University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 2023 John Kubiatowicz

Midterm III
SOLUTION

April 27th, 2023

CS162: Operating Systems and Systems Programming

Your Name:	
Your SID:	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book** exam. You are allowed 3 pages of notes (both sides). You have 2 hours to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming. Write all of your answers directly on this paper. *Make your answers as concise as possible.* If there is something about the questions that you believe is open to interpretation, please ask us about it!

Problem	**Possible**	**Score**
1	20	
2	20	
3	23	
4	20	
5	17	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [20 pts]

Please *EXPLAIN* your answer in TWO SENTENCES OR LESS (Answers longer than this may not get credit!). Also, answers without an explanation *GET NO CREDIT.*

Problem 1a[2pts]: The clock algorithm provides an approximation to a FIFO allocation policy, since it links physical pages in a large circular queue.

☐ True [✓ False]

Explain: *The clock algorithm provides an approximate to LRU. The fact that physical pages are linked in a large circular queue is merely a consequence of the algorithm: the clock algorithm walks through all of the pages in a loop looking for an "old" page rather than the "oldest" page by looking at the Use bit.*

Problem 1b[2pts]: An empty Amazon Kindle gains weight when you add electronic books to it.

[✓ True] ☐ False

Explain: *As Professor Kubi discussed in class, a FLASH memory that is holding contents (such as books) is in a higher energy state than an erased FLASH memory and thus is very slightly heavier than an empty Kindle. (NOTE: The key being very, very, very slightly, of the order of attograms, i.e. 10^{-18} grams).*

Problem 1c[2pts]: When producing a queueing model of some system (such as a filesystem) and the distribution of request arrival times is unknown, the best estimate is to use a *deterministic* distribution.

☐ True [✓ False]

Explain: *Few if any systems are deterministic. As we discussed in class, an exponential ("memoryless") distribution could be a good choice when you don't have a lot of information about the system other than that there are a bunch of events combined together from multiple independent source.*

Problem 1d[2pts]: For disks with constant *areal bit density* (bits stored/unit area of disk media), the disk head reads bits at a different rate on the outer tracks than on the inner tracks.

[✓ True] ☐ False

Explain: *Since the rotation speed of a disk is constant during operation, the media is flying underneath the disk heads faster on the outer tracks than on the inner tracks. Because the density of bits is the same throughout the disk, this means that bits pass underneath the disk head faster on the outer tracks than on the inner tracks.*

Problem 1e[2pts]: The Bitcoin blockchain could be used for distributed decision making.

[✓ True] ☐ False

Explain: *The bitcoin blockchain can be used to take a set of proposals from different participants and fix them in a serial order. As a result, it is possible for all parties to look at the serial order and come to the same decision independently via some deterministic function of proposals.*

Problem 1f[2pts]: Direct Memory Access (DMA) can reduce the load on the processor when accessing block devices such as disks.

[✓ True] ☐ False

Explain: *Rather than transferring large blocks of data from a block device (i.e. 4KiB or more) to memory using a loop, the processor can hand the transfer off to a DMA device and do something else while the transfer is happening.*

Problem 1g[2pts]: Supposing that during Two Phase Commit for transaction T, the following occurs:

1. The Coordinator sends VOTE-REQ to all workers
2. Worker A sends VOTE-COMMIT to the Coordinator
3. Worker A dies
4. The Coordinator sends GLOBAL-COMMIT to all workers
5. Worker A recovers

Then Two Phase Commit for T fails because Worker A never received the GLOBAL-COMMIT message, so T was never committed to its log.

☐ True [✓ False]

Explain: *Once the coordinator has started sending GLOBAL-COMMIT, we know that it has stored that fact in its log. Consequently we know that (1) the result of 2PC is commit and (2) Worker A can know that it crashed before getting the Coordinator's decision, but can ask the Coordinator for the result to find out that it is supposed to commit.*

Problem 1h[2pts]: Network Address Translation (NAT) automatically switches the IP address of a remote server to spread load over multiple servers (thereby improving overall response time).

☐ True [✓ False]

Explain: *Network Address Translation (NAT) is a mechanism for sharing a single IP address among multiple clients.*

Problem 1i[2pts]: Since the File Allocation Table (FAT) does not actually store file contents, the FAT is not stored on disk.

☐ True [✓ False]

Explain: *Since the FAT keeps track of which blocks belong to which files (and in which order), it is an essential part of the file system. It must be stored on the disk – otherwise the file system cannot survive a power outage.*

Problem 1j[2pts]: Because RAID 5 is an instance of an erasure code, it requires individual disks to be able to identify when they have failed using a different code.

[✓ True] ☐ False

Explain: *In order for RAID 5 to recover from a failure, it must know which disk to exclude when XORing the remaining disks together. Consequently, each disk must employ some other error correction code on its data to recognize failure.*

Problem 2: Multiple Choice [20pts]

Problem 2a[2pts]: Which of the following is *true* about Pintos' file system? ***(Choose one):***

A: ☐ Directory data (i.e., directory entries) are stored on the directory's inode on disk.

B: ☐ Two threads can always write to the same file concurrently.

C: ☑ The seek syscall should never require blocks in the file to be allocated or freed.

D: ☐ The directory path "/../../" is invalid.

E: ☐ None of the above

A is false because inodes hold metadata, not data. B is false because the basic file system implementation is not threadsafe; simultaneous modification by concurrent threads can cause inconsistency. C is true because setting the offset to any location in the file does not cause the underlying blocks allocated of the file to change (this only changes on a write or rollback). D is false because ".." in the root directory simply points back at the root directory.

Problem 2b[2pts]: Which of the following is *false* about Pintos' file system? ***(Choose one):***

A: ☐ To determine the end of a directory when reading directory entries, PintOS relies on an offset with a file length.

B: ☐ Once the buffer cache has been implemented, inodes should no longer store the data for inode_disk.

C: ☑ To enforce synchronization, we add a lock to inode_disk.

D: ☐ Extending a file requires that you fill up the empty space with zeros.

E: ☐ None of the above are false (i.e. all of the above are true).

A is true because of the inode format. B is true because inode_disk data should be stored in the cache. C is false because synchronization should be done in memory, not on disk. D is true because basic file format does not support disjoint file segments and you cannot leave uninitialized data in a file (when you skip past end of file before writing).

Problem 2c[2pts]: Which of the following is *true* about distributed decision making protocols ***(Choose one):***

A: ☐ The two-phase commit protocol is tolerant to Byzantine failures.

B: ☑ In the two-phase commit protocol, failure of the coordinator at any time before it writes "commit" into its log will cause the transaction to be aborted.

C: ☐ If a worker in the two-phase commit protocol fails *after* voting to commit, it will commit the transaction when it later recovers.

D: ☐ The PAXOS protocol is less robust than two-phase commit because it can block on a failed coordinator.

E: ☐ None of the above.

A is false because two-phase commit only functions if all participants are honest and functioning (cannot even deal with permanent crashes). B is true because it is always correct for the coordinator to send an abort up until it sends its first commit message. In the basic protocol given in class, the coordinator only marks major transitions in phase in its log. The fact that it crashed before writing the final commit in its log means that it lost the responses from workers and thus must abort. C is false because it is not even true when the worker doesn't fail (i.e. just because a particular worker votes to commit doesn't mean that the 2pc will choose to commit). D is false because PAXOS is can choose a new coordinator and thus is not vulnerable to a failed coordinator.

Problem 2d[2pts]: Which of the following are true of the Fast File System (FFS)? ***(Mark all that apply):***

A: ☐ FFS changed the inode format from the BSD 4.1 file system to better reflect the overwhelming presence of small files in a typical UNIX filesystem.

B: ☑ Direct pointers in the inode allow for reduced access times for small files.

C: ☑ Free blocks are tracked using a bitmap, allowing for files to be more easily allocated in contiguous memory.

D: ☐ It placed all the inodes together on the inner tracks of the disk for better performance.

E: ☑ FFS keeps 10% of disk blocks free to help with contiguous block allocation of files (and to improve overall read performance).

A is false because FFS (in BSD 4.2) kept the same inode format from BSD 4.1. B is true for this shared inode structure. C is true, since free block tracking using a bitmap was a change for the FFS that allowed better recognition of groups of free blocks (for locality of allocation). D is false, since FFS spread inodes throughout block group. Finally, E is true because reserving free space makes it more likely for there to be runs of free blocks throughout the disk.

Problem 2e[2pts]: Which of the following are true about modern hard disks? ***(Mark all that apply):***

A: ☐ They have an independent disk head on every surface of every platter. As a result, the disk can simultaneously read or write from different tracks on different platters.

B: ☑ Their internal controllers can queue requests and perform variants of the elevator algorithm without consulting the operating system.

C: ☑ They have internal caching, allowing them to read a whole track at a time.

D: ☐ They have a lower bit density on the outside tracks from the inside tracks (because the surface of the outside tracks move under the disk head faster than that of the inside tracks).

E: ☐ None of the above.

A is false because disk heads are locked together in a single disk arm and move as a unit. B and C are true because modern disk controllers have internal queueing and caching for track buffers. D is false because the density of bits on the surface of a platter is constant throughout the disk.

Problem 2f[2pts]: Little's Theorem has the following properties ***(Mark all that apply):***

A: ☐ It applies only to memoryless arrival distributions.

B: ☑ It can be used to compute the average time spent in a queue, given the average length of the queue and average arrival rate.

C: ☐ It shows why the average length of time spent in a queue grows without bound as the system utilization approaches 100%.

D: ☑ It applies only to systems in equilibrium.

E: ☐ None of the above.

A is false because Little's Law applies to any distribution of arrivals. B is true simply by rearranging Little's law in the form: $T_{queue} = \frac{L_{queue}}{\lambda}$. C is false because Little's law has nothing to do with utilization. D is true because Little's law only applies to systems in equilibrium.

Problem 2g[2pts]: Which of the following statements about I/O performance are true? ***(Mark all that apply)*:**

A: ☑ The elevator algorithm can improve hard disk drive performance by handling I/O requests in order of physical location rather than in order of arrival.

B: ☑ For many I/O devices, the effective bandwidth of a request can be improved by increasing the size of the request (in bytes).

C: ☑ If the requests entering a queue are combined from many different (uncorrelated) sources, then the arrival distribution can be roughly modeled by an exponential (memoryless) distribution.

D: ☐ SSDs have a seek time that is higher than the time to perform a write.

E: ☐ None of the above.

A is true because rearranging requests can allow much less time devoted to seeking. B is true because increasing request size can decrease the fraction of time devoted to overhead. C is true; it is the justification for presenting the queueing models that we did (with a memoryless input). D is false because SSDs have no seek time; they are direct access devices.

Problem 2h[2pts]: Which of the following statements about I/O performance are *false*? ***(Choose one)*:**

A: ☐ Little's law applies to all stable systems, regardless of structure, bursts of requests, or variation in service.

B: ☑ It is impossible for utilization to be 1 in any stable system.

C: ☐ Utilization can be computed by λ/μ, where μ is the completion rate in jobs/second and λ is the arrival rate in jobs/second.

D: ☐ The equation $T_{ser} \times u / (1 - u)$ calculates the amount of time that a job spends waiting in the queue for a device with an exponential service time.

E: ☐ None of the above are false (i.e. all of the above are true).

A is true: Little's law is independent of the actual distributions of arrival or service times. <u>B is the false choice</u> because a deterministic arrival process with a deterministic service process can be stable with 100% utilization. Note that the queueing equations that we gave in class were for a memoryless (non-deterministic) arrival process. C is true by definition of utilization. D is also true – this is the C=1 queueing equation that we gave in class.

Problem 2i[2pts]: Memory-mapped I/O is ***(Choose one):***

A: ☐ A security mechanism for I/O devices that that prevents user-mode applications from directly accessing these devices, forcing device access to go through the system call interface.

B: ☐ A software protocol that constructs a coherent distributed shared memory between multiple physical nodes, allowing communication between nodes to occur as reads and writes to the shared memory (address) space.

C: ☐ A technique for communication between processes using the mmap() system call. As a result, processes can interact via reads and writes to shared memory addresses.

D: ☑ A hardware mechanism for assigning physical memory addresses to devices such that processor read and write operations to these addresses become commands to devices.

E: ☐ None of the above.

A, B, and C are false because they are not descriptions of memory-mapped I/O. <u>D is a true description of memory-mapped I/O.</u> Regular load and store instructions are used to interact with I/O devices.

Problem 2j[2pts]: The Meltdown security flaw (made public in 2018) was able to gain access to protected information in the kernel because ***(Choose one)*:**

A: ☐ Data-specific variability in the processing of system-calls (related to how Intel processors handled synchronous exceptions) by one process (the victim) allowed another process (the attacker) to successfully guess values stored in the kernel stack of the victim process.

B: ☐ A POSIX-compatible system call implemented by multiple operating systems neglected to properly check arguments and could thus be fooled into returning the contents of protected memory to users.

C: ☐ There was a wide-spread bug in the x86 architecture that caused certain loads to ignore kernel/user distinctions in the page table under the right circumstances. This allowed user-code to directly load and use data that was supposed to be protected in kernel space.

D: [✓] Many processors allowed timing windows in which illegal accesses could be performed speculatively and made to impact cache state – even though the speculatively loaded data was later squashed in the pipeline and could not be directly used.

E: ☐ None of the above.

A and C are false because the Meltdown bug was not about incorrect operation of the processor API; instead, it was a far more subtle consequence of speculative execution causing changes to processor state that is normally not accessible to user-level attackers but could be detected via timing differences. B is false because Meltdown was not a software bug.

D is the true description of the Meltdown security bug. It was a hardware bug that was a consequence of speculative execution: since the outcome of branches and protection violations were predicted to allow higher performance, illegal accesses to kernel memory were temporarily allowed but later squashed in the pipeline when the violation was processed. As discussed in class, the trick was to get the temporary accesses to affect cache state in a way that could be viewed by an attacker.

Problem 3: Short Answer Potpourri [23pts]

Problem 3a[5pts]: Suppose that a system uses an Nth-chance clock algorithm for demand paging. Also assume that physical memory has 8 pages and each PTE has a dirty bit, a use bit, and a counter. Assume that the clock algorithm evicts dirty pages when its counter reaches 2, and evicts clean pages when its counter reaches 1. Assume that the clock hand starts at PPN 7 (wrapping around to PPN 0) and that the algorithm increments *first* before examining control bits. Assume that the physical pages are allocated to virtual pages as follows:

PPN	Contains VPN	Dirty	Use	Counter
0	1	1	~~1~~ *0*	0 *1*
1	~~3~~ *10*	0	~~1~~ *0*	0
2	~~5~~ *6*	0	0	0
3	2	0	~~0~~ ~~*1*~~ *0*	0
4	8	0	~~1~~ *0*	0
5	~~12~~ *4*	~~1~~ *0*	0	~~1~~ *0*
6	9	1	0	~~0~~ *1*
→ 7	7	0	~~1~~ *0*	0

For the following access pattern, provide the PPN of the page frame at the time that the access succeeds. Assume all accesses are reads. We have filled the first entry for you:

Accessed VPN 3: PPN =? **1**

Accessed VPN 2: PPN =? *3 [Already there, (Use⇒1)]*

Accessed VPN 6: PPN =? *2 [Skip: PPN 0, 1 (Use⇒0)]*

Accessed VPN 8: PPN =? *4 [Already there, Use bit already set]*

Accessed VPN 4: PPN =? *5 [Skip: PPN 3, 4 (Use ⇒ 0)], PPN 5 evicted (Dirty/Counter⇒0)]*

Accessed VPN 10: PPN =? *1 [Skip PPN 6 (Counter⇒1); PPN 7 (Use⇒0), PPN 0 (Counter⇒1)]*

Problem 3b[8pts]: For the following problem, assume a hypothetical machine with 4 pages of physical memory and 7 pages of virtual memory. Given the access pattern:

```
A B D D E F A A C F G D A C G D C E
```

Indicate in the following table which pages are mapped to which physical pages for each of the following policies. Assume that a blank box matches the element to the left. We have given the FIFO policy as an example. *[Clarification during exam: break ties by choosing PPN with smallest ID]*

Access→		A	B	D	D	E	F	A	A	C	F	G	D	A	C	G	D	C	E
FIFO	1	A					F						D						
	2		B					A											E
	3			D						C									
	4					E						G							
MIN	1	*A*												*All 4 options satisfy MIN, but clarification ⇒ choosing page 1*					*E*
	2		*B*				*F*					*G*							*E*
	3			*D*															*E*
	4					*E*				*C*									*E*
LRU	1	*A*					*F*								*C*				
	2		*B*					*A*					*D*						
	3			*D*						*C*				*A*					*E*
	4					*E*						*G*							

Problem 3c[3pts]: What is a *precise exception* and why is it the concept helpful for handling page faults during demand paging? (2 pts for definition, 1 pt for reason it is helpful).

Precise exceptions are exceptions for which there is a clearly identifiable exception point in the instruction stream (identified by a current instruction) for which: (1) all instructions prior to the current instruction have completed and committed their state, and (2) neither the current instruction nor any following instructions have started. This concept is helpful for handling page faults, because the page fault handler can simply restart execution at the current instruction.

Problem 3d[3pts]: What is the mmap() system call? How could it be used to set up shared memory between two processes so that they could share a linked list between them (here sharing means that the two of them could add items to the list and traverse the list assuming that suitable synchronization was employed to avoid inconsistencies):

The `mmap()` system call maps an open file into a chunk of the virtual address space such that reads and writes to addresses in that portion of the address space cause reads and writes to the mapped file. If two processes map the same file to the same address in their respective virtual address spaces, then the two processes can share a linked list by simply reading and writing to the shared address range. [Note: you can actually use `mmap()` without a file, instead mapping "anonymously" for pure shared memory. This was not required to answer this questions.]

Problem 3e[2pts]: Prior to the discovery of the Meltdown exploit, operating systems typically mapped some or all of kernel memory into the top of each user's page table (with PTEs that had the "User accessible" bit set to 0). What was the advantage of this practice?

The advantage of including PTEs that map kernel addresses in the user's page table is that system calls, traps, and interrupts gain immediate access to kernel memory as soon as they transition to kernel mode. One particularly important part of the kernel's address space is all of the code segments…

Problem 3f[2pts]: Explain why two-phase commit is not subject to the General's Paradox.

The General's Paradox involves a multi-party decision problem in which two parties try to agree on an exact time *in which something will happen using only unreliable messages. In contrast, two-phase commit does not involve time; it only involves getting a set of participants to agree to all commit or all abort* eventually. *Two-phase commit is resilient to participants crashing and recovering, which would not work with the setup of the General's Paradox.*

Problem 4: File Systems [20pts]

Problem 4a[3pts]: Suppose that you are producing a Pintos-like file system for which the `inode_disk` contains the following data pointers:

```
#define BLOCKSIZE 1024
typedef uint32_t block_sector_t;
// Assume that this is one BLOCKSIZE in size
struct inode_disk {
      block_sector_t direct_pointers[12];
      block_sector_t indirect_pointer;
      block_sector_t double_indirect_pointer;
      block_sector_t triple_indirect_pointer;
      ...
}
struct indirect_block { // indirect blocks look like this
      block_sector_t block_nums[BLOCKSIZE/sizeof(block_sector_t)];
}
```

What is the maximum file size (in bytes) that can be supported by this filesystem? (You can leave an unsimplified expression for your answer).

Every indirect block can hold pointers to 1024/sizeof(uint32_t)=1024/4=256 blocks
So, maximum file = 1024 bytes/block × [12 + 256 + $(256)^2$+$(256)^3$]blocks

Problem 4b[4pts]: Utilizing the inode structure from Problem 4a, what is the *minimum* and *maximum* number of disk accesses required to read 512 bytes from a file? Assume that the buffer cache has not been implemented, but that the contents of an inode and various types of indirect blocks can be reused during the processing of a single request. Explain your answer. *(Hint: assume that you are starting from anywhere within a very large file, so that all indirect pointers are in use).*

Minimum: 512 bytes fully within one direct bloc k= 1 inode + 1block = 2 disk accesses
Maximum: 512 bytes contiguous in file but straddle 2 blocks: Two options (both valid):

1) *blocks are at end of double-indirect and beginning of triple indirect:*
   1 inode +1 double-indirect + 1 indirect + 1 data block +
   1 triple-indirect + 1 double-indirect + 1 indirect + 1 data block = 8 disk accesses
2) *blocks within triple-indirected-block. First block is last block listed in one indirect block and first block in another indirect block:*
   1 inode+1 triple-indirect+1 double-indirect+1 indirect+1 data block +
   1 double-indirect + 1 indirect + 1 data block = 8 disk accesses

Problem 4c[4pts]: Suppose that you are building a file system on top of an SSD using the inode structure from Problem 4a. Assume that the SSD has the following parameters:

- Page size (i.e. minimum size of items transferred): 1024B/
- Controller time: 10 μsec
- Page read time: 40 μsec
- Page write time: 90 μsec

Further, assume that reads or writes to the SSD are *not pipelined*, which means that every access incurs latency through the controller. Assuming there is no buffer cache, how much time is required to create a new file that is 256 KiB in size? Assume that this is the second file created in the root directory. Show your work and give the answer in milliseconds (ms):

Read Time =T_{ctrl}+T_{read}=50 μsec=0.05ms/page; Write time = T_{ctrl}+T_{write}=100 μsec=0.1ms/page
256KiB file ⇒ 256 blocks. Initial File Write = 0.1ms/block×(1 inode + 1 indirect + 256 blocks)=25.8ms
To update root directory, we need to read directory inode+first block, then rewrite first block:
Directory update = 0.05ms (1 inode + 1block) + 0.1ms (1 block) = .2ms
Total time for operation: 25.8ms + 0.2ms = 26ms

Problem 4d[4pts]: At a particular point in time, the buffer cache has dirty data that needs to be flushed to disk. Suppose that the identities of these blocks can be listed in [track:sector] form as follows:

[10:5], [22:9], [11:6], [2:10], [20:5], [32:4], [32:5], [6:7]

Assume that the disk head is currently positioned over track 20. What is the sequence of writes under the following disk scheduling algorithms:

a) Shortest Seek Time First:

[20:5], [22:9], [32:4], [32:5], [11:6], [10:5], [6:7], [2:10]
In breaking the tie between [32:4] and [32:5], we picked sector order, so that we can pull them consecutively from disk without extra rotations.

b) Scan (initially moving upwards):

[20:5], [22:9], [32:4], [32:5], [11:6], [10:5], [6:7], [2:10]

In this particular case, notice that both SSTF and Scan yield the same result.

Problem 4e[3pts]: Rather than writing updated files to disk immediately when they are closed, many UNIX systems use a delayed *write-behind policy* in which dirty disk blocks are flushed to disk once every 30 seconds. List two advantages and one disadvantage of such a scheme (one sentence each):

Advantage 1: *You can perform multiple (small) writes to a block without lots of disk traffic.*
Advantage 2: *You can create and delete small files without ever needing to flush to disk.*
There are other advantages which we will be willing to take. For instance: By having multiple outstanding dirty blocks, you can reorder them via an elevator algorithm for better performance. Some students contrasted the guaranteed flush every 30 seconds with waiting to flush until close (on files opened for a really long time); we accepted this as an advantage.

Disadvantage: *The principle disadvantage is durability, i.e. the possible loss of data if the system crashes before data is flushed to disk.*

Problem 4f[2pts]: What is a "Journaled" filesystem and how does it improve the durability of data on a disk relative to a system such as (4a). Give your answer in two sentences or less.

A journaled file system is one in which file system modifications are first written to a log as transactions before they are transferred to the primary file system. This methodology improves durability of the file system since writes must first be committed to the log before the writer is told that they are committed, making sure that they are on a persistent media. We also took answers that addressed "consistency" or "integrity" instead of "durability", if it was said that the atomicity of transactions in the log made sure that the file system was always in a consistent state – even in the presence of unexpected crashes.

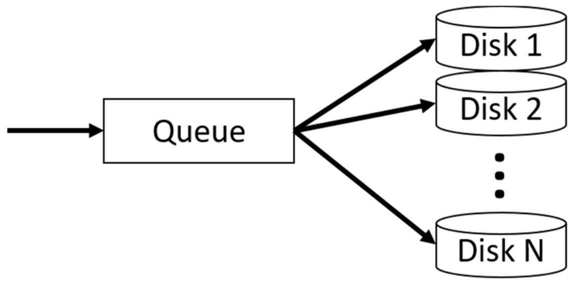
Problem 5: Parallel Disk Systems [17pts]

NOTE: IN THE FOLLOWING PROBLEM, ALL OF THE NUMBERS HAVE BEEN CHOSEN SO THAT ANSWERS SIMPLIFY EASILY WITHOUT THE NEED FOR A CALCULATOR.

Suppose that we build a disk array to handle a high rate of I/O by coupling many disks together. Properties of this system are as follows:

- Has a total of 16 disks, each of which is 2TiB in size
- Uses disks that rotate at 15,000 RPM, have a data transfer rate of 32768 KBytes/s (for each disk), and have an 3.5 ms average seek time, 4096 Byte sector size.
- Has a SCSI interface with a 500µs controller command time. Assume that a group of consecutive sectors can be fetched with a single request.
- Is limited only by the disks (assume that no other factors affect performance).

Each disk can handle only one request at a time, but each disk in the system can be handling a different request. The data is not striped (all I/O for each request has to go to one disk).



EACH OF THE FOLLOWING ANSWERS SHOULD BE SIMPLIFIED TO SINGLE NUMBERS. HOWEVER, YOU MUST SHOW YOUR WORK TO GET CREDIT.

Problem 5a[6pts]: What is the average *service time* to retrieve a *single* sector from a random location on a *single* disk, assuming no queuing time (i.e. the unloaded request time)? *Hint: there are four terms in this service time!*

$$T_{ser} = T_{ctrl} + T_{seek} + T_{rot} + T_{xfer} =$$
$$= 0.5ms + 3.5ms + ½ \frac{60{,}000\ ms/min}{15{,}000\ rev/min} + \frac{4096\ bytes}{32768\times10^3 bytes/s\times10^{-3} s/ms}$$
$$= 0.5ms + 3.5ms + 2ms + .125ms = 6.125ms$$

Problem 5b[3pts]: Assume that we plan to build a file system on top of this disk which achieves considerably more locality than spreading sectors randomly across the disk. How many sectors would it need to read in a row in order to achieve an effective bandwidth of 25% of the transfer rate from the disk? Express your answer in number of sequential sectors it would need to read at a time. *Hint: Express your latency from 5a in a form that is a fixed overhead + a component that is linearly related number of sequential sectors to fetch at a time. Solve in a way that the linear component is 25% of the total amount. Show your work!*

Let N be #sectors so that transfer time is 25% of total time:

$$(N \times T_{xfer}) = 25\%\ T_{ser} = .25\ [(T_{ctrl} + T_{seek} + T_{rot}) + (N \times T_{xfer})] \Rightarrow$$
$$(N \times T_{xfer}) = \frac{1}{3}(T_{ctrl} + T_{seek} + T_{rot}) \Rightarrow$$
$$N = \frac{1}{3}(0.5ms + 3.5ms + 2ms)/0.125ms = 2ms/0.125ms = 16\ sectors$$

Problem 5c[2pts]: Let's use your answer from Problem 5b as our file-system block size. What is the average *service time* to retrieve a *block* from a random location on a *single* disk? Note that the answer to this problem is actually part of the work to solve Problem 5b.

$$T_{ser} = [(T_{ctrl} + T_{seek} + T_{rot}) + (N \times T_{xfer})] = 0.5ms + 3.5ms + 2ms + 16 \times 0.125ms = 8ms$$

Problem 5d[2pts]: Let's use your answer from Problem 5c. Now, assume that we build a file system on the disk array that can have block-sized requests outstanding for all disks. Allowing each disk request to be for a random block on disk, what is the maximum number of I/Os per second (IOPS) for the whole disk subsystem (an "I/O" here is a block request)? We will give partial credit, so make sure to show your work.

1 Disk IOP takes 8ms = 0.008s $\Rightarrow$ (1/0.008) IOPS = 125 IOPS
16 Disks IOPS = 16 $\times$ 125 IOPS = 2000 IOPS

Problem 5e[4pts]: Treat the entire system as an M/M/m queue (that is, a system with one queue but m servers rather than one), where each disk is a server. For simplicity, assume that any disk can service any request. Assuming FIFO scheduling by the OS again, what is the maximum request rate (λ) that the system can handle while increasing the total time to service a block request so that the queue adds no more than 25% to the overall service time from Problem 5c. Show your work!

You might find the following equation for an M/M/m queue (where any server can handle any request from the queue) useful:

$$\text{Server Utilization}\,(\zeta) = \frac{\lambda}{\dfrac{1}{\text{Time}_{\text{server}}/m}} = \lambda \times \frac{\text{Time}_{\text{server}}}{m}$$

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \left[\frac{\zeta}{m(1-\zeta)}\right]$$

The total time to complete a request would be: $T_{system} = T_{server} + T_{queue}$.
So, to increase service time by no more than 25% would be to say that
$T_{queue} = 0.25 \times T_{server} \Rightarrow \left[\frac{\zeta}{m(1-\zeta)}\right] = 0.25 = \frac{1}{4}$
Here, m=16, so we know: $4\zeta = 16(1-\zeta) \Rightarrow \zeta = \frac{4}{5}$
From first equation: $\frac{4}{5} = \lambda \times \frac{T_{server}}{m} = \lambda \times \frac{8ms/request}{16} = \lambda \times 0.5ms \Rightarrow$
Finally, (converting to sec^{-1}): $\lambda = \frac{8}{5}\frac{requests}{ms} = 1.6\frac{requests}{ms} \times 10^3 \frac{ms}{s}$
Answer: 1600 req/sec

[Scratch Page (feel free to remove)]

[Scratch Page (feel free to remove)]