CS 111 Final Examination
Spring Quarter, 2022
Solutions

You have 3 hours (180 minutes) for this examination; the number of points for each question indicates roughly how long we think it will take to answer that question. Make sure you print your name and sign the Honor Code below. During this exam you are allowed to consult three double-sided pages of notes that you have prepared ahead of time, as well as any of the .c and .h files from your solution to Project 7. Other than these materials, you may not consult any other sources, including books, notes, your laptop, or phones or other personal devices. If there is a trivial detail that you need for one of your answers but cannot recall, you may ask the course staff for help.

Note: do not write on the backs of any pages! We'll be scanning the exams into Gradescope and won't see anything on the back sides. There are extra pages at the end if you need more space.

I acknowledge and accept the Stanford University Honor Code. I have neither given nor received aid in answering the questions on this examination, and I have not consulted any external information other than three double-sided pages of prepared notes.

____
(Signature)

____
(Print your name, legibly!)

____
(SUNet email id)

Problem	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	Total
Score												
Max	12	10	10	5	5	8	8	18	12	20	50	158

Problem 1 (12 points)

Indicate whether each of the statements below is true or false, and explain your answer briefly.

(a) The target of a symbolic link can be a directory (i.e. it is legal to invoke the command "ln -s x/y z" where x/y refers to a directory).

Answer: true. Hard links cannot refer to directories because that could result in circular link structures, which would confuse the inode reclamation mechanism, which uses reference counts. Symbolic links do not affect reference counts, so there's no problem with circularities.

(b) In a system using paging, each thread in a process has its own set of page maps.

Answer: false. All of the threads in a process share the same virtual address space, which means they use the same page maps.

(c) When the operating system allocates a physical page of memory on behalf of a user process's stack, it must zero out the contents of that physical page before allowing the user process to access it.

Answer: true. If the page were not zeroed, then the new owner would be able to read any contents left there by the last owner of the page; this could leak sensitive information.

(d) The primary reason to use a multi-level page table instead of a single-level page table is to speed up address translation.

Answer: false. Multi-level page tables actually make address translation slower. The main reason for using them is to save memory (there's no need to allocate page tables for unused portions of the virtual address space).

(e) Indirect blocks are never used in inodes representing directories.

Answer: false. Although most directories are small, it is possible for directories to grow arbitrarily large (and some directories are quite large). If a director gets large enough, it will need to use indirect blocks.

(f) In a file system that uses a write-ahead log, the operation described by each log entry must be idempotent.

Answer: true. It is possible that the operation described by a log entry has already been completed when the log is replayed; or, if there is a crash during log replay, parts of the log may get replayed multiple times. Thus, it must be safe to replay a given log entry multiple times.

Problem 2 (10 points)

(a) (5 points) You have been complaining to a friend that your file system, which uses the 4.3 BSD multi-level index mechanism, limits the sizes of files, and you are reaching the point where you would like to create files larger than the current limit. Your friend suggests that you should double the block size in the file system, and that this would increase the maximum file size by more than a factor of 2x. Is your friend right or wrong, and why?

Answer: your friend is right. If you double the block size, not only will each block hold twice as much data, but each indirect block will hold twice as many block pointers. Thus, the maximum file size will increase by roughly a factor of 8x: the double re-indirect block pointer will hold twice as many pointers to index blocks, each of which will hold twice as many pointers to data blocks, each of which will hold twice as much data.

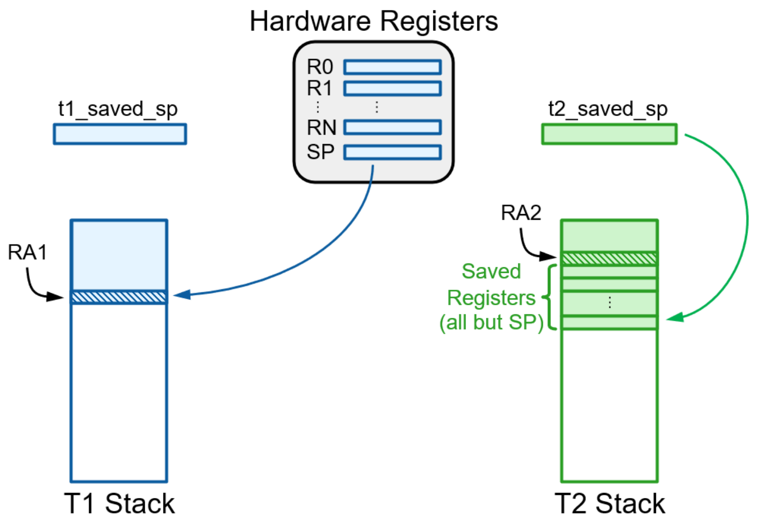
An alternative answer is that the maximum file size can't be increased because it's already 2^{32}, and only 32 bits are allotted in the inode for representing it. This isn't the answer we hoped for (we forgot about this limit in designing the question), but we accepted this answer.

(b) (5 points) One of the page replacement policies discussed in class was LRU (least recently used). Another policy, which we did not discuss in class, is MRU (most recently used): this policy chooses the most recently accessed page to evict. Assuming there are only 3 page frames in physical memory, give an access pattern, or sequence of virtual page accesses (e.g. ABCDEAAABFE), in which MRU results in fewer evictions than LRU.

One possible answer: ABCDABCD

Problem 3 (10 points)

Consider a point in time where your code for Project 2 (Thread Dispatcher) has just invoked the `stack_switch` method. Suppose that the invoking thread is T1 and it wishes to switch to thread T2. The state of the world will look like this when `stack_switch` starts executing:



In particular:

- `stack_switch` will be invoked with the addresses of `t1_saved_sp` and `t2_saved_sp` as parameters (these values are probably in the `Thread` structs for the two threads).
- T2's stack will contain saved registers from when it last invoked `stack_switch`, and `t2_saved_sp` will contain T2's saved stack pointer.
- As part of invoking `stack_switch`, the caller saves its return address on the stack, just as would happen for any method call. Thus T1's stack contains a return address `RA1`, saved when it invoked `stack_switch`, and T2's stack also contains a return address `RA2`, saved when `stack_switch` was last invoked in that thread.
- `stack_switch` will push the values of the hardware registers (except for the stack pointer) onto the current stack, then save the stack pointer register in `t1_saved_sp`.
- `stack_switch` will then load `t2_saved_sp` into the stack pointer register, load other hardware registers by popping their saved values off of T2's stack, and then return.

(a) (5 points) Suppose that T1 has invoked `stack_switch` as described above. When this call to `stack_switch` returns, which return address will it use, `RA1` or `RA2`?

Answer: RA2 (any given call to `stack_switch` will return using the return address for the *new* thread).

(b) (5 points) When will the other return address be used, if ever?

Answer: RA1 will be used later, when the dispatcher eventually context-switches back to T1.

Problem 4 (5 points)

Consider two threads running on a single-core system that uses the 4.4 BSD scheduling algorithm. Thread A has been compute-bound and has been using most of the CPU time of the core. Thread B has been mostly blocked, and has used CPU time only in short bursts of 1 millisecond with long gaps in between bursts.

(a) (2 points) Which thread will receive higher priority for execution, and why?

Answer: thread B will receive higher priority, since its recent CPU usage is less than that of thread A.

(b) (3 points) Suppose thread A blocks briefly, then wakes up and needs only one millisecond of CPU time before it blocks again. When thread A wakes up after that short blockage, will its priority be lower than that of thread B, about the same, or higher? Why?

Answer: thread B will still have higher priority, because CPU usage is averaged over a period of time. A single short blockage by thread A will not lower its average CPU usage enough for it to become less thanthe CPU usage of thread B.

Problem 5 (5 points)

Write "Internal" or "External" beneath each of the following descriptions to indicate whether it is an example of internal fragmentation or external fragmentation.

(a) A system has 4096-byte pages, but a process needs 6000 bytes for its code.

Answer: internal.

(b) A system uses static relocation and finds itself in a state where a new process needs 10000 bytes of memory. The system has only two free regions in physical memory, one with 7000 bytes and one with 5000 bytes; as a result, it cannot accommodate the new process.

Answer: external.

(c) In a slab-based memory allocator, a program allocates several 50-byte objects, frees every other one of them, then allocates several 70-byte objects.

Answer: external.

(d) A memory allocator using the Best Fit approach ends up with lots of small holes.

Answer: external.

(e) A memory allocator always rounds each requested size up to a multiple of 8 bytes.

Answer: internal.

Problem 6 (8 points)

Suppose that `fsck` is being used to restore file system consistency after a crash, and it discovers that a particular block is referenced by two different inodes, A and B. Suppose also that inode B has a later creation time than inode A. In discussion during class, a student suggested the the best way for `fsck` to handle this situation is to remove the block from inode A but leave it in inode B: since B has the later creation time, it's likely that file A has been deleted, but its inode was not written back to disk to reflect that before the crash.

However, it is possible that the block actually belongs to inode A, not B (i.e., if the system had shut down cleanly, the block would appear only in A and not in B). Describe a specific sequence of events that could result in the appearance seen by fsck.

> **Answer: this could happen in several different ways. One way is that file A is created first; then file B is created, written, and deleted; finally file A is extended. All of the metadata for file B was written to disk before it was deleted, but it was not overwritten after B was deleted; one of the blocks from file B was reused to extend file A, and the file A metadata was written to disk before the crash along with the contents of the block.**
>
> **Here is another way. File A is created, written, and deleted. Then file B is created written and deleted, and it uses the same blocks that were previously in file A. All of the metadata for file A was flushed to disk after it was created, but not after it was deleted. All of the metadata for file B was flushed to disk after it was created, but not after it was deleted. The data blocks were flushed to disk after they were written to file A, but not after they were written to file B.**
>
> **To get full credit, you must be precise about which blocks were flushed to disk, and when.**

Problem 7 (8 points)

(a) (4 points) Consider a system that uses demand paging and 4 KB pages, and is experiencing thrashing under a particular workload. If the page size were increased from 4 KB to 1 MB, would this make the system's performance better, worse, or have no impact? Explain your answer.

> **Answer: system performance will get worse. The larger page size will end up using more physical memory for the same amount of useful data (even if only 4 KB of a given 1 MB page is needed, the entire page will have to be present in memory).**

(b) (4 points) Now imagine that the same (original) system is experiencing poor performance under a different workload because of too many TLB misses. Would increasing the page size from 4 KB to 1 MB make the TLB performance better, worse, or have no impact? Explain your answer.

> **Answer: TLB performance will get better, because the same number of TLB entries will "cover" more virtual memory, thereby reducing the likelihood of TLB misses.**

Problem 8 (18 points)

Which of the following operations must be privileged (they may only be performed in kernel mode)? Explain briefly why it is or isn't safe for user programs to execute each operation.

(a) Invalidate the TLB entry corresponding to a particular virtual address.

Answer: this operation need not be privileged, since the only harm it does is to reduce performance by causing an extra TLB miss (and this only affects the thread that invoked the operation).

(b) The instruction that invokes a kernel call.

Answer: this operation need not be privileged. In fact, it *must* not be privileged, since if it were, user programs would not be able to invoke kernel calls. The operation cannot cause harm, since the address of the trap handler is determined by the kernel, not the invoking thread.

(c) The instruction that returns from a kernel call back to user space.

Answer: this operation must be privileged, because it changes the processor status register, which includes the user/kernel mode bit (and the new value is determined by whoever invokes the instruction). If a user program could invoke this instruction, it could switch to kernel mode while continuing to run in the user program; it could then do really naughty things.

(d) Set the PML4 base register, which determines the location of the page maps for the current thread.

Answer: this operation must be privileged. If user programs could execute this instruction, they could change the virtual-to-physical mappings and gain access to any physical memory in the system.

(e) Fill in the entries in the jump table used for dynamic linking.

Answer: there is no need for this operation to be privileged (and it is not privileged in actual systems). It's possible that the user program could corrupt its jump table, but that would only affect the particular process, not anything else.

(f) Disable interrupts.

Answer: disable interrupts must be privileged. Otherwise, a user program could disable interrupts and hijack a core (there would be no way for the operating system to regain control).

Problem 9 (12 points)

There is a file in a Unix V6 file system with no journaling (as in Project 7) that uses 25600 blocks for data (that is, its size is 25600*512 bytes). Suppose we want to erase N bytes from the beginning of the file, so that the byte that used to be at offset N in the file will now be at offset 0. Furthermore, 0 < N <= 512. How many disk blocks must be modified to accomplish this (hint: this depends on N)? Show your work so we can see how you computed your answer.

Answer: first, some background. In the Unix V6 file system, each indirect or doubly-indirect block holds 256 block pointers. Since the file has 25600 blocks, it will be a "large" file and will require 100 indirect blocks (25600/100). The first seven indirect blocks will have their pointers in the inode, and the other 93 will be referenced in the doubly-indirect block.

In order to erase bytes, the contents of the file must be shifted down to "cover up" the erased bytes. If N is 512 (i.e. exactly one full block), we can do the erasure by moving block pointers: block 1 becomes block 0, block 2 becomes block 1, and so on. This means we'll need to modify each of the 100 indirect blocks, plus the inode, for a total of 101 modified blocks. Note that we don't need to modify the doubly-indirect block.

If N < 512, then we can't just move block pointers; we will have to copy all of the file data. This means that 25600 data blocks will be modified. In addition, the inode will have to be rewritten to update its size, so a total of 25601 blocks will be modified. No indirect or doubly-indirect blocks will need to be modified.

Problem 10 (20 points)

(a) (15 points) Describe an algorithm that, given the inumber of a file and the inumber of its parent directory, reconstructs an absolute pathname identifying that file. If there are multiple absolute pathnames identifying the file, then reconstruct any one of them. You do not need to write code, but you must describe the algorithm precisely.

Answer: First, set up a running cur pathname and set it equal to the empty string.

Next, read the inode of the parent directory and loop through the directory (in a fashion similar to directory_findname). For each entry, see if its d_inumber matches the inumber of our file. When a match is found, prepend the name in the direntv6 to the cur pathname. Prepend onto this updated cur a '/'.

Next, find the inumber of the parent's parent directory (looking for .. in the parent directory). Loop through this grandparent directory; look for the direntv6 corresponding to the inumber of the parent, and prepend the parent directory's name and a '/'.

Continue in this way until reaching the root directory (inode 1).

(b) (5 points) Is it possible that the path produced by your algorithm in part (a) could contain a symbolic link? Explain your answer.

Answer: yes, but only if the starting file is a symbolic link. None of the parents can possibly be a symbolic link: we know by assumption that the first parent is a directory, and we know that ".." entries always refer to directories.

Problem 11 (50 points)

Extend your solution for Project 7 by implementing the following function:

```
int hard_link(struct unixfilesystem *fs, char *target_path, char *link_path);
```

This function will create a hard link (not a symbolic link) at the location given by `link_path`. The link must refer to the file at `target_path`; upon successful completion of this function, either `target_path` or `link_path` may be used to refer to the file that was originally at `target_path`. The function must return 0 on success and -1 if an error is detected.

Here are some additional details and simplifications:

- Both `target_path` and `link_path` will be absolute paths.
- You may assume that there is currently no file with the name given by `link_path`.
- You may assume that there is at least one unused entry in the directory containing `link_path`. Unused entries are entirely zeroes (both `d_inumber` and `d_name`); they come about when files are deleted.
- Your solution need not write any changes back to disk; it can simply update the relevant file-related data structures in memory.
- For this problem you do not need to print a message for each error condition; returning -1 is sufficient.
- You may use any of the functions defined for Project 7, and you may assume they have been implemented correctly. You may consult your solution to Project 7 when working on this problem.
- You may find the following C library functions useful in your solution:

```
/* Copy the C string at source to dest, including the terminating
 * NULL character. Returns dest. */
char *strcpy(char *dest, const char *source)

/* Same as strcpy, but stop after copying max_chars characters if the
  * end of source hasn't been reached (this may leave dest without a
 * NULL terminating character). */
char *strncpy(char *dest, const char *source, size_t max_chars)

/* Return the address of the first instance of character ch in str,
 * or nullptr if there is no such character in str. */
char *strchr(char *str, int ch);

/* Return the address of the last instance of character ch in str,
 * or nullptr if there is no such character in str. */
char *strrchr(char *str, int ch);
```

Use this page for your work on Problem 11.

```
int hard_link(struct unixfilesystem *fs, char *target_path, char *link_path) {
    if (link_path[0] != '/') {
        return -1;
    }

    // Separate link_path into a directory and an element in that directory,
    // then find the inode for the directory.
    char clone[strlen(link_path) + 1];
    strcpy(clone, link_path);
    char *link_dir = link_path;
    char *last_slash = sttrchr(clone, '/');
    char *elem = last_slash+1;
    if (strlen(elem) > MAX_COMPONENT_LENGTH) {
        return -1;
    }
    int link_in;
    if (last_slash != clone) {
        *last_slash = '\0';
        link_in = pathname_lookup(fs, clone);
        if (link_in < 0) {
            return -1;
        }
    } else {
        // Special case: link_path is in the root directory.
        link_in = 1;
    }
    struct inode link_node;
    if ((inode_get(fs, link_in, &link_node) < 0)
            || ((link_node.i_mode & IFMT) != IFDIR)){
        return -1;
    }
    char *elem = last+slash+1;
    if (strlen(elem) > MAX_COMPONENT_LENGTH) {
        return -1;
    }

    // Get the inode for the target (must not be a directory!).
    target_in = pathname_lookup(fs, target_path);
    if (target_in < 0) {
        return -1;
    }
    struct inode target_node;
    if (inode_get(fs, target_in, &target_node) < 0) {
        return -1;
    }
    if ((target_node.i_mode & IFMT) == IFDIR) {
        return -1;
    }

    // Continued on next page...
    // Find an empty slot in the directory of the link. Each iteration of
    // this outer loop scans one block of the directory file. Note: we don't
    // need to worry about the exact length of the directory file, since we
    // can assume we'll find an empty slot before we get there.
    for (int i = 0; ; i++) {
        struct direntv6 entries[NUM_DIRENTS_PER_BLOCK];
        if (file_getblock(fs, link_in, i, entries) < 0) {
            return -1;
        }
        for (slot = 0; slot < NUM_DIRENTS_PER_BLOCK, slot++) {
            if (entries[slot].d_inumber == 0) {
                // Found a free slot!
                strncpy(entries[slot].d_name, elem, MAX_COMPONENT_LENGTH);
                entries[slot].d_inumber = target_in;
                target_node.i_nlink++;
                break;
            }
        }
    }
    return 0;
}
```