Fill out the bottom of this page with your computing ID.
Write your computing ID at the top of each page in case pages get separated.

On my honor as a student I have neither given nor received aid on this exam.

---

1. In POSIX sockets, sockets used by clients are represented as a file descriptor on which one can call `read()` or `write()`.

   For these questions, consider using such sockets implemented on top of typical network that uses a mailbox-like model of communication. Assume the sockets are used in a "stream" mode where they resemble bidirectional pipes (and not in "datagram" mode POSIX supports which we did not discuss in lecture).

(a) (5 points) Consider a client program making a `write()` call on a socket to send a 4096-byte message to a server program. Which of the following is true about this operation? **Select all that apply.**

- ◯ The client message will transmit the message on the network at most once.
- ◯ If the `write()` call returns successfully, then the server program successfully processed the message and sent any response
- √ The server to which the socket is connected will read the message at most once.
- ◯ If the `write()` call successfully writes the entire 4096-byte message, then the server will read the entire message in one call to `read()`.
- √ If the `write()` call returns an error, then the message was not received by the server. *accept either way b/c didn't cover this enough detail*

(b) (4 points) Consider two clients, running on different machines, both making `write()` calls to a socket connected to the same server process. Assume both of these sockets were created by using `connect()` to create connections to the same server address. What is true about how the server can distinguish between the two messages? **Select all that apply.**

- √ when the messages are sent over the network, each client's socket implementation will include the address of each client machine (or equivalent information)
- √ it will read them from different socket file descriptors
- ◯ the return value of `read()` will include the client address
- ◯ the client address will be added to the buffer passed to `read()`

2. (4 points) Consider systems using NFS (Network File System) which has close-to-open consistency model. Which of the following are sufficient conditions for two clients (on different machines) that both access a shared file on a single NFS server to observe the same behavior (that is, the same values being read) that would happen on a local filesystem? **Select all that apply.**

   ◯ clients always wait at least one second between writing the file and reading it

   ◯ the two clients always close the file after modifying it *if one client is reading the file at the same time, may not notice changes indefinitely due to caching even though close forces those changes to the server b/c the reading client won't check for updates*

   √ both clients have caching entirely disabled

   √ the two clients never have any file open at the same time

3. (5 points) The network filesystem AFS uses *callbacks*, where a client can register with filesystem server to be notified when the server's copy of a file is updated. In contrast, to maintain the same open-to-close consistency model, NFS requires clients to check with the server each time a file is opened. What is true about AFS's callbacks *compared to NFS's alternate strategy to ensure consistency*? **Select all that apply.**

   ◯ AFS's callbacks more easily handle (without violating the consistency model) cases where the server crashes and is rebooted *don't need to reconstruct the list of callbacks to ensure consistency*

   √ AFS's callbacks are likely to lower the number of times clients need to contact the server *don't need to contact on open() if already cached*

   ◯ AFS's callbacks ensure that, even if a another client is modifying its cached copy of the file, a client will immediately learn about the modifications *doesn't change that clients are allowed to modify their cached copy without writing it to the server*

   √ AFS's callbacks are likely to increase the performance of opening and reading a file on a client *don't need to contact on open() if already cached*

   ◯ AFS's callbacks are likely to increase the performance of writing and closing an open file on a client *still need to write out file when closing to server*

4. (4 points) Suppose the coordinator in a two-phase commit system restarts after a power failure that affects the coordinator but not any of the workers. It reads a message in its log that must have been written just before sending prepare messages to its workers. Which of the following would make most sense for the coordinator to do next?

   ◯ send an 'agree to commit' vote to each of the workers *coordinator doesn't send votes*

   ◯ send a 'committed' message to the workers *have no way of knowing the workers will all operating normally*

   ◯ discard the transaction the log message is about and wait for the next transaction to be started *workers may be confused indefinitely wondering if transaction will finish*

   √ resend the prepare messages to the workers

   ◯ wait for votes (to commit or to abort) from the workers *don't know prepare messages actually sent*

5. Consider a system which uses the following page replacement policy for its page cache with **3 physical pages**:

   - physical pages are kept in an ordered list
   - immediately after each page replacement occurs (but before the access that triggered the replacement happens), the physical page used is added to the top of the list and *all physical pages are marked as unaccessed*
   - whenever a physical page is accessed, it is marked as accessed (if it was not already)
   - to choose a page to replace, first, the operating system removes a physical page off the bottom of the list. Then, if it is unaccessed, it chooses this page. Otherwise, it marks the page as unaccessed and places it at the top of the list, and repeats this process starting by removing another physical page from the bottom of the list.

   This policy is the same as the second-chance policy we discussed in lecture, except that pages have their accessed or referenced bit cleared more frequently.

(a) (10 points) Complete the following chart indicating how the replacement policy described above will execute with the following access pattern. Capital letters represent virtual pages, and the access pattern lists the order in which the program performs accesses. Assume all physical pages are initially empty. The first five are done for you.

page accessed	required replacement?	replaces virtual page
A	✓	(empty)
B	✓	(empty)
C	✓	(empty)
B		—
D	✓	A
A	*yes*	*B*
C	*no*	
B	*yes*	*D*
A	*no*	
D	*yes*	*C*

(b) (10 points) Give an example of an access pattern where the above policy would perform a different page replacement than LRU. Identify one page replacement which would differ from an LRU policy. (If the access pattern from the previous part is an example which differs from LRU, you may use it to answer this question.)

Write the access pattern as sequence of virtual pages identified by unique letters. Circle a page replacement whose outcome will differ.

Solution: As above:
A B C B D Ⓐ …

For the page replacement you circled, identify what will be replaced with LRU and with the policy specified above:

LRU: C ; above policy: B

6. Consider a process running on a system with 4KB (2^{12} byte) pages, and two-level page tables where page tables at each level have 1024 entries and virtual addresses are 32 bits. The process has the following memory layout:

virtual address range	usage
`0x00000000-0x00000FFF`	inaccessible
`0x00001000-0x00003FFF`	code (read-only)
`0x00004000-0x00006FFF`	global variables (read/write)
`0x00007000-0x00007FFF`	heap (read/write)
`0x00008000-0x7FFF0FFF`	inaccessible
`0x7FFF1000-0x7FFF2FFF`	stack (read/write)
`0x7FFF3000-0xFFFFFFFF`	inaccessible

(a) (5 points) Based on the memory layout above, how many pages are accessible?

9 (3 code, 3 globals, 1 heap, 2 stack)

(b) (5 points) Suppose all the addresses the process can access are loaded into memory and can be accessed without triggering a page fault. What is the minimum amount of space that must be allocated to store page tables to allow this? (For this question, ignore page table entries that would be allocated to point to non-user-accessible operating system memory (like exception handlers).)

3 pages (12 KB) (1 page table for 1st level, 2 page tables for second level; each page table is one page)

(c) Suppose this process forks, and then:

- the child process modifies 8 bytes on the stack at address `0x7FFF1080` through `0x7FFF1088`,
- the child process modifies 5120 bytes in a global array at addresses `0x00004F00` through `0x00006300`, and
- the parent process modifies 1024 bytes in a global array at addresses `0x00004F00` through `0x00005300`

Assume the operating system uses copy-on-write and that all accessible pages in either the child or parent process can be read without a page fault.

i. (5 points) How many more pages of data that can be acccesed by the child or parent process must be allocated compared to how many pages were allocated before forking?

4 pages (1 for stack, 3 for globals); also give credit for 6 (assuming you need to copy pages weven if there's onl

ii. (5 points) How many more pages must be allocated to store additional page tables? (For this question, ignore space required to store page table entries that would point to operating system code like exception handlers.)

same as part b

7. Consider a filesystem which uses redo-logging. A program asks the filesytem to rename a file in a directory on this filesystem, which requires updating the directory entry for the file to contain a new name and updating the modification time stored in the directory's inode. The filesystem uses redo logging to ensure these two updates occur in a consistent way (so it is not possible for only one of the updates to appear to be done in the event of a crash).

(a) (4 points) A useful redo-log entry to represent for the update to the directory entry will primarily include _____. *on original exam, there were some typos in the incorrect options below*

- ◯ the old (pre-rename) contents of the directory entry for the file and its location on disk
- ◯ directory entry representing a symbolic link from the old to new inode
- ◯ a new inode number for the file being moved
- √ the new (post-rename) contents of the directory entry for the file and its location on disk
- ◯ none of the above; this update should not appear in the log

(b) (5 points) Suppose the machine unexpectedly loses power, is rebooted when power is restored, and filesystem redo-logging-based recovery code runs on boot. What must have happened before the machine lost power for the file to be present under its new name (and not its old one)?

Solution: the 'commit' entry for the transaction was written to the log

8. (4 points) Suppose we want to allow all users to append to a log file but allow none of them (except the system administrator) to tamper with the log file. Which of the following are viable mechanisms for doing this on a POSIX-like system which has support for passing file descriptors between programs? **Select all that apply.**

- √ create a set-user-ID program owned by a system administator user which reads from its input and appends to the log file, and the log file is set as only writeable by that system administrator user
- ◯ create an access control list for the file which permits all users to write the file, but makes it not readable by any but the system administrator *users can overwrite data in the log*
- ◯ mark the log file as set-user-ID and not writeable by anyone but its owner
- √ create a directory that is marked as world-writeable and have a special program (run by the system administrator) periodically append the contents of files users write in that directory to the log file (located in another directory and only writeable by the system administrator user) *accept EITHER true or false — other users can delete files in the world-writable directory before they are added to the log file, which arguably tampers with the log*

9. Suppose we knew that an inode-based filesystem with a similar design to the xv6 filesystem has:
   - no support for fragments or extents or similar
   - 4KB blocks
   - 4 byte block pointers
   - an indirect and double-indirect pointer in each inode

   and is used to store only:
   - 50 000 1KB files
   - 50 000 4KB files
   - 2 000 16KB files
   - 200 128KB files; and
   - 10 8MB files

   (where 1KB is 2^{10} bytes and 1MB is 2^{20} bytes.) For each of the questions below, you may leave your answers as unsimplified arithmetic expressions.

(a) (5 points) If there were 100 direct pointers per inode, then how many blocks would be allocated to store pointers to other blocks?

30 (for 8MB files; one indirect + double-indirect pointing to one indirect (with rest of double-indirect block unused

(b) (6 points) if there were 100 direct pointers per inode, then how much space would be used for direct block pointers *that did not point to anything* within the inodes allocated to the files specified above?

Solution: $99 \cdot 100\ 000$(for 1KB+4KB files) $+ 96 \cdot 2\ 000 + (100 - 32) \cdot 200 = 10\ 105\ 600$ pointers times 4 bytes per pointer

(c) (8 points) What number of direct pointers per inode would result in the least space being allocated to store information related to the location of data for files? Count both blocks allocated to store block pointers and space in used inodes dedicated to store block pointers (regardless of whether they actually contain a valid block pointer). Don't worry about space in inodes that are not assigned to any file mentioned above. Assume it is not possible for inodes to have variable numbers of direct pointers.

Solution: 4 (approx $(200 + 30) \cdot 4K$ (blocks of ptrs)) + $4 \cdot (103K)$ (ptrs in inodes) $\approx 1332K$); half credit for 32 (approx $30 \cdot 4K$ (blocks of ptrs) + $32 \cdot 103K$ (ptrs in inodes) $\approx 3416K$)

(d) (4 points) Given the number of direct pointers you specified above, how many blocks *outside of inodes* would be allocated to store pointers to blocks? (You may leave your answer as an unsimplified arithmetic expression.)

Solution: 200 (for the 128KB files) + 30 (for 10MB files)

10. Consider a hypervisor (also known as a virtual machine monitor) that:

- uses a trap-and-emulate strategy;
- use shadow page tables to implement virtual memory (without any special hardware support for virtual machine page tables);
- fills in shadow page table entries on demand (that is, in response to page faults on the underlying hardware, rather than trying to detect modifications to page table entries directly); and
- maintains two versions of the shadow page table, one for the guest OS kernel and one for the guest OS's user mode, and, whenever it modifies the shadow page tables, modifies both versions

The guest operating system has not been modified to run on the virtual machine monitor (so it runs the same code that would run on real hardware).

Consider the following scenario: The guest operating system is running a process. The process tries to access a virtual page which will be allocated on demand. The guest operating system allocates a physical page from its page fault handler, then resumes the process. Immediately after being resumed, the process writes a value to the newly allocated page.

(a) (6 points) Suppose the page table entry the guest OS writes to its page table as a result of the allocation-on-demand is:

- be located at the index in the page table corresponding to page number 100
- and contains page number 50

and that a partial mapping between guest OS's physical pages and the underlying hardware's machine (physical) addresses is:

physical (from guest OS's view) page	machine (hardware physical) page
50	500
100	501

The corresponding shadow page table entry for the guest OS's user mode should (fill in the blanks below):

- be located at the index in the page table corresponding to page number 100 .
- and contain page number 500 .

(b) (10 points) What exceptions (of all kinds, including traps, interrupts, etc.) will occur as part of this process on the real hardware? Identify them briefly.

Solution: page fault for access to-be-allocated page; protection fault for returning from exception handler; page fault to fill in the shadow PT entry. No deduction for also listing a protection fault for the guest OS invaliding the TLB or changing the page table base pointer.

11. (25 points) Consider using monitors (mutexes and condition variables) to build a map (hashtable-like key/value data structure) that has support for waiting for a particular key to be inserted. The hashtable provides the following operations:

    - `void Put(string key, string value)`
    - `string WaitForAndGetValue(string key)`

    Calling `WaitForAndGetValue(key)` should return immediately if `key` has already been set at least once, otherwise it should wait for a call to `Put` for `key` to complete and then return the value set.

    On the next page, complete the implementation of this in terms of the C++ standard library `map<Key, Value>` type which provides the following relevant methods:

    - `void insert(Key key, Value value)` — insert the key/value pair (key, value)
    - `at(Key key)` — retrieve the value with key key
    - `contains(Key key)` — return true if a key is present, false otherwise

    The implementation uses a `ValueHolder` struct to hold both a string value and relevant metadata. In our reference solution, we added a field that struct to implement `Put` and `WaitForAndGetValue`.

    The implemention also has a utility function `GetHolderForKey` which retreives a ValueHolder object and is responsible for allocating and initializing it, if necessary.

    If you don't know the exact name or syntax for a pthreads API function you require, make a reasonable guess. We will not take off points for minor syntax errors or differences in names/argument order/whether arguments are pointers/etc.

```
struct ValueHolder {
    string value;
    bool value_set;
    pthread_cond_t cv;
};
pthread_mutex_t lock;
map<string, ValueHolder*> internal_map;

ValueHolder *GetHolderForKey(string key) {
    if (!internal_map.contains(key)) {
        ValueHolder* holder = new ValueHolder;
        holder->value_set = false;
        pthread_cond_init(&holder->cv, NULL);
        internal_map.insert(key, holder);
    }
    return internal_map.at(key);
}

void Put(string key, string value) {
    pthread_mutex_lock(&lock);
    ValueHolder *holder = GetHolderForKey(key);
    holder->value = value;
    holder->value_set = true;
    pthread_cond_broadcast(&holder->cv);
    pthread_mutex_unlock(&lock);
}

string WaitForAndGetValue(string key) {
    pthread_mutex_lock(&lock);
    ValueHolder *holder = GetHolderForKey(key);
    while (!holder->value_set) {
        pthread_cond_wait(&holder->cv, &lock);
    }
    string result = holder->value;
    pthread_mutex_unlock(&lock);
    return result;
}
```