**QUESTION 1**

At timestamp 1:21:33 in the Feb 15 Echo360 recording of class, I talk about a line of code in poolalloc that rounds the size of the request.  Explain how that works and justify that it will work in all cases.

Ans: size = (size + BHDRSIZE + p⃗ quanta ) & ∼(p⃗ quanta );

Here, size is the size of the request, BHDRSIZE is the size of the block header structure, and p⃗ quanta is the quantum size. The & ∼(p⃗ quanta ) part rounds the size up to the nearest greater quantum. The ∼ operator is the bitwise NOT operator, which inverts all the bits of the p⃗ quanta value. The & operator is the bitwise AND operator, which sets all the bits of the result to 1 if the corresponding bits of both operands are 1. This has the effect of rounding the size up to the nearest greater quantum.

As in lecture you said to use the value of p⃗ quanta 31 and do the binary calculation so here is the example of it. if p⃗ quanta is 31, then its binary representation is 00011111. The ∼ operator inverts all the bits of this value, so we get 11100000. The & operator sets all the bits of the result to 1 if the corresponding bits of both operands are 1. So, if we apply the & operator to 11100000 and size + BHDRSIZE + p⃗ quanta, we get a value that is rounded up to the nearest greater multiple of 32. For example, if size + BHDRSIZE is 100, then size + BHDRSIZE + p⃗ quanta is 131, which in binary is 10000011. If we apply the & operator to 11100000 and 10000011, we get 10000000, which is 128 in decimal. So, the rounded value is 128, which is the nearest greater multiple of 32 to 100.

As we did the programming assignment 3 rounded the size of slabs to power of 2 it will always work in all cases. Same here the code will work in all cases because it always rounds up the size to the nearest greater multiple of p⃗ quanta. This ensures that the requested size is always aligned to the boundary specified by p⃗ quanta. If the size is already a multiple of p⃗ quanta, then the rounding operation will not change the size. If the size is not a multiple of p⃗ quanta, then the rounding operation will increase the size to the nearest greater multiple of p⃗ quanta. This guarantees that the block header and the requested memory block will be aligned to the specified boundary.

**QUESTION 2**

In the course of discussing the memory management assignment, I've made reference on several occasions to checking for whether a number is a power of two by using the following C expression:

n & (n - 1)

Explain how to use this test to determine whether n is a power of two.  Explain why this will always work correctly.

Ans: n & (n-1) as I remember when you told me to use size & (size-1) in memory management assignment to round the slabs using this expression. Which is very simple and easy to understand.

As we have slabs sizes 2,4,8,16,32,64, up to 4k. To round the slab size to power of 2 we can use this expression in pooladd when the size is not rounded make it round using this expression and add it to the slabs.

The expression n & (n - 1) can be used to check whether a number n is a power of 2. If n is a power of 2, then it has only one bit set to 1 in its binary representation. As we use bitwise operations to round the value to nearest of power of 2 and the binary number helps me to understand. If the n is 8 which is power of 2^3 in binary it is 100 and their n-1 is 7 which is 111. If its power of 2 the first bit is 1 and rest are 0 and in n-1 all bits are 1. 16 is a power of 2, and its binary representation is 10000. If we subtract 1 from 16, we get 15, which is 1111 in binary. If we perform a bitwise AND operation between 16 and 15, we get 0, which means that 16 is a power of 2. This is because the only bit that is set to 1 in 16 is the fifth bit from the right and subtracting 1 from 16 flips all the bits to the right of the fifth bit. Performing a bitwise AND operation between 16 and 15 sets all the bits to 0 in the result because the only bit that is set to 1 in 15 is the fourth bit from the right, which is not set to 1 in 16.

If n is not a power of 2, then it has more than one bit set to 1 in its binary representation. For example, 6 is not a power of 2, and its binary representation is 0110. If we subtract 1 from 6, we get 5, which is 0101 in binary. If we perform a bitwise AND operation between 6 and 5, we get 4, which means that 6 is not a power of 2. When we perform a bitwise AND operation between n and n-1, the result is 0 if size is a power of 2, and non-zero otherwise. This is because the AND operation only sets the bits to 1 in the result where both operands have 1s.

This test will always work correctly because it relies on the fact that a power of 2 has only one bit set to 1 in its binary representation.

### QUESTION 3

In the Mar 15 Echo360 recording, at timestamp 1:42:24, I discuss the way in which blocks are located in the kfs file system.  Write an expression that gives the maximum size of a file where the parameter B is the size of the block in bytes.  While we didn't cover it in class, I'll tell you that the overhead related to tags is eight bytes.

Ans: So, in the Mar 15 lecture as you talked about the disk structure in limbo using # at beginning of variables. # long dblock[NDBLOCK]; # long iblock; # long idblock; In the diagram of block you showed dentry structure there is an array of 6 block numbers each of this points to block of file if the file is xyz then that file is block 0 of xyz. You also mentioned if the block is 1kb then it will be 6kb of data, but it has some overhead we did not get that much 8 bytes used for something else. If the data block is less thn 6k we can find it from dblock but if its greater than 6k we can find it from iblock.

So basically, the iblock is the block full of block numbers. So the xyz file we have if we look into iblock one of the block in iblock will be block 6 of xyz. If the block number is not small enough to be directly addressed but is small enough to be handled by the single-indirect block, then the code handles that case. The idea is that the indirect block contains a number of pointers to additional data blocks in the file. To be more concrete, the number of direct blocks that can be accessed (NDBLOCK) is 6, and blocks are 1024 bytes each. So, the first 6 KB of the file can be accessed directly. Because each block index is 4 bytes, a disk block can hold 256 of these block indices. The indirect block then lists the block numbers that make up file positions from the 6 KB point to the 262 KB point. This as you explained in the lecture.

In the kfs file system, for example the blocks are located using a block number that is stored in the iblock. The block number is used to index into a block map that is stored in the iblock. The block map contains the disk addresses of the blocks that make up the file. Let's assume that each block has a size of B bytes and that the overhead for tags is eight bytes. This means that the actual size of each block is B + 8 bytes. If we have 1kb blocks each of this block number is 4bytes so that will give us 256k. so the final expression will be like this as this from my knowledge what I understand is this

Max file size = (B+8) \* (maximum number of blocks that can be addressed by the iblock);

where B is the size of the block in bytes, and 8 is the overhead related to tags. Multiplying this by the block size B gives the maximum size of a file in the kfs file system.

### QUESTION 4

Suppose we have a computer where, from the CPU's perspective, there is only one very high-speed serial I/O device.  (Think USB on steroids, but with a cleaner and more sane design.)  In terms of a controller, there is just a single memory location that can be read and written.  We'll say that the address of this register is 0xffff ffff 0000 0000 (as a 64-bit address).  The 64-bit value written to that address is interpreted as the address of a linked list of operations to perform.  Reads from that address return 64 bits of status information.  This computer is the basis of the next several questions.

The data structures and global variables you will need are declared as follows:

typedef struct Req Req;

struct Req {

int async;

vlong addr;

void \*cmd;

void \*data;

void \*up;

Req \*next;

};

Req \*curq, \*nexthd, \*nexttl;

The curq pointer is the head of the list that's currently being processed by the controller.  Nexthd and hexttl define the queue that we are adding requests to while the controller is busy.

For the first question, if we have some operations in the next queue and we see that the controller is idle, we can start the controller running the next list.  Write a few lines of code that do this.  I'm not going to be overly picky about C details here, but you've been writing in C all quarter, so I do expect you to at least be able to get close without running out and looking things up.  (Note that you are allowed to reference the text and the lectures, so there are plenty of example of code available to you.  There's no excuse for going out to Google or anything else for that.)

Ans: So, start with the struct I will write it down what I have understand a struct Req with fields async, addr, cmd, data, up, and next. The Req struct is used to represent a request to the controller. The async field is a flag that indicates whether the request should be processed asynchronously or synchronously. The addr field is the address of the I/O device that the request is for. The cmd field is a pointer to the command that should be executed. The data field is a pointer to the data that should be transferred. The up field is a pointer to user data that should be passed to the completion routine. The next field is a pointer to the next request in the queue. It has declared three pointers curq, nexthd, and nexttl. curq is a pointer to the head of the list that’s currently being processed by the controller. nexthd and nexttl define the queue that we are adding requests to while the controller is busy.

We'll say that the address of this register is 0xffff ffff 0000 0000 (as a 64-bit address).  The 64-bit value written to that address is interpreted as the address of a linked list of operations to perform.  Reads from that address return 64 bits of status information. To start the controller running the next list, we need to check if the controller is idle and if there are operations in the next queue. If both conditions are true, we can set the current queue to the next queue and update the next queue pointers.

if (curq == nil && nexthd != nil) {

curq = nexthd;

nexthd = nexthd->next;

curq->next = nil;

// Write the address of the current queue to the controller

vlong addr = 0xffff ffff 0000 0000 | (vlong)curq;

write\_to\_controller(addr);

// Wait for the controller to finish processing the queue

while (read\_from\_controller() != 0) {

// reading the status information

}

if (nexttl == nil) {

nexttl = curq;

} else {

nexttl->next = curq;

nexttl = curq;

}

}

### This code checks if the current queue is empty and if there are operations in the next queue. If both conditions are true, it sets the current queue to the next queue and updates the next queue pointers. It then writes the address of the current queue to the controller by setting the upper 32 bits to 0xffff ffff and the lower 32 bits to the memory address of the current queue casted to a 64-bit integer. This allows the controller to access the linked list of operations to perform.

### We use write to controller and read from controller to function which we help while reading and writing the address. After writing the address to the controller, the code enters a loop that waits for the controller to finish processing the queue. This is done by repeatedly reading the status information from the controller until it returns 0, indicating that the queue has been fully processed.

### Once the controller has finished processing the queue, the code updates the next queue tail pointer if it is nil or appends the current queue to the end of the next queue if it is not nil. This ensures that any new requests added to the queue will be processed in the correct order.

### This question also works with real life example like factory that produces cars. One of the operations in the production process is painting the car body. There is a queue of cars waiting to be painted and a controller who operates the painting machines. If the controller is currently idle and there are cars waiting in the queue to be painted, it would make sense to start the controller running the next list of cars to be painted.

### This ensures that the painting process keeps moving smoothly and there is no delay in production. So, the next list of cars in the queue is started and the controller begins painting the cars. This way, the production process continues without any delay, and the cars move through the painting process efficiently. Same example goes with printer or xerox machines will work.

### QUESTION 5

The async member of the Req structure is a flag that is 1 if the request is asynchronous and 0 if it is synchronous.  This affects the interrupt behavior.  In particular, after all requests have been initiated and after all synchronous requests have been completed, the controller generates an interrupt to alert the CPU of the completion.  (For the purposes of this exam, we won't deal with the details of asynchronous requests.)  Also significant is the pointer, up, which the OS can use as it sees fit.  We will use it to point to a rendezvous which is the structure on which the requesting process is waiting.  Describe in pseudo-code what the interrupt handler for this controller needs to do.

Ans: So, here we are using the async flag that is 1 if the request is asynchronous and 0 if it is synchronous. After all requests have been initiated and after all synchronous requests have been completed, the controller generates an interrupt to alert the CPU of the completion. Also significant is the pointer, up, which the OS can use as it sees fit.  We will use it to point to a rendezvous which is the structure on which the requesting process is waiting. We can create a function called interrupt\_handler() to perform the interrupt behavior.s

void interrupt\_handler() {

if (curq->async == 0) {

// Find the completed request

Req \*completed\_req = find\_completed\_request();

// Wake up the process waiting on the rendezvous

wakeup(completed\_req->up);

// Remove the request from the queue

remove\_request(completed\_req);

// If there are more requests, start processing them

if (curq != nil) {

start\_processing\_requests();

} else {

// Generate interrupt to signal completion of synchronous requests

generate\_interrupt();

}

} else {

// Handle asynchronous requests

handle\_asynchronous\_requests();

}

}

The handler first checks whether the current request is synchronous (async flag is 0). If it is, the handler proceeds to find the completed request, wake up the waiting process on the rendezvous pointed to by the up field of the completed request, and remove the request from the queue.

After that, the handler checks if there are more requests in the queue. If there are, it starts processing them. If not, the handler generates an interrupt to signal the completion of all synchronous requests.

If the current request is asynchronous (async flag is 1), the handler simply passes control to the function handle\_asynchronous\_requests() to handle the request appropriately.

This question I can relate with the printer driver in computer system example. The printer has controller which has both synchronous and asynchronous requests. In the driver code, you define a structure called Req that represents a print request. The Req structure has two members: async, which is a flag that is 1 if the request is asynchronous and 0 if it is synchronous, and up, which is a pointer to a rendezvous structure on which the requesting process is waiting.

When a print request is made, the driver adds a Req structure to a queue of requests. If the request is synchronous, the driver blocks the requesting process until the request is completed. If the request is asynchronous, the driver immediately returns control to the requesting process. The controller generates an interrupt to alert the CPU of the completion of a print request. The interrupt handler checks the async flag of the completed request to determine whether it is synchronous or asynchronous.

If the request is synchronous, the interrupt handler wakes up the process waiting on the rendezvous pointed to by the up field of the completed request. The handler then removes the completed request from the queue and checks if there are more requests in the queue. If there are, the handler starts processing them. If not, the handler generates an interrupt to signal the completion of all synchronous requests. If the request is asynchronous, the interrupt handler simply removes the completed request from the queue and checks if there are more requests in the queue. If there are, the handler starts processing them.

### QUESTION 6

The addr member of the Req structure gives the 64-bit address of the device that this command packet is directed to.  For simplicity, let's assume we have only one disk drive and it's at address 0x0000 0001 0000 0000 and a command packet for the disk has 16 bytes.  The first byte is the command.  It has the value 0x21 for a read and the value 0x31 for a write.  The next three bytes are unused in this example.  The next four bytes are the count of the number of sectors to transfer.  The final eight bytes of the command packet are the starting sector number.  All of the address, the number of sectors, and the sector number are given in big endian (most significant byte first) order.  Assume that the sectors are the classic 512 byte size.  The cmd member of the Req structure points to the command packet and the data member points to the memory address where the data is to be transferred.  Given that we are implementing a driver function similar to those in Inferno declared as:

vlong

diskwrite(Chan \*, void \*va, vlong n, vlong off)

write the code to insert a new request into the queue.  For simplicity, assume that both the number of bytes, n, and the offset, off, are multiples of 512.

Ans: So, the addr member of the Req structure gives the 64-bit address of the device that this command packet is directed to. It has the value 0x21 for a read and the value 0x31 for a write. Here is the code to create a new request and insert it into the queue for the disk driver:

vlong

diskwrite(Chan \*, void \*va, vlong n, vlong off)

{

// Allocate a new request

Req \*new\_req = malloc(sizeof(Req));

// Allocate memory for the command packet

new\_req->cmd = malloc(16);

// Set the address of the device

\*(uint64\_t \*)new\_req->cmd = 0x0000 0001 0000 0000;

// Set the command

new\_req->cmd[0] = read ? 0x21 : 0x31;

// Set the number of sectors to transfer

\*(uint32\_t \*)(new\_req->cmd + 4) = n / 512;

// Set the starting sector number

\*(uint64\_t \*)(new\_req->cmd + 8) = off / 512;

// Set the data pointer

new\_req->data = va;

// Add the request to the queue

add\_request(new\_req);

}

In this code the function diskwrite, it is creating a new request and inserting it into the queue for the disk driver. The parameters of the function are a channel (c), a pointer to the memory address of the data to be transferred (va), the number of bytes to transfer (n), the offset (off), and a boolean flag indicating whether the operation is a read or a write (read).

The code first allocates memory for a new request using the malloc() function. It also allocates memory for the command packet using the malloc() function, and sets the address of the device to 0x0000 0001 0000 0000, which is the address of the disk drive in this example.

It then sets the first byte of the command packet to either 0x21 or 0x31 depending on whether the operation is a read or a write, respectively. It sets the next four bytes to the number of sectors to transfer, which is calculated by dividing the number of bytes by 512. Finally, it sets the last eight bytes to the starting sector number, which is calculated by dividing the offset by 512.

In code I create function add\_request() that adds the new request to the queue. I also assumes that Chan is a structure that represents a channel and vlong is a 64-bit integer type. The n and off are multiples of 512. The read parameter is a boolean that indicates whether the operation is a read or a write. The code then sets the data pointer to point to the memory address of the data to be transferred, and adds the new request to the queue using the add\_request() function.

### QUESTION 7

The previous question assumed that the request was aligned on a sector boundary.  Write out the pseudo-code for how you'd handle a request that is not so aligned.

Ans: As we do align on a sector boundary in previous question I understand that if we have to show a request is not so aligned first I think it does not make any sense to do that but as we deal with conditions it can happened some sector boundary which will not be aligned. For that basically first thing comes in my mind is if else statement.

if (request is not aligned on a sector boundary) {

read the sector containing the first byte of the request

read the sector containing the last byte of the request

modify the data in the first sector and the last sector to include the request data

write the first sector back to the storage device

write the last sector back to the storage device

} else {

write the data to the storage device

}

The given pseudo-code describes how to handle unaligned requests on a sequential storage device. First we will start to check if the request is aligned on a sector boundary: If the request is not aligned on a sector boundary, it means that the request spans across multiple sectors, and special handling is required.

Read the sector containing the first byte of the request: In order to modify the data, the first sector containing the first byte of the request needs to be read from the storage device.

Modify the sector to include the data from the request: Once the sector is read, the data from the request can be modified and written to the sector.

Write the sector back to the storage device: After the data has been modified, the sector needs to be written back to the storage device.

Read the sector containing the last byte of the request: Similarly, the last sector containing the last byte of the request needs to be read from the storage device.

Modify the sector to include the data from the request: The data from the request is then modified and written to the sector.

Write the sector back to the storage device: Finally, the sector is written back to the storage device.

Write the data to the storage device: If the request is aligned on a sector boundary, the data can be directly written to the storage device without any special handling.

The code reads and modifies the sectors containing the first and last bytes of the unaligned request and then writes them back to the storage device. For aligned requests, the data can be directly written to the storage device without any extra handling. This approach ensures that the data is stored correctly on the storage device, even if the requests are not aligned on sector boundaries.

### QUESTION 8

The previous few questions did not mention locking.  Discuss what type of locking you need to use here and where the locking and unlocking will take place.

Ans: When multiple threads or processes are accessing a storage device, it is essential to ensure that they do not interfere with each other and corrupt the data. To achieve this, proper locking mechanisms need to be used. As I know whenever we perform something which is changed in between while running code we can put locks in that code, and it will work how we wants to workflow.

Checking whether a request is aligned on a sector boundary or not does not require any specific type of lock. The purpose of using a lock in this case is to ensure that other threads or processes do not interfere with the data while it is being read or written. As I know the locks which I used before is the mutex and read-write lock. But in this case, I will use mutex because I know in kernel mostly mutex locks are used.

Pthread\_mutex\_lock and Pthread\_mutex\_unlock these two are very helpful whenever we required them for threads code, we can use them and set them lock before start of the code and unlock once the code perform their work. If I’m not wrong, we can use read-write locks also I’m this case read-write locks which allow multiple threads to read a shared resource simultaneously while ensuring that only one thread can write to it at a time. Where mutex locks provide exclusive access to a shared resource while read-write locks allow concurrent read access.

In the case of handling unaligned requests, the locking and unlocking would take place as follows:

acquire lock on the file.

if (request is not aligned on a sector boundary) {

read the sector containing the first byte of the request.

read the sector containing the last byte of the request.

modify the data in the first sector and the last sector to include the request data.

write the first sector back to the storage device.

write the last sector back to the storage device.

} else {

write the data to the storage device.

}

release lock on the file

In this code, I acquire a lock on the file before performing any read-write operations. If the request is not aligned on a sector boundary, you read the sectors containing the first and last bytes of the request, modify them to include the request data, and then write them back to the storage device. If the request is aligned on a sector boundary, you simply write the data to the storage device. Finally, you release the lock on the file. This type of locking mechanism can help ensure data integrity when multiple threads or processes are accessing a storage device.

### QUESTION 9

Suppose we have a storage device that can be read randomly but must be written sequentially.  I.e., each block that is written must be the next physical block after the previous one written.  Describe how you would design a kfs-like file system for such a storage device.

Ans: Designing a kfs-like file system for a storage device that supports random reads but requires sequential writes would involve implementing mechanisms to ensure that data is written sequentially to the storage device. This could be achieved by keeping track of the current write position and only allowing writes to occur at the next physical block after the previous one written.

In assignment 5 we modify the skelread and skelwrite function for devskel.c file. In that and in lecture you teach us that how the zeros file works using dd command we can read the data from randomly and write the data sequentially.

In skelread, instead of reading the data from the RAID devices sequentially, we can use the kpread function to read data from a random offset within the file. To achieve this, we need to modify the offset argument of the kpread function to a random offset.

For doing this first we have to Open the file to read from the file or write to the file, we must first open it. In the case of the "mirror1" file created using the "zeros" command, we can use the command "cat mirror1" to read the contents of the file. Read data randomly: Since the file system supports random access, we can use the "dd" command to read a specific part of the file. For example, if we want to read 512 bytes from the middle of the file, we can use the dd command to read the data from 512 bytes and skip some parts directly using the help of command.

In skelwrite, we can write data sequentially by setting the offset argument of kpwrite function to the current length of the file. This ensures that the data is always written at the end of the file, thus writing sequentially. We also need to update the length of the file in skeltab[Qdata].length after writing data to the file.

Same for writing we can do it write data sequentially to write to the file, we must ensure that the data is written sequentially. We can use the "dd" command to write to the file in blocks. For example, if we want to write 512 bytes to the end of the file, we can use the command and start the writing position to that 512-byte location, and it will start writing to file. This command will write 512 bytes of zeros to the end of the file.

Once everything is done, we can close the file. Once we are done with reading or writing to the file, we must close it using the appropriate command, such as "kclose" we use to close the file in the kernel programming.

As we talked in last lecture that truncate the data or information when writing in file is one of the approaches, where all writes are appended to a block on the storage device. This would ensure that data is written sequentially while still allowing for efficient random access when reading data. To ensure that data is in sequential writes, the file system would need to check the directory structures that each block of data has its id and address correctly, and data integrity checks is data is leak or corrupt. These features could be implemented in a similar manner to other kfs-like file systems.

Another thing comes in mind while doing the last assignment every time we echo some text it will show in file but truncate or write from start not understanding. As I know the file is getting new input everytime when we do echo >> file1. But if we want to truncate the data, we can set the last location address of the va to the next data written like my friend he did 1024 set it to the va offset so eveytime he written something it goes to showed on 1024 bytes. Similar we can do that for to write the data sequentially. We can set the parameter everytime to the last location where data is written is my understanding.

In a kfs-like file system, we can implement similar functionality by ensuring that data is written sequentially to the storage device. This could be achieved by keeping track of the current write position and only allowing writes to occur at the next physical block after the previous one written. When reading data from the file system, you could use random access to read specific parts of the file as needed. This would allow for efficient access to data stored on the storage device.

By doing these steps, we can read data randomly and write data sequentially to a file system that supports random access but requires sequential writing.

This question is about file system so we can say here more about how the file system works and what are the main components of it. inferno file server communication diagram in textbook the kfs is just one section and it has his many sub sections.

I found this useful information in textbook about creating a new file system is referred to as reaming in Inferno. On disk, the first block, called the superblock, contains metadata describing the file system as a whole. Free space is represented by free block lists similar to that illustrated in Figure 17-4. Blocks in files are managed using a hybrid list-structured allocation and tree-structured allocation in a way that is very similar to UNIX file systems. Unlike UNIX, however, Inferno expects all of a file’s metadata to be in its directory entry, as we saw in devdir ( ) in Section 19.3. This leads to the kfs directory structure where all of a file’s metadata is kept in its on-disk directory entry. Therefore, there are no i-nodes as found in UNIX file systems.

From this block of information, I understand if I am going to create new file system like kfs which will needs the superblock which will store the metadata of my new file system. The other thing in file system required is tree structure which will have bunch of directories. Where all file’s metadata stored on disk directory entry.

Superblocks and inode in chapter 20 When a file system is mounted, a file system–specific function is called to load an internal representation of the file system’s metadata. Named after the original UNIX on-disk metadata, this is called the superblock. A member of the internal superblock structure points to a structure of type structs super operations. This structure contains a number of function pointers that are needed to carry out operations on a mounted file system. Although the name suggests that these functions are primarily related to the superblock, most are actually functions needed to fetch and manipulate other metadata structures called i-nodes.

Another thing is textbook which is important is allocation and management of free space on the storage device. Divide the storage device into fixed-size blocks. Each block should be large enough to hold at least one file system block, which typically ranges from a few hundred bytes to a few kilobytes. Reserve the first block of the storage device for the superblock. The superblock should contain metadata about the file system, such as the number of blocks, the block size, and the location of the root directory. Reserve a portion of the storage device for the free block list. The free block list should be a linked list of blocks that are currently not in use. Each block in the free block list should contain a pointer to the next free block.

Reserve the next block after the free block list for the root directory. The root directory should be a special directory that contains entries for all the files and directories in the file system. Each entry should contain the file or directory name, the location of the file's or directory's metadata, and the size of the file or directory. Allocate free blocks sequentially as files are created or extended. When a new file is created or an existing file is extended, allocate a sequence of free blocks starting from the last block allocated to the file. Update the file's metadata, including its size and location of its blocks, in the directory entry.

When a file is deleted, mark its blocks as free in the free block list. Update the directory entry to remove the file's metadata. This approach will help to designing a kfs-like file system for a storage device that can be read randomly but must be written sequentially ensures that files are written sequentially and improves performance. It also ensures that free blocks are allocated efficiently, which maximizes the use of the storage device.