CS 162 Fall 2020 — Operating Systems and System Programming

Final Exam

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

○ You must choose either this option

○ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

☐ You could select this choice.

☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

This is a **proctored, closed-book exam**. During the exam, you may **not** communicate with other people regarding the exam questions or answers in any capacity. If there is something in a question that you believe is open to interpretation, please use the "Clarifications" button to request a clarification. We will issue an announcement if we believe your question merits one. We will overlook minor syntax errors in grading coding questions. You do not have to add necessary `#include` statements. For coding questions, the number of blank lines you see is a suggested guideline, but is not a strict minimum or maximum. There will be no limit on the length of your answer/solution.

a)

Name

b)

Student ID

c)

Please read the following honor code: "I understand that this is a closed book exam. I hereby promise that the answers that I give on the following exam are exclusively my own. I understand that I am allowed to use three 8.5x11, double-sided, handwritten cheat-sheet of my own making, but otherwise promise not to consult other people, physical resources (e.g. textbooks), or internet sources in constructing my answers." Type your full name below to acknowledge that you've read and agreed to this statement.

1. (20.0 points) True/False

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

a) 1

It's important in 2PC for workers to reply to the coordinator before logging the coordinator's message, for otherwise an untimely crash can cause the message to never be sent to the coordinator, blocking the entire system.

○ True

[✓ False]

2

Explain.

Must log before responding, otherwise may be unable to recover a situation like the following: (1) send response, (2) crash before logging, (3) response packet is dropped.

b) **1**

In 2PC, when the coordinator receives a request, the first thing it does is send a PREPARE message to workers.

○ True

[✓ False]

2

Explain.

The first thing it does is log the request.

c) **1**

In 2PC, after the coordinator sends a request to workers, if it has not received a reply after some time, it will try to resend the request.

○ True

[✓ False]

2

Explain.

It will GLOABL-ABORT.

d) **1**

AFS implements callbacks so that a write to a file from Node A instantly propagates to Node B, without Node B having to poll to update its cache.

○ True

[✓ False]

2

Explain.

Almost true, AFS does implement callbacks, but it has write through on close semantics, so all openers of a file are only updated of changes when they reopen.

e) **1**

The Berkeley FFS discussed in class does not include the file name in the inode.

[✓ True]

False

2

Explain.

File names are included in the directory entry instead.

f) **1**

The Berkeley FFS discussed in class includes the file name in the inode.

○ True

[✓ False]

2

Explain.

File names are included in the directory entry instead.

g) **1**

In the Berkeley FFS, each file descriptor uniquely corresponds to its own inode.

○ True

[✓ False]

2

Explain.

Different file descriptors may correspond to the same inode. (In the Unix system, file descriptors correspond to an entry in the global file table which contains info like permissions; that entry then identifies the inode.)

h) **1**

The FAT file system can suffer from external fragmentation if we delete a lot of small files.

○ True

[✓ False]

2

Explain.

We can still use those unallocated blocks for large files.

i) **1**

Soft links could lead to dangling references if a file is removed.

[✓ True]

False

2

Explain.

Soft links do not include a reference count, so this is true.

j) **1**

For a distributed hash table, as compared to recursive queries, iterative queries are more scalable (since clients do more work) and often faster (since the directory server is typically closer to storage nodes). However, it's often harder to enforce consistency for iterative queries.

○ True

[✓ False]

2

Explain.

False, the second reason (speed) is a con of iterative queries and a pro of recursive queries.

k) **1**

Consider System A that has one queue serviced at a rate of N jobs per second, and System B that has N queues, each of which is serviced at the rate of one job per second. True or False: System A will have lower queueing delays on average than System B.

[✓ True]

False

2

Explain.

One server that can process N jobs per second is faster. It has better utilization (no load balancing problems), which gives lower queuing delays on average.

l) **1**

Consider System A that has one queue serviced at a rate of N jobs per second, and System B that has N queues, each of which is serviced at the rate of one job per second. True or False: System A will have higher queueing delays on average than System B.

○ True

[✓ False]

2

Explain.

One server that can process N jobs per second is faster. It has better utilization (no load balancing problems), which gives lower queuing delays on average.

m) 1

As utilization approaches 100%, response time goes to infinity for any deterministic arrival process.

○ True

[✓ False]

2

Explain.

Given a deterministic arrival process and service time, it is possible to sustain a utilization of u = 1 with a bounded response time.
Consider the trivial scenario of: 1 job arrives every 5 seconds, and it takes exactly 5 seconds to service that job. Utilization is 1, but response time is boudned by 5 seconds.

n) **1**

As utilization approaches 100%, the response time of a real system will approach infinity.

○ True

[✓ False]

2

Explain.

As utilization approaches 100%, the finite queues will fill up, causing requests to be dropped rather than delayed indefinitely.

o) **1**

Since there is a disk head for every surface of every platter, it is possible to operate on each platter independently.

○ True

[✓ ● False]

2

Explain.

All of the disk heads are ridgedly tied together in a single head assembly that moves in and out as a unit.

p) **1**

The disk head assembly must move for each sector that is read from the disk.

○ True

[✓ False]

2

Explain.

Since the disk platters are spinning, the head can stay stationary while a whole cylinder-worth of sectors are read.

q) **1**

When devices use DMA, they typically work in terms of physical addresses.

[✓ True]

○ False

2

Explain.

The DMA controller is not associated with any particular processes, and thus does not need to translate virtual addresses. Instead, it is given direct access to the memory bus, so that it can copy data (using physical addresses) without tying up the CPU.

r) **1**

When devices use DMA, they typically work in terms of virtual addresses.

○ True

[✓ False]

2

Explain.

The DMA controller is not associated with any particular processes, and thus does not need to translate virtual addresses. Instead, it is given direct access to the memory bus, so that it can copy data (using physical addresses) without tying up the CPU.

s) **1**

On a low-bandwidth network interface, it makes sense to poll for the first packet and use interrupts on subsequent packets

○ True

[✓ False]

2

Explain.

On a low-bandwidth network, polling will waste many CPU cycles, and we should use interrupts for all packets. Remember: interrupts are useful for low-frequency events (as we have here) and polling is good for high-frequency events.
Note: it's not valid to say that we should use interrupts for the first packet and polling for the rest, because the arrival of the first packet doesn't imply that more packets will arrive soon on a low-bandwidth network.

t) **1**

On a high-bandwidth network interface, it makes sense to poll for the first packet and use interrupts on subsequent packets

True [✓ False]

2

Explain.

On a high-bandwidth network, interrupts will have too high an overhead. We should use polling for all packets. Remember: interrupts are useful for low-frequency events (as we have here) and polling is good for high-frequency events. Note: we can alterntaively use a hybrid approach where interrupts are used for the first packet (to get started) and polling is used for all subsequent packets.

u) **1**

Programmed IO requires special load and store instructions in the ISA that do not use the data cache so that updates are made directly to main memory and thus the device.

○ True

[✓ False]

2

Explain.

When using programmed IO, we want to ensure that our loads/stores are not cached in the data cache. If they were, then they would not make immediate changes to RAM, which are required to effect changes on the device. However, we don't need special load/store instructions in order to avoid caching the data; most modern implementations have a bit in the page-table entry that tells the MMU not to cache the data.

v) 1

On modern SSDs, large contiguous reads are much faster than large random reads

○ True

[✓ False]

2

Explain.

SSDs don't have moving parts so sequential reads/writes take the amount of time as random reads/writes

w) **1**

On modern SSDs, large contiguous writes are much faster than large random writes

○ True

[✓ False]

2

Explain.

SSDs don't have moving parts so sequential reads/writes take the amount of time as random reads/writes

x) **1**

Modern operating systems must include an elevator scheduling algorithm within a device driver in order to get the benefits of track-local request scheduling.

○ True

[✓ ● False]

2

Explain.

Modern disk controllers can handle multiple requests at the same time and incorporate optimizations such as elevator scheduling inside the controller.

y) **1**

It is possible to design a disk storage system that can recover from more than two simultaneously failing disks without losing information.

[✓ True]

○ False

2

Explain.

Using an erasure code such as Reed-Solomon coding allows one to add any number of parity disks to a system, over and above the two that are tolerated with RAID 6.

z) **1**

It is impossible to design a disk storage system that can recover from more than two simultaneously failing disks without losing information.

○ True

[✓ False]

2

Explain.

Using an erasure code such as Reed-Solomon coding allows one to add any number of parity disks to a system, over and above the two that are tolerated with RAID 6.

aa) **1**

It is possible for some SSD manufacturers to guarantee that their devices will not fail for some amount of time (say 5 years), even if a user writes continuously to the device.

[✓ True]

○ False

2

Explain.

Given the presence of wear-leveling functionality, some SSD devices are so large that a user would be unable to write enough data over 5 years to wear out any particular FLASH bit. (Note: this is not the same as being unable to write enough data to fill the SSD).

ab) 1

Suppose Host A is trying to transfer a 1000 byte file over a TCP connection to Host B and the Initial Sequence Numbers (ISNs) in both directions are 0. Suppose 5 packets, each of size 200 bytes, are sent over the connection from Host A to Host B. If Host B sends back a packet with the ACK field set to 600, then only the first three packets have made it through to Host B.

○ True

[✓ False]

2

Explain.

TCP uses cumulative ACKs, where each ACK says it's received all bytes up to the sequence number X - 1. Packet #5 could have been received even thought #4 has not.

ac) **1**

Suppose Host A is trying to transfer a 1000 byte file over a TCP connection to Host B and the Initial Sequence Numbers (ISNs) in both directions are 0. Suppose 5 packets, each of size 200 bytes, are sent over the connection from Host A to Host B. If Host B sends back a packet with the ACK field set to 400, then only the first two packets have made it through to Host B.

○ True

[✓ False]

2

Explain.

TCP uses cumulative ACKs, where each ACK says it's received all bytes up to the sequence number X - 1. Packet #4 or #5 could have been received even thought #3 has not.

ad) 1

TCP and UDP are both reliable transport-level protocols, but UDP is faster because we don't care about the order of the packets.

○ True

[✓ False]

2

Explain.

UDP is not reliable.

ae) **1**

Because UDP provides unreliable, best-effort delivery, using UDP is no different than using native IP.

True [✓ False]

2

Explain.

UDP is on the transport layer and allows us to connect processes to other processes, whereas IP does not give that functionality.

af) **1**

The journal log may be located on a different disk than the contents of the filesystem it backs.

[✓ True]

False

2

Explain.

There is no requirement for the journal to be on the same disk as the file system it backs. The journal must simply be append-only, non-volatile memory. Journaling file systems often use a faster, smaller disk (e.g. an SSD) for log writes.
Common Mistake: the intent of journaling file systems is not to recover from a *disk* crash, but instead to recover from a system crash (e.g. the CPU).

ag) **1**

The journal log must be located on the same disk as the contents of the filesystem it backs.

True [✓ False]

2

Explain.

There is no requirement for the journal to be on the same disk as the file system it backs. The journal must simply be append-only, non-volatile memory. Journaling file systems often use a faster, smaller disk (e.g. an SSD) for log writes.
Common Mistake: the intent of journaling file systems is not to recover from a *disk* crash, but instead to recover from a system crash (e.g. the CPU).

ah) 1

Because of the changes that are made to the buffer cache in Pintos Project 3, writes to a file are transactional. That is, if Thread A and Thread B attempt to write to a file at the same time, Thread C reading from the file after both writes complete should see either the contents of Thread A's write followed by the contents of Thread B's write OR the contents of Thread B's write followed by the content of Thread A's write. In particular, C cannot see a mix of writes from Thread A and Thread B.

○ True

[✓ False]

2

Explain.

This is true at a per-block level in the project, but nothing in the spec/design guarantees this sort of isolation for multi-block writes.

2. (16.0 points) Multiple Choice

a) (2.0 pt)

Which of the following are *true* about Two-Phase Commit?

- [✓] TPC makes sure that all nodes either commit or abort despite possibility of nodes crashing.
- ☐ TPC can still function correctly if it does not have a logging mechanism.
- ☐ TPC is a distributed consensus algorithm that allows for progress despite the presence of malicious nodes (i.e TPC can solve the Byzantine Generals Problem).
- ☐ Two-Phase Commit that can make progress despite indefinitely crashed or blocked nodes.

A logging mechanism is fundamental for TPC because that is how the process will come to a unanimous decision despite failures. TPC can handle failures, but it cannot handle nodes that are actively trying to compromise the decision making process (malicious nodes). Two-Phase Commit cannot continue to make progress in the presence of crashed and/or blocked nodes.

b) (2.0 pt)

Which of the following are *false* about Two-Phase Commit?

- ☐ TPC makes sure that all nodes either commit or abort despite possibility of nodes crashing.
- [✓] TPC can still function correctly if it does not have a logging mechanism.
- [✓] TPC is a distributed consensus algorithm that allows for progress despite the presence of malicious nodes (i.e TPC can solve the Byzantine Generals Problem).
- [✓] Two-Phase Commit that can make progress despite indefinitely crashed or blocked nodes.

A logging mechanism is fundamental for TPC because that is how the process will come to a unanimous decision despite failures. TPC can handle failures, but it cannot handle nodes that are actively trying to compromise the decision making process (malicious nodes). Two-Phase Commit cannot continue to make progress in the presence of crashed and/or blocked nodes.

c) (2.0 pt)

For Two-Phase Commit, given that all follower nodes will vote to COMMIT and all crashed nodes will eventually recover, in which cases *could* there be a GLOBAL-COMMIT?

- [✓] After receiving a COMMIT vote from each follower, the coordinator sends a GLOBAL-COMMIT, and crashes.
- [✓] The coordinator sends a VOTE-REQ to all followers. All (N - 1) followers log a VOTE-REQ. (N - 1) followers vote to commit, but 1 follower crashes before sending a COMMIT.
- [✓] The coordinator crashes before sending a VOTE-REQ to all followers.
- ☐ The coordinator crashes right after sending a VOTE-REQ to all followers.
- ☐ All of the above.

Option 1: In recovery mode, the coordinator looks into the log and sees that it sent out a GLOBAL-COMMIT. It sends out another GLOBAL-COMMIT to all of its followers, and this can keep occurring until the coordinator receives all N ACKS. Option 2: Suppose the coordinator allotts X seconds for the followers to send a COMMIT or ABORT in response to their VOTE-REQ. If a follower crashes but recovers and sends a response within X seconds, it will be as if the follower didn't crash. Option 3: On reboot, because the log doesn't have a VOTE-REQ, the coordinator will resend a VOTE-REQ to all followers. Option 4: Because the coordinator hasn't logged a global action after a VOTE-REQ, on reboot, the coordinator will send out a GLOBAL-ABORT.

d) (2.0 pt)

For Two-Phase Commit, given that all follower nodes will vote to COMMIT and all crashed nodes will eventually recover, in which cases is a GLOBAL-COMMIT *guaranteed*?

- 🟥 After receiving a COMMIT vote from each follower, the coordinator sends a GLOBAL-COMMIT, and crashes.
- ☐ The coordinator sends a VOTE-REQ to all followers. All (N - 1) followers log a VOTE-REQ. (N - 1) followers vote to commit, but 1 follower crashes before sending a COMMIT.
- ☐ The coordinator crashes before sending a VOTE-REQ to all followers.
- ☐ The coordinator crashes right after sending a VOTE-REQ to all followers.
- ☐ All of the above.

Option 1: In recovery mode, the coordinator looks into the log and sees that it sent out a GLOBAL-COMMIT. It sends out another GLOBAL-COMMIT to all of its followers, and this can keep occurring until the coordinator receives all N ACKS. Option 2: If a follower crashes and doesn't recover within the timeout the coordinator specified, the coordinator will send out a GLOBAL-ABORT. Option 3: On reboot, because the log doesn't have a VOTE-REQ, the coordinator will resend a VOTE-REQ to all followers. However, since a follower can timeout, a GLOBAL-ABORT can occur. Option 4: Because the coordinator hasn't logged a global action after a VOTE-REQ, on reboot, the coordinator will send out a GLOBAL-ABORT.

e) (2.0 pt)

Which of the following statements about AFS are correct?

- ☐ If multiple clients write to the same file concurrently, parts of both writes may be persisted.
- ☐ The client contacts the server to check for changes.
- 🟥 AFS has a lower server load than NFS.
- 🟥 AFS is not stateless.

Since writes occur on `close()` and are only visible after `close()`, multiple clients' writes to the same file would result in only one of the writes being persisted. The server contacts clients to notify them of changes when a client calls `close()` on a file. Clients cache files locally, so AFS has a lower server load than NFS. AFS is not stateless, because the server keeps track of the files that each client has open in order to notify them of changes.

f) (2.0 pt)

Which of the following statements about AFS are correct?

- 🟥 If multiple clients write to the same file concurrently, only one of the writes will be persisted.
- 🟥 The server contacts clients to notify them of changes.
- ☐ AFS has a higher server load than NFS.
- ☐ AFS is stateless.

Since writes occur on `close()` and are only visible after `close()`, multiple clients' writes to the same file would result in only one of the writes being persisted. The server contacts clients to notify them of changes when a client calls `close()` on a file. Clients cache files locally, so AFS has a lower server load than NFS. AFS is not stateless, because the server keeps track of the files that each client has open in order to notify them of changes.

g) (2.0 pt)

Which of the following statements about NFS are correct?

- 🟥 NFS utilizes write-through caching.
- 🟥 The client contacts the server to check for changes.
- ☐ If multiple clients write to the same file concurrently, only one of the writes will be persisted.
- ☐ NFS is stateless, so a networked `open()` call returns a file descriptor that any client can use.

NFS utilizes write-through caching, meaning that writes may take a while. The client periodically polls the server to check for changes. If multiple clients write to the same file, parts of both writes may be persisted, due to the NFS protocol's weak consistency. NFS is stateless, which means it does not require a networked `open()` call: clients send `inumbers` with requests (keeping track of open file descriptors would require state).

h) (2.0 pt)

Which of the following statements about NFS are correct?

- ☐ NFS utilizes write-back caching.
- ☐ The server contacts clients to notify them of changes.
- 🟥 If multiple clients write to the same file concurrently, parts of both writes may be persisted.
- 🟥 NFS is stateless, so it does not require a networked `open()` call.

NFS utilizes write-through caching, meaning that writes may take a while. The client periodically polls the server to check for changes. If multiple clients write to the same file, parts of both writes may be persisted, due to the NFS protocol's weak consistency. NFS is stateless, which means it does not require a networked `open()` call: clients send `inumbers` with requests (keeping track of open file descriptors would require state).

i) (2.0 pt)

Which of the following are *true*?

- ☐ FFS, FAT and NTFS, can all support hard links.
- 🟥 The FFS is generally efficient for small files.
- 🟥 To avoid a situation in which the file system must wait one complete rotation for each block it reads, the FFS can take advantage of RAM on disk controllers.
- ☐ The FFS takes advantage of temporal locality by placing a directory and its files near each other.

Option 1: FAT cannot support hard links. Option 2: For most small files, FFS can fit all blocks for the same file in one cylinder, so there aren't any seeks. Option 3: Subsequent reads can be pulled off the track buffer without worrying about physical rotation. Option 4: The FFS takes advantage of spatial locality by placing a directory and its files near each other.

j) (2.0 pt)

Which of the following are *true*?

☐ Berkeley FFS uses a linked-list file allocation strategy because indirect pointers point to direct pointers, which point to data blocks.

☐ A file in a FAT file system will always take longer to read than in a FFS file system.

☐ In the FAT file system, file attributes, such as file permissions, are stored in the directory rather than the file.

🟥 Appending a block to a large file using FAT is slower than if you were to use the FFS.

Option 1: The Berkeley FFS uses an indexed file allocation strategy. Option 2: If the file consists of only one disk block, then traversal of pointers in FAT is not necessary. Thus, the file will be read at the same speed in both file systems. Option 3: A significant security threat FAT faces stems from not supporting file permissions. Option 4: FAT needs to follow pointers block by block to find the end of the file, while the FFS can find the end of the file much more quickly by looking at the inode.

k) (2.0 pt)

Which of the following are *true* about Quorum Consensus?

☐ Quorum Consensus will result in consistent data for get() requests despite node failures / crashes.

☐ Having W = 1 will always perform just as well as W > 1.

☐ In a system with no failures or crashes, there would be no need for Quorum Consensus.

☐ Having W = R is always the most optimal strategy.

🟥 None of the above.

Option 1: It is possible that the X nodes that sent X ACKS for the put() request are also the X nodes that fail when responding to a get() request. Option 2: A write might happen at a single replica that then fails. If that replica were to lose its state, the outcome of the write would be lost. Option 3: Quorum Consensus overall goal is to speed up performance for put() and get() requests and to combat failure. Option 4: In the case where reads are more common than writes, it would be better if R < W because this improves performance for reads. It is similar for when writes are more common than reads: W should be less than R.

l) (2.0 pt)

Which of the following are *false* about Quorum Consensus?

☐ Quorum Consensus will result in consistent data for get() requests despite node failures / crashes.

☐ Having W = 1 will always perform just as well as W > 1.

☐ In a system with no failures or crashes, there would be no need for Quorum Consensus.

☐ Having W = R is always the most optimal strategy.

🟥 All of the above.

Option 1: It is possible that the X nodes that sent X ACKS for the put() request are also the X nodes that fail when responding to a get() request. Option 2: A write might happen at a single replica that then fails. If that replica were to lose its state, the outcome of the write would be lost. Option 3: Quorum Consensus overall goal is to speed up performance for put() and get() requests and to combat failure. Option 4: In the case where reads are more common than writes, it would be better if R < W because this improves performance for reads. It is similar for when writes are more common than reads: W should be less than R.

m) (2.0 pt)

Which of the following are features implemented in Unix FFS to help optimize performance on traditional HDDs?

- 🟥 Attempting to keep files that are in the same directory in the same block group.
- ☐ Always keeping file contents contiguous.
- ☐ Utilizing an indexed file allocation strategy.
- ☐ Offsetting sector numbers from one track to the next.

FFS tries to keep file contents contiguous via first-fit block allocation, and attempts to keep a directory's files in the same block group. The indexed file allocation strategy does not optimize for performance, but optimizes disk usage. Track skewing is a feature of HDD controllers, not of FFS.

n) (2.0 pt)

Which of the following are features implemented in Unix FFS to help optimize performance on traditional HDDs?

- ☐ Always keeping files that are in the same directory in the same block group.
- 🟥 Attempting to keep file contents contiguous.
- ☐ Utilizing an indexed file allocation strategy.
- ☐ Offsetting sector numbers from one track to the next.

FFS tries to keep file contents contiguous via first-fit block allocation, and attempts to keep a directory's files in the same block group. The indexed file allocation strategy does not optimize for performance, but optimizes disk usage. Track skewing is a feature of HDD controllers, not of FFS.

o) (2.0 pt)

Which of the following are **TRUE** about SSDs?

- ☐ The lack of moving parts eliminates controller latency.
- 🟥 Sequential reads and random reads both maximize bandwidth.
- ☐ The read/write interface allows interacting with individual bytes, in contrast to the chunk-based interface of traditional HDDs.
- 🟥 Erasures are provided as a separate interface.

The lack of moving parts eliminates seek and rotational latency, but SSDs still have controller latency. Without seek & rotational latency, both sequential access and random access to have the maximal bandwidth. SSDs provide the same chunk-based interface of traditional HDDs, and also provide a separate erasure interface, generally about 10x the size of writes.

p) (2.0 pt)

Which of the following are **FALSE** about SSDs?

- 🟥 The lack of moving parts eliminates controller latency.
- ☐ Sequential reads and random reads both maximize bandwidth.
- 🟥 The read/write interface allows interacting with individual bytes, in contrast to the chunk-based interface of traditional HDDs.
- ☐ Erasures are provided as a separate interface.

The lack of moving parts eliminates seek and rotational latency, but SSDs still have controller latency. Without seek & rotational latency, both sequential access and random access to have the maximal bandwidth. SSDs provide the same chunk-based interface of traditional HDDs, and also provide a separate erasure interface, generally about 10x the size of writes.

q) (2.0 pt)

When writing applications that communicate over the internet, the network layers offer us many abstractions. Which of the following physical limitations does the Internet Protocol (IP) provide abstractions to overcome?

- 🟥 Intermediate hops on an end-to-end communication have a variety of values for MTU (Maximum Transmission Unit), while we would like to send IP datagrams without considering the path they will take.
- ☐ Physical connections have a limited bandwidth, while we want arbitrary concurrent messages.
- ☐ Packets may be dropped, while we want reliable messages.
- ☐ Data is routed from machine to machine, while we want messages to be delivered from process to process.

IP splits data into packets, allowing us to send arbitrary size messages. The upper layer protocols are responsible for resolving reliability issues, including network bandwidth and routing from process to process.

r) (2.0 pt)

Which of the following are issues in RPC that proper marshalling **would** solve?

- 🟥 Client and server machines use different architectures.
- 🟥 Data transmitted includes pointers in a machine's address space.
- ☐ Non-atomic failure results in inconsistency.
- ☐ A single server has multiple clients, and a single client has multiple servers.

Marshalling can address different architectures by converting data to a canonical form. Marshalling can also copy arguments that are passed by reference to address pointers. Non-atomic failures must be addressed for the entire system, i.e. with distributed transactions. Multiple clients/servers are addressed at binding time, which should be independent of marshalling.

s) (2.0 pt)

Which of the following are issues in RPC that proper marshalling **would NOT** solve?

☐ Client and server machines use different architectures.

☐ Data transmitted includes pointers in a machine's address space.

🟥 Non-atomic failure results in inconsistency.

🟥 A single server has multiple clients, and a single client has multiple servers.

Marshalling can address different architectures by converting data to a canonical form. Marshalling can also copy arguments that are passed by reference to address pointers. Non-atomic failures must be addressed for the entire system, i.e. with distributed transactions. Multiple clients/servers are addressed at binding time, which should be independent of marshalling.

t) (2.0 pt)

Which of the following statements describing features of the Transmission Control Protocol (TCP) are **TRUE**?

🟥 The receiver detects bad packets via a checksum.

🟥 The sender automatically retransmits packets after a timeout if no ACK is received.

☐ The receiver must send an ACK for every packet received.

☐ The window size should be increased when the sender detects more dropped packets.

TCP headers contain a checksum for the receiver to detect bad packets. TCP transmitters should retransmit packets after a timeout. TCP has window-based acknowledgements. Dropped packets indicate that the window size should be decreased.

u) (2.0 pt)

Which of the following statements describing features of the Transmission Control Protocol (TCP) are **FALSE**?

☐ The receiver detects bad packets via a checksum.

☐ The sender automatically retransmits packets after a timeout if no ACK is received.

🟥 The receiver must send an ACK for every packet received.

🟥 The window size should be increased when the sender detects more dropped packets.

TCP headers contain a checksum for the receiver to detect bad packets. TCP transmitters should retransmit packets after a timeout. TCP has window-based acknowledgements. Dropped packets indicate that the window size should be decreased.

v) (2.0 pt)

Which of the following are *true* about the ACID properties of transactions?

- 🟥 Atomicity guarantees that all updates in a transaction happen, or none happen.
- ☐ Consistency refers to allowing the database to go from consistent state to inconsistent state, and then to another consistent state due to having techniques / algorithms to deal with the inconsistent state.
- ☐ If a distributed file system is isolated, concurrent transactions can affect each other.
- 🟥 Durability refers to persistence of committed transactions, even when the system crashes.

Consistency ensures that transactions will only let the database go from consistent state to consistent state–data written must be valid. Isolation ensures that concurrent execution of transactions do not affect one another.

w) (2.0 pt)

Which of the following are *false* about the ACID properties of transactions?

- ☐ Atomicity guarantees that all updates in a transaction happen, or none happen.
- 🟥 Consistency refers to allowing the database to go from consistent state to inconsistent state, and then to another consistent state due to having techniques / algorithms to deal with the inconsistent state.
- 🟥 If a distributed file system is isolated, concurrent transactions can affect each other.
- ☐ Durability refers to persistence of committed transactions, even when the system crashes.

Consistency ensures that transactions will only let the database go from consistent state to consistent state–data written must be valid. Isolation ensures that concurrent execution of transactions do not affect one another.

3. (28.0 points) Short Answer

a) Bottleneck in Distributed File Systems

1

What's the central bottleneck in both NFS and AFS? Provide at least one reason to back your choice and explain why AFS has less of this bottleneck than NFS.

The central file server.
Reasons: – Performance: all writes go to server, cache misses go to server; with AFS, writes are sent in bulk, and most reads after an initial "“cache miss”" go to the local disk. – Availability: Server is single point of failure; with AFS, files are cached on the local disk, so may allow activity to continue during short periods when server is down. – Cost: server machine's high cost relative to workstation; AFS is more efficient per client, so can operate with either more clients or a less powerful server.

b) End to End Principle

1

Does the following scenario follow the end-to-end principle: When downloading files across the internet, the end hosts use checksums to validate whether or not the file got downloaded correctly? Explain.

Yes, since the endhosts are the one validating the data, not the intermediate layers (servers).

c) Index vs Linked File Systems

1

Suppose that you start with an indexed file system (like FFS) and you swapped to a linked file system (like FAT). If you noticed a drastic decrease in performance, what could cause this?

The access pattern of the average workload on the machine involves random access to data.

d) AFS

1

Why is there no polling needed from the client in AFS?

The server will notify all clients when a file has been changed.

e) AFS

1

What is an advantage of having write on close in AFS?

There will never be partial updates, it will always be all or nothing (i.e only other clients see changes if the server see changes).

f) RAID

1

Why is NFS's cache consistency considered "“weak consistency”"?

When changes are committed on the file server, there may be a period of time where a client will read invalid data from their cache (until they pull the latest data from the system).

g) RAID

1

Why might RAID 5 no longer be sufficient (in terms of redundancy) for today's users?

Only one disk can fail in RAID 5 and since the time to repair the disk takes a long time, it's possible another disk may fail.

h) RAID

1

Why might RAID 1 be considered "“faster”" than RAID 5 for live data (such as in a data base)? Give two reasons.

1 Writes in a RAID 1 system can occur by directly writing to each of the mirrorred disks rather than performing a read-modify write on the parity disk as required by RAID 5. (2) Reads can occur to either of the mirrored drives, thus giving twice the read bandwidth; in the case of RAID 5, data is not directly replicated as it is in RAID 1.

i) SSD

1

Explain at least two reasons that SSDs must have a Flash Translation Layer (FTL).

1 Physical pages in an SSD cannot be rewritten, but must be erased as part of a large block before being reused. So, the FTL permits remapping of a logical page to a different physical page when it is rewritten. (2) The Flash Translation Layer is a fundamental component of wear-leveling.

j) SSD

1

What is Wear-Leveling and why is it necessary?

Wear-Leveling is a process used in an SSD to even out the total number of times each physical page of an SSD is written. It is necessary because each bit of FLASH can only be written a finite number of times before it begins to fail.

k) SSD

1

What on-device filesystem structure is well-suited for an SSD and why?

A Log-Structured Filesystem works well with an SSD. Since pages in an SSD cannot be overwritten (but have to be collected into blocks and erased first), the log-structuring allows new blocks to be continuously written and later garbage-collected.

l) Journaling File System Crash

1

After a COMMIT was written for a transaction in the log, your computer crashes. Assuming your computer is using a journaling file system, what can we assume about the operations within the transaction? What should you do during recovery?

All, some, or none of the operations have been applied to the file system. You should replay the transaction.

m) Journaling File System Crash

1

Before a COMMIT was written for a transaction in the log, your computer crashes. Assuming your computer is using a journaling file system, what can we assume about the operations within the transaction? What should you do during recovery?

None of the operations have been applied to the file system. You should discard the transaction.

n) Two Phase Commit

1

Is 2PC subject to the General's Paradox? Why or why not?

No, because in 2PC we eventually reach the same decision (commit/abort) rather than in the General's Paradox where we must arrive at the same decision at the same time.

o) Virtual File System

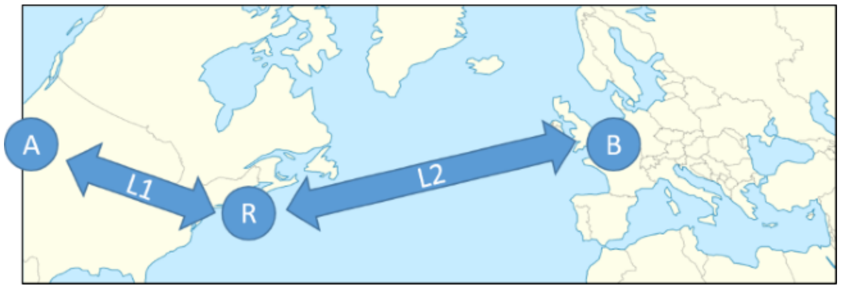
1

Name an advantage of having the Virtual File System in Linux.

Reasons: – Provides single syscall API for all file system commands for all file servers – Abstracts out file system API to make the development of distributed file systems easier

p) (6.0 points) TCP Window

Consider the following communication path from node A on the West Coast of the United States to node B in Europe traversing one cross-continent link L1, a router R, and an trans-atlantic link L2. The topology is as shown below:



Assume the following information:

- Information about L1:
- Bandwidth over link L1: 100Gb/s
- Latency over link L1: 100ms
- Maximum Transfer Unit (MTU) for L1: 1500 bytes
- Information about L2:
- Bandwidth over link L2: 640 Gb/s
- Latency over link L2 (fiber): 100ms
- Maximum Transfer Unit (MTU) for L2: 9000 bytes
- Additional Info:
- Router latency: 10ms
- Router bandwidth: line rate to all ports
- TCP/IP Header size: 40 bytes

In the following, make sure to show your work if there are any calculations. Please pay attention to units!

1

What is the maximum packet size that can be transported from A to B without requiring any fragmentation (splitting of packets) in the network?

Maximum packet = 1500 bytes

2

Assuming that the router can process packets a full line rate (i.e. as fast as the network requests of it), what is the maximum sustained bandwidth that two applications using TCP/IP could achieve between nodes A and B?

100 Gb/s * (1500-40)/1500 = 97.33 Gb/s

3

Assuming that A and B communicate via TCP/IP and that the network and router are fully pipelined (meaning that packets do not have to be fully received by the router before they can be forwarded along the next link), how many packets should be in the network to maximize the throughput of the link? Show your work.

RTT=2 * (100ms + 10ms + 100ms) = 420ms = .42 s * 100 Gb/s = 42 Gbits 1 max packet = 1500 * 8 = 12000 bits So, total packets = 42 * 10^9/12000 = 3.5 million packets

4. (12.0 points) Disk Design

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

• Uses 10GB disks that rotate at 15,000 RPM, have a data transfer rate of 8 MBytes/s (for each disk), and have a 9 ms average seek time, a 4 KiByte sector size, 64 KiByte block size.

• Has a SCSI interface with a 808μs controller command time.

• Is limited only by the disks (assume that no other factors affect performance).

• Has a total of 25 disks.

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). *Note: Sizes are in powers of 2, bandwidths are in powers of 10.*

a)

What is the average service time (ms) to retrieve a single disk block from a random location on a single disk, assuming no queuing time?

$$\frac{2^{16}}{(8*10^6*10^{-3})} + 0.808 + 8 + \frac{60000}{15000} * \frac{1}{2} = 20ms$$

b)

Assume that the OS is not particularly clever about disk scheduling and passes requests to the disk in the same order that it receives them from the application (FIFO). If the application requests are randomly distributed over a single disk, what is the bandwidth (bytes/sec) that can be achieved?

$$\frac{2^{16}}{(20*10^{-3})} = \frac{3276800bytes}{second}$$

c)

What is the average number of I/Os per second (IOPS) that the whole disk system can handle (assuming that I/O requests are a block size at a time, evenly distributed among the drives, and uncorrelated with one another)?

$$25 * \frac{1}{20ms * \frac{1000ms}{sec}} = 1,250IOPS$$

d)

Assume that there is a queue in front of the controller. Assume that the distribution of service times has C = 1.5 (i.e. not quite memoryless). Also assume that requests for blocks arrive via an exponential (memoryless) process with an average arrival rate of λ blocks/second. What is the arrival rate λ such that the queue has an average length of 10 blocks? You don't need to actually compute this number as long as you give an explicit equation for λ.

Hint 1: You should be able to reuse some result from a previous subpart.

Hint 2: use Little's law: $L_q = \lambda * T_q$ and $u = \lambda * T_{ser}$.

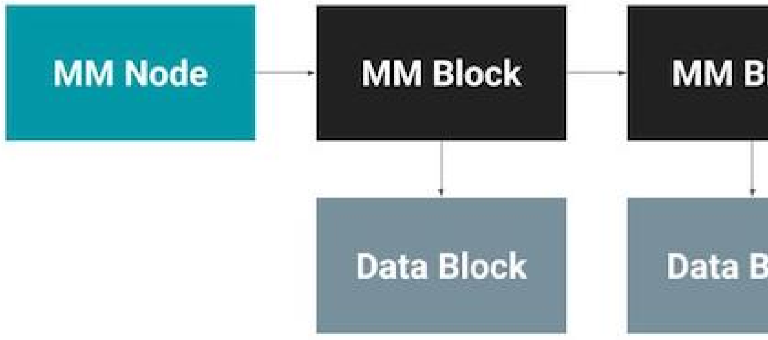
Hint 3: Use the quadratic formula,

$$Ax^2 + Bx + C = 0 \implies x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

$u = 2 * sqrt(6) - 4$ **(See Spring 2006 MT2).** $u = \lambda\ T_{ser} \implies \lambda = \frac{2*sqrt(6)-4}{20} * 1000$

5. (24.0 points) Monty Mole File System

CS162 course staff is developing a revolutionary new file system called MMFS (the Monty Mole File System) that is optimized for sequential reads. It is similar to the FAT filesystem in that it is a linked filesystem. However, it is also like the inode-based filesystem you are implementing for Project 3, in that it utilizes an MMnode to store file metadata.



On disk, MMFS represents a file in the following way:

Assume that the data structures below have been declared for you.

```
/* On-disk MM node.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct mm_node_disk
  {
    off_t length;                       /* File size in bytes. */
    block_sector_t head;                /* Pointer to first MM block */
    char unused[126];
  };

/* In-memory MM node. */
struct mm_node
  {
    struct list_elem elem;              /* Element in list. */
    block_sector_t sector;              /* Sector number of disk location. */
    int open_cnt;                       /* Number of openers. */
    struct mm_node_disk mm_node_disk;
  };

/* On-disk MM block.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct mm_block_disk
  {
    block_sector_t data;                /* Pointer to corresponding data block */
    block_sector_t next_mm_block;       /* Pointer to next MM block */
    char unused[126];
  };
```

For this problem, assume that there is no buffer cache (you will be accessing the file system device directly using the `block_*` functions). In addition, assume all calls to `free_map_allocate` and `malloc` succeed. Finally, do not worry about synchronization for this problem, and assume no functions are called concurrently.

NOTE: Feel free to use as many lines of code as you need for each blank. The number of blank lines in our starter code should be used as a guide (i.e. that's how many lines our staff solution uses).

SUGGESTION: Have the PDF copy of your exam open to reference the starter code as you're generating and submitting answers on ExamTool.

You can use the following functions, in addition to any functions listed on the reference sheet:

```
/* Allocates CNT consecutive sectors from the free map and stores
   the first into *SECTORP. All allocated sectors are zero-filled.
   Assume free_map_allocate always succeeds. */
void free_map_allocate (size_t cnt, block_sector_t *sectorp);

/* Reads sector SECTOR from the file system device into BUFFER, which must
   have room for BLOCK_SECTOR_SIZE bytes. */
void block_read(block_sector_t sector, void *buffer);

/* Write sector SECTOR to the file system device from BUFFER, which must contain
   BLOCK_SECTOR_SIZE bytes. Returns after the file system device has
   acknowledged receiving the data. */
void block_write(block_sector_t sector, const void *buffer);

/* Reads LENGTH bytes of sector SECTOR, starting at byte OFFSET, from the file system device into BUFF
   which must have room for LENGTH bytes. */
void block_partial_read(block_sector_t sector, void *buffer, off_t offset, size_t length);

/* Write LENGTH bytes of sector SECTOR, starting at byte OFFSET, to the file system device from BUFFER
   which must contain BLOCK_SECTOR_SIZE bytes. Returns after the file system device has
   acknowledged receiving the data. */
void block_partial_write(block_sector_t sector, const void *buffer, off_t offset, size_t length);
```

a) First, we will implement `mm_create` (similar to `inode_create`), which will initialize an MMnode with the given length and write it to the specified sector on the file system device. You should allocate the minimum number of MM blocks and data blocks needed to accommodate the length of the file. Remember that you can assume `free_map_allocate` already zero fills allocated disk blocks for you. Also, you must set any unused or invalid `block_sector_t` attributes to 0.

```
/* Initializes an MMnode with LENGTH bytes of data and
   writes the new MMnode to sector SECTOR on the file system
   device. */
void
mm_create (block_sector_t sector, off_t length)
{
  ASSERT (length >= 0);

  struct mm_node_disk *mm_node_disk = malloc (sizeof(struct mm_node_disk));
  _____________________________[A]
  _____________________________[A]

  if (length > 0) {
    free_map_allocate(1, &mm_node_disk->head);
    block_sector_t next_sector = mm_node_disk->head;
```

```
      struct mm_block_disk *mm_block_disk = malloc (sizeof(struct mm_block_disk));
      off_t pos = 0;
      while (next_sector != 0) {
        pos += BLOCK_SECTOR_SIZE;
        if (pos < length) {
          _____________________________[B]
        } else {
          _____________________________[C]
        }

        _____________________________[D]
        _____________________________[D]
        _____________________________[D]
      }
      free (mm_block_disk);
    }

    _____________________________[E]
    free (mm_node_disk);
  }
```

1

[A]

```
mm_node_disk->length = length;
mm_node_disk->head = 0;
```

2

[B]

```
free_map_allocate (1, &mm_block_disk->next_mm_block);
```

3

[C]

```
mm_block_disk->next_mm_block = 0;
```

4

[D]

```
free_map_allocate (1, &mm_block_disk->data);
block_write(next_sector, mm_block_disk);
next_sector = mm_block_disk->next_mm_block;
```

5

[E]

```
block_write(sector, mm_node_disk);
```

b) Next, we will implement `mm_open` and `mm_close` (similar to `inode_open` and `inode_close`). `mm_open` reads an MMnode from the specified sector on the file system device and initializes/returns an in-memory `struct mm_node`. Like `inode_open`, `mm_open` should keep track of a list of open MMnodes, and reopen/return an MMnode that has already been opened previously. `mm_close` closes a previously opened MMnode, and frees the in-memory `struct mm_node` if there are no more references to it.

NOTE: You can assume that `mm_init` has already been called prior to `mm_open`. `mm_reopen` has been implemented for you, and can be used as part of your solution.

```
static struct list open_mm_nodes;

/* Initializes the MMnode module. */
void
mm_init (void)
{
  list_init (&open_mm_nodes);
}

/* Reopens and returns MM_NODE. */
struct mm_node *
mm_reopen (struct mm_node *mm_node)
{
  if (mm_node != NULL)
    {
      mm_node->open_cnt++;
    }
  return mm_node;
}

/* Reads an MMnode from SECTOR
   and returns a `struct mm_node' that contains it. */
struct mm_node *
mm_open (block_sector_t sector)
{
  struct list_elem *e;
  struct mm_node *mm_node;

  for (e = list_begin (&open_mm_nodes); e != list_end (&open_mm_nodes);
       e = list_next (e))
    {
      mm_node = list_entry (e, struct mm_node, elem);
      if (_____________________________[A])
        {
          _____________________________[B]
          _____________________________[B]
        }
    }

  mm_node = malloc (sizeof(struct mm_node));

  _____________________________[C]
  _____________________________[C]
  _____________________________[C]
  _____________________________[C]

  return mm_node;
}

/* Closes MM_NODE.
   If this was the last reference to MM_NODE, frees its memory. */
void
mm_close (struct mm_node *mm_node)
{
  if (mm_node == NULL)
    return;

  _____________________________[D]

  if (______________________________[E])
    {
      list_remove (&mm_node->elem);
      free (mm_node);
    }
}
```

1

[A]

```
mm_node->sector == sector
```

2

[B]

```
mm_reopen (mm_node);
return mm_node;
```

3

[C]

```
list_push_front (&open_mm_nodes, &mm_node->elem);
mm_node->sector = sector;
mm_node->open_cnt = 1;
block_read(sector, &mm_node->mm_node_disk);
```

4

[D]

```
int open = --mm_node->open_cnt;
```

5

[E]

```
open == 0
```

c) Finally, we will implement `mm_read_at` and `mm_length` (similar to `inode_read_at` and `inode_length`).

```
/* Returns the length, in bytes, of MM_NODE's data. */
off_t
mm_length (const struct mm_node *mm_node)
{
  _____________________________[A]
}

/* Reads SIZE bytes from MM_NODE into BUFFER, starting at position OFFSET.
   Returns the number of bytes actually read, which may be less
   than SIZE if an error occurs or end of file is reached. */
off_t
mm_read_at (struct mm_node *mm_node, void *buffer, off_t size, off_t offset)
{
  if (offset >= mm_length (mm_node))
    return 0;

  off_t bytes_read = 0;
  struct mm_node_disk *mm_node_disk = &mm_node->mm_node_disk;
  block_sector_t sector = mm_node_disk->head;

  struct mm_block_disk *mm_block_disk = malloc(sizeof(struct mm_block_disk));
  block_read(mm_node_disk->head, mm_block_disk);

  _____________________________[B]
  while (_____________________________[C]) {
    sector = mm_block_disk->next_mm_block;
    block_read(sector, mm_block_disk);

    _____________________________[D]
  }

  while (size > 0)
    {
      block_sector_t data_sector = mm_block_disk->data;
      int sector_ofs = offset % BLOCK_SECTOR_SIZE;

      off_t inode_left = mm_length (mm_node) - offset;
      int sector_left = BLOCK_SECTOR_SIZE - sector_ofs;
      int min_left = inode_left < sector_left ? inode_left : sector_left;

      int chunk_size = size < min_left ? size : min_left;

      if (chunk_size <= 0)
        break;

      if (sector_ofs == 0 && chunk_size == BLOCK_SECTOR_SIZE) {
        block_read (data_sector, buffer + bytes_read);
      }
      else {
        _____________________________[E]
      }


      _____________________________[F]
```

```
            ______________________________[F]

            size -= chunk_size;
            offset += chunk_size;
            bytes_read += chunk_size;
          }

        free (mm_block_disk);
        return bytes_read;
      }
```

1

[A]

```
return mm_node->mm_node_disk.length;
```

2

[B]

```
off_t bytes_scanned = BLOCK_SECTOR_SIZE;
```

3

[C]

```
bytes_scanned < offset
```

4

[D]

```
bytes_scanned += BLOCK_SECTOR_SIZE;
```

5

[E]

```
block_partial_read (data_sector, buffer + bytes_read, sector_ofs, chunk_size);
```

6

[F]

```
sector = mm_block_disk->next_mm_block;
block_read(sector, mm_block_disk);
```

6. Reference Sheet

```
/******************************* Threads *******************************/
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                      void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int sem_init(sem_t *sem, unsigned int value);
int sem_up(sem_t *sem);
int sem_down(sem_t *sem);

/****************************** Processes *******************************/
pid_t fork(void);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int execv(const char *path, char *const argv[]);

/*************************** High-Level I/O *****************************/
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fclose(FILE *stream);

/******************************** Sockets *******************************/
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, structure sockaddr *addr, socklen_t *addrlen);
int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);

/**************************** Low-Level I/O *****************************/
int open(const char *pathname, int flags);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);

/****************************** PintOS *********************************/
void list_init(struct list *list);
struct list_elem *list_head(struct list *list);
struct list_elem *list_tail(struct list *list);
struct list_elem *list_begin(struct list *list);
struct list_elem *list_next(struct list_elem *elem);
struct list_elem *list_end(struct list *list);
struct list_elem *list_remove(struct list_elem *elem);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
void list_insert(struct list_elem *before, struct list_elem *elem);
void list_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);
void sema_init(struct semaphore *sema, unsigned value);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);
void lock_init(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);
void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
```

No more questions.