Lecture 21: April 8

Lecturer: Prashant Shenoy *TA: Vimal Mathew & Tim Wood*

21.1 I/O Systems

The I/O system responsible for managing input and output devices attached to a computer. A computer system may have many types of devices ranging from ordinary disk drives or peripherals such as printers and display cards, to more unusual devices such as joysticks, robot actuators, or fuel injection systems in a car. A typical I/O system is composed of multiple components. The **system bus** is a shared communication channel that allows devices to talk directly to the CPU. A **device port** can be thought of as an outlet to which a device can be attached. The port typically consists of four registers, one to indicate the status of the device, one for sending control messages, and one each for data being sent to or from the device. An I/O **controller** is responsible for translating the commands that are transmitted over the system bus into actions such as reads or writes that a device will perform.

The PCI bus is an example of a standard I/O bus that negotiates access between the CPU and a large number of possible devices. In order to control these devices, the kernel I/O subsystem must interact with device drivers designed for each device. These drivers translate high level commands from a user process into messages sent to the device controller. The device controller in turn translates the commands into the low level operations that the device must perform.

The operating system kernel provides a large number of services to support I/O. These include negotiating access to devices between competing processes, buffering and caching to improve I/O performance, error handling in the case of device failure, as well as I/O scheduling.

21.2 I/O Communication

The CPU needs some way to communicate with devices in order to determine when data is ready to be read from the device, or to alert the device that it has commands to send to it. The simplest communication method is **polling**. In this case, the CPU constantly "polls" the device to try to determine when it has completed its most recent access. A typical polling scheme would go through the following steps:

1. CPU loads a register on the device controller with the command to be performed
2. The CPU changes the status of the device port to "command-ready", in turn the controller changes its status to "busy"
3. The controller reads the command register and performs the operation, and places the output of the command into a buffer if required
4. If the command was successfully run, the controller changes its status back to "idle"
5. The CPU continuously checks the controller status until it sees "idle", at that point, it reads the data from the buffer, or sends a new command

While this approach allows the CPU to immediately obtain data once it is ready from the device, it makes the CPU busy-wait. We have shown previously that this is wasteful, since it would be best if the CPU could perform other activities while the I/O was processed.

Rather than making the CPU busy-wait, communication using **interrupts** can be used to allow the CPU to continue running other processes while I/O is performed. Once the I/O completes, the I/O controller sends an interrupt to the CPU which alerts the OS that I/O is ready to be handled. To handle the interrupt, the OS determines which device caused the interrupt and either reads the data produced by the device or issues another command.

Although interrupt based I/O improves performance, it can still be slow since the CPU can only read in a small amount of data per interrupt (based on the size of the data in register in the control port). This can result in very poor performance for devices such as disks which must transfer very large amounts of data. The common solution to this problem is **Direct Memory Access (DMA)**. DMA requires a more sophisticated device controller that is able to read or write directly to memory. This allows the CPU to provide the DMA controller with an address telling it where to read or write data in memory. This allows the DMA controller to read an entire request into memory before triggering the interrupt that tells the CPU it has finished. This can dramatically reduce the number of interrupts the CPU must deal with, but it causes some extra contention on the memory bus since both the CPU and DMA controller may be reading or writing to memory at once (generally in different locations).

21.3 Programmer's View of I/O

Operating systems mask the low level details of I/O devices from programmers. They provide a high-level interface which simplifies the programmer's job by creating a standard interface used across many types of devices. Different devices may have different characteristics. The basic transfer unit may be a **block** for a disk drive, but it could be a **character** in a device such as a modem. Another key difference between devices is whether they support random access (e.g. a hard disk) or only sequential access (e.g. a keyboard). Devices also may support either synchronous or asynchronous calls; typically devices are technically asynchronous, but the I/O calls provided by the OS are synchronous in order to provide a simpler programming model.

21.4 Buffering & Caching

The nature of I/O requests is such that either the same or nearby blocks may be accessed multiple times in a short interval. To improve performance, I/O devices typically include a small on-board memory where they can temporarily store data before transferring it to or from the CPU. This allows a disk to buffer data to be read from or written to the device while the DMA controller transfers it to memory.

Using buffers both within the OS and on a device allows the system to improve I/O performance or consistency. This is particularly important when the speed of the device and the CPU are very different—the CPU may quickly write a disk block to a buffer, and then the device will more slowly spool it out to disk.

The use of caches also helps the OS improve performance by reducing the number of device operations that must actually be performed. For example, the OS can keep a memory cache holding recently used disk blocks. When a new disk read comes in, the OS checks whether the disk block is already stored in the cache, allowing it to be returned immediately. Storing blocks in a cache can significantly improve read performance, but it also complicates writes. In a **write-through** policy, when the OS makes a write it applies it both to the memory holding the block and to the disk itself immediately. This provides high reliability since all writes are known to have been made to disk. However, faster write performance can be obtained with a

write-back policy which only writes the update to the memory at first, and queues the write to disk to be performed sometime later. This can lead to very fast writes, but results in a weaker reliability model since the programmer cannot be sure that a disk write has truly gone to disk.