I/O Software

CS 537 – Introduction to Operating Systems

Programmed I/O

- Basic idea:
  - CPU does all communication directly with device
  - CPU waits for device to complete one operation before issuing another request
- Flow of events
  - user program issues a request for I/O
  - OS copies a single byte (word) to/from device
  - OS waits for device to become ready again
  - OS then copies the next byte (word) to/from device
  - repeat this process until all the data copied
  - return control to the user

Programmed I/O

- psuedo-code for reading from device

```
for(i=0; i<count; i++) {
   while(device_status_register != READY);
   device_data_register = buffer[i];
}
return_to_user();
```

Programmed I/O

- Advantages:
  - simple to implement
  - very little hardware support
- Disadvantages:
  - busy waiting (polling)
    - ties up CPU for long periods with no useful work (maybe)

Inerrupt Driven I/O

- Basic idea:
  - similar to programmed I/O but instead of busy waiting, block the process and have the I/O device interrupt when it is ready
- Flow of events:
  - user program issues a request for I/O
  - OS copies a single byte (word) to/from device
  - OS schedules another process
  - device interrupts the CPU
  - OS then copies the next byte (word) to/from device and reschedules again
  - repeat this process until all the data copied
  - return control to the user

Interrupt Driven I/O

- psuedo-code for reading from device
  - code executed on system call

```
block_user();
count = n;          i = 0;
while(device_status_register != READY);
device_data_register = buffer[i];
schedule();
```

  - code executed on interrupt from device

```
if(count == 0) { unblock_user(); }
else {
        device_data_register = buffer[i];
        count--;   i++;
}
return_from_interrupt();
```

Interrupt Driven I/O

- Advantages:
  - system can do useful work while device not ready
- Disadvantage
  - lots of interrupts
    - interrupts are expensive to do
    - have to run interrupt code fragment on everyone
    - what if the device is not slow?

DMA

- Basic idea:
  - exactly like programmed I/O but instead of the device communicating with CPU, it communicates with DMA controller
- Flow of events:
  - user program issues a request for I/O
  - OS blocks calling process
  - OS programs DMA controller
  - OS schedules another process
  - DMA controller then does programmed I/O
    - with busy waits and all
    - it may actually be able to handle more than one device at a time
  - DMA interrupts CPU when data transfer complete
  - OS unblocks user process

DMA

- psuedo-code for reading from device
  - code executed on system call

```
block_user();
set_up_DMAC();
scheduler();
```

  - code executed on interrupt from device

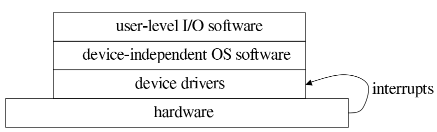
```
unblock_user();
return_from_interrupt();
```

DMA

- Advantages
  - only one interrupt to the CPU for a single I/O operation
    - CPU only bothered when I/O finished
- Disadvantages
  - Memory conflicts between CPU and DMAC
  - DMAC is much slower than CPU
    - what if device is very fast?
    - what if there is no other work for CPU?

I/O Software Layers

- Modern I/O software is broken into layers
  - a common interface from one layer to the next allows for high degree of flexibility
    - abstraction
  - you don’t need to know how each layer works – you just need to know how to interact with it



User Level Software

- library calls
  - users generally make library calls that then make the system calls
  - example:
    - int count = *write*(fd, buffer, n);
    - *write* function is run at the user level
    - simply takes parameters and makes a system call
  - another example:
    - printf(“My age: %d\n”, age);
    - takes a string, reformats it, and then calls the write system call

User Level Software

- Spooling
  - user program places data in a special directory
  - a *daemon* (background program) takes data from directory and outputs it to a device
    - the user doesn't have permission to directly access the device
    - daemon runs as a privileged user
  - prevents users from tying up resources for extended periods of time
    - printer example
  - OS never has to get involved in working with the I/O device

Device Independent OS Software

- Make devices look like files
  - this is the Unix and Windows approach
- You can open, read, write, close, etc. a device
  - some devices may be read only (keyboard)
  - others may be write only (monitor)
- Example – writing to a disk

```
fd = open("/dev/hda1", O_RDONLY);
lseek(fd, 1024, SEEK_SET);
write(fd, buffer, size);
close(fd);
```

Device Independent OS Software

- OS knows the file represents a device because the meta data says so
  - in Unix, there is no file – just an inode
- How does OS know what to do on a read?
  - meta data includes a *major* and a *minor* number
    - major: what category of device it is
    - minor: what specific device in the category
- Protection of devices is now simple
  - put the access rights for the device in the meta data
- How to deal with errors?
  - let the lower level, device specific software deal with it
  - return standard error codes to indicate a failure

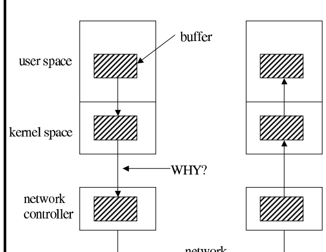
Device Independent OS Software

- Many devices require buffering
  - want to make this interface common for all devices as well
- Where to buffer the data?
  - usually, not in user space
    - page might get swapped out
  - usually place buffered data in the kernel space
    - locked in memory
  - double buffering
    - while copying data to/from user space, more data may arrive
    - keep a second buffer for new data while the other is being transferred
    - switch back and forth between the two buffers

Device Independent OS Software

- Advantages of buffering
  - increases efficiency
    - can transfer entire blocks instead of single words
  - allows for asynchronous operation
    - user transfers data to the kernel and moves on
- Disadvantages of buffering
  - lots of copying can reduce performance

Device Independent OS Software



Data gets copied 5 times!

Drivers

- At some level we must deal with the fact that I/O devices are different
- Drivers are the level of software that do this
  - usually provided by the device vendor
- A single driver handles a single device
  - or maybe a class of devices
    - imagine IDE disks of different sizes and speeds
  - use the major number to indicate which driver to run
  - the minor number is passed as a parameter to the driver
    - which specific device of a particular class

Drivers

- Basic driver functions
  - accept read/write commands from the device independent layer and translate into actual commands for the device
    - must write/read the various I/O ports of the device
  - translate input parameters and check for errors
  - handle device errors if possible
    - maybe retry a read request if checksum fails

Drivers

- In today's world, drivers are built into the operating system
- Better solution would be to put them in I/O space and provide system calls for I/O port interaction
  - kernel wouldn't crash because of a buggy driver
- Drivers are often implemented as separate processes
  - allows them to block and let the OS reschedule
  - makes interface between drivers and OS much simpler
- Devices are constantly added and removed
  - must allow drivers to be dynamically plugged into the system