Input/Output – 3
I/O Software

CS 342 – Operating Systems
Ibrahim Korpeoglu
Bilkent University
Department of Computer Engineering

Programmed I/O Disadvantage

- Assume printer prints at a rate 100 chars/second
  - Each character takes 10ms to print.
- During this 10 ms, CPU will be busy with checking the status register of printer (controller).
  - This is waste of CPU
  - During 10ms period something else can be made (An other process can be run).

Interrupt Driven I/O

- After copying application buffer content into kernel buffer, and sending 1 character to printer,
  - CPU does not wait until printer is READY again.
  - Instead the scheduler() is called and some other process is run.
    - The current process is put into *blocked* state.
  - Printer controller generates an hardware interrupt:
    - When the character is written to the printer and when printer becomes READY again.

Interrupt Driven I/O

```
copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY)
{
};  /* loop until printer controller becomes READY */
*printer_data_register = p[0]; // send to the first character
                                              to the printer
controller */
scheduler(); /* do some other task */
```

Code executed in Kernel when the print() system call is made by the application process.

Interrupt Driven I/O

```
If (count == 0) {      /* we are finishes printing the string */
        unblock_user(); /* unblock the user process that
                                     wanted to print the string */
} else {
        /* copy one more character to the controller buffer */
        *print_data_register = p[i];
        count = count – 1;
        i = i + 1;
}
acknowledge interrupt();
return_from_interrupt(); /* finished serving the interrupt */
```

Interrupt Service Routine *that is executed when the printer controller becomes READY for one more byte again.*

I/O Using DMA

- Disadvantage of Interrupt Driven I/O is that interrupt occurs on every character.
- Interrupt take time, so this scheme wastes some amount of CPU time.
  - Solution is use of DMA
- DMA controller will feed the characters from *kernel buffer* to the *printer controller*.
  - CPU is not bothered during this transfer
- After transfer of whole buffer (string) is completed, then the CPU is interrupted.

I/O Using DMA

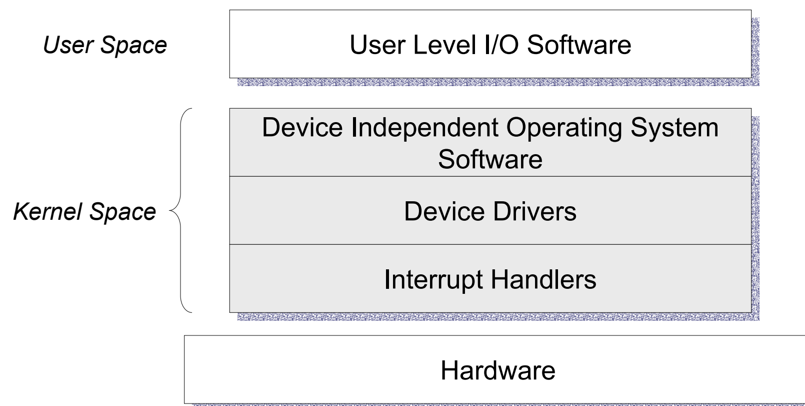
```
copy_from_user(buffer, p, count)
{
        setup_DMA_controller();
        scheduler();
}
```

Code executed in Kernel when the print() system call is made by the application process.

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

Interrupt Service Routine *that is executed when a 'transfer completed interrupt' is received from DMA.*

I/O Software Layers



Interrupt Handlers

- The interrupts are handled at the lowest possible layer.
- A device derives initiates an I/O operation.
  - Then driver blocks (if it is implemented as a separate process) by using one of the following ways:
    - Do a *down*() operation on a semaphore
    - *Wait*() on a condition variable
    - Call *receive*() on a message
- When I/O completes, interrupt occurs.
  - Interrupt Handler Routine is exeuted
  - Driver is unblocked (by up() on semaphore, etc.) so that it can continue to run.

Processing Interrupts

- Not an easy task. It involves al lot of critical steps.
- Following are steps executes in *software* after hardware part of interrupt handling is done.

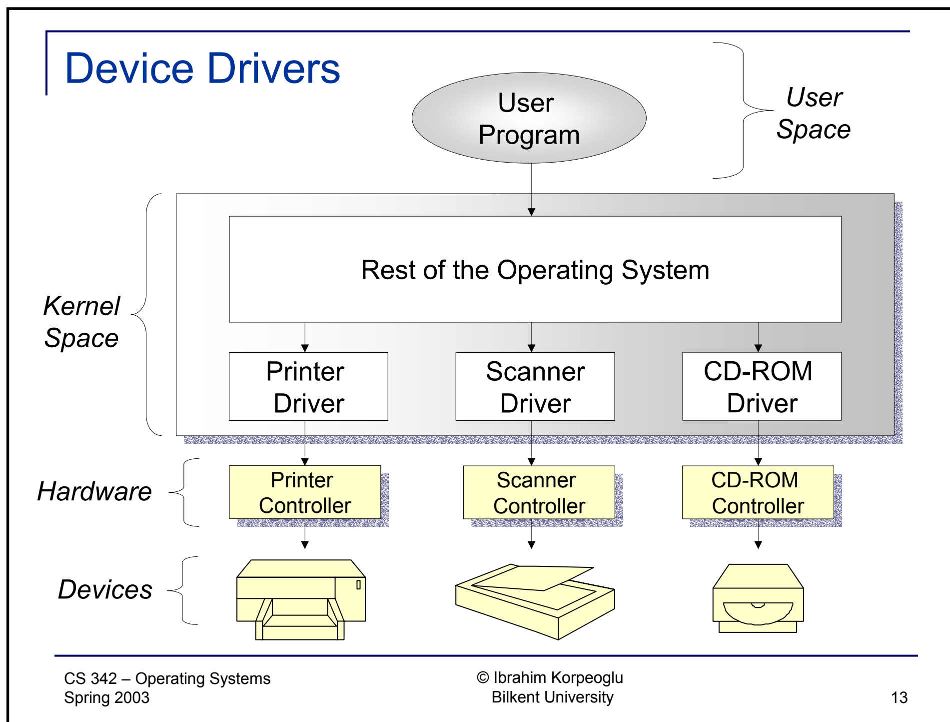
1. *Save any registers including the PSW*
2. *Setup context for the ISR: setup TLB, MMU, page table, etc.*
3. *Setup a stack for interrupt service routine (ISR)*
4. *Acknowledge the interrupt controller (re-enables interrupts)*
5. *Copy registers from where they were saved by hardware to the process table.*
6. *Run the interrupt service routine. ISR will examine the device controller registers.*
7. *Choose which process to run next.*
8. *Set up the MMU context for the process to run.*
9. *Load the new process' registers, including it PSW.*
10. *Start running the new process.*

Device Drivers

- A device controller has:
  - Control registers
  - Status registers
  - Data buffers
- This structure changes from device to device
  - A mouse driver should know how far the mouse has moved and which buttons are pressed
  - A disk driver should know about sectors, tracks, cylinders, heads, arm motion, motor drives, head settling times, …
- Each I/O device need some device specific code for controlling it.
  - This code is called *device driver*.

Device Drivers

- Generally written by device manufacturers
  - Not necessarily by OS writers.
- A driver can be written for more than one OS.
- Driver is then integrated to the OS.
- Each driver can handle one type of device
  - A SCSI disk
  - A floppy drive
  - A network card
  - Mouse
  - Joystick
    - Each will have different drivers
- Device Drivers are usually part of the OS (although they may be integrated later)



Device Drivers

- We need to have a well-defined model of what a driver does and how it integrates with the rest of the OS.
  - This is because, OS implementers and derivers implementers may be different.
- OSs usually classify drivers into two classes
  - Drivers for block devices
  - Drivers for character devices.

Device Drivers

- Most OSs define a *standard interface*
  - that all block drivers must support, and
  - that all character drivers must support.
- The rest of the OS uses this *interface* to talk to the drivers.
  - A driver implementer should follow this interface.
  - This interface include procedures such as:
    - *Read a block,*
    - *Write a stream of bytes (string),*
    - ….

Device Driver Functions

- Accept *abstract* read and write requests from rest of the OS (device-independent part).
  - Carry these requests out.
    - Translate from abstract terms to concrete terms.
      - Example: A disk driver translates from a *linear block address* to a physical *head, track, sector, and cylinder number.*
- Initialize the devices
- Check if the device is currently use for some other request.
    - If so, *enqueue* the request.
- Issue sequence of commands to control the device.

Device Independent I/O Software

- Part of Operating system has device-independent I/O software. This is the software that is nearly all for all I/O devices and that does some common tasks.
  - Uniform interfacing to device derivers
  - Buffering
  - Error reporting
  - Allocating and releasing dedicated devices
  - Providing a device-independent block size.

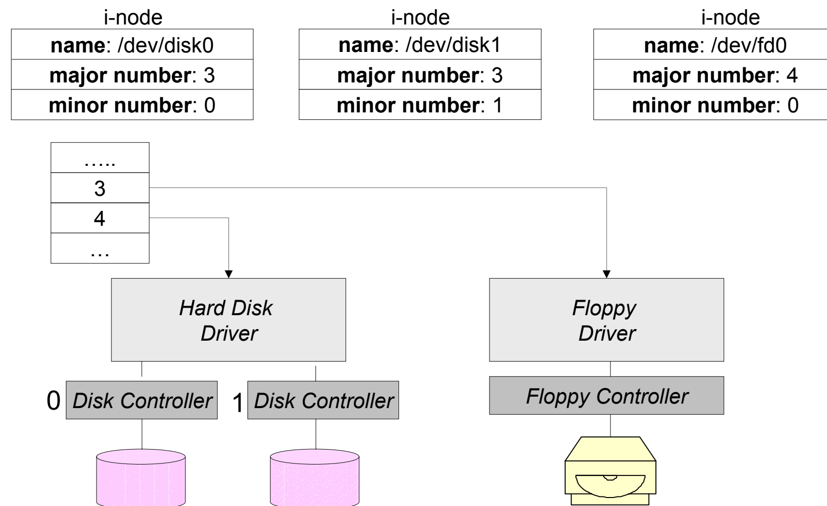
Device Independent I/O Software-Functions

- Uniform interfacing to device derivers
- Buffering
- Error reporting
- Allocating and releasing dedicated devices
- Providing a device-independent block size.

Uniform Interfacing

- Drives need to have uniform interface. The benefits are:
  - The driver implementers know what is expected from them
  - The OS implementers can developed device-independent I/O function on top of a well-defined lower-layer driver interface.
    - They know which functions each driver implements and the pro-types of these function.

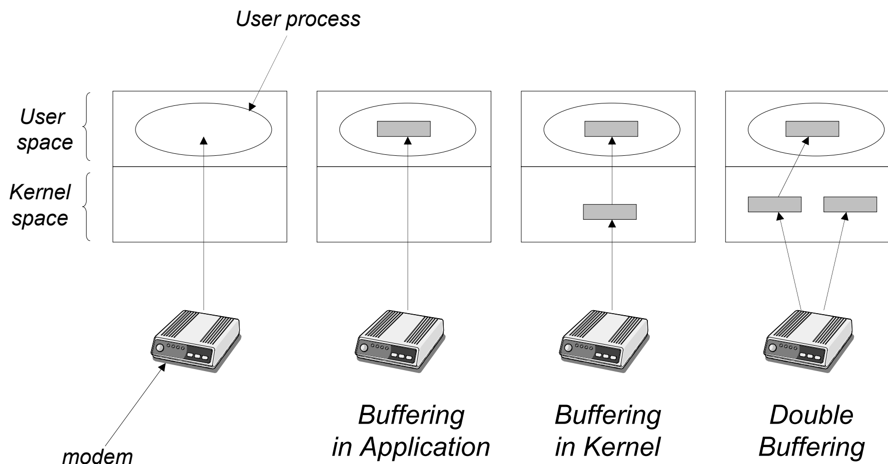
Uniform Interfacing: Mapping symbolic I/O device names to their appropriate drives



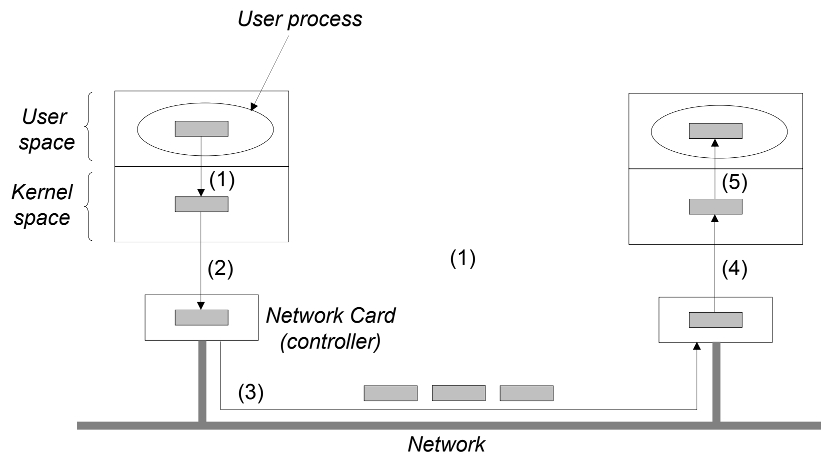
Buffering

- Buffering in kernel (in device independent I/O software layer) is required in order:
  - To reduce the interrupts that will be given to applications when data is available for application to read
  - To not unblock applications unnecessarily for write operations

Buffering



Buffering – Multiple copy operations



Error Reporting

- Some errors are handled by device-controllers
  - *Example*: Checksum incorrect, re-correct the block by using redundant bits
- Some by device derivers
  - *Example*: disk block could not be read, re-issue the read operation for block from disk
- Some by the device-independent software layer of OS.
  - Programming errors:
    - *Example*: Attempt to write to a read-only device
  - Actual I/O errors
    - *Example*: The camcorder is shut-off, therefore we could not read. Return an indication of this error to the calling application.

Allocating Dedicated Devices

- Do not allow concurrent accesses to dedicated devices such as CD-RWs.
  - Each application should issue an open() system call before using a device.
  - Implement open() system call such that it returns an error if the device is currently busy.

Device-independent Block Size.

- There are different kind of disk drives. Each may have a different physical sector size.
- A file system uses a block size while mapping files into logical disk blocks.
- This layer can hide the physical sector size of different disks and can provide a fixed and uniform disk block size for higher layers, like the file system
  - Several sectors can be treated as a single logical block.

User-space I/O software

- This includes
  - The I/O libraries that provides the implementation of I/O functions which in turn call the respective I/O system calls.
    - These libraries also implemented formatted I/O functions such as *printf*() and *scanf*()
  - Some programs that does not directly write to the I/O device, but writes to a spooler.

Layers of I/O system inside an OS