Overview

- This Lecture
  - Advanced char driver
  - Source: LDD ch6, ch9, WLDD ch21

ioctl

- User space library
  - int ioctl(int fd, unsigned long cmd, void *argp);
- Kernel interface to device driver
  - int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
  - arg is either an integer or a pointer
- The function is normally implemented using a switch statement

ioctl commands

- ioctl commands should be unique!
- To avoid command clashes, the following rules are recommended
  - Four bit fields are used
    - Type: magic number, we recommend 'k' (consult *Documentation/ioctl_number.txt*)
    - Number: command ordinal number (0-255)
    - Direction: for data transfer. _IOC_NONE, _IOC_READ, _IOC_WRITE, and _IOC_READ|_IOC_WRITE
    - Size: size of user data involved. This field has 13 or 14 bits
  - The macros used are
    - _IO(type, nr), _IOR(type, nr, datatype), _IOW(type, nr, datatype), _IOWR(type, nr, datatype)
    - E.g #define my_cmd1 _IOW(MY_MAGIC, 1, int)

Predefined commands

- The predefined commands are not passed to drivers!
- They are divided into three groups:
  - Those that can be issued on any file (regular, device, FIFO, or socket)
  - Those that are issued only on regular files
  - Those specific to the file system type
- Device drivers are only interested in the first group, whose magic number is ‘T’
  - It is essential to never use ‘T’ as your magic number

Predefined commands (cont.)

- The following predefined commands are interesting to device drivers
  - FIOCLEX: Set the close-on-exec flag. Setting this flag causes the file descriptor to be closed when the calling process executes a new program.
  - FIONCLEX: Clear the close-on-exec flag
  - FIOASYNC: Set or reset asynchronous notification.
  - FIOQSIZE: This command returns the size of a file or directory; when applied to a device file, however, it yields an ENOTTY error return.
  - FIONBIO: This call modifies the O_NONBLOCK flag in filp->f_flags. The third argument to the system call is used to indicate if the flag is to be set or cleared.

Data transfer

- Care must be taken when transferring data between user space and kernel space!
- *copy_to_user* and *copy_from_user* checks the validation of user space address, but too expensive
- For ioctl, light-weight functions are used to copy single values
  - put_user(data, ptr);
  - get_user(local, ptr);
- Direct validation check can be done with
  - int access_ok(int type, const void *addr, unsigned long size);
  - __put_user and __get_user can be used with it.

Capabilities

- Linux uses more capabilities rather than root and normal user
  - man *capget* and *capset*
- The capabilities interested to device drivers are
  - CAP_NET_ADMIN: The ability to perform network administration tasks.
  - CAP_SYS_MODULE: The ability to load or remove kernel modules.
  - CAP_SYS_RAWIO: The ability to perform "raw" I/O operations.
  - CAP_SYS_ADMIN: A catch-all capability that provides access to many system administration operations.

Capabilities (cont.)

- Capabilities are checked with the function
  - int capable(int capability);
  - Example
    - if (! capable (CAP_SYS_ADMIN)) return -EPERM;
- Issues with capabilities
  - How to decide what operations need which capabilities?
  - How to make sure there is no security hole with more complicated capability scheme?

Blocking I/O

- When the O_NONBLOCK flag in filp->f_flags is set, operations are non-blocking; otherwise they are blocking.
- What the driver should respond if it can't immediately satisfy the request?
  - Put the process to sleep
- A wait queue is needed to keep track of sleeping processes
  - DECLARE_WAIT_QUEUE_HEAD(name);
  - wait_queue_head_t my_queue;
    init_waitqueue_head(&my_queue);

Sleep and wake-up

- The following functions put a process to sleep
  - wait_event(queue, condition);
  - wait_event_interruptible(queue, condition);
  - wait_event_timeout(queue, condition, timeout);
  - wait_event_interruptible_timeout(queue, condition, timeout)
  - Interruptible is recommended, otherwise the process can't be killed.
- The following functions wake up a process
  - void wake_up(wait_queue_head_t *queue);
  - void wake_up_interruptible(wait_queue_head_t *queue);

Non-blocking operations

- For blocking mode, the process waits until some data are read or written
- For non-blocking mode, the process simply return –EAGAIN or -EWOULDBLOCK if there is no data read or written

Blocking example

- How to implement a pipe with blocking mode?

Advanced sleep

- How to manually put a process to sleep?
- How to manipulate sleep on multiple wait queues?

Exclusive wait

- Thundering herds problem
  - Multiple processes are waken up, but only one process can get the resources.
- How to solve thundering herds problem?

Seek a device

- Use *llseek* to change current position of the device
  - loff_t llseek(struct file *filp, loff_t off, int whence)
  - whence should be one of the following values
    - 0: SEEK_SET
    - 1: SEEK_CUR
    - 2: SEEK_END
  - filp->f_pos is changed

Single open device

- Use a counter (set to 1) to record the number of processes allowed to use the device
- When a process opens the device, decreases the counter by one.
- If there is a process already using the device, return -EBUSY
- When a process releases the device, increase the counter by one.

Single user access

- How to allow only processes from a single user?
  - In open function

```
spin_lock(&scull_u_lock);
if (scull_u_count && (scull_u_owner != current->uid) &&  /* allow
   user */
(scull_u_owner != current->euid) && /* allow whoever did su */
!capable(CAP_DAC_OVERRIDE)) { /* still allow root */
spin_unlock(&scull_u_lock);
return -EBUSY;   /* -EPERM would confuse the user */     }
if (scull_u_count == 0)   scull_u_owner = current->uid; /* grab it */
scull_u_count++;
spin_unlock(&scull_u_lock);
```

  - In release function?

Blocking open

- How?

```
spin_lock(&scull_w_lock);
while (! scull_w_available()) {
spin_unlock(&scull_w_lock);
if (filp->f_flags & O_NONBLOCK) return -EAGAIN;
if (wait_event_interruptible (scull_w_wait, scull_w_available()))
    return -ERESTARTSYS; /* tell the fs layer to handle it */
spin_lock(&scull_w_lock); }
if (scull_w_count == 0) scull_w_owner = current->uid; /* grab it
    */
scull_w_count++;
spin_unlock(&scull_w_lock);
```

- What to do at release?

Check read/write flags

- To know if a device file is opened write only?
  - if ((filp->f_flags & O_ACCMODE) == O_WRONLY) {

Clone a device

- We can clone a device at open function
- Clone the device structure for each open
- How can other functions such as read know which device it is operating?
  - Use filp->private_data to store device data structure when open
  - Get the device structure through filp when read or write.

Microkernel

- OS abstractions are implemented with user-space servers
- The kernel provides minimum functions
  - Address space
  - Process creation
  - IPC (inter-process communication)

Why microkernel?

- Isolation (for bugs)
- Fault tolerance
  - Easy to restart a faulty server
- Modular
  - Understandable and replaceable
- Suitable for distributed systems
  - IPC allows servers on other hosts
- Natural concurrency on multiprocessor/multicore
  - Multiple independent servers
- Easy to provide scheduling, priority
- Easy to provide security

History of microkernel

- Individual ideas around at the beginning
- Lots of research projects starting early 1980s
- Big hit with CMU's Mach in 1986
- Was too slow in early 1990s
- Now slowly returning (L4, OKL4, seL4, QNX in embedded systems/routers, etc)
- Ideas very influential on non-microkernel

Microkernels in practice

- Big issue: Unix compatibility
  - It is critical to widespread adoption
  - But Unix not designed in a modular fashion
- Mach, L4: one big Unix server
  - not a big practical difference from a single Linux kernel
  - Mach in particular was quite slow, but L4 improved a lot
- KeyKOS: more interesting structure
  - split up Unix into many entities

The problem

- KeyKOS proposed to solve the access control problem
- Traditional access control model
  - A process has some privileges based on U/GID
  - For each syscall, kernel checks the process' privileges allow it
- What problem the KeyKOS authors faced?
  - Fortran compiler need one more privilege to access /sysx/stat
  - Allowed write access to /sysx
  - user executed "/sysx/fort code.f -o /sysx/bill"

Whose problem?

- The problem was that the compiler was given more privileges than necessary
  - Only /sysx/stat needs to be written
  - But to avoid mistakes, the compiler has to check all places when opening files

Proposal

- Explicitly specify privileges to use for every operation
- Pros: easier to write secure, privileged programs
  - Program will not grant its privileges to things done on user's behalf
- Cons
  - Invasive design implications
  - Have to pass around capabilities instead of user-meaningful names
  - Notion of a user identity is at odds with a pure-capability design

Capability

- Capability (AKA key)
  - Communicable, unforgeable token of authority
  - A value that references an object along with an associated set of access rights
- Capabilities used in many settings
  - Hardware: Cambridge CAP system, x86 segments (in a way)
  - OS kernel: Hydra, KeyKOS (and its successors)
  - Distributed systems: Amoeba
  - Others: Unix FDs, URLs, Java object refs

KeyKOS

- Capability
  - Used as the base access control mechanism, more structural than Mach
- Very small kernel
  - Provides a few kinds of objects to applications
    - Devices - access to underlying hardware, used by device driver processes
    - Pages - 4KB of data, a page of memory
    - Nodes - a block of 16 keys (key is the term for a capability)

KeyKOS (cont.)

- More objects provided
  - Segments - a virtual address space, like a page table or page directory. It is implemented using nodes, mapping to page keys at the leaves. Segments can be constructed out of other segments
  - Meters - CPU time allocation (CPU-time explicitly allocated!)
  - Domains - something like a Unix process

Domain

- Domains are the most interesting object
  - 16 general-purpose key slots (similar to capability registers). It is effectively an implicit node object
  - Address slot: key to entire virtual memory of process
  - Meter slot: key to CPU meter for process
  - Keeper slot: key to another domain that will handle exceptions

Objects

- Objects are named by keys
  - Key is a 12-byte blob, but its bytes can't be handled directly
  - Key bytes are manipulated through explicit operations, like FDs (open, close, dup)
  - The KeyBits service returns the actual 12 bytes behind any key given to it
  - Unknown: can you supply a virtualized KeyBits to hide the fact that other keys are being virtualized as well?

Kernel API

- At a low level, 3 system calls are provided
  - void FORK(key k, msg m): send m to k's domain, continue running afterwards
  - msg *CALL(key k, msg m): send m to k's domain (+ newly-formed resume key for sender) and suspend
  - msg *RETURN(key k, msg m): send m to k (which will be returned from its CALL) and dequeue the domains waiting for the calling domain.

Domain states

- Domain/process has three states
  - Available, running, waiting
  - State transition diagram?
  - Why require receiver to be available?
- Other system calls are implemented through messages to kernel objects (e.g. devices)
- Kernel design suggests an object-oriented structure for applications

Key operations

- How to give keys to others?
  - Clone keys (like dup on Linux FD)
- Keys include access restrictions like RO
- IPC primitive allowed passing 4 keys
  - They can refer to nodes for more keys
- Can append a byte onto a key on creation
  - Useful for keeping track of the origin of the key
  - Distinguish multiple callers of the same service domain

Object keeper

- Handle exceptions
- Segment keeper
  - A domain to handle page faults
- Meter keeper
  - A domain to call when CPU time expires, controls CPU time allocation
- Domain keeper
  - A domain to call for other CPU exceptions
- Faults look like a CALL from the faulting domain to keeper
  - Invoked keeper gets a "service key" to the faulting object, to fix it up as needed

Bank

- Bank is a domain
  - Top-level bank has all nodes and pages in a system
  - Other objects (e.g. domain) are special forms of a node
- Invoke a bank to allocate a new object
  - Returns a “service key” to the object
  - For domain object, with the service key, it can populate the domain’s slots and make it running

Persistent single-level store

- Applications have no notion of a disk, just memory
  - Kernel periodically saves complete system state to disk
- Applications store data in memory
  - Kernel saves it to disk eventually
  - To access data, just access memory and the kernel will demand the page if necessary

Difference from Linux

- Differences of KeyKOS
  - No file system root, every user has their own home directory (names mapped to keys)
  - Keys include both files and processes that can be invoked
  - One user's home dir is not namable by other users (unless granted)
  - Every user has a persistent shell domain that keeps a key to the user's home dir
  - A login process keeps a password and start key for each user's shell domain
  - User types in password, login does an RPC call into user's shell, passing in a key for user's terminal
  - When invoking a command in shell, must say if argument is a string or a capability for file of that name
  - Must specify all capabilities at invocation time

KeyNIX

- Use Unix keeper for each Unix process to emulate Unix
  - Shared state stored in explicitly shared segment
- File system
  - Separate domain for each inode
  - Implement Unix access control model
- Capabilities have to be implemented at all levels
- Performance is ok, but hard to know why

Discussion

- Comparison with other microkernels
- Comparison with exokernel
- Comparison with Unix/Linux
- Why didn't pure-capability systems catch on?
- How about microkernels?