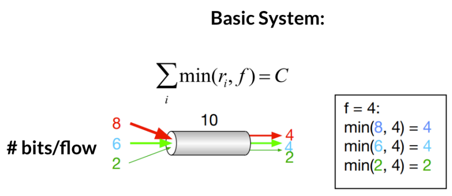
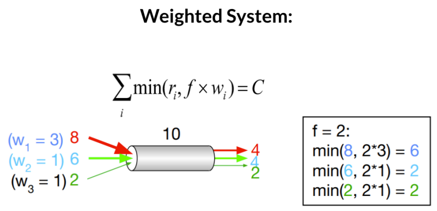
Discussion 7: EEVDF, I/O

Oct 30, 2024

Contents





1 Earliest Eligible Virtual Deadline First (EEVDF)

EEVDF is scheduling algorithm that divides the available CPU time fairly among the clients that are contending for it. For example, if there are five threads contending to run on 1 CPU, each thread should receive 20% of the available time.

The CPU is allocated to each thread for a fixed time quanta q, and each thread is assigned a weight that determines the share of the CPU they are entitled to receive.

Fluid Flow Systems

A **fluid flow system** is an idealized model where multiple flows of data can be served simultaneously. In the context of CPU scheduling, each flow represents a thread's CPU usage. Here, a resource is assumed to be granted to potentially multiple flows at once, and in arbitrarily small units of time. Note that this system is unrealistic for CPU scheduling, since typically we cannot schedule more than 1 thread at a time per core. We will later address how this system is applied to EEVDF, and how this limitation is overcome.

In a fluid flow system, we assume there are multiple flows transmitting data through a communication link. In this model, each flow transmits *packets* of data across the link, where each packet contains ≥ 1 *bit(s)* of data.

In a basic system (where each flow is weighted equally), ach flow can transmit a service rate of min(r_i, f) bits/time_unit where,

1. r_i = flow arrival rate (the rate at which bits are sent to flow i)
2. f = link fair rate (the rate of bits that can be sent simultaneously across the link, per flow)

In the example below of the **basic system**, the link has a capacity of 10, meaning it can send at most 10 bits/time_unit. Since there are 3 flows transmitting data simultaneously, each flow should be able to send $f = 10/3 = 3.33$ bits at a time. However, since the 3rd flow only needs to submit 2 bits at a time, the remaining 8 bits can be fairly redistributed to the other 2 flows, increasing the link fair rate f to 4.

Weighted System

Each flow i can be assigned a weight w_i. In this case, each flow will transmit data at a rate proportional to its weight.

The link fair rate here is determined by the following equation: $f = C/\sum_i w_i$ where C represents the capacity of the link. This represents the rate of data transmitted across the link if $w_i = 1$. Each flow now transmits $min(r_i, f * w_i)$ bits of data at a time across the link. In this example, $f = C/(w_1 + w_2 + w_3) = 10/5 = 2$ bits/time_unit.

$$f_i(t) = \frac{w_i}{\sum_{j \in \mathcal{A}(t)} w_j}.$$

$$S_i(t_0, t_1) = f_i(t_0) * (t_1 - t_0).$$

Generalized Processor Sharing (GPS)

GPS or **Generalized Processor Sharing** extends the fluid flow system to sharing the processor among multiple threads. We can consider multiple threads on a unicore processor as an analogue to ”flows” and ”links” in the ideal fluid flow system. In this case, only one thread can be serviced at a time and gains exclusive access to CPU cycles in discrete time quanta.

$f_i(t)$, the proportional share of the processor that thread i should receive at time t can be computed as the following equation, where $A(t)$ refers to set of active threads at time t.

Service time represents access duration to a resource (e.g., CPU). In an ideal system, each active thread should receive a service time proportional to its weighted share. The ideal service time $S_i(t_0, t_1)$ that **thread *i*** should receive from times t_0 to t_1 can be calculated by the following equation.

Weighted Fair Queueing (WFQ)

We further construct a weighted fair queueing model upon the GPS model, but schedule threads in order of *earliest finishing time* in a fluid flow model.

Here’s the issue: When new threads enter the system, we would have to recompute the **finishing time** for every thread, since the total weight of the system increases.

However, the key observation here is that even if a thread’s physical finishing time changes, **the *order* in which clients finish does not change**. This allows us to create an efficient scheduling algorithm.

$$V(t_1) = \int_0^{t_1} \frac{1}{\sum_{j \in A(t)} w_j} \cdot dt$$

Virtual Time

The virtual time of a system scales inversely proportional to the sum of weights of active threads. More specifically, the *rate* of virtual time is the *inverse* of the total weight of active threads.

The rate of virtual time (dVdt) corresponds to $1/\sum_{j\in A(t)} w_j$ where A(t) refers to the set of active threads at time t.

1 virtual time unit represents the physical time it takes to schedule all active threads based on their weights. Suppose the following example:

1. Thread 1 and Thread 2 are the only active threads in the system, with weights 2 and 3, respectively.
2. Within one **virtual time unit** or **round**, both threads must be scheduled, and should be allotted time proportional to their weights.
3. Suppose 1 virtual time unit corresponds to t physical time units, and that each unit of weight corresponds to 1 time unit of CPU execution.
4. Thus, Thread 1, whose weight $w_1 = 2$, should be granted 2 units of CPU execution time. Thread 2, whose weight $w_2 = 3$, should be granted 3 units of execution time.
5. In total, scheduling each thread exactly once should account for 5 time units of running on the CPU. This time corresponds to **1 virtual time unit**, which is also equivalent to the **sum of the weights of the active threads in the system**.

To calculate the virtual time from time 0 to t_1, $V(t_1)$, we compute the integral of dVdt from t=0 to t_1: Don't be scared by the integral. If the set of active threads changes from time 0 to t, we can easily calculate the integral by taking the virtual time in terms of fragments.

Consider the following system...

1. Thread 1 with weight w_1 enters the system at time 0.
2. Thread 2 with weight w_2 enters the system at time t_0
3. Both threads are the only active threads from time t_0 to t_1.

We can thus calculate the virtual time $V(t_1)$ as the following: $V(t_1) = 1/w_1 * (t_0 - 0) + 1/(w_1 + w_2) * (t_1 - t_0)$

Figure 1: EEVDF Recurrence Relation

$$\begin{aligned} \textbf{Eq. 1} \quad ve^{(1)} &= V(t_0^i), \\ \textbf{Eq. 2} \quad vd^{(k)} &= ve^{(k)} + \frac{r^{(k)}}{w_i}, \\ \textbf{Eq. 3} \quad ve^{(k+1)} &= vd^{(k)}. \end{aligned}$$

Modeling EEVDF as a Fluid Flow System

Recall that in a fluid flow system, each flow would send packets of data across a link. When we extend this analogy to threads running on a processor, each flow represents a thread, and each packet represents an individual thread's request. Each thread can submit multiple requests, but these should be scheduled in the order in which they are submitted.

To create an actual scheduling algorithm modeled after a fluid flow system, we first define the following specifications.

1. Each thread's request is considered **eligible** in a fluid flow system after the previous thread's request finishes executing (i.e. after the previous request's deadline).
2. **Eligible time** = time when a request can be scheduled.
3. **Deadline** = time by which a request must be served.
4. Use the eligible time and deadline for each thread in terms of **virtual time** to make scheduling decisions.

Scheduling Decision

1. At each time step, only consider threads that have eligible requests. (i.e. v(e) < v(t), where v(e) = virtual eligible time, t = current time)
2. Choose the thread with the earliest virtual deadline v(d)... hence the name of the algorithm, ***earliest eligible virtual deadline first***.
3. Given the initial request of a thread, we can determine the eligible time of the *subsequent* request of a thread based on the initial request's deadline. Apply the same logic for all future requests.
4. Each scheduling decision schedules a thread based on a specific request. For simplicity, we assume that each thread makes a series of periodic requests. However, this assumption is not generally true in practice: rather, the scheduler can submit requests on behalf of threads.

Formally, we can write the following recurrence relation for the eligible time and eligible deadline of each request for each thread.

1. t_0^i refers to the time of the first request made by thread i.
2. ve^1 refers to the virtual eligible time of the first request, and is thus simply the virtual time corresponding to when thread i made its first request.
3. vd^k refers to the virtual deadline of the kth request, which is initially set to the virtual eligible time of that request, plus the request length divided by the thread's weight.
4. ve^{k+1} denotes the virtual eligible time of the $k+1$th request, which is initially the deadline of the previous request.
5. r^k denotes the request length of the kth request.

If the kth request takes up less or more than r^k time units, ve^{k+1}, the eligible time of the next request for that thread, will be updated to account for the thread's lag. More specifically, ve^{k+1} is updated to $ve^{k+1} = ve^k + u^k/w_i$, where u^k represents the time the thread took to complete the kth request.

Lag

Imagine a time period of one second; during that time, in our five-process scenario, each thread should have gotten 200ms of CPU time.

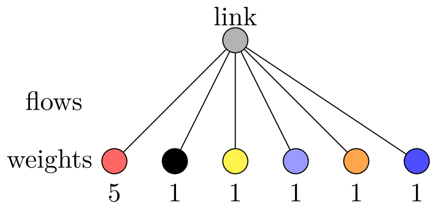
However, physical reality does not allow us to hand out time slices in real-valued intervals. We will hand out quanta in discrete time units, and as a consequence some threads will run beyond its ideal service time and others will run below.

For each thread, the **lag** calculates the difference between the time that thread should have gotten and how much it actually got. A thread with a **positive lag** has **not received its fair share and should be scheduled sooner**. On the other hand **negative lag** indicates that the thread has exceeded its portion. Intuitively, lag represents the CPU time that a thread is owed.

1. **Ideal service time**: service time client i should receive based on an ideal system (i.e. client's weighted share * time interval). $\implies S_i(t_0, t_1)$
2. **Real service time** : service time client i actually receives in a non-ideal system in a given time interval. $\implies s_i(t_0, t_1)$
3. **Lag**: ideal service time - real service time: $S_i(t_0, t_1) - s_i(t_0, t_1)$
4. If a thread i has **positive lag**, the eligible time for its subsequent request is pushed back, such that the thread is eligible earlier to execute its next request.
5. If a thread i has **negative lag**, its eligible time is pushed forward, meaning it becomes eligible *later* in time, to account for using up too much CPU time.

Service Time Calculations

Figure 2: Fluid Flow System Diagram

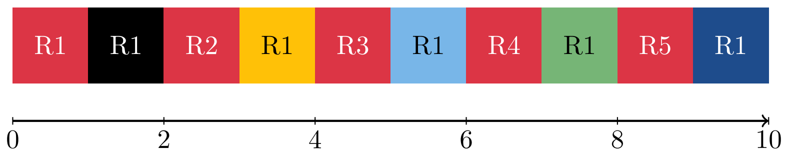


1. What is the ideal service time from t=0 to 1s for the red flow? What does this value represent?

   $S_i(0, 1) = 0.5 * (1\text{-}0) = 0.5$s This means that the red flow should receive 1s of service time from t=0-2s. From t=0-1s, the red flow is able to transmit 0.5 bits (i.e. half of a packet). This is equivalent to having full access to the processor for 0.5s.

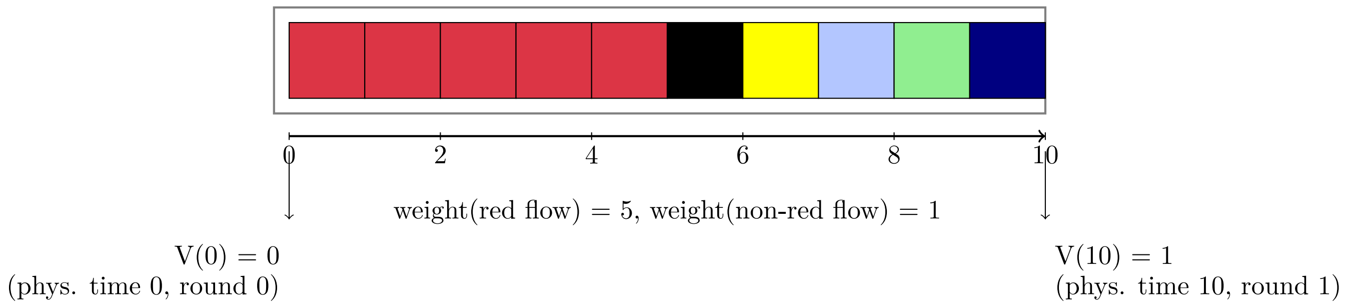
2. What is the ideal service time from t=0 to 1s for any non-red flow?

   $S_i(0, 1) = 0.1 * (1\text{-}0) = 0.1$s Same explanation as above.



Virtual Time Calculations

Weighted Fair Queueing Example



1. In this example of weighted fair queueing, what is the virtual time at physical time t=5 in this example? Assume the set of active clients include all flows and stays constant from t=0 to 10.

   V(t) = 1/(sum of weights of active clients) * t V(t) = 1/10 * t V(t) = 1/10 * 5 = 0.5

2. Now, assume that the yellow client enters the system at t=6, while all others are already present. What is the virtual time at physical time t=5?

   V(t) = 1/(sum of weights of active clients) * t V(t) = 1/9* t V(t) = 1/9 * 5 = 0.55

3. From the scenario above, what is the virtual time at time t=7?

   V(5) = 0.55, and virtual time from t=5 to 7 is 1/10 * 2s = 0.2 V(7) = V(5) + (V(7) - V(5)) = 0.2 + 0.55 Note that you cannot just take 1/10 * 7s because the yellow client only enters the system at t=6.

EEVDF Scheduling

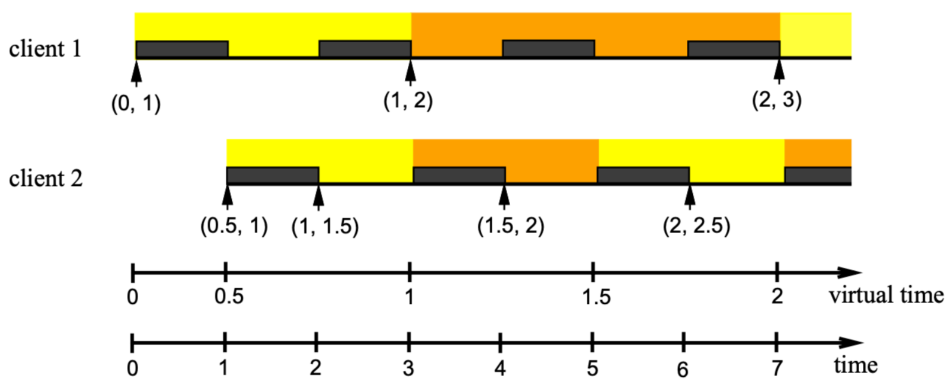
1. From t=0 to 2s, why must red packet 1 be scheduled before any non-red packet? Why does this also apply for t=4 to 6s?

   From t=0 to 2s, red packet 1 and all non-red packets are eligible, but red packet 1 has the earliest deadline. Thus, red packet 1 must be scheduled first. The same applies for t=4 to 6s because red packet 4 has a deadline of t=8s, whereas all non-red packets have a deadline of t=10s.

2. Now, assume we scheduled the blue packet at t=8 and red packet 5 at t=9. What is the lag from t=8-9s for the red flow?

   Ideal service time = 0.5*1s = 0.5s, Received service time = 0s → lag = 0.5s - 0s = 0.5s → positive lag (red flow “owed” CPU time)

Figure 3: EEVDF Simple Example



3. Fill out the table based on the diagram above.

t	V(t)	Active Threads	$\frac{dV(t)}{dt}$	Thread 1 (C1)	Thread 2 (C2)
0					
1					
2					
3					
4					
5					
6					
7					

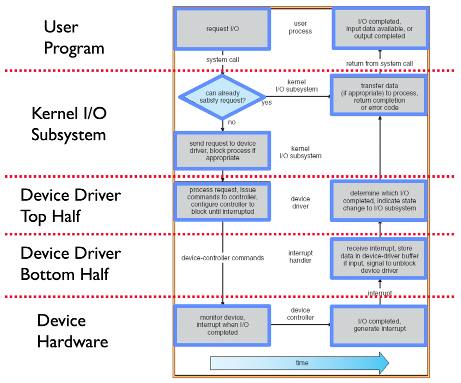
Add your calculations below.

2 I/O

I/O devices can be categorized based on how they access data, also known as **access patterns**. **Character devices** (e.g. keyboard, printer) access data as a character stream, meaning data is not addressable. On the other hand, **block devices** (e.g. disk) access data in fixed-sized blocks. Data is addressable (i.e. able to `seek`). **Network devices** (e.g. ethernet, wireless, Bluetooth) have a separate interface for networking purposes (e.g. socket interfaces like `select`).

I/O devices can also be categorized based on the **access timing**. **Synchronous** or **blocking** interfaces wait until an I/O request is fulfilled. On the other hand, **non-blocking** interfaces return quickly from a request without waiting. **Asynchronous** interfaces allow other processing to continue while waiting for the request to complete. For instance, when requesting data, the request will return immediately with a pointer to a buffer. This buffer will be filled asynchronously. When the request is completed, the process will be notified with a signal It's important to note that while synchronous and blocking are synonymous, non-blocking and asynchronous are two distinct types of non-synchronous I/O.

To provide an easy to use abstraction for I/O, **device drivers** connect the high-level abstractions implemented by the OS and hardware specific details for I/O devices. They support a standard interface allowing the kernel to use hardware without knowing the specific implementation. Device drivers are split into two halves. The **top half** is used by the kernel to start I/O operations, while the **bottom half** services interrupts produced by the device. It's important to note that in Linux, the terms are flipped.


User Program
Kernel I/O Subsystem
Device Driver Top Half
Device Driver Bottom Half
Device Hardware
request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
yes
no
kernel I/O subsystem
transfer data (if appropriate) to process, return completion or error code
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
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

Device Access

Processors talk to I/O devices through **device controllers**, which contain a set of registers that can be read from and written to. When using **programmed I/O (PIO)**, the processor is involved in every byte transfer. This can be done using **port-mapped I/O (PMIO)** which uses special memory instructions (e.g. `in`/`out` in x86). The memory address space is distinct from the physical memory address space. This is useful on older/smaller devices with limited physical address space, but it complicates CPU logic with special instructions. Alternatively, **memory-mapped I/O (MMIO)** maps control registers and displays memory into the physical address space. It uses standard memory instructions (`load`, `store`). As a result, it will use up a portion of the physical memory address space. However, this simplifies CPU logic by putting the responsibility on the device controller.

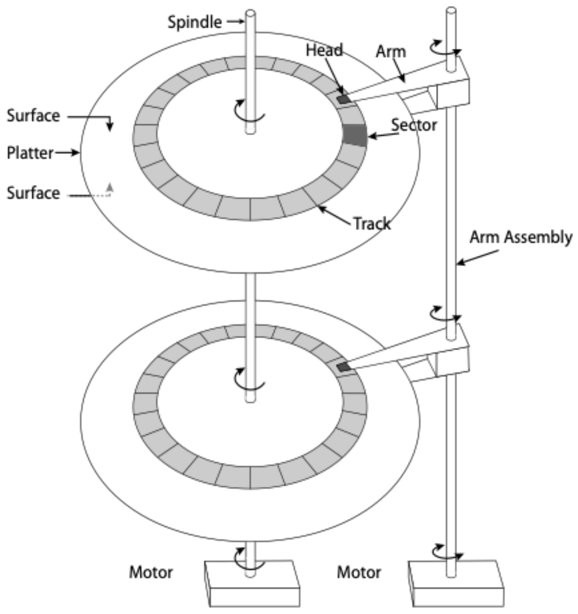
On the other hand, **direct memory access (DMA)** allows the I/O device to directly write to main memory without CPU intervention. It uses memory addresses within the physical memory region contrary to MMIO which just shares the same physical address space.

When an I/O operation is completed, the OS needs to be notified regarding its success. **Polling** puts this responsibility on the OS, requiring the OS to repeatedly check a memory-mapped status register. This provides a low-cost method since it would effectively be a memory access. However, it's not a very efficient use of CPU cycles, especially for infrequent or unpredictable I/O. On the other hand, the I/O device could generate an **interrupt**. This is much more expensive due to context switching overhead, but it's easier to handle infrequent or unpredictable requests.

Storage Devices

Magnetic Disks

Magnetic disks use magnetization to read and write data. Each magnetic disk is comprised of multiple **platters**, single thin round plates that store information. Each platter spins on a **spindle**, where each platter has two **surfaces**. Data is read and written with the **head**. Each head attaches to an arm, and each arm attaches to the arm assembly. Data is stored in fixed size units called **sectors**, which are the minimum units of transfer. A circle of sectors makes up a track. Due to the geometry of a disk, the track length varies from the inside to the outside of the disk.



As seen in the picture above, magnetic disks have lots of moving parts, which can lead to many physical issues. Compared to flash memory in the next section, the read/write times are a lot slower. However, they are much cheaper compared to flash memory, offering a better data size to price ratio.

Flash Memory

Flash memory uses electrical circuits (e.g. NOR, NAND) for persistent storage. While NOR flash allows individual words to be read/written, we will focus on NAND flash which reads/writes in fixed size units called pages (typically 2 KB to 4 KB). Flash memory involves much fewer moving parts compared to magnetic disks, leading to more durability. Its read and write speeds are also much faster. However, writing to a cell requires erasing it first (i.e. can't override the value). Erasing can only be done in large units called **erasure blocks** (typically 128 KB to 512 KB). Furthermore, each page has a finite lifetime, meaning it can only be erased and rewritten a fixed number of times (e.g. 10K).

To address the concern of erasure blocks, flash memory typically uses a **flash translation layer (FTL)** maps logical flash pages to different physical pages on the device. This allows the device to relocate data without the OS knowing or having to worry about it. As a result, writing over an existing page would involve writing to a new location, updating the mapping in the FTL, and then erasing the old page in the background. To ensure all physical pages are used roughly equally, the FTL uses **wear leveling**.

Performance

It's important to minimize how long it takes to access data from a storage device. The time required to retrieve data typically involves **queuing time**, **controller time**, and **access time**. Queuing time refers to how long a data request spends in the OS queue before actually getting fulfilled. Controller time refers to how long the device controller spends processing the request. Access time refers to how long it takes to access data from the device. When discussing performance between different devices, the access time is what will generally be focused on.

For magnetic disks, the access time involves the **seek time**, **rotation time**, and **transfer time**. Seeking moves the arm over to the desired track. Then, we need to wait for the target sector to rotate under the head. Finally, the disk must transfer data to/from the buffer for read/write. When maximizing disk performance, the key is to minimize seek and rotation times since transfer times are fixed for a given disk.

To improve disk performance, disks employ several intelligence techniques. **Track skewing** staggers logical sectors of the same number on each track by the time it takes to move across one track. **Sector sparing** remaps bad sectors to spare sectors on the same surface. Disk manufacturers typically include spare sectors distributed across each surface for this. To preserve sequential behavior, disks may use slip sparing which remaps all sectors when there is a bad sector. Disks often include a few MB of **buffer memory** which is used by the disk controller to buffer data for reads and writes. As a result, **track buffering** can improve performance by storing sectors that have been read by the disk head but not requested by the OS, taking advantage of physical spatial locality.

2.1 Concept Check

1. If a particular I/O device implements a blocking interface, do you need multiple threads to have concurrent operations which use that device?

   Yes. Only with non-blocking or asynchronous IO can you have concurrency without multiple threads.

2. For I/O devices which receive new data very frequently, is it more efficient to interrupt the CPU than to have the CPU poll the device?

   No. It is more efficient to poll, since the CPU will get overwhelmed with interrupts.

3. When using SSDs, which between reading or writing data is complex and slow?

   SSDs have complex and slower writes because their memory can't easily be mutated.

4. Why might you choose to use DMA instead of MMIO? Give a specific example where one is more appropriate than the other.

   DMA is more appropriate when you need to transfer large amounts of data to/from main memory without occupying the CPU, especially when the operation could potentially take a long time to finish (accessing a disk sector, for example). The DMA controller will send an interrupt to the CPU when the DMA operation completes, so the CPU does not need to waste cycles polling the device. While memory mapped I/O can be used to transfer device data into main memory, it must involve the CPU. MMIO is useful for accessing devices directly from the CPU (writing to the frame buffer or programming the interrupt vector, for example).

5. Usually, the OS deals with bad or corrupted sectors. However, some disk controllers magically hide failing sectors and re-map to back-up locations on disk when a sector fails.

   (a) If you had to choose where to lay out these "back-up" sectors on disk - where would you put them? Why?

   Should spread them out evenly, so when you replace an arbitrary sector you find one that is close by.

   (b) How do you think that the disk controller can check whether a sector has gone bad?

   Using a checksum, this can be efficiently checked in hardware during disk access.

   (c) Can you think of any drawbacks of hiding errors like this from the OS?

   Excessive sector failures are warning signs that a disk is beginning to fail.

6. When writing data to disk, how can the buffer memory be used to increase the perceived write speed from the OS viewpoint?

   Data written to disk can actually be stored in the buffer memory. Once this buffer memory is written to, the disk can acknowledge to the OS that the write has completed before it is actually stored on persistent storage. The buffer would get flushed to the platter at some later time. This technique is known as write acceleration.

2.2 Hard Drive Performance

Assume we have a hard drive with the following specifications.

- An average seek time of 8 ms
- A rotational speed of 7200 revolutions per minute (RPM)
- A controller that can transfer data at a maximum rate of 50 MiB/s

We will ignore the effects of queuing delay for this problem.

1. What is the expected throughput of the hard drive when reading 4 KiB sectors from a random location on disk?

   The time to read the sector can be broken down into three parts: seek time, rotation time, and transfer time.

   We are already given the expected seek time: 8 ms.

   On average, the hard disk must complete 1/2 revolution before the sector we are interested in reading moves under the read/write head. Given that the disk makes 7200 revolutions per minute, the expected rotation time is

$$\frac{1}{2} \times \frac{1}{7200 \text{ RPM}} \approx 4.17 \text{ ms}$$

   If the controller can transfer 50 MiB per second, it will take

$$4 \text{ KiB} \times \frac{1}{50 \text{ MiB}} \approx 0.078125 \text{ ms}$$

   to transfer 4 KiB of data.

   In total, it takes

$$8 \text{ ms} + 4.17 \text{ ms} + 0.078125 \text{ ms} \approx 12.248 \text{ ms}$$

   to read the 4 KiB sector, yielding a throughput of

$$\frac{4 \text{ KiB}}{12.248 \text{ ms}} \approx 326.6 \text{ KiB/s}$$

2. What is the expected throughput of the hard drive when reading 4 KiB sectors from the same track on disk (i.e. the read/write head is already positioned over the correct track when the operation starts)?

Now, we can ignore seek time and only need to account for rotation time and transfer time. Therefore, it takes a total of

$$4.17 \text{ ms} + 0.078125 \text{ ms} \approx 4.24 \text{ ms}$$

to read the 4 KiB sector, yielding a throughput of

$$\frac{4 \text{ KiB}}{4.24 \text{ ms}} \approx 943.4 \text{ KiB/s}$$

.

3. What is the expected throughput of the hard drive when reading the very next 4 KiB sector (i.e. the read/write head is immediately over the proper track and sector at the start of the operation)?

Now, we can ignore both seek and rotation times. The throughput of the hard disk in this case is limited only by the controller, meaning we can take full advantage of its 50 MiB/s transfer rate. Note that this is roughly a 156× improvement over the random read scenario!