Today's plan

- Virtual Memory in seL4
- Minix device drivers
- Minix block driver and block I/O
- Minix RAM disk driver

Virtual Memory in seL4

- seL4 Reference Manual, Chapter 7
- a `Page` represents a physical memory frame
  - Pages support `map`, `unmap`, `remap`, `GetAddress`
- ASID is an Address Space ID, usually mapped to a hardware resource
- two-level page tables for 32-bit architectures, 4-level page tables for 64-bit architectures
  - some levels use 8 bits, some 9, some 10, some 12
- Page faults send a message to the exception handler for the thread
  - replying to the message restarts the thread

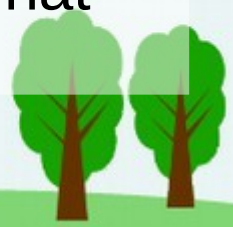
Minix I/O

- structure reflects conceptual structure of:

1. device-specific I/O: drivers, implemented as driver processes
2. device-independent I/O: server layer
3. user-level I/O: user process layer

Minix device drivers

- device drivers execute as separate processes, and (after initialization) only when they receive a message
- user processes send messages to the file system or PM server, and servers send messages to the device drivers
- device drivers may also receive messages from the interrupt handlers
- interrupt handler messages carry no data, they are just notifications
- messages from the servers may carry information about the (minor) device number, which process is requesting the I/O, how many bytes are requested, where on the device and where in memory the bytes are or should be placed, and especially what operation is requested



Minix device driver threading

- each device driver may be interrupted, but no variables are shared with other device drivers, so there can be no race conditions: each device driver is much like a monitor
- device drivers may receive messages, and therefore suspend
- the block-device drivers will only accept messages from the interrupt handler while they are waiting for a result, so (like a non-waiting monitor) these device drivers are not re-entrant
- the terminal task will accept keyboard input while waiting for a response from a serial line read, but will not accept further requests for keyboard input until the first one has been satisfied

Minix device driver structure

- a device driver must:

1. initialize its device(s)
2. loop forever,
   A) receiving a message (usually device drivers block here)
   B) carrying out the message (occasionally device drivers block here)
   C) sending a response (device drivers should never block here)

- all the block device drivers (RAM, floppy, hard disk, CD, SCSI) share similar functions, and are implemented with a single, generic main loop in `drivers/libdriver/driver.c`, p. 773
- the generic main loop uses an array of function pointers (of type `struct driver`, p. 770) to execute device-specific operations

Minix block driver

- for example, a `dev_open` message results in a call to `(*dp->dr_open)(dp, &mess)`
- `drivers/libdriver/driver.c` contains generic versions for some of these functions, for example `do_rdwt`, which calls

```
(*dp→dr_prepare)(mp→DEVICE);
...
(*dp->dr_transfer)(mp->PROC_NR, opcode, mp->POSITION, &iovec1, 1);
...
```

- at the end of each request, the device gets to

```
(*dp->dr_cleanup)();
```

Minix block driver, details

- if a block device chooses to use `do_rdwt`, it can implement `dr_prepare`, `dr_transfer`, and `dr_cleanup` to provide the desired functionality
- for example, for the RAM disk, `m_transfer` can perform the I/O, and `m_prepare` just verifies the device number and returns the device base and size
- in contrast, for the hard disk (`at_wini`), `w_transfer` sets up the operation, then waits for the transfer to complete
- the wait can be waiting for an interrupt (line 12898) in case of read, or waiting in a busy loop (line 12909) in case of a write
- in-class question: the "interrupt wait" waits for an interrupt, but will receive from ANY (line 13132), which means it may discard a message. Why not listen to a message from the hardware only? Why does the driver never get a message from another source while waiting for its interrupt? (5 minutes)

Structure of File I/O and Block I/O

- all device-independent file operations are done by the FS server
- this includes interfacing to device drivers, buffering, error reporting, and using a device-independent block size

Block I/O operations

- open a device -- check to make sure it is actually there, initialize it, read the partition table
- close a device
- read a block, specifying the start location on disk, the number of bytes, and the buffer address
- write a block, specifying the start location on disk, the number of bytes, and the buffer address
- IOCTL a device, getting or setting the partition table to/from memory
- do a scattered read or a scattered write, providing an array of blocks to be read or written

Scattered I/O operations

- `do_vrdwt`, p. 775
- each scattered operation is a collection of reads or writes of individual (variable sized) blocks
- each Minix 3 scattered operation is all reads or all writes
- each Minix 3 scattered operation is sequential on the device, but the source/results may be scattered among different buffers
- optional reads allow for prefetching, and the device driver can decide whether prefetching is a good idea (e.g. not beyond the end of a track)

Partition Table

- a partition table for the x86 is a list of 4 partition entries at offset 446 (x1BE) in the first (boot) sector of a disk
- each partition entry has a 16 bytes of data (the last 2 bytes of the first sector hold the magic number 55AA, line 11571)
- the first 8 bytes include flags (e.g. 0x80 for a bootable disk) and geometry (heads, sectors, cylinders)
- the final 8 bytes are two 4-byte numbers recording the linear number of the start and size of the partition, in sectors
- partitions may be nested, e.g. the first partition may have another partition table in its first sector

Minix RAM disk

- RAM disk is block-oriented, just like the real disks, even the block size is the same
- RAM disk is stored in main memory: a 1MB RAM disk takes up 1MB of main memory
- write requests write directly to memory
- read requests copy data directly from memory
- six different Minix RAM devices, all with major device number 1 and using the same driver:
  - minor device 0: `/dev/ram`, a true RAM disk on which the root file system is usually mounted. Memory for this is allocated at boot time
  - minor device 1: `/dev/mem`, a pseudo RAM disk corresponding to the physical memory of the PC
  - minor device 2: `/dev/kmem`, a pseudo RAM disk corresponding to the kernel virtual memory of Minix
  - minor device 3: `/dev/null`, a pseudo RAM disk that discards all its data and is always empty
  - minor device 4: `/dev/boot`, a pseudo RAM disk that gives access to the in-memory copy of the boot device
  - minor device 5: `/dev/zero`, a pseudo RAM disk that discards all its data and returns as many bytes of zero as requested
- the first three RAM disks and `/dev/boo`t have some overlap: aliasing, different names for the same underlying objects

Minix RAM disk implementation

- pages 782-787
- many functions (e.g. `close`, `cleanup`) are no-ops
- `main` in the memory driver calls `m_init` (which sets up each device's pointers and sizes) and then goes to the shared `driver_task` function
  - open checks the device number
  - exactly one ioctl is supported, to set the size of a RAM disk (only FS may do this)
- doing this requires allocating memory, which eventually leads to a call to the process manager (PM)
- the code in `servers/vm/alloc.c` implements the first-fit memory allocation algorithm (`findbit`, lines 369-399)
  - current code calls this via `mmap` (line 588 in `drivers/storage/memory/memory.c`)
- `m_geometry` returns the (completely fake) geometry
- reading or writing of `/dev/null` is a question of returning the correct number of bytes read or written (0 read, all written)

Reading or writing a RAM disk block

- `m_transfer`, p. 783-785
- ignore the optional bit -- all operations on a RAM disk complete immediately
- one iteration per request in the vector
- for "normal" memory devices, check for address legality, i.e. within the RAM disk, and truncate the transfer if necessary
- select the appropriate segment
- perform the virtual or physical copy
- update the number of bytes copied