**Problem Set 2 CS543 Mangesh Raut(mbr63)**

1. When adding a block back to the free tree for a pool, we check to see if adjacent blocks are also free and coalesce them if they are. Is it possible that we might accidentally coalesce free blocks belonging to different pools in this way? Why or why not?

Ans: If the mechanism used to keep track of which blocks belong to which pool is not properly implemented, it is possible that free blocks belonging to various pools may unintentionally coalesce. Blocks from separate pools may coalesce, for instance, if the memory management system does not keep track of which pool a block belongs to or does not make sure that they cannot be combined.

The memory management system should provide a mechanism to identify the pool that each block belongs to and make sure that blocks from various pools can't be combined to avoid this from happening. This might be accomplished, for instance, by adding a pool identity in each block's header, which is verified before merging blocks.

In conclusion, by adding appropriate tracking and validation procedures in the memory management system, the probability of unintentionally coalescing free blocks from distinct pools can be decreased. To prevent the inadvertent coalescence of free blocks from various pools, the memory management system should incorporate adequate tracking and validation methods. The system can confirm the pool that a block belongs to before merging blocks by including a pool identification in the header of each block. This will lessen the chance of unintentional free blocks from several pools coalescing.

2. Consider an RS-232 controller with no interrupt capability. If we are to support a 19,200-bps data rate with seven data bits, even parity, and one stop bit, how often should we poll the controller? If each polling operation takes 200 μS, what fraction of the system’s time is spent polling?

Ans: It takes at least 10/19200 = 521 s to poll the controller to support a 19,200-bps data rate with seven data bits, even parity, and one stop bit. This would guarantee that we could receive each incoming byte within the necessary window of time. This is merely a minimum polling rate, though, and a greater rate would be necessary to consider processing overhead and any additional system-level variables.

You have given the data rate as 19,200 bps (bits per second) and the format as seven data bits, even parity, and one stop bit. This means that each byte of data is transmitted in 10 bits (7 + 1 + 1 + 1). Therefore, the transmission rate is 1,920 bytes per second (19,200 / 10). This means that one byte of data takes 1/1920 seconds to be transmitted.

If each polling operation takes 200 μS (microseconds), then you need to poll the controller at least every 521 μS (1/1920 \* 10^6) to ensure that you do not miss any incoming byte. This is because if you poll too late, you might miss some bits of the byte or receive an incomplete byte.

However, this is only a minimum polling frequency, and it does not account for any processing overhead or system-level variables that might affect the timing. Therefore, you might want to poll more frequently than this to avoid any errors or delays.

19,200 bps corresponds to 19,200 bits per second of data transmission. Each byte is communicated in 10 bits with 7 data bits, even parity, and 1 stop bit, resulting in a transmission rate of 1,920 bytes per second (19,200 bits per second / 10 bits/byte).

Therefore, 1 byte needs to be transmitted in 1,920 seconds.

The fraction of time spent polling is calculated by dividing the time spent on each polling operation by the time interval between two consecutive polls. In this case,

Fraction = Time spent on each polling / Time interval between polls = 200 μS / Polling frequency If we use the minimum polling frequency of 521 μS,

Fraction = 200 μS / 521 μS = 0.384 This means that if you poll every 521 μS, you will spend about 38.4% of your time polling.

3. What is the average access time for a disk drive that spins at 3600 RPM with an average seek time of 50 mS? What if the drive is spun at 10,000 RPM? Does the speed of rotation make a significant difference? What if the drive technology advances and makes the average seek time 9 mS?

Ans: The average time to find the requested sector plus the time to rotate the disk such that the head is over the desired sector are added to determine the average access time for a disk drive that rotates at 3600 RPM and has an average seek time of 50 mS. Seek Time + (Rotational Latency / 2) is the equation used to calculate average access time. Rotational Latency, which is computed as (1 / (number of sectors per rotation) \* 60 seconds per minute), is the amount of time it takes for the required sector to rotate beneath the head.

The average access time for a drive with a 3600 RPM can be computed as follows:

Average Access Time is equal to 50 mS plus (60 / (3600 \* 2)) to produce 50.0167 mS.

Following are the formulas to determine the average access time when the drive is spinning at 10,000 RPM:

Average Access Time is equal to 50 mS plus (60 / (10,000 \* 2)) to produce 50.006 mS.

Therefore, the average access time is significantly affected by the rotational speed. The average access time will be shorter on a faster spinning disk because of the lower rotational latency.

The average access time can be estimated as follows if drive technology improves and the average search time is lowered to 9 mS:

Average Access Time (3600 RPM) is equal to 9 mS + (60 / (3600 \* 2)) = 9 mS + 0.0167 mS = 9.0167 mS.

Average Access Time (10,000 RPM) = 0.006 mS + (60 / (10,000 \* 2)) = 9.006 mS

In this scenario, a faster average seeks time results in a faster average access time, regardless of the speed of rotation.

4. If an Ethernet controller does not use DMA but generates an interrupt for every byte, and if each interrupt takes 10 μS to process, then can the system effectively use the 10 Mbit/S data rate? Why or why not?

Ans: No, the 10 Mbit/s data rate cannot be used by the system efficiently. The minimum frame size required by the Ethernet standard is 64 bytes, therefore if an interrupt is created for each byte, the processing cost for just one frame would need 640 S, which is substantially longer than the allowed 9.6 S inter-frame interval. The number of active devices on the system would also raise the processing overhead for interruptions, making it challenging to keep up with the incoming data stream. Devoid of DMA, the system would therefore be unable to manage the maximum data rate of 10 Mbit/s.

If an Ethernet controller does not use DMA but generates an interrupt for every byte, it will create a lot of overhead for the CPU and slow down the data transfer. The CPU will have to process each interrupt and copy each byte from or to memory. This will take more time than the actual data transmission.

For example, if we assume that each interrupt takes 10 μs to process, then transferring one byte will take 10 μs + 0.8 μs (the time it takes to transmit one bit at 10 Mbit/s) = 10.8 μs. This means that transferring one packet of 64 bytes (the minimum frame size for Ethernet) will take 64 \* 10.8 μs = 691.2 μs.

However, according to Ethernet standard, there should be a minimum inter-frame gap of 9.6 μs between two packets. This means that if we want to use the maximum data rate of 10 Mbit/s, we should be able to transmit one packet in less than 9.6 μs.

Clearly, this is impossible if we generate an interrupt for every byte and process it for 10 μs. Therefore, an Ethernet controller that does not use DMA but generates an interrupt for every byte cannot effectively use the 10 Mbit/s data rate.

A 10 Mbit/s Ethernet connection has a maximum data transfer rate of 10 million bits per second. This equates to 10 million / 8 = 1.25 million bytes per second. If each byte generates an interrupt and takes 10 μs to process, the system will spend 10 μs \* 1.25 million = 12.5 seconds processing interrupts, leaving very little time for actual data transfer. This effectively limits the data transfer rate to a much lower value and makes it impossible to take full advantage of the 10 Mbit/s Ethernet connection.

5. Using old-style cylinder-head-sector addressing, consider a disk with 1024 cylinders, 4 two-sided platters, and 34 sectors per track. What is the total number of sectors on this disk? What is the cylinder, head, and sector number corresponding to block number 100,000?

Ans: Cylinder-head-sector (CHS) addressing is a method of accessing data on a disk drive by specifying its physical location using three parameters: cylinder number, head number, and sector number.

A cylinder is a set of tracks that are at the same distance from the center of the disk. A track is a circular path on a disk where data is stored. A head is a device that reads or writes data on a track. A sector is a segment of a track that holds a fixed amount of data.

The total number of sectors on a disk is equal to the product of the number of cylinders, heads, and sectors per track. The block number (or logical block address) is an abstract way of referring to a sector without specifying its physical location.

The total number of sectors on this disk is calculated as follows:

1024 cylinders \* 4 heads \* 34 sectors per track = 13824 sectors.

To determine the cylinder, head, and sector number corresponding to block number 100,000, we use the following formula:

c = lba / (s \* h)

h = (lba / s) % h

s = lba % s + 1

Where lba is the block number, c is the cylinder number, h is the head number, and s is the sector number. Plugging in the values:

c = 100,000 / (34 \* 4) = 59.76 = 60

h = (100,000 / 34) % 4 = 2

s = 100,000 % 34 + 1 = 17

So, the cylinder number is 60, the head number is 2, and the sector number is 17 for block number 100,000 using the cylinder-head-sector addressing.

6. If disk blocks are 1024 bytes in size, what percentage of the disk is overhead imposed by a free bitmap?

Ans: It depends on the size of the free bitmap in relation to the entire size of the disk what portion of the disk is overhead caused by the free bitmap.

A free bitmap is a method of tracking free space on a disk by using a bit array, where each bit represents a block on the disk. A bit value of 0 means that the block is free, while a bit value of 1 means that the block is allocated.

The size of the free bitmap depends on the number of blocks on the disk and the size of each bit. The number of blocks on the disk can be obtained by dividing the disk size by the block size. The size of each bit can vary depending on how it is stored (e.g., 1 byte = 8 bits).

The overhead imposed by a free bitmap is the ratio of its size to the disk size. The smaller this ratio, the less space is wasted by storing information about free space.

For example, if we have a 100 GB disk with 1024 bytes block size and we use 1 byte to store each bit in the free bitmap, then:

The number of blocks on the disk is 100 \* 10243 / 1024 = 107

The size of each bit in the free bitmap is 1 byte = 8 bits

The size of the free bitmap is (107 \* 8) / (8 \* 1024) = (107 / (8 \* 1024)) KB

The overhead imposed by the free bitmap is (107 / (8 \* 1024)) / (100 \* 10243) = (107 / (800 \* 10246)) = ~10-7

This means that only about ~10-7% of the disk space is used for storing information about free space using a free bitmap.

The size of the free bitmap would be 1024 \* 8 = 8192 bits (assuming 1 byte = 8 bits) if each block were represented by a single bit in the free bitmap.

When the disk size is divided by the block size, the result is the total number of blocks on the disk. As a result, if the disk has a size of 100 GB, there will be 107 blocks overall since 100 \* 10243/1024 = 107 blocks.

As a result, the free bitmap would have a size of 8192 \* 107 bits, or 8192 \* 107 / 8 = 107 KB = 104 MB.

The free bitmap's overhead can then be estimated using the formula free bitmap size / disk size. Therefore, the overhead for a 100 GB drive would be 104 / (100 \* 1024) = 10-4, or 0.01%.

7. What percentage of the storage space is overhead in a list-structured allocation structure, as illustrated in Figure 17-6?

Ans: A linked list is a natural means of representing the blocks of a file while allowing them to be spread out across the disk. As with representing a free list, we can either embed the links into the data blocks themselves, or we can maintain a separate data structure. As an example of the first case, suppose each disk block is 512 bytes and blocks are identified with a 4-byte index. In each of these blocks, 508 bytes can be used for data and the first four bytes give the location of the next block in the file. Figure 17-5 shows this approach. A variation on this second approach uses a single array of pointers, one for each data block in the file system.

The array is indexed by the same block numbers as the set of data blocks. Each entry in the array gives the next block in the file. The last block in a file is identified with a special value. Some systems refer to this as a file allocation table (FAT).

Suppose we have two files in a very small file system. The first file uses blocks 5, 3, 2, and 6, in that order. Blocks 4, 1, and 8 comprise the second file. The only free block in this small file system is 7. The table for this case looks like Figure 17-7.

The overhead in a list-structured allocation structure is typically the size of the block pointer (4 bytes) multiplied by the number of data blocks in the file. In the case of a linked list structure with 512-byte blocks, each block can store 508 bytes of data and 4 bytes for the block pointer, so the overhead is approximately 0.8% ((4/512)\*100).

8. If we have a three-level tree-structured allocation structure as in Figure 17-8, where each block is 1024 bytes and each block pointer is 4 bytes, how large is the largest file that can be represented?

Ans: Suppose we have a tree with two levels of index blocks. With our 512-byte block size, the root index block contains 128 indirect pointers addressing 128 index blocks, each of which indexes 128 data blocks of 512 bytes each. This arrangement allows for files up to 8 MB in size. If we design our tree to have three levels of index blocks, we can have files as large as 1 GB. In this case, the root block contains double-indirect pointers to blocks of single-indirect pointers to data blocks.

Figure 17-8 shows this structure. Although this structure allows us to manage large files, randomly addressing any block in the file requires four disk reads regardless of how large or how small the file is. In Chapters 18, 19 and 20, we see examples of hybrid designs that provide random access of smaller files with fewer disk accesses while still allowing large files to be represented.

the root block contains 256 pointers to index blocks at level 2. Each index block at level 2 contains 256 pointers to index blocks at level 3. Each index block at level 3 contains 256 pointers to data blocks. Therefore, the total number of data blocks that can be addressed by the tree is 256 \* 256 \* 256 = 16,777,216.

Multiplying this number by the block size of 1024 bytes gives us the maximum file size of 16 GB.

For the three-level tree-structured allocation structure as shown in Figure 17-8, with each block being 1024 bytes and each block pointer being 4 bytes, the root block contains 1024/4 = 256 pointers to index blocks at level 2. Each index block at level 2 contains 1024/4 = 256 pointers to index blocks at level 3. Each index block at level 3 contains 1024/4 = 256 pointers to data blocks. Therefore, the total number of data blocks that can be addressed by the tree is 256 \* 256 \* 256 = 16,777,216.

Thus, the largest file that can be represented in this three-level tree-structured allocation structure is 16,777,216 \* 1024 = 17,179,869,184 bytes (approximately 16 GB).

9. The inner for loop (the one iterating over the variable i) in devwalk ( ) doesn’t have a terminating condition. Is there a danger of an infinite loop here? Why or why not?

Ans: No, there is no danger of an infinite loop in the inner for loop of devwalk(). This is because the loop uses the null-terminated string convention in C, where the loop will automatically terminate once it reaches the end of the string. In the loop, the variable name[i] is used as the condition for the loop, and since name is a null-terminated string, the loop will exit once it reaches the null character at the end of the string.

The devwalk function takes a gen function argument, which returns a Dirtab structure that contains information about the file in the device. The gen function also returns -1 when there are no more files to walk. Therefore, the inner for loop will terminate when gen returns -1.

10. Given the default block size of 1024 bytes, what is the largest file that can be represented in kfs?

Ans: In kfs, the file size is represented using a 32-bit integer, which can store values up to 2^32 - 1.

The block size is 1024 bytes, which means that each block can store up to 1024 bytes of data. Therefore, the maximum number of blocks that can be used to store a file is (2^32 - 1) / 1024.

To find the maximum file size that can be represented in kfs, we multiply the maximum number of blocks by the block size:

Max file size = Max number of blocks × Block size

= ((2^32 - 1) / 1024) × 1024

= 2^32 - 1 bytes

≈ 4 GB

the file size is represented using a 32-bit integer, which can store values up to 2^32 - 1. The block size is 1024 bytes, which means that each block can store up to 1024 bytes of data. Therefore, the maximum number of blocks that can be used to store a file is (2^32 - 1) / 1024.

Multiplying this number by the block size gives us the maximum file size of 2^32 - 1 bytes or approximately 4 GB.