1 VMs

Let's say you have a single layer of virtualization, i.e. a VMM on top of hardware, which is running a single VM and OS:

$$\text{Hardware} -> \text{VMM} -> \text{VM} -> \text{OS} -> \text{Applications}$$

In this case, if a sensitive instruction is encountered in the VM, since the VM is always running in (real) user mode, this will trap into the VMM. The VMM will check to see whether the VM was operating in (virtual) privileged or (virtual) user mode. If it's the former, the VMM will verify and emulate execution of the instruction. If it's the latter, the VMM will cause a trap to occur in the guest OS running on the VM, effectively switching the virtual mode of the VM from user mode to privileged mode. If the guest OS then tries to execute a privileged instruction, this will again trap into the VMM, but this time the VMM will verify and emulate the instruction since we're now in (virtual) privileged mode.

Now let's think about this in terms of nested virtualization, with something like this:

$$\text{Hardware} -> \text{VMM}_0 -> \text{VM}_0 -> \text{VMM}_1 -> \text{VM}_1 -> \text{VMM}_2 -> \text{VM}_2 -> ... \text{VMM}_i -> \text{VM}_i$$

In this case, if a sensitive instruction is encountered in VM$_i$, since VM$_i$ is running in (real) user mode, this will cause a trap to VMM$_i$. As the problem says, if we're running the virtual machine in (virtual) privileged mode, VMM$_i$ will verify this and try to emulate the instruction. VMM$_i$, however, is running on VM$_{(i-1)}$, which means VMM$_i$, trying to emulate this privileged instruction, is still running in (real) user mode.

This will cause a trap into VMM$_{(i-1)}$, which will verify that VMM$_i$ was running in (virtual) privileged mode (which it was as it's a VMM), and therefore try to emulate the instruction. Again, however, since VMM$_{(i-1)}$ is running on VM$_{(i-2)}$, this is still (real) user mode, so we'll have to imitate the above steps.

This continues until you get up to VMM$_0$, which is the OS on top of hardware. Once you trap into VMM$_0$, this will be in (real) privileged mode and actually be able to emulate the instruction on the real machine.

2 Stride Scheduling

2.1 1

In terms of which threads run when, there would be a difference. For example, with normal stride scheduling the thread with 5 tickets would run first, whereas in adapted stride scheduling the thread with four tickets in the process with 10 tickets would run first.

After 16 clock ticks, however, adapted and normal stride scheduling will produce the same results; all threads (and processes) will have had their meters increased by 1.

This works, since even though there's an additional layer of scheduling based on the tickets held by the processes, the structure of the problem is the exact same. Concretely, you can think of the processes in adapted stride scheduling as being threads with 10 and 6 tickets, respectively. This means that after 16 clock ticks, each processes' meter will have increased by 1, meaning each process got access to the processor for 10 and 6 ticks, respectively.

Each process has a number of threads, and if the process has fully distributed its tickets among its threads, this produces a problem analogous to the previous one – a smaller version of stride scheduling. Since we already know that if a group of threads with 'n' tickets distributed gets 'n' clock ticks it'll increase the meter of all the threads by one, each thread in the above 2 processes will have it's meter increased by 1.

2.2 2

As mentioned above, in adapted stride scheduling the thread with 4 tickets in the process with 10 tickets would run first.

2.3 3

Building off part 1, each process will run based on the number of tickets it has. So from the example, process 1 would run 10 times and process 2 would run 6 times after 16 clock ticks. The number of times the threads run, however, will equal:

$$\text{(number of times process runs)} * \frac{\text{number of tickets held by a thread}}{\text{total tickets given out in a process}}$$

Generalizing even more:

$$\frac{\text{number of tickets a process holds}}{\text{total tickets}} * \text{clock ticks} * \frac{\text{number of tickets held by a thread}}{\text{total tickets given out in a process}}$$

So if the process with 6 tickets only hands out 1 ticket to each of its threads, then each thread will get the processor (6 * 1/2) = 3 times in 16 clock ticks.

For the process with 10 tickets, each thread with 2 tickets will get the processor 2 times and the thread with 4 tickets will get it 4 times.

3 Rhinopias

3.1 1

Looking at the original Rhinopias I, if is spins at 6 milliseconds / revolution, with, on average, 750 sectors each with 512 bytes, then using this average number of sectors (instead of 1000 sectors / track for the outer regions):

$$\text{Rhinopias I}$$

$$\frac{\text{1 revolution (track)}}{\text{6 ms}} * \frac{\text{1000 ms}}{\text{1 s}} * \frac{\text{1 (750 * 512) bytes}}{\text{track}} = \text{64 million bytes}$$

$$\text{Rhinopias II}$$

$$\frac{\text{1 revolution (track)}}{\text{4 ms}} * \frac{\text{1000 ms}}{\text{1 s}} * \frac{\text{1 (750 * 512) bytes}}{\text{track}} = \text{96 million bytes}$$

This means that by increasing the rotational speed by 33% you increase the maximum one-track transfer speed by roughly 50%.

3.2 2

This answer to this question relies on what we use as the average seek time. In the table, it says it's 4ms for both Rhinopias I and II. In the textbook, however, it states that this estimate might be pessimistic since the data region doesn't occupy the entire disk, and therefore uses 2ms. I'll provide calculations for both.
First, assuming 4ms average seek time:

Rhinopias I:

.007 time per sector = .004s average seek time + .003 average rot. latency.

$$\frac{512 \text{ bytes}}{\text{sector}} * \frac{\text{sector}}{.007\text{s}} = 73142.9 \text{ bytes per second}$$

Rhinopias II:

.006 time per sector = .004s average seek time + .002 average rot. latency.

$$\frac{512 \text{ bytes}}{\text{sector}} * \frac{\text{sector}}{.006\text{s}} = 85333.3 \text{ bytes per second}$$

Which means the Rhinopias II is roughly 17% faster than the I.

Assuming the average seek time is closer to 2ms instead of 4ms:

Rhinopias I:

.005 time per sector = .002s average seek time + .003 average rot. latency.

$$\frac{512 \text{ bytes}}{\text{sector}} * \frac{\text{sector}}{.005\text{s}} = 102400 \text{ bytes per second}$$

Rhinopias II:

.004 time per sector = .002s average seek time + .002 average rot. latency.

$$\frac{512 \text{ bytes}}{\text{sector}} * \frac{\text{sector}}{.004\text{s}} = 128000 \text{ bytes per second}$$

Which means the Rhinopias II is roughly 25% faster than the I.

3.3 3

Since this is making use of a log-structure system, where you're writing at the end of where ever you currently are on disk, there shouldn't be any seek or rotational delays. This means it should write at the same speed as the maximum transfer speed, meaning the Rhinopias II should write 50% faster than the Rhinopias I.

4 Soft Updates

So, let's say I'm creating a directory with a file in it: I need to 1) initialize a file inode, 2) modify the data block containing the new directory entry, and 3) modify the directory inode. Let's say that the (1) file inode and the (3) directory inode are on the same disk block - this makes writing to disk in the above order impossible as there's a circular dependency.

To recap, we have 2 disk blocks: one containing the file inode and directory inode and one referencing the modification to the data block containing the new directory entry. In Async FFS there would be 2 writes to disk for these two blocks. Using soft updates, to mitigate the circular dependency the first block is written to disk but not before the update modifying the directory inode is undone. The first block can now be written to disk without causing the ordering problem described above. The second block can now be written out to disk, and the first block, with the update of modifying the directory inode redone, written again, thus ensuring the correct order of writes.

Using soft updates in this scenario, then, would result in 3 total disk writes to move everything from the cache to disk as opposed to only 2 in Async FFS.