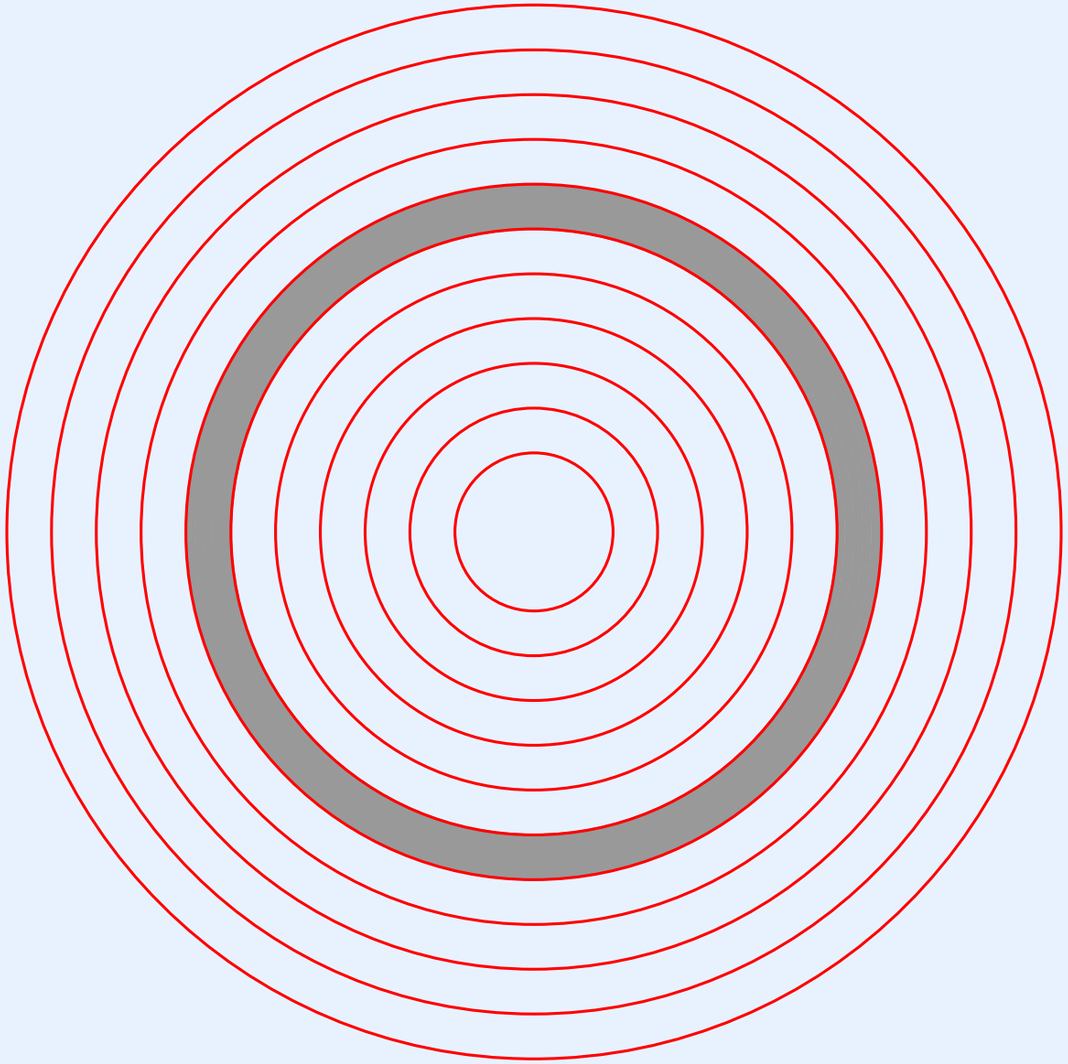
Module XIV

Device Management
Device Drivers
Device-Independent I / O

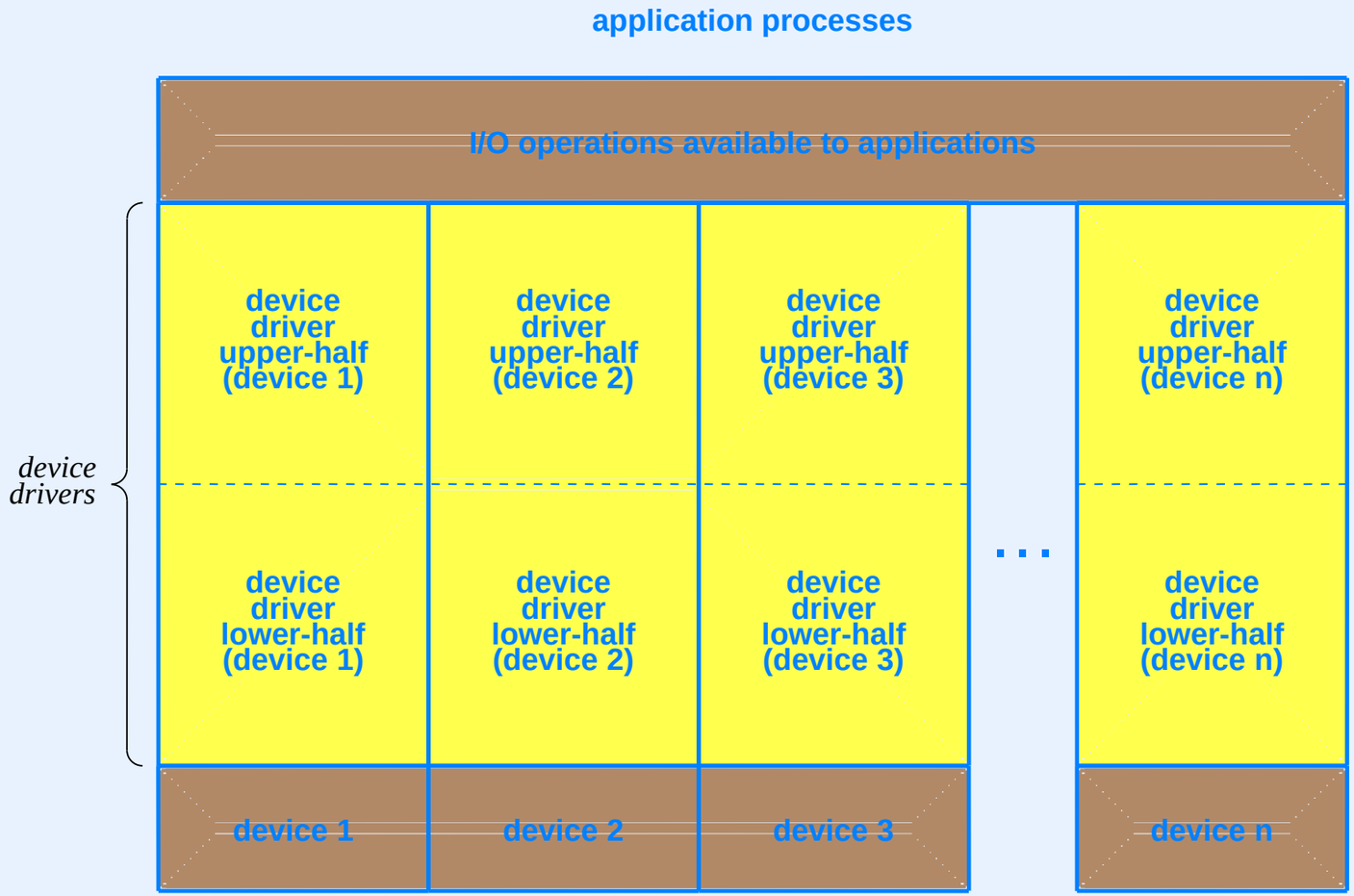
Location Of Device Management In The Hierarchy



Conceptual Organization Of Device Software

- Three conceptual pieces
  - Abstract interface (high-level I/O operations)
  - Set of physical devices
  - Device driver software that connects the two
- We will see that each device driver can be divided into two parts
  - An upper-half that applications call
  - A lower-half that handles interrupts

Conceptual Organization Of Device Software (continued)



Interface And Driver Abstractions

- Two abstractions are needed
  - The I/O interface the operating system offers to applications
  - The interface offered by the underlying device driver functions

Goals For The Device Interface Applications Use

- Isolation from hardware: ensure that applications do not contain details related to device hardware
- Portability: allow applications to run on any brand or model of equivalent device unchanged
- Elegance: limit the interface to a minimal number of orthogonal functions
- Generality: use a common paradigm across all devices
- Integration: integrate the device manager with the process manager and other operating system facilities

Achieving The Goals

- Devise a small set of functions that applications use to
  - Obtain incoming data from a device
  - Transfer outgoing data to a device
  - Control the device
- Examples of controlling a device
  - Adjust the volume on headphones
  - Turn off character echo when reading a password
  - Eject a USB drive
- The approach is known as a *device-independent* interface

Achieving Device-independent I / O

- Define a set of abstract operations
- Build a function for each operation
- Have each function include an argument that a programmer can use to specify a particular device
- Arrange an efficient way to map generic operation onto code for a specific device

Definition Of A Device Driver

- A *device driver* consists of a set of functions that perform I/O operations on a given device
- The code is device-specific
- The set includes
  - An interrupt handler function
  - Functions to control the device
  - Functions to read and write data. The code is divided into two conceptual parts

The Two Conceptual Parts Of A Device Driver

- The upper-half
  - Functions that are executed by an application
  - The functions usually perform data transfer (*read* or *write*)
  - The code copies data between the user and kernel address spaces
- The lower-half
  - Is invoked by the hardware when an interrupt occurs
  - Consists of a device-specific interrupt handler
  - May also include dispatcher code, depending on the architecture
  - Executed by whatever process is executing
  - May restart the device for the next operation

Division Of Duties In A Driver

- The upper-half functions
  - Have minimal interaction with device hardware
  - Enqueue a request, and may start the device
- The lower-half functions
  - Have minimal interaction with application
  - Interact with the device to
    * Obtain incoming data
    * Start output
  - Reschedule if a process is waiting for the device

Xinu's Device-Independent I/O Primitives

Operation	Purpose
close	Terminate use of a device
control	Perform operations other than data transfer
getc	Input a single byte of data
init	Initialize the device at system startup
open	Prepare the device for use
putc	Output a single byte of data
read	Input multiple bytes of data
seek	Move to specific data (usually a disk)
write	Output multiple bytes of data

- Xinu adopts the open-read-write-close paradigm of Unix
- Some abstract functions may not apply to a given device

Synchronous Vs. Asynchronous Semantics

When using a synchronous I/O interface, a process is blocked until the operation completes. When using an asynchronous I/O interface, a process continues to execute and is notified when the operation completes.

Xinu's Synchronous Semantics

- Like many modern systems, Xinu uses *synchronous* semantics
  - When a process attempts to receive incoming data from a device, the process blocks until the data arrives
  - When a process attempts to send outgoing data to a device, the process blocks until data can be transferred or placed in a buffer where it stays until the device finishes transferring it

Coordination Of Processes Performing Synchronous I/O

- A device driver must be able to block and later unblock application processes
- Good news: there is no need to invent new coordination mechanisms because standard process coordination mechanisms suffice
  - Message passing
  - Semaphores
  - Suspend/resume
- We will see examples later

Implementation Of Device-Independent I/O In Xinu

- An application process
  - Makes calls to device-independent functions (e.g., *read*)
  - Supplies the device ID as parameter (e.g., ETHER or CONSOLE)
- The device-independent I/O function
  - Uses the device ID to identify the correct hardware device
  - Invokes the appropriate device-specific function to perform the specified operation
- Examples
  - When a process reads from the ETHER device, the device manager invokes *ethread*
  - When a process reads from the CONSOLE, the device manager invokes *ttyread*

Mapping A Generic I/O Function To A Device-Specific Function

- The mapping must be extremely efficient
- Solution: use a two-dimensional array known as a *device switch table*
- The device switch table
  - Is a kernel data structure that is initialized at compile time
  - Has one row for each device
  - Has one column for each possible I/O operation
- An entry in the table points to a function to be called to perform the operation on the device
- A device ID is chosen to be an index into rows of the table

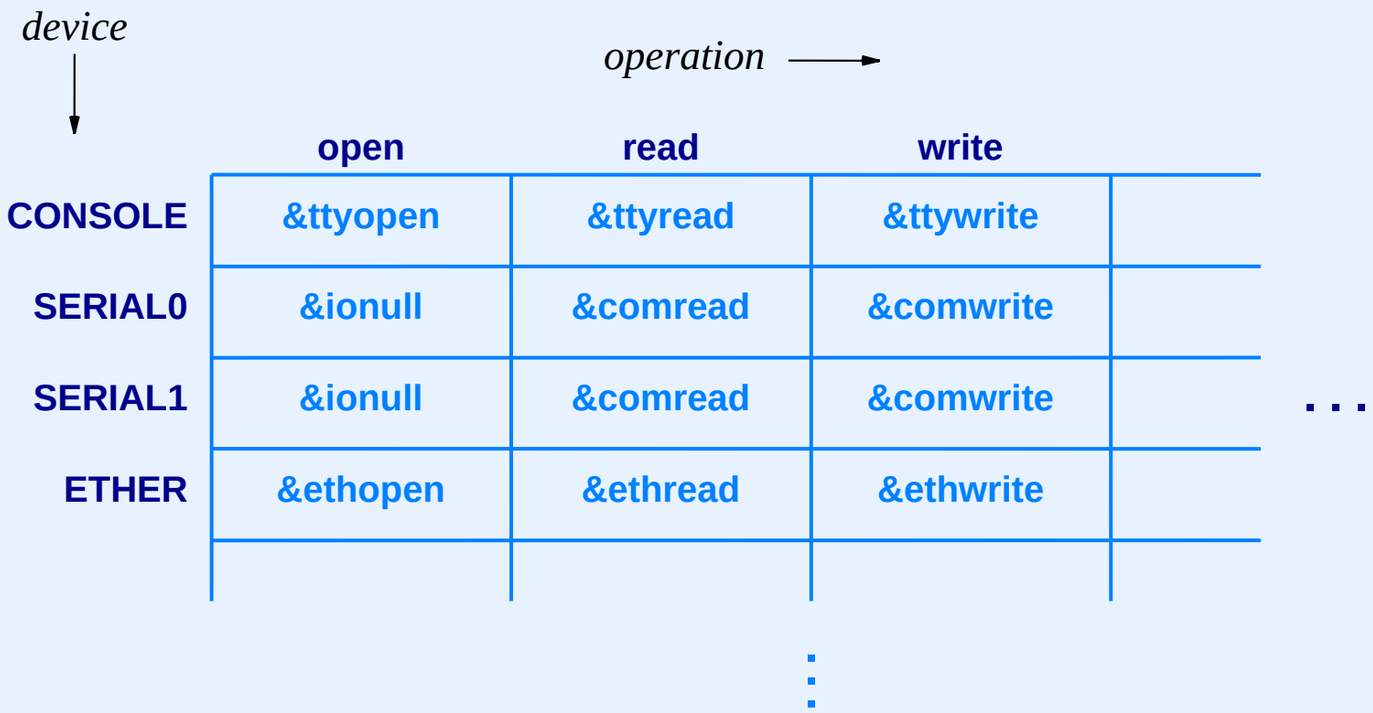
Entries In The Device Switch Table

- A given device-independent operation may not make sense for some devices
  - *Seek* on a keyboard, network interface, or audio output device
  - *Close* on a mouse
- To avoid special cases in the code
  - Make each entry in the device switch table point to a valid function
  - Use special functions for cases where an operation does not apply to a specific device

Special Entries Used In The Device Switch Table

- *ionull*
  - Used for an innocuous operation (e.g., *open* for a device that does not really require opening)
  - Simply returns *OK*
- *ioerr*
  - Used for an incorrect operation (e.g., *putc* on disk)
  - Simply returns *SYSERR*

Illustration Of A Device Switch Table



- Each row corresponds to a device and each column corresponds to an operation
- An entry specifies the address of a function to invoke
- The example uses *ionull* for *open* on devices *SERIAL0* and *SERIAL1*

Replicated Devices And Device Drivers

- A computer may contain multiple copies of a given physical device
- Examples
  - Two Ethernet NICs
  - Two disks
  - Two monitors
- Goal: have one copy of device driver code for each type of device and use the same code with multiple devices

The Point About Devices And Drivers

Instead of creating a device driver for each physical device, an operating system maintains a single copy of the driver for each type of device and supplies an argument that permits the driver to distinguish among multiple copies of the physical hardware.

Parameterized Device Drivers

- A device driver must
  - Know which physical copy of a device type to use (e.g., which disk)
  - Keep information about each physical copy of a device separate from information for other physical devices (e.g., maintain separate information for each disk)
- To accommodate multiple copies of a device
  - Assign each instance of a replicated device a unique number (0, 1, 2, ...) known as its *minor device number*
  - Store the minor device number in the device switch table
  - Example 1: for two disks of the same type, assign minor numbers 0 and 1
  - Example 2: for three NICs of the same type assign minor numbers 0, 1, and 2
- The point: minor numbers only distinguish among devices of the same type

Device Names

- Previous examples have shown examples of device names used in code (e.g., *CONSOLE*, *SERIAL0*, *SERIAL1*, *ETHER*)
- The device switch table is an array, and each device name is an index into the array
- How does the system know how many rows to allocate in the table?
- How are unique values assigned to device names?
- How are minor device numbers assigned for replicated devices?
- Answer: it's automatic — a configuration program takes device information as input, including names to be used for devices, and generates the definitions and the device switch table entries automatically

Device Configuration

- We will see more details later; for now, it is sufficient to know that
  - The OS designer creates a file named *Configuration* that
    * Lists devices in the system and gives each a name (e.g., CONSOLE)
    * Specifies a *type* for each device
    * Specifies the driver functions to use for each operation on the device (open, close, read, write, putc, getc, etc.)
  - The config program generates two files
    * A file named *conf.h* that contains declarations for data structures used in the device switch table
    * A file named *conf.c* that contains a definition of the device switch table with initial values specified, including minor numbers

Initializing The I/O Subsystem

- At system startup
  - Entries in the device switch table are already initialized
  - The interrupt vectors (and perhaps the bus) must be initialized
  - The *init* function is called for each device, which initializes both the device hardware and the driver (e.g., creates the semaphores the driver uses for coordination)
- In lab, you will create a driver and understand how an array can hold information for a set of replicated devices and how the minor number of the device corresponds to an index into the array

The Implementation Of High-Level I/O Functions

- Each high-level function (e.g., *open*, *close*, *read*, *write*)
  - Takes a device name as an argument
    - Uses the device switch table to choose the device-specific driver function to use
  - Invokes the driver function
- Basically, a high-level I/O function uses one level of indirection to invoke an underlying driver function
- An example will clarify the approach used

An Example High-Level I/O Function

```
/* read.c - read */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  read  -  Read one or more bytes from a device
 *------------------------------------------------------------------------
 */
syscall read(
          did32         descrp,         /* Descriptor for device        */
          char          *buffer,        /* Address of buffer            */
          uint32        count           /* Length of buffer             */
        )
{
        intmask         mask;           /* Saved interrupt mask         */
        struct dentry   *devptr;        /* Entry in device switch table */
        int32           retval;         /* Value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvread) (devptr, buffer, count);
        restore(mask);
        return retval;
}
```

Summary

- The *device manager* in an operating system provides an interface that applications use to request I / O
- Device-independent I / O functions
  - Provide a uniform interface
  - Define generic operations that must be mapped to device-specific functions
- Xinu uses a device switch table to map a device-independent operation to the correct driver function
- A device driver for each device consists of
  - Upper-half functions that each implement a high-level operation on the device (e.g., *read* and *write*)
  - A lower-half function that handles interrupts

Questions?