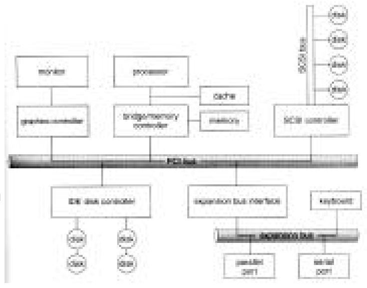
Lecture 20: I/O

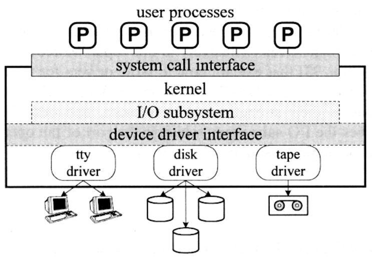
- I/O hardware
- I/O structure
- communication with controllers
- device interrupts
- device drivers
- streams

I/O hardware



- *bus* - a set of wires and a protocol that defines the messages that can be set over the wires
- *controller* - a collection of electronics that can operate a device (a port, a bus, a hard disk)
- if controller is complex it can be implemented as a separate circuit board called *adapter*
- note that the controllers are located on both “sides” of the bus: device controller - bus - host controller

(Unix) I/O Structure



- To decrease complexity of I/O design Unix I/O management is layered
- user applications communicate with peripheral devices via the kernel through system calls
- *I/O subsystem* handles the I/O requests and uses the device driver interface to communicate with the devices
- *device driver* is an independent part of the kernel that contains a collection of data structures and functions that controls one or more devices and interacts with the kernel through a well defined interface

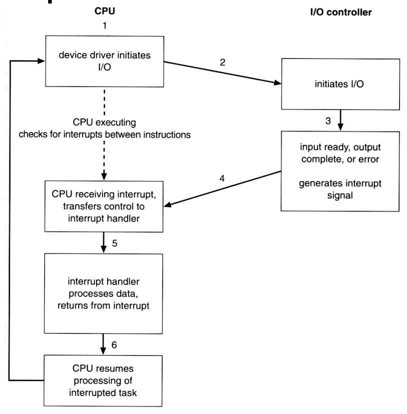
Communication with controllers

- Controller has a set of *control and status registers* (CSR)
- CSRs are device-dependent
- driver writes to CSRs to issue commands and reads them to obtain completion status and error information
- two ways of I/O space configuration:
  - separate I/O space - separate (CPU) instructions are needed to move data to and from controllers
  - memory-mapped device I/O - CSRs are assigned regular memory addresses and data in CSRs can be manipulated by regular memory operations (like store and load); examples - video memory in PCs
- two ways of transferring data between kernel and device:
  - *programmed I/O* (PIO) - data has to be written to the device byte by byte. Whenever the device is ready for the next byte it issues an interrupt; examples - printers, modems, mice, keyboards
  - *direct memory access* (DMA) – CPU just gives the description of the data to be transferred (location, size, etc.) and the DMA controller does the rest communicating with memory independently of CPU
    - variant - direct *virtual* memory access (DVMA) – controller copies data between two memory-mapped devices

Polling and device interrupts

- The kernel communicates with devices by:
  - polling - CPU regularly *polls* CSRs of a device to check if the I/O operation has completed
  - interrupts - a device *raises* an interrupt to alert the CPU that I/O operation has completed
- interrupt handler - a short self-contained routine that responds to an interrupt
- interrupts can be:
  - polled - there is one interrupt handler that upon startup checks all the devices to see which of them needs attention
  - vectored - the device is assigned to a specific interrupt handler and a *interrupt vector table* is kept. The vector contains the addresses of interrupt handlers by the interrupts. When an interrupt is raised the vector is consulted by hardware (rather than CPU) and the corresponding interrupt handler is started

Interrupt flowchart

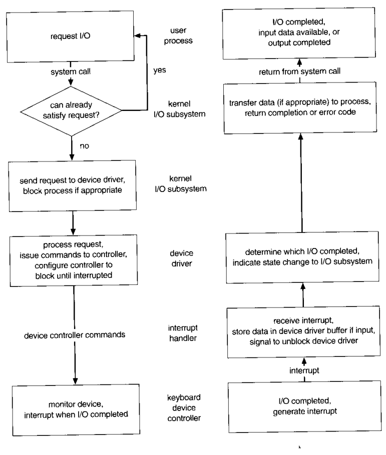


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
3
input ready, output complete, or error
generates interrupt signal
CPU executing checks for interrupts between instructions
4
CPU receiving interrupt, transfers control to interrupt handler
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

(Unix) device drivers

- there is a variety of external devices and ways of communicating with them, to simplify programming – use device drivers
  - *device driver* – kernel module that is coded to communicate with a particular device
- from the I/O subsystem's perspective the device driver is 'black box" that supports a standard set of operations (each device may implement these operations differently)
- Unix supports two device types
  - *block* - stores data and performs I/O in fixed-size, randomly accessible blocks; examples - hard disks, floppy drives, CD-ROMs; due to the structured nature of the I/O operations on block devices efficient cache/buffering algorithms can be used;
  - *character* - can store and transfer arbitrary sized data. May transfer one byte at a time (generating an interrupt after every byte) or perform some internal buffering; examples - keyboard, mouse, clock, modem

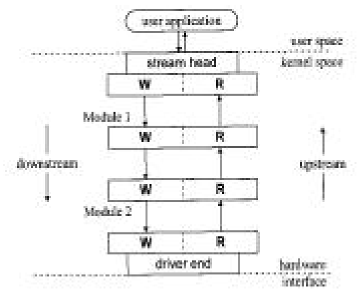
I/O request flowchart and hierarchy


request I/O
system call
can already satisfy request?
yes
no
user process
kernel I/O subsystem
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
device controller commands
interrupt handler
monitor device, interrupt when I/O completed
keyboard device controller
I/O completed, input data available, or output completed
return from system call
transfer data (if appropriate) to process, return completion or error code
determine which I/O completed, indicate state change to I/O subsystem
receive interrupt, store data in device driver buffer if input, signal to unblock device driver
interrupt
I/O completed, generate interrupt

Why STREAMS?

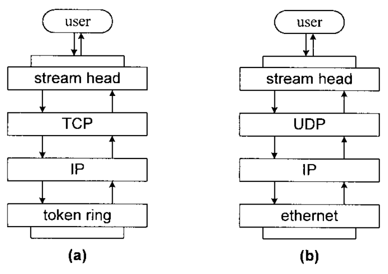
- Kernel interacts with drivers at a very high level leaving the device driver to do most of I/O processing. It provides flexibility of the design, yet only part of the work of the driver is hardware dependent; the other part - high-level I/O processing: queue management, buffering, caching, etc. Every vendor writes their own device drivers
  - code duplication
    - large kernel
    - greater risk of conflict
  - complex drivers
- This problem is especially apparent in network driver design: network protocols are complex and designed in (interchangeable layers); this suggests a modular approach to driver design

STREAMS



- STREAMS - a full-duplex (bidirectional) data transfer path between a driver and a user application
- Consists:
  - module - consists of a pair of queues:
    - read - pass data (in the form of messages) *upstream* - to application
    - write - pass messages *downstream* - to device
  - stream head - handles system calls, may block
  - driver end - handles interrupts, communicates to actual device
- except for stream head the modules communicate asynchronously: module code may be executed in the context of different process

Reusing modules



- TCP/IP protocol *stack* consists of a layers of protocols where an entity on one layer communicates with a *peer* entity on the same layer and provides services to the upper layer protocol and utilizes services of the lower layer without concern to its internal structure
- using streams modules can be assembled to fit the network configuration: note how IP module is reused