In Minix, device drivers are processes that run in user-space. Each driver usually handles multiple devices of the same type (e.g., all disks, all keyboards, all memory devices, etc). These processes perform privileged (kernel-mode) operations by sending and receiving messages to/from the system task.

This is (mostly) unlike monolithic kernel systems where user processes go into kernel mode for short bursts by making system calls.

Minix' process-structured approach tends to be less efficient but more reliable and modular (clearer distinction between system components).
The clock task is the exception: it is part of the kernel.

Clock task registers a customized interrupt handler that does the following work without sending a message to the clock task every time:

- increment realtime variable
- decrements ticks_left for currently scheduled user process
- check whether a timer has expired

The interrupt handler sends a message to clock task only if a timer has expired or scheduled process used up its time quantum.

The clock task then decides what to do (reschedule, send message, etc.)

Every other interrupt-driven device driver runs in user space, and can therefore not directly access IO ports or kernel memory.

   (this is actually possible, but against Minix design principles)
Interrupts

In Minix, user-space processes (drivers, services, applications) cannot specify interrupt handlers to the CPU. Instead they can ask the kernel to send a message when a specified interrupt occurs.

(Almost) all interrupts are caught by a single kernel routine, which then analyzes the interrupt and dispatches a message to the process that requested the interrupt of that type. It is in `kernel/system/do_irqctl.c`

Down side: it is not possible to specify interrupt handlers that do more than just pass a message.

Most IO drivers usually
- start an IO device
- block, waiting for a message from the kernel
- perform some operation in response

The message is usually generated by an interrupt from the device.

Some IO drivers do not use physical IO (eg. the memory virtual device driver for `/dev/mem` etc), and do not wait for interrupts (ie. don't wait for an interrupt message from the kernel)
IO access

Device drivers might need access to IO and memory in ways that aren't normally possible for user mode processes:

- Access to memory outside normal data space (eg. memory driver)
- Read/write to IO ports to communicate with device controllers (eg. hard disk)
  - CPU instructions for doing this only available in kernel mode
- Responding to expected ("predictable") interrupts
  - eg. disk driver issues a write, waits for interrupt upon completion
- Responding to "unpredictable" interrupts
  - eg. keyboard driver may receive interrupts at any time (or none at all!)

These are all supported by *kernel calls* handled by the *system task*. 
Interrupt Handlers and I/O Access in MINIX 3

- **Case 1**: Access to memory outside its normal data space
  - A normal process
    - Has access only to its own text, data, and stack segments
  - The system task
    - Allows other segments to be defined and accessed by user-space processes
    - For example, the memory driver
      - Can access a memory region reserved for use as a RAM disk
      - And also access other memory regions designated for special access

- **Case 2**: Reading and writing to I/O ports
  - Kernel calls are provided to use I/O instructions
  - System task does the actual I/O for less-privileged processes
Case 3: Responding to predictable interrupts

- An interrupt is initialized on behalf of a user space process
  - This is done by a device driver making a kernel call to the system task
  - The handler routine for the interrupt is always \texttt{generic_handler}
    - A function defined as part of the system task
    - Converts the interrupt into a notification message to the process on whose behalf the interrupt was set
- The device driver must
  - Initiate a \texttt{receive} operation after the kernel call that issues the command to the controller (effectively blocking the driver until...)
- When the notification is received
  - The device driver can do what must be done to service the interrupt
  - Deal with “expected” interrupt failure, using a watchdog timer to notify
  - Interrupt policy can be set to automatically reenable an interrupt.
Interrupt Handlers and I/O Access
in MINIX 3

• **Case 4:** Responding to unpredictable interrupts
  • When you don’t know when, if ever, an interrupt might occur (typically the keyboard)
    • The driver can’t just make a `receive` call to accept input from a single source (receive blocks indefinitely; remember that each driver may service multiple devices of the same type)
    • The same process may need to respond to other input and output sources
    • Instead, non-blocking, non-queued message calls are used (notify)

However: this means that a notification may arrive while another is being processed, so interrupts might be lost (and therefore keystrokes, in the case of the keyboard).

• MINIX tries to make the interrupt response as fast as possible (little code in notification message handler)
When a user program wishes to open a certain file, it calls eg. the C stdlib fopen call, which sends a message to the filesystem server, which sends a message to the appropriate block device's driver (eg. hard disk).

The IO driver unblocks when receiving that message, then performs some action and sends a message back to the filesystem, which sends a message to the user process, and the fopen call returns.

```c
message mess;           /* message buffer*/

void io_driver() {
  initialize();         /* only done once, during system init.*/
  while (TRUE) {
    receive(ANY, &mess); /* wait for a request for work*/
    caller = mess.source; /* process from whom message came*/
    switch(mess.type) {
      case READ:                 rcode = dev_read(&mess); break;
      case WRITE:                rcode = dev_write(&mess); break;
      /* Other cases go here, including OPEN, CLOSE, and IOCTL*/
      default:                   rcode = ERROR;
    }
    mess.type = DRIVER_REPLY;
    mess.status = rcode;      /* result code*/
    send(caller,&mess);       /* send reply message back to caller*/
  }
}
```

Figure 3-18: Simplified version of a block device driver
Device-independent IO code is contained in the filesystem.

A core piece of Unix philosophy is: "Everything is (accessible as) a file"

Since the FS and IO systems are so closely related, the FS and device-independent IO are implemented inside the same program in Minix (the filesystem server).

Minix' FS can currently not handle device-locking
  eg. a single process taking exclusive control of a CD writer
However this should be straightforward to implement if needed in the future.

In practice, even though mutual exclusion may not be enforced by the operating system, user-processes would typically be written not to mess with eachother's devices
  eg. Minix has a printer spooling daemon (lpd), so other programs should rather make use of that instead of writing to the printer device directly.

User-level functions ("above" device-independent code) are provided in the standard C library (printf, scanf, fopen, etc.)
  (higher-level languages' IO facilities typically just perform C standard library calls as it is relatively simple to call C functions from most other languages)
Deadlock handling

- MINIX 3 *mostly* ignores deadlocks
- (Currently) no mutual exclusion on resources
- The only place deadlocks can occur is with system processes
  - *Implicitly* shared resources
    - Process table slots
    - i-node table slots
    - And so on...
- No deadlock algorithm can handle such resources anyway
- Message passing is designed to avoid deadlock
  - Only `send` or `sendrec` can be used from higher-level processes
  - Therefore, no `receive` without a driver process that wants to reply
  - Sending is handled by `mini_send` in `kernel/proc.c:192` which fails if a process tries to block by sending a message to another process which is also blocking trying to send a message to the former.