103011227 周延儒 CS18

1. [20 points] (from Chapter 12) Explain why SSTF scheduling tends to favor middle cylinders over the innermost and outermost cylinders.

   This is because the center of the disk is the location having the **smallest average distance to all other tracks**. Thus, the disk head tends to move away from the edges of the disk.

   Also, there is another explanation. The current location of the head **partitions the cylinders into two groups**. If the head is not in the center of the disk and a new request arrives, the new request is more likely to be **in the group that includes the center of the disk**; As a result, the head is **more likely to move in that direction.**

2. [20 points] **12.8** Requests are not usually uniformly distributed. For example, we can expect a cylinder containing the file-system metadata to be accessed more frequently than a cylinder containing only files. Suppose you know that 50 percent of the requests are for a small, fixed number of cylinders.
   a. Would any of the scheduling algorithms discussed in this chapter be particularly good for this case? Explain your answer.

      Well, in my opinion, **SSTF would be the best scheduling algorithm** in this case. FCFS, however, could cause **unnecessary head movement** if references to the “**high- demand” cylinders** were **interspersed with references to cylinders far away.**

   b. Propose a disk-scheduling algorithm that gives even better performance by taking advantage of this “hot spot” on the disk.

      We could distribute the **hot data near the middle of the disk**. And we could **modify SSTF** to prevent starvation. In addition, Add the policy that if the **disk becomes idle for more than**, the OS generates **an anticipatory seek to the hot region,** because it is more likely that the next request will be there.

3. [40 points] **13.3** Consider the following I/O scenarios on a single-user PC:

   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 (with no device preallocation available)

   Buffering may be needed to manage **throughput difference between the tape drive and the source or destination of the I/O**. And **caching** can be adopted to hold copies of data residing on the tape, for **faster access**. **Spooling** could be used to stage data to the device when **multiple users want 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 to increase 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 could 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 only useful for input and for I/O completion detection. And neither of both is needed for a memory-mapped device.**

   For each of these scenarios, would you design the operating system to use **buffering, spooling, caching, or a combination**? Would you use **polled I/O or interrupt-driven** I/O? Give reasons for your choices.

4. [20 points] **13.7** Typically, at the completion of a device I/O, a single interrupt is raised and appropriately handled by the host processor. In certain settings, however, the code

that is to be executed at the completion of the I/O can be broken into two separate pieces. The first piece executes immediately after the I/O completes and schedules a second interrupt for the remaining piece of code to be executed at a later time. What is the purpose of using this strategy in the design of interrupt handlers?

The purpose of this strategy is to **ensure that the most critical section of the interrupt handling code is performed first** and **the less critical portions of the code are delayed** for the future.
For example, 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 this case, lower-priority interrupt handler could just perform the copy operation