CS 4410 Operating Systems Prelim 2, Fall 2015
Profs. Bracy and Van Renesse

NAME: ____ NetID:___________

- **This is a closed book examination. You have 120 minutes. No electronic devices of any kind are allowed.**
- You must fill in your name and NETID above and at the top of each (odd-numbered) page. If you fail to do so, we will take off 1 point for each omission.
- Show your incomplete work for partial credit. Make any other assumptions as necessary and document them. Brevity is key.
- Please write your solutions within the provided boxes as much as possible. Write clearly. Use the scratch paper at the end if you need to practice your answer first.

Question	**Points Possible**
0: Omitting Name or NetID	−2
1: Multiple Choice	20
2: Alternating Bit Protocol	20
3: Hand-over-Hand	20
4: Oldie but Goodie	20
5: All’s Well That Ends Well	20

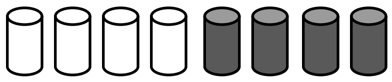
[20 pts] 1. Multiple Choice Questions

[2] a) Networking. Which of the following are true of UDP? Select all that apply.
(A) UDP is reliable.
(B) UDP is ordered.
(C) UDP has a smaller header than TCP.
(D) UDP controls network congestion.
(E) UDP is the predominant protocol for all web traffic.

Your Answer:

C

[3] b) RAID. Recall that RAID Level 1 mirrors disks. In the picture, there are four stripes, each of which is mirrored once. Which of the following statements about this RAID Level 1 set-up are true? Select all that apply.



(A) RAID Level 1 can always detect when a single bit flips.
(B) RAID Level 1 can always detect and correct when a single bit flips.
(C) RAID Level 1 can always detect when two bits flip (note that the two bits may or may not be on different disks).
(D) RAID Level 1 can always detect and correct when two bits flip.
(E) RAID Level 1 supports a 2x read performance over an un-mirrored disk even while detecting bit flip errors.

Your Answer:

A

[2] c) Caching and Page Tables. Which of the following statements about caching and page tables are true? Select all that apply.
(A) A Page Table is a cache for memory.
(B) A Level 2 Cache is a cache for a Level 1 cache.
(C) A 2-Level Page Table is a cache for a 1-Level Page Table.
(D) A TLB is a cache for a Page Table.
(E) DRAM can be used as a cache for disk.

Your Answer:

D, E

[3] d) RPC. Which of the following statements about Remote Procedure Calls (RPC) are true? Select all that apply.
(A) RPCs require the programmer to construct correctly formatted network messages.
(B) A Remote Procedure Call could support a Python program running on Linux to provide a service to a C++ program running on Windows.
(C) After the client and server stubs are compiled, the server program must then define the server's interface using an Interface Definition Language (IDL).
(D) If you look at code that invokes a Remote Procedure Call, the call would be indistinguishable from a local procedure call.

Your Answer:

B, D

NETID:________

1. (Continued)

[2] e) Page Tables. In a clip of the movie The Social Network that was shown in class, the Harvard professor asks a question about single-level page tables, paraphrasing: "assuming PTEs have 8 status bits, what would those status bits be?" Mark Zuckerberg answered, correctly according to the professor, "1 valid bit, 1 modify bit, 1 reference bit and 5 permission bits." The professor's question is: (Select all that apply.)

(A) impossible to answer as there is not enough information
(B) a very difficult question that only a serious geek could answer
(C) so easy that anyone with a basic knowledge of computer science could have answered it
(D) really outdated

Your Answer: A

[3] f) Page Tables. Which of the following statements about page tables are true? Select all that apply.

(A) Multi-level page tables always require more space than single-level page tables.
(B) Multi-level page tables generally have a slower look-up time than single-level page tables.
(C) All page table structures that are not a simple single-level page table have the fundamental structure of an array of arrays.
(D) Page tables need to be invalidated on a context switch.

Your Answer: B

[2] g) File Systems. Which of the following statements about Unix-like File Systems (UFS) are true? Select all that apply.

(A) A File System Consistency Checker detects random bit flips in the data blocks of the file system.
(B) When completed, every block in a consistent File System must be marked as either free or in use.
(C) In a correctly structured Directory System, two distinct paths cannot lead to the same file.
(D) A Directory can consist of indirect blocks and data blocks, just like a File.

Your Answer: B, D

[3] h) Networking. Which of the following statements about Ethernet's Carrier Sense Multiple Access / Collision Detection (CSMA/CD) Protocol are true? Select all that apply.

(A) multiple hosts use CSMA/CD to share the same Ethernet physical network.
(B) CSMA/CD has senders sense whether the Ethernet is currently in use.
(C) CSMA/CD has senders sense to determine whether a collision has transpired.
(D) CSMA/CD prevents any single host from monopolizing the network.
(E) CSMA/CD guarantees packet delivery.

Your Answer: A, B, C

[20pts] 2. Alternating Bit Protocol

This question is to test your understanding of retransmission protocols such as TCP. Suppose there are two computers, X and Y, connected by a single physical network link. Packets can flow in both directions. Packets can get lost, but they can't get re-ordered or damaged. While unreliable, if one computer keeps retransmitting the same packet (with the same contents), eventually at least one copy will arrive at the other computer. The minimum latency on the link (the time between sending a packet and receiving it) is 1 millisecond and the maximum packet size is 101 bytes. The bandwidth is unlimited. Note that because of the set-up, packets do not need addresses: a packet sent on one end of the link is automatically destined for the other. The length of a packet *p* is given by function length(*p*).

Pat designs an "alternating bit protocol" for reliable communication from X to Y: X and Y both maintain a sequence number that counts the number of packets sent and received, respectively. A packet has two fields: a 1 byte header and a payload of at most 100 bytes. Having only limited size, the header cannot store the entire sequence number. In this case, the 1-byte header stores the sequence number **mod 2**, that is, the header only contains the least significant bit of the sequence number. Packets from X to Y are data packets, and packets from Y to X are acknowledgment packets.

The send function on X is as below:

```
var send_seq initially 0;

fun reliable_send(payload):
  if length(payload) > MTU – 1:
    return ERROR(“payload too large”)

  # Keep trying until an acknowledgment is received
  for ever:
    # Send a data packet
    var data = new Packet()
    data.seq = send_seq mod 2
    data.payload = payload
    link.send(data)

    # Wait for an ack packet with the same sequence
    # number, timing out after 5 seconds. If successful
    # increment send_seq and return SUCCESS.
    var ack = link.receive(5)
    if ack != TIMEOUT:
      if ack.seq == send_seq mod 2:
        send_seq += 1
        return SUCCESS
```

The corresponding receive function on Y is:

```
var recv_seq initially 0;

fun reliable_receive():
  # Keep receiving packets until a packet
  # arrives with the expected sequence number
  for ever:
    # Wait for data packet and prepare ACK
    var data = link.receive(∞)
    var ack = new Packet()
    ack.seq = data.seq
    ack.payload = None

    # If the data packet has the right sequence
    # number, increment recv_seq,
    # send the ack, and return the payload
    if data.seq == recv_seq mod 2:
      recv_seq += 1
      link.send(ack)
      return data.payload

    # send acknowledgment in any case
    link.send(ack)
```

Basically, the sender sends even packets (0, 2, 4, …) with a header containing 0 and odd packets (1, 3, 5, …) with a header containing 1. For each packet, the sender keeps sending the same packet until it gets an acknowledgment with the same bit in the header. The receiver acknowledges all packets it receives. It delivers the first packet with a 0 header, then the first packet with a 1 header, and then it goes back to 0 and so on, alternating between 0 and 1.

Answer the following questions:

a) [3] True or False: It is an invariant that ((*send_seq* == *recv_seq*) or (*send_seq* + 1 == *recv_seq*)).

True

a) [2] What layer protocol is this: Data Link, Network, Transport, or Application?

Transport

NETID:________

c) [3] What is the maximum payload transmission rate in bytes / second from the sender's perspective? (Think about the best case in which no packets get lost.)

50,000

Briefly explain (or provide the work for) your answer:

100 / (2 x .001)

d) [3] True or False: if packets could be re-ordered on the link, the protocol still works.

False

d) [3] True or False: if the protocol used all 8 bits in the header and used an 8-bit sequence number (0 … 255) instead of a 1-bit sequence number (i.e., replacing **`mod`** 2 with **`mod`** 256), the protocol would work even if packets could get arbitrarily re-ordered?

False

f) [3] Suppose the protocol used an 8-bit sequence number and windows of at most 10 packets so that up to 10 packets could be sent before an acknowledgment was required, what would the maximum payload transmission rate be (in bytes/sec)? (Recall that the bandwidth is unlimited, but the end-to-end latency is not.)

500,000

Briefly explain (or provide the work for) your answer:

10 x 100 / (2 x .001)

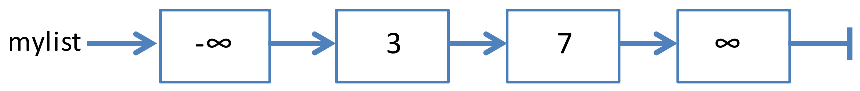
g) [3] Which of the following statements are consistent with the end-to-end design principle? Check either True or False (no points if you check both):

	True	False
• If the probabilities of packet loss on the links that connect a source and destination are independent of one another, then it is not strictly necessary to implement per-link reliability: end-to-end retransmission is sufficient to provide reliability. For some applications an end-to-end acknowledgment is even necessary, for example in the case of reliable file transfer between hosts that may crash.	✓	
• Implementing reliability on intermediate links is useless and one should never do it.		✓
• Implementing reliability on intermediate links induces overhead (for example, buffering for retransmission or computing checksums) even for end-hosts that don't need it.	✓	

[20pts] 3. Hand-over-Hand

```
1   class Node:                 # node in linked list
2     def __init__(self, value, next):
3       self.lock = Lock()
4       self.value = value
5       self.next = next        # another node or None

7   def newList():
8     return Node(−∞, Node(∞, None))

10  def helper_find(list, value):
11    before = list
12    before.lock.acquire()
13    after = before.next
14    after.lock.acquire()
15    while after.value < value:
16      before.lock.release()
17      before = after
18      after = before.next
19      after.lock.acquire()
20    return before, after

22  def contains(list, value):
23    node = list
24    while node.value < value:
25      node = node.next
26    return node.value == value
```

```
28  def size(list):
29    total = −1
30    node = list
31    while node.value < ∞:
32      node = node.next
33      total += 1
34    return total

36  def add(list, value):
37    before, after = helper_find(list, value)
38    if after.value != value:
39      before.next = Node(value, after)
40    before.lock.release()
41    after.lock.release()

43  def remove(list, value):
44    before, after = helper_find(list, value)
45    if after.value == value:
46      before.next = after.next
47    before.lock.release()
48    after.lock.release()

50  mylist = newList()   # example starts here
51  add(mylist, 3); add(mylist, 7)
52  print size(mylist)    # prints "2"
```



Lines 1-48 in the code above implement a concurrent, thread-safe, and deadlock-free sorted linked list of distinct numbers, assuming the following two conditions hold:

- ∞ is larger than any number, and −∞ is smaller than any number;
- reading and writing object references (such as `node.next`) are atomic.

The list interface is as follows (lines 50-52 serve to illustrate some of these):

`mylist = newList()`	# creates a new list
`contains(mylist, x)`	# returns whether mylist contains x, where −∞ < x < ∞
`size(mylist)`	# returns the size of the list
`add(mylist, x)`	# adds number x to the list, where −∞ < x < ∞
`remove(mylist, x)`	# removes number x from the list, where −∞ < x < ∞

The list implementation uses two administrative "book-end" nodes with values −∞ and ∞. The `add()` and `remove()` operations are updates, and these require locks. Locks are acquired in a "hand-over-hand" fashion: going through the list, the lock on the next node is acquired before the lock on the last node is released. A thread may thus hold up to two locks at a time. The helper function `helper_find(list, value)` finds, locks, and returns two adjacent nodes `before` and `after` in the list such that `before.value < value ≤ after.value`. Because of the book-end nodes, `helper_find` is always successful.

The read-only `size()` and `contains()` functions do not acquire any locks and can run *lock-free*!

NETID:________

True	False	Check True or False for each of the following questions:
☐	✓	[0pts] The implementation of the list is in the style of Mesa monitors. *(This is an example question and answer.)*
☐	✓	[2pts] The order in which locks are released in lines 40/41 or 47/48 matters for correctness.
✓	☐	[2pts] The code is deadlock-free because if two threads that run `add()` or `remove()` acquire the same two locks, they acquire them in the same order.
☐	✓	[1pt] A thread running `add()` or `remove()` acquires locks in a nested fashion, i.e., if it requires lock L1 and then L2, it will first release L2 and then L1.
✓	☐	[2pts] It's an invariant at line 20 that `after == before.next` holds.
✓	☐	[2pts] In this code it's an invariant that if a thread holds locks on two nodes containing x and y resp., x < y, then no other thread can insert or remove nodes with values in the range [x, y].
☐	✓	[2pts] A list may contain duplicates in case threads invoke `add(list, x)` multiple times (possibly concurrently) with the same x.
☐	✓	[2pts] The code may accidentally remove a bookend node if a thread invokes `remove(x)` on a value x that is not currently in the list.
☐	✓	[2pts] If one thread invokes `add(list, x)`, and another thread invokes `remove(list, x)` concurrently, then, assuming there are no other operations on the list, it is guaranteed that x is in the list after both operations complete.
☐	✓	[1pt] If one thread invokes `add(list, x)`, and another thread invokes `remove(list, x)` concurrently, then, assuming there are no other operations on the list, it is guaranteed that x is *not* in the list after both operations complete.
✓	☐	[2pts] Suppose a thread invokes `add(list, x)` at time t1, and the call returns at time t2. Also suppose any call to `remove(list, x)` finished before t1 and no thread invokes `remove(list, x)` at time t1 or later. Then any call to `contains(list, x)` after time t2 will return `True`.
☐	✓	[2pts] Same setting as the previous question. Any call to `contains(list, x)` after time t1 (instead of after time t2) is guaranteed to return `True`.

[20pts] 4. Oldie but Goodie

The **PDP11** was a series of computers sold by Digital Equipment Corp. (DEC) from 1970 and into the nineties. A PDP11 computer has a 16-bit virtual address space, where each address identifies a byte, for a total of 64 Kbytes. A page is 2^{13} bytes = 8 Kbytes, and thus the virtual address space of a process consisted of 8 pages. A page table entry (PTE) had a 9-bit frame (= physical page) number, a Valid bit, and a Writable bit.

a) [5pts] What is the maximum physical memory (in Kbytes) in a PDP11?
(A Kbyte is 1024 bytes.)

$\boxed{2^{13+9} = 4{,}096 \text{ Kbytes}}$

b) [9pts] Consider the following page table of a process:

Page	Valid	Frame	Writable
0	yes	0x003	no
1	yes	0x001	no
2	yes	0x008	yes
3	no	N/A	N/A

Page	Valid	Frame	Writable
4	no	N/A	N/A
5	no	N/A	N/A
6	no	N/A	N/A
7	yes	0x004	yes

Fill in the following table:

Virtual Address	**Valid (yes, no)**	**Physical Address (if valid) in hexadecimals**	**Writable (yes, no)**
0x1234	yes	0x07234	no
0x4321	yes	0x10321	yes
0x8888	no	N/A	no

c) [6pts] The Bogux O/S running on the PDP11 uses "Local Replacement", meaning that it assigns a certain number of physical frames to each process. As a result, two processes never contend for the same frame. However, if a lot of processes are running, the number of frames per process may well be fewer than 8. Assume a situation in which each process has three frames. Suppose the page reference string of some process is

0, 7, 2, 0, 7, 1, 0, 3, 1, 2

Initially no pages are mapped to physical frames. Now consider the state of the process's page table after the first 7 references (i.e., after page accesses 0 7 2 0 7 1 0). Which (up to three) pages are mapped at this time assuming one of the following page replacement schemes, and how many page faults have occurred then. Also show in the last column how many page faults occur in total after all 10 references?

Scheme	all page numbers of mapped pages **after 7 references** (3 max.)	#page faults **after 7 references**	#page faults total (**after 10 references**)
First In First Out (by way of example)	0 1 2	5	7
LRU (Least Recently Used)	0 1 7	4	6
OPT (Belady)	0 1 2	4	5

[20pts] 5. All's Well That Ends Well

NETID:________

Suppose you have a Terabyte partition on a disk. To be precise, the partition has 2^{40} bytes on it, subdivided into blocks of 4Kbytes ($4096 = 2^{12}$ bytes).

a) [2] How many blocks are on the partition? (Write your answer in the format 2^{xxx}.)

2^{28} (≈ 256 million)

Briefly explain (or provide the work for) your answer:

$2^{40} / 2^{12}$

You want to put a Unix-like file system on the partition, with one superblock in position 0, followed by a sequence of blocks filled with i-nodes. Each i-node is $128 = 2^7$ bytes. You want to have enough i-nodes to store 2^{20} (about a million) files.

b) [3] How many blocks do you need to store all these i-nodes? (Answer in 2^{xxx} format.)

2^{15}

Briefly explain (or provide the work for) your answer:

$2^{20} \times 2^7 / 2^{12}$

A block pointer identifies a block on the partition, and is 4 bytes long (enough to identify 2^{32} blocks). An "indirect block" (a block filled with block pointers) can have 4096 / 4 = 1024 (2^{10}) block pointers.

Suppose now that an i-node contains 13 block pointers. The first 10 point to the first 10 data blocks. The next three point to an indirect block, a double indirect block, and a triple indirect block. The maximum file size can be approximated by just the number of data blocks reachable from the triple indirect block pointer (the rest is negligible).

c) [3] In theory, how much data (in bytes) could be accessed from the triple indirect block pointer in the i-node? For this question, assume the size of the disk is unbounded. (Answer in 2^{xxx} format.)

2^{42}

Briefly explain (or provide the work for) your answer:

$= {2^{10}}^3 \times 2^{12}$

Question 5. (cont'd)

d) [3] Assume now that the file system cache is empty except for the superblock. Assume the file with i-node #2015 has the string "Hello World" in it (that is, the file is just 11 bytes long). How many disk accesses would be necessary to read the contents of this file, given that you know the i-node number?

2

Briefly explain your answer:

- read block containing inode #2015 to find the data block
- read the data block itself

e) [3] In reality this file's i-node number has to be retrieved first. Suppose the name of the file is /etc/test.txt. Assume that the contents of each directory fits in a single block. The root directory / is described in i-node #2, and /etc is in i-node #5. Again, assuming only the superblock is in the cache and a cache large enough so the same block never has to be read more than once, how many disk accesses are required to read the file?

5

Briefly explain your answer:

- ~~read superblock to find inode of root directory (#2)~~ – *already in cache*
- read block with inode of root directory (#2) to find location of root directory
- read root directory to find inode # of etc. (#5)
- ~~read block with etc. inode (#5) to read etc. to find location of ect directory~~ – *already in cache*
- read etc. directory to find inode # of test.txt
- read block with inode #2015
- read the content of the file itself

NETID:________

f) [3] File /etc/shakespeare.txt (i-node #7) contains the complete works of Shakespeare (2^{22} bytes or about 4 Megabytes). Assuming only the superblock is in the cache, how many disk accesses are required to retrieve the whole thing?

1028

Briefly explain your answer:

- ~~read superblock to find inode of root directory (#2)~~ – *already in cache*
- (1) read block with inode of root directory (#2) to find location of root directory
- (1) read root directory to find inode # of etc (#5)
- ~~read block with etc inode (#5) to find location of ect directory~~ – *already in cache*
- (1) read etc directory to find inode # of shakespaere.txt
- ~~read block with shakespeare inode (#7)~~ – *already in cache*
- (10) read the first 10 direct blocks
- (1) read the indirect block
- (1014) read the remaining (1024 – 10 = 1014 blocks)

g) [3] Suppose somebody wants to add the text "All's Well That Ends Well." to the end of the complete works of Shakespeare (the new text will be contained in a new data block at the very end). Suppose that the file system has only the superblock in its cache and can allocate free blocks without going to the disk. How many disk reads and how many disk writes are necessary (assuming a "write-through" cache? (Assume that among the i-nodes only i-node #7 has to be updated for this operation.)

4 reads / 3 writes

Briefly explain your answer:

- ~~read superblock to find inode of root directory (#2)~~ – *already in cache*
- (1) read block with inode of root directory (#2) to find location of root directory
- (1) read root directory to find inode # of etc (#5)
- ~~read block with etc inode (#5) to find location of ect directory~~ – *already in cache*
- (1) read etc directory to find inode # of shakespaere.txt
- ~~read block with shakespeare inode (#7)~~ – *already in cache*
- (1) read the indirect block
- (1) allocate and write the new data block
- (1) write the updated indirect block
- (1) update the last modified time in the inode and write the block containing the inode