CHAPTER 13

I/O Systems

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. Although related topics appear in other chapters, here we bring together the pieces to paint a complete picture. In this chapter we describe I/O structure, devices, device drivers, caching, and terminal I/O.

13.1 Each channel would run the following algorithm:

```
        /** data definitions **/

// a list of interrupts sorted in earliest-time-first order
List interruptList

// the list that associates a request with an entry in
interruptList
List requestList

// an interrupt-based timer
Timer timer

while (true) {
    /** Get the next earliest time in the list **/
    timer.setTime = interruptList.next();

    /** An interrupt will occur at time timer.setTime **/

    /** now wait for the timer interrupt
    i.e. for the timer to expire **/

    notify( requestList.next() );
}
```

13.2 The advantage of supporting memory-mapped I/O to device-control registers is that it eliminates the need for special I/O instructions from the instruction set and therefore also does not require the enforcement

of protection rules that prevent user programs from executing these I/O instructions. The disadvantage is that the resulting flexibility needs to be handled with care; the memory translation units need to ensure that the memory addresses associated with the device control registers are not accessible by user programs in order to ensure protection.

13.3 The purpose of this strategy is to ensure that the most critical aspect of the interrupt handling code is performed first and the less critical portions of the code are delayed for the future. For instance, when a device finishes an I/O operation, the device-control operations corresponding to declaring the device as no longer being busy are more important in order to issue future operations. However, the task of copying the data provided by the device to the appropriate user or kernel memory regions can be delayed for a future point when the CPU is idle. In such a scenario, a lower-priority interrupt handler is used to perform the copy operation.

13.4 Polling can be more efficient than interrupt-driven I/O. This is the case when the I/O is frequent and of short duration. Even though a single serial port will perform I/O relatively infrequently and should thus use interrupts, a collection of serial ports such as those in a terminal concentrator can produce a lot of short I/O operations, and interrupting for each one could create a heavy load on the system. A well-timed polling loop could alleviate that load without wasting many resources through looping with no I/O needed.

13.5 When an interrupt occurs the currently executing process is interrupted and its state is stored in the appropriate process control block. The interrupt service routine is then dispatched in order to deal with the interrupt. On completion of handling of the interrupt, the state of the process is restored and the process is resumed. Therefore, the performance overheads include the cost of saving and restoring process state and the cost of flushing the instruction pipeline and restoring the instructions into the pipeline when the process is restarted.

13.6 Three pros of the UNIX method: Very efficient, low overhead and low amount of data movement. Fast implementation—no coordination needed with other kernel components. Simple, so less chance of data loss

Three cons: No data protection, and more possible side effects from changes so more difficult to debug. Difficult to implement new I/O methods: new data structures needed rather than just new objects. Complicated kernel I/O subsystem, full of data structures, access routines, and locking mechanisms

13.7 The user program typically specifies a buffer for data to be transmitted to or from a device. This buffer exists in user space and is specified by a virtual address. The kernel needs to issue the I/O operation and needs to copy data between the user buffer and its own kernel buffer before or after the I/O operation. In order to access the user buffer, the kernel needs to translate the virtual address provided by the user program to the corresponding physical address within the context of the user

program's virtual address space. This translation is typically performed in software and therefore incurs overhead. Also, if the user buffer is not currently present in physical memory, the corresponding page(s) need to obtained from the swap space. This operation might require careful handling and might delay the data copy operation.

13.8 A hybrid approach could switch between polling and interrupts depending on the length of the I/O operation wait. For example, we could poll and loop N times, and if the device is still busy at N+1, we could set an interrupt and sleep. This approach would avoid long busy-waiting cycles. This method would be best for very long or very short busy times. It would be inefficient it the I/O completes at N+T (where T is a small number of cycles) due to the overhead of polling plus setting up and catching interrupts.

Pure polling is best with very short wait times. Interrupts are best with known long wait times.

13.9 a. A mouse used with a graphical user interface

Buffering may be needed to record mouse movement during times when higher-priority operations are taking place. Spooling and caching are inappropriate. Interrupt-driven I/O is most appropriate.

b. A tape drive on a multitasking operating system (assume no device preallocation is available)

Buffering may be needed to manage throughput difference between the tape drive and the source or destination of the I/O. Caching can be used to hold copies of data that resides on the tape, for faster access. Spooling could be used to stage data to the device when multiple users desire to read from or write to it. Interrupt-driven I/O is likely to allow the best performance.

c. A disk drive containing user files

Buffering can be used to hold data while in transit from user space to the disk, and visa versa. Caching can be used to hold disk-resident data for improved performance. Spooling is not necessary because disks are shared-access devices. Interrupt-driven I/O is best for devices such as disks that transfer data at slow rates.

d. A graphics card with direct bus connection, accessible through memory-mapped I/O

Buffering may be needed to control multiple access and for performance (double-buffering can be used to hold the next screen image while displaying the current one). Caching and spooling are not necessary due to the fast and shared-access natures of the device. Polling and interrupts are useful only for input and for I/O completion detection, neither of which is needed for a memory-mapped device.

13.10 It is possible, using the following algorithm. Let's assume we simply use the busy-bit (or the command-ready bit; this answer is the same regardless). When the bit is off, the controller is idle. The host writes to data-out and sets the bit to signal that an operation is ready (the

equivalent of setting the command-ready bit). When the controller is finished, it clears the busy bit. The host then initiates the next operation.

This solution requires that both the host and the controller have read and write access to the same bit, which can complicate circuitry and increase the cost of the controller.

13.11 Reliable transfer of data requires modules to check whether space is available on the target module and to block the sending module if space is not available. This check requires extra communication between the modules, but the overhead enables the system to provide a stronger abstraction than one which does not guarantee reliable transfer. The stronger abstraction typically reduces the complexity of the code in the modules. In the STREAMS abstraction, however, there is unreliability introduced by the driver end, which is allowed to drop messages if the corresponding device cannot handle the incoming data. Consequently, even if there is reliable transfer of data between the modules, messages could be dropped at the device if the corresponding buffer fills up. This requires retransmission of data and special code for handling such retransmissions, thereby somewhat limiting the advantages that are associated with reliable transfer between the modules.

13.12 Direct virtual memory access allows a device to perform a transfer from two memory-mapped devices without the intervention of the CPU or the use of main memory as a staging ground; the device simply issues memory operations to the memory-mapped addresses of a target device and the ensuing virtual address translation guarantees that the data is transferred to the appropriate device. This functionality, however, comes at the cost of having to support virtual address translation on addresses accessed by a DMA controller and requires the addition of an address-translation unit to the DMA controller. The address translation results in both hardware and software costs and might also result in coherence problems between the data structures maintained by the CPU for address translation and corresponding structures used by the DMA controller. These coherence issues would also need to be dealt with and results in further increase in system complexity.

13.13 Consider a system which performs 50% I/O and 50% computes. Doubling the CPU performance on this system would increase total system performance by only 50%. Doubling both system aspects would increase performance by 100%. Generally, it is important to remove the current system bottleneck, and to increase overall system performance, rather than blindly increasing the performance of individual system components.

13.14 A number of issues need to be considered in order to determine the priority scheme to be used to determine the order in which the interrupts need to be serviced. First, interrupts raised by devices should be given higher priority than traps generated by the user program; a device interrupt can therefore interrupt code used for handling system calls. Second, interrupts that control devices might be given higher priority than interrupts that simply perform tasks such as copying data served up a device to user/kernel buffers, since such tasks can always be delayed.

Third, devices that have real-time constraints on when their data is handled should be given higher priority than other devices. Also, devices that do not have any form of buffering for its data would have to be assigned higher priority since the data could be available only for a short period of time.

13.15 Generally, blocking I/O is appropriate when the process will be waiting only for one specific event. Examples include a disk, tape, or keyboard read by an application program. Non-blocking I/O is useful when I/O may come from more than one source and the order of the I/O arrival is not predetermined. Examples include network daemons listening to more than one network socket, window managers that accept mouse movement as well as keyboard input, and I/O-management programs, such as a copy command that copies data between I/O devices. In the last case, the program could optimize its performance by buffering the input and output and using non-blocking I/O to keep both devices fully occupied.

Non-blocking I/O is more complicated for programmers, because of the asynchronous rendezvous that is needed when an I/O occurs. Also, busy waiting is less efficient than interrupt-driven I/O so the overall system performance would decrease.