select`, `epoll`
- `dup`, `dup2`

- `read`, `write`
- `seek`
- `preadv`, `pwritev`
- `stat`
- Blocking or non-blocking
- `poll`, `select`, `epoll`
- `dup`, `dup2`
- Send over a UNIX socket via `SCM_RIGHTS`

- `read`, `write`
- `seek`
- `preadv`, `pwritev`
- `stat`
- Blocking or non-blocking
- `poll`, `select`, `epoll`
- `dup`, `dup2`
- Send over a UNIX socket via `SCM_RIGHTS`
- Inherited over `exec`

- `read`, `write`
- `seek`
- `preadv`, `pwritev`
- `stat`
- Blocking or non-blocking
- `poll`, `select`, `epoll`
- `dup`, `dup2`
- Send over a UNIX socket via `SCM_RIGHTS`
- Inherited over `exec`
- `mmap`

- `read`, `write`
- `seek`
- `preadv`, `pwritev`
- `stat`
- Blocking or non-blocking
- `poll`, `select`, `epoll`
- `dup`, `dup2`
- Send over a UNIX socket via `SCM_RIGHTS`
- Inherited over `exec`
- `mmap`
- `sendfile`, `splice`, `tee`

- `read`, `write`
- `seek`
- `preadv`, `pwritev`
- `stat`
- Blocking or non-blocking
- `poll`, `select`, `epoll`
- `dup`, `dup2`
- Send over a UNIX socket via `SCM_RIGHTS`
- Inherited over `exec`
- `mmap`
- `sendfile`, `splice`, `tee`
- `openat`

- `read`, `write`
- `seek`
- `preadv`, `pwritev`
- `stat`
- Blocking or non-blocking
- `poll`, `select`, `epoll`
- `dup`, `dup2`
- Send over a UNIX socket via `SCM_RIGHTS`
- Inherited over `exec`
- `mmap`
- `sendfile`, `splice`, `tee`
- `openat`
- ...

- `read`, `write`
- `seek`
- `preadv`, `pwritev`
- `stat`
- Blocking or non-blocking
- `poll`, `select`, `epoll`
- `dup`, `dup2`
- Send over a UNIX socket via `SCM_RIGHTS`
- Inherited over `exec`
- `mmap`
- `sendfile`, `splice`, `tee`
- `openat`
- ...
- `ioctl`

Use file descriptors!

What interesting file descriptors exist?

eventfd

- 64-bit counter used as an event queue

eventfd

- 64-bit counter used as an event queue
- `write`: Add value to counter

eventfd

- 64-bit counter used as an event queue
- `write`: Add value to counter
- `read`: Block until non-zero; read value and reset to 0
  - "Semaphore mode": Read 1 and decrement by 1

eventfd

- 64-bit counter used as an event queue
- `write`: Add value to counter
- `read`: Block until non-zero; read value and reset to 0
  - "Semaphore mode": Read 1 and decrement by 1
- `poll`: Ready for reading if non-zero

eventfd

- 64-bit counter used as an event queue
- `write`: Add value to counter
- `read`: Block until non-zero; read value and reset to 0
  - "Semaphore mode": Read 1 and decrement by 1
- `poll`: Ready for reading if non-zero
- Several drivers use eventfd to signal events between kernel and userspace

timerfd

- Allows handling timers as file descriptors
- Throw them in the `poll` loop with everything else
- Create with specified timeout
- `read`: Block until timeout; return number of times expired
- `poll`: Reading for reading if timeout passed

Signals

Signals

- Receive asynchronous events in a process

Signals

- Receive asynchronous events in a process
- Suspend execution, save registers, move execution to handler
- Restore registers and resume execution when handler done

Signals

- Receive asynchronous events in a process
- Suspend execution, save registers, move execution to handler
- Restore registers and resume execution when handler done
- Assume a userspace stack to push and pop state

Signals

- Receive asynchronous events in a process
- Suspend execution, save registers, move execution to handler
- Restore registers and resume execution when handler done
- Assume a userspace stack to push and pop state
- `sigaltstack` sets an alternate stack to switch to

Signals

- Receive asynchronous events in a process
- Suspend execution, save registers, move execution to handler
- Restore registers and resume execution when handler done
- Assume a userspace stack to push and pop state
- `sigaltstack` sets an alternate stack to switch to
- Set up stack to return into call to `sigreturn` for cleanup

Signals

- Receive asynchronous events in a process
- Suspend execution, save registers, move execution to handler
- Restore registers and resume execution when handler done
- Assume a userspace stack to push and pop state
- `sigaltstack` sets an alternate stack to switch to
- Set up stack to return into call to `sigreturn` for cleanup
- Can receive signals while in a kernel syscall

Signals

- Receive asynchronous events in a process
- Suspend execution, save registers, move execution to handler
- Restore registers and resume execution when handler done
- Assume a userspace stack to push and pop state
- `sigaltstack` sets an alternate stack to switch to
- Set up stack to return into call to `sigreturn` for cleanup
- Can receive signals while in a kernel syscall
- Some syscalls restart afterward
- Syscalls with timeouts adjust them (`restart_syscall`)
- Other syscalls return `EINTR`

Signals

- Receive asynchronous events in a process
- Suspend execution, save registers, move execution to handler
- Restore registers and resume execution when handler done
- Assume a userspace stack to push and pop state
- `sigaltstack` sets an alternate stack to switch to
- Set up stack to return into call to `sigreturn` for cleanup
- Can receive signals while in a kernel syscall
- Some syscalls restart afterward
- Syscalls with timeouts adjust them (`restart_syscall`)
- Other syscalls return `EINTR`
- Can mask signals to avoid interruption

Signals

- Receive asynchronous events in a process
- Suspend execution, save registers, move execution to handler
- Restore registers and resume execution when handler done
- Assume a userspace stack to push and pop state
- `sigaltstack` sets an alternate stack to switch to
- Set up stack to return into call to `sigreturn` for cleanup
- Can receive signals while in a kernel syscall
- Some syscalls restart afterward
- Syscalls with timeouts adjust them (`restart_syscall`)
- Other syscalls return `EINTR`
- Can mask signals to avoid interruption
- Special syscalls that also set signal mask (`ppoll`, `pselect`, `KVM_SET_SIGNAL_MASK ioctl`)

Signals

- Receive asynchronous events in a process
- Suspend execution, save registers, move execution to handler
- Restore registers and resume execution when handler done
- Assume a userspace stack to push and pop state
- `sigaltstack` sets an alternate stack to switch to
- Set up stack to return into call to `sigreturn` for cleanup
- Can receive signals while in a kernel syscall
- Some syscalls restart afterward
- Syscalls with timeouts adjust them (`restart_syscall`)
- Other syscalls return `EINTR`
- Can mask signals to avoid interruption
- Special syscalls that also set signal mask (`ppoll`, `pselect`, `KVM_SET_SIGNAL_MASK ioctl`)
- "async-signal-safe" library functions

Signed-off-by:  <(;,;)@r'lyeh>

signalfd

- File descriptor to receive a given set of signals
- Block “normal” signal delivery; receive via signalfd instead

signalfd

- File descriptor to receive a given set of signals
- Block “normal” signal delivery; receive via signalfd instead
- `read`: Block until signal, return `struct signalfd_siginfo`
- `poll`: Readable when signal received

How do you build a new type of file descriptor?

Semantics

- `read` and `write`
  - Nothing
  - Raw data
  - Specific data structure

Semantics

- `read` and `write`
  - Nothing
  - Raw data
  - Specific data structure
- `poll`/`select`/`epoll`
  - Must match `read`/`write` blocking behavior if any
  - Can have pollable fd even if `read`/`write` do nothing

Semantics

- `read` and `write`
  - Nothing
  - Raw data
  - Specific data structure
- `poll`/`select`/`epoll`
  - Must match `read`/`write` blocking behavior if any
  - Can have pollable fd even if `read`/`write` do nothing
- `seek` and file position

Semantics

- `read` and `write`
  - Nothing
  - Raw data
  - Specific data structure
- `poll`/`select`/`epoll`
  - Must match `read`/`write` blocking behavior if any
  - Can have pollable fd even if `read`/`write` do nothing
- `seek` and file position
- `mmap`

Semantics

- `read` and `write`
  - Nothing
  - Raw data
  - Specific data structure
- `poll`/`select`/`epoll`
  - Must match `read`/`write` blocking behavior if any
  - Can have pollable fd even if `read`/`write` do nothing
- `seek` and file position
- `mmap`
- What happens with multiple processes, or `dup`?

Semantics

- `read` and `write`
  - Nothing
  - Raw data
  - Specific data structure
- `poll`/`select`/`epoll`
  - Must match `read`/`write` blocking behavior if any
  - Can have pollable fd even if `read`/`write` do nothing
- `seek` and file position
- `mmap`
- What happens with multiple processes, or `dup`?
- For everything else: `ioctl`

Implementation

- `anon_inode_getfd`
  - Doesn't need a backing inode or filesystem
  - Provide an ops structure and private data pointer
  - Private data points to your kernel object

Implementation

- `anon_inode_getfd`
  - Doesn't need a backing inode or filesystem
  - Provide an ops structure and private data pointer
  - Private data points to your kernel object
- `simple_read_from_buffer`, `simple_write_to_buffer`

Implementation

- `anon_inode_getfd`
  - Doesn't need a backing inode or filesystem
  - Provide an ops structure and private data pointer
  - Private data points to your kernel object
- `simple_read_from_buffer`, `simple_write_to_buffer`
- `no_llseek`, `fixed_size_llseek`

Implementation

- `anon_inode_getfd`
  - Doesn't need a backing inode or filesystem
  - Provide an ops structure and private data pointer
  - Private data points to your kernel object
- `simple_read_from_buffer`, `simple_write_to_buffer`
- `no_llseek`, `fixed_size_llseek`
- Check `file->f_flags & O_NONBLOCK`
  - Blocking: `wait_queue_head`
  - Non-blocking: return `-EAGAIN`

What interesting file descriptors don't exist yet?

Child processes

- `fork/clone`

- `fork`/`clone`
- Parent process gets the child PID

- `fork`/`clone`
- Parent process gets the child PID
- Parent uses dedicated syscalls (`waitpid`) to wait for child exit

- `fork`/`clone`
- Parent process gets the child PID
- Parent uses dedicated syscalls (`waitpid`) to wait for child exit
- When child exits, parent gets `SIGCHLD` signal

- `fork`/`clone`
- Parent process gets the child PID
- Parent uses dedicated syscalls (`waitpid`) to wait for child exit
- When child exits, parent gets `SIGCHLD` signal
- Parent makes `waitpid` call to get exit status

- `fork`/`clone`
- Parent process gets the child PID
- Parent uses dedicated syscalls (`waitpid`) to wait for child exit
- When child exits, parent gets `SIGCHLD` signal
- Parent makes `waitpid` call to get exit status

Problems:

- `fork`/`clone`
- Parent process gets the child PID
- Parent uses dedicated syscalls (`waitpid`) to wait for child exit
- When child exits, parent gets `SIGCHLD` signal
- Parent makes `waitpid` call to get exit status

Problems:

- Waiting not integrated with `poll` loops

- `fork`/`clone`
- Parent process gets the child PID
- Parent uses dedicated syscalls (`waitpid`) to wait for child exit
- When child exits, parent gets `SIGCHLD` signal
- Parent makes `waitpid` call to get exit status

Problems:

- Waiting not integrated with `poll` loops
- Signals

- `fork`/`clone`
- Parent process gets the child PID
- Parent uses dedicated syscalls (`waitpid`) to wait for child exit
- When child exits, parent gets `SIGCHLD` signal
- Parent makes `waitpid` call to get exit status

Problems:

- Waiting not integrated with `poll` loops
- Signals
- Process-global; libraries can't manage only their own processes

Alternatives

- Set `SIGCHLD` handler, write to pipe or eventfd
  - Still process-global; gets all child exit notifications
  - Requires coordinating global signal handling between libraries
  -  Signals

Alternatives

- Set `SIGCHLD` handler, write to pipe or eventfd
  - Still process-global; gets all child exit notifications
  - Requires coordinating global signal handling between libraries
  -  Signals
- `signalfd` for `SIGCHLD`
  - Still process-global; gets all child exit notifications
  - Requires coordinating global signal handling between libraries
  - Must block `SIGCHLD`; breaks code expecting `SIGCHLD`

clonefd

clonefd

- New flag for `clone`
- Return a file descriptor for the child process

clonefd

- New flag for `clone`
- Return a file descriptor for the child process
- `read`: block until child exits, return exit information

clonefd

- New flag for `clone`
- Return a file descriptor for the child process
- `read`: block until child exits, return exit information
- `poll`: becomes readable when child exits

clonefd

- New flag for `clone`
- Return a file descriptor for the child process
- `read`: block until child exits, return exit information
- `poll`: becomes readable when child exits
- Maintains a reference to the child's `task_struct`

clonefd

- New flag for `clone`
- Return a file descriptor for the child process
- `read`: block until child exits, return exit information
- `poll`: becomes readable when child exits
- Maintains a reference to the child's `task_struct`
- Relatively simple, except...

Complications

- Need a new `clone` system call for the `fd` out parameter

Complications

- Need a new `clone` system call for the `fd` out parameter
- `clone` syscall parameters vary by architecture

Complications

- Need a new `clone` system call for the `fd` out parameter
- `clone` syscall parameters vary by architecture
  - Avoided in the new syscall

Complications

- Need a new `clone` system call for the `fd` out parameter
- `clone` syscall parameters vary by architecture
  - Avoided in the new syscall
- `clone` is out of parameters (6) on some architectures

Complications

- Need a new `clone` system call for the `fd` out parameter
- `clone` syscall parameters vary by architecture
  - Avoided in the new syscall
- `clone` is out of parameters (6) on some architectures
  - Pass parameters via a struct and size

Complications

- Need a new `clone` system call for the `fd` out parameter
- `clone` syscall parameters vary by architecture
  - Avoided in the new syscall
- `clone` is out of parameters (6) on some architectures
  - Pass parameters via a struct and size
- Low-level `copy_thread` function grabbed `tls` parameter directly from syscall register arguments; couldn't move it

Complications

- Need a new `clone` system call for the `fd` out parameter
- `clone` syscall parameters vary by architecture
  - Avoided in the new syscall
- `clone` is out of parameters (6) on some architectures
  - Pass parameters via a struct and size
- Low-level `copy_thread` function grabbed `tls` parameter directly from syscall register arguments; couldn't move it
  - Pass parameter normally via C, fix assembly syscall entry
  - Fixed with `copy_thread_tls` (merged in 4.2)

Complications

- Need a new `clone` system call for the `fd` out parameter
- `clone` syscall parameters vary by architecture
  - Avoided in the new syscall
- `clone` is out of parameters (6) on some architectures
  - Pass parameters via a struct and size
- Low-level `copy_thread` function grabbed `tls` parameter directly from syscall register arguments; couldn't move it
  - Pass parameter normally via C, fix assembly syscall entry
  - Fixed with `copy_thread_tls` (merged in 4.2)
- `ptrace` and reparenting

Complications

- Need a new `clone` system call for the `fd` out parameter
- `clone` syscall parameters vary by architecture
  - Avoided in the new syscall
- `clone` is out of parameters (6) on some architectures
  - Pass parameters via a struct and size
- Low-level `copy_thread` function grabbed `tls` parameter directly from syscall register arguments; couldn't move it
  - Pass parameter normally via C, fix assembly syscall entry
  - Fixed with `copy_thread_tls` (merged in 4.2)
- `ptrace` and reparenting
  - Work in progress

History and status

- Thiago Macieira originally proposed `forkfd` to simplify Qt
- Josh and Thiago started on `clonefd` earlier this year
- Some infrastructure merged into 4.2
- Syscall aimed for future kernel after resolving `ptrace` issues

File descriptor: Userspace reference to kernel object

What else can we do with a reference to `task_struct`?

Process IDs

Process IDs

- Small integers used to reference processes
- Used pervasively in process syscalls
- Enumerated as directories in `/proc`

Process IDs

- Small integers used to reference processes
- Used pervasively in process syscalls
- Enumerated as directories in `/proc`
- Unique within root container
- Container PID namespaces map a subset of these

Process IDs

- Small integers used to reference processes
- Used pervasively in process syscalls
- Enumerated as directories in `/proc`
- Unique within root container
- Container PID namespaces map a subset of these
- PIDs do not hold a reference; can be reused
- Race condition if used from non-parent process

clonefd as process identifier

clonefd as process identifier

- Unique across the entire system

clonefd as process identifier

- Unique across the entire system
- Holds a reference to the process
- Race-free

clonefd as process identifier

- Unique across the entire system
- Holds a reference to the process
- Race-free
- Can pass via exec, UNIX sockets

clonefd as process identifier

- Unique across the entire system
- Holds a reference to the process
- Race-free
- Can pass via exec, UNIX sockets
- Allows non-parent processes to obtain exit information

Next steps

- Merge `clonefd`
- For each PID syscall, add an fd variant
- Add ioctls to obtain process information
- Add process enumeration (next, child, root)

Other future file descriptors

Other future file descriptors

Warning: wild speculation and conjecture ahead

User and group IDs

User and group IDs

- Suppose users and groups were unique kernel objects?

User and group IDs

- Suppose users and groups were unique kernel objects?
- Unique across container user namespaces
- “Get unused user/group”

User and group IDs

- Suppose users and groups were unique kernel objects?
- Unique across container user namespaces
- “Get unused user/group”
- Set up arbitrary mappings when mounting a filesystem

User and group IDs

- Suppose users and groups were unique kernel objects?
- Unique across container user namespaces
- "Get unused user/group"
- Set up arbitrary mappings when mounting a filesystem
- Allow a process to hold multiple credentials (like `setgroups`)

Filesystem mounts

Filesystem mounts

- Suppose `mount` returned a directory file descriptor

Filesystem mounts

- Suppose `mount` returned a directory file descriptor
- `openat` relative to the filesystem

Filesystem mounts

- Suppose `mount` returned a directory file descriptor
- `openat` relative to the filesystem
- Separate call to bind into the filesystem namespace
- Bind existing dirfd for bind mounts

Summary

- File descriptor: Userspace reference to kernel object

Summary

- File descriptor: Userspace reference to kernel object
- Reference-counted, race-free, unambiguous ID

Summary

- File descriptor: Userspace reference to kernel object
- Reference-counted, race-free, unambiguous ID
- Well-defined semantics
- Extensive operations

Summary

- File descriptor: Userspace reference to kernel object
- Reference-counted, race-free, unambiguous ID
- Well-defined semantics
- Extensive operations
- `poll` and blocking

Summary

- File descriptor: Userspace reference to kernel object
- Reference-counted, race-free, unambiguous ID
- Well-defined semantics
- Extensive operations
- `poll` and blocking
- Use file descriptors in new APIs
- Don't invent new identifier namespaces