CS 111 Final Examination
Spring Quarter, 2023
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 Assignment 7. Other than these materials, you may not consult any other sources, including books, notes, your laptop, or phones or other personal devices. You may use the calculator app on your phone. 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 will 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	#12	Total
Score													
Max	10	10	16	10	6	9	8	6	15	3	24	40	157

Problem 1 (10 points)

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

(a) When a thread is preempted by the scheduler, any locks it holds are automatically released.

Answer: false. A lock can only be released when the owning thread decides it is safe to do so; if the operating system were to release the lock automatically, it could cause synchronization errors in the application.

(b) When the dispatcher performs a context switch and the new thread is in the same process as the old one, it need not flush the TLB.

Answer: true. The TLB only needs to be flushed when a context switch is causing a change of address space, but all of the threads in a given process have the same virtual address space.

(c) Circular chains of references can cause memory leaks if reference counts are used for storage reclamation.

Answer: true. With reference counting, an object is freed only when its reference count becomes zero, but a circular chain will result in all of the objects in the chain having nonzero reference counts, even if there are no external references to the chain.

(e) A base-and-bound address translation mechanism can support shared libraries.

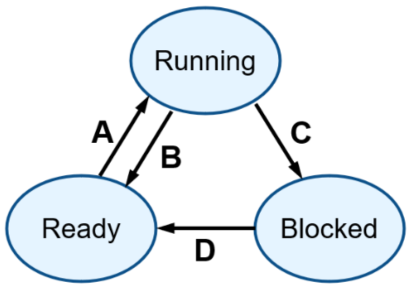
Answer: false. With base-and-bound, a process can access only a single contiguous region of physical memory, but each shared library will occupy a separate contiguous region.

(e) If the block size is increased in a file system, internal fragmentation will also increase.

Answer: true. On average, internal fragmentation will waste one half block in each file; thus, as blocks get larger the total amount of wasted space will increase.

Problem 2 (10 points)

Consider the following diagram, which illustrates the various scheduling states that a thread can be in, as well as transitions between them:



(a) (2 points) What is the initial state of a new thread?

Answer: Ready.

(b) (2 points) Give an example of an event that would cause a thread to undergo transition A (from Ready to Running).

Answer: the thread currently executing on a core exits or blocks, so the dispatcher on that core runs some other Ready thread.

(c) (2 points) Give an example of an event that would cause a thread to undergo transition B (from Running to Ready).

Answer: a time slice ends, so the dispatcher moves the currently running thread back to the Ready queue and runs some other thread.

(d) (2 points) Give an example of an event that would cause a thread to undergo transition C (from Running to Blocked).

Answer: a thread attempts to acquire a lock, but the lock is owned by some other thread.

(e) (2 points) Give an example of an event that would cause a thread to undergo transition D (from Blocked to Ready).

Answer: a thread is blocked waiting on a condition variable, and some other thread notifies that condition variable.

Problem 3 (16 points)

(a) (4 points) You have been discussing your implementation of the Party class from Assignment 2 with a friend, and they suggest that instead of using a new thread for each incoming guest, it would be better to spawn a new child process for that guest. Would this work? Why or why not?

Answer: this would not work. Each process has a separate address space, so threads in different processes would not be able to access shared variables needed to implement the Party class.

(b) (4 points) In C++ making a copy of a condition variable is prohibited. For example, code like the following generates a compiler error:

```
std::condition_variable a;
std::condition_variable b;
...
b = a;
```

Why would it be a bad idea to copy a condition variable?

Answer: there might be threads waiting on a condition variable at the time it is copied: copying the condition variable would cause the threads to be queued on 2 different condition variables simultaneously, which does not make sense.

(c) (4 points) Suppose that a collection of threads shares access to a set of locks. One of the threads executes code that may hold any number of locks at once, acquiring them in an unpredictable order. The other threads may acquire any of the locks, but none of these threads ever holds more than one lock at a time. Can deadlock occur over this lock usage? Explain your answer.

Answer: deadlock could not occur. Suppose the first thread is holding one lock while waiting for another. The thread that holds the other lock will not attempt to acquire additional locks, so it can execute until it eventually releases the lock held by the first thread. For a deadlock to appear, each of the threads in the cycle must be holding one resource while trying to acquire another.

(d) (4 points) The journaling mechanism in Assignment 8 logs only metadata such as inodes; it does not log the data of regular files. Suppose you decided to log file data as well as metadata. Could you use the existing log entry types for this? If so, explain how. If not, explain why not and describe one or more new log entry types that would be needed to log file data.

Answer: yes, the `LogPatch` type could be used with an `offset_in_block` of 0 and a `length` of 512. The disadvantage of this approach is that it would consume a lot more log space (the log entry would need to contain 512 zero bytes).

Problem 4 (10 points)

Your disk needs to service this list of requested sectors: 15, 1567, 85, 778. In addition:

- The disk head is currently at sector 800.
- The requests arrived in the order listed (sector 15 arrived first); no additional requests arrive while these are being serviced.
- You can assume that the seek time from one sector to another is proportional to the difference between the sector numbers.
- For the scan algorithm, sectors are serviced in order from lower sector numbers to higher sector numbers.

(a) (6 points) For each disk scheduling algorithm (FIFO, shortest positioning time first, and scan), list the order in which the requests will be serviced.

Answer:

FIFO: 15, 1567, 85, 778

SPTF: 778, 85, 15, 1567

SCAN: 1567, 15, 85, 778

(b) (4 points) Which disk scheduling algorithm will result in the shortest total seek time? Explain your answer.

Answer: FIFO will seek for a total of 4512 sectors, SPTF for a total of 2337 sectors, and SCAN for total of 3082 sectors. Thus SPTF has the shortest total seek time.

Problem 5 (6 points)

(a) (2 points) What is the difference between deadlock and starvation?

Answer: deadlock occurs when each in a collection of threads is waiting for a resource owned by some other thread in the collection, so none make any progress. Starvation is when some threads make progress, but their progress makes it impossible for one or more other threads (the starving ones) to make progress.

(b) (2 points) Describe (briefly) a situation that results in deadlock.

Answer: thread 1 owns lock A and is attempting to acquire lock B. Thread 2 owns lock B and is attempting to acquire lock A.

(c) (2 points) Describe (briefly) a situation that results in starvation.

Answer: a system with one core uses a FIFO scheduler, and there is a thread that is completely CPU-bound and will run for days. Once this thread starts running on the core, no other thread will be able to run until the CPU-bound thread completes.

Problem 6 (9 points)

LRU (least recently used) was introduced in class as a potential page replacement policy, but it is not practical to implement in practice. The clock algorithm was then introduced as an easier-to-implement approximation to LRU. Consider each of the following modifications to the clock algorithm: if that modification were made independently, would the clock algorithm behave more or less like LRU? Explain your answers briefly.

(a) (3 points) Eliminate reference bits entirely (i.e. the algorithm behaves as if reference bits are always cleared for every page).

Answer: the algorithm will behave less like LRU than the original clock algorithm. Without reference bits, the algorithm will not be able to consider usage at all, so it will behave the same as FIFO.

(b) (3 points) Whenever the algorithm reaches physical page 0, it switches direction (first it sweeps pages in increasing order from 0 to N, then it sweeps back from N-1 to 0, then up from 1 to N, and so on).

Answer: the algorithm will behave less like LRU than the original clock algorithm. When the algorithm switches direction, it will sweep pages that it just swept a few moments ago, so they may be replaced even if they have only been idle for a very brief period, and even if there are pages at the other side of the clock that have been idle much longer.

(c) (3 points) Instead of replacing a page immediately when its reference bit is zero, the algorithm only replaces a page if the reference bit was zero on 3 consecutive scans of the page.

Answer: the algorithm will behave more like LRU than the original clock algorithm. The new algorithm will be able to distinguish between pages that have been idle for one complete scan, two complete scans, and so on, so it is more likely to find the true least recently used page to replace.

Problem 7 (8 points)

For each of the following pieces of information, which are stored in page map entries, indicate whether the item must be included in TLB entries, and explain why or why not.

(a) The physical page number for the page.

Answer: yes. Without this information it wouldn't be possible to compute physical addresses during references to the page.

(b) The "present" bit, which indicates whether the page is currently present in physical memory.

Answer: no. If a page isn't present, then there is no need for it to be present in the TLB at all.

(c) The "kernel" bit, which allows access to the page only when in kernel mode.

Answer: yes. If this bit were not present, then the TLB would have to be flushed at the beginning and end of each kernel call.

(d) The "read-only" bit, which either allows or disallows writes to the page.

Answer: yes. If this bit were not present, then once a page was loaded into the TLB it wouldn't be possible to trap write accesses to read-only pages.

Problem 8 (6 points)

You are developing a software package that uses another package developed separately by someone else. Your package will contain a file that is actually a link to a file `libnet.a` in the other package and you will use this link when compiling your package. You know that the other developer will occasionally delete `libnet.a` and replace it with a newer version.

(a) Suppose that you want your link to automatically refer to the newest version of `libnet.a` Should you use a hard link or a symbolic link? Explain your answer briefly.

Answer: a symbolic link. A symbolic link includes the textual name of the target file, so if the file is deleted and rewritten, the symbolic link will refer to the new version.

(b) Now suppose that the other developer tends to introduce bugs in new versions of `libnet.a`. You have tested the current version and know that it works, and you want to keep using that same version, even if the developer "upgrades" the file. Should you use a hard link or a symbolic link? Explain your answer briefly.

Answer: a hard link. A hard link contains the inumber of the target file and increments its reference count. If the original link to the target file is deleted, the file contents will remain and the hard link will continue to refer to those old contents. When a new file is recreated with the same name as the original, the new version will use a different inumber.

Problem 9 (15 points)

For each of the following approaches to managing memory or storage, indicate whether the approach suffers from significant performance degradation as memory becomes nearly full. Explain each answer briefly. You may assume that the memory or storage doesn't ever fill completely.

(a) First-fit memory allocation

Answer: there will be performance degradation due to fragmentation. When allocating space for a new object, allocator will have to examine more and more elements in the furthest to find a hole that is large enough.

(b) Memory allocation based on garbage collection

Answer: there will be performance degradation. As memory utilization increases, the cost of each scan of the garbage collector will increase, since there will be more live objects to scan and copy, but the number of free bytes found will reduce. Because of this, the garbage collector will have to run more frequently.

(c) A paged virtual memory system

Answer: there will be no performance degradation. This is because all pages are the same size, so free pages can be stored in a simple linked list and each allocation can always return the first free page from the list.

(d) A file system that uses a bitmap to keep track of free blocks

Answer: there will be performance degradation. As the amount of free space on the disk drops, it will be necessary to scan farther and farther in the freemap to find a free block.

(e) A flash translation layer, which manages storage of a flash memory device

Answer: there will be performance degradation. Flash devices use a garbage collector to collect free space, and the overhead for that will increase in the same way as described in (b) above.

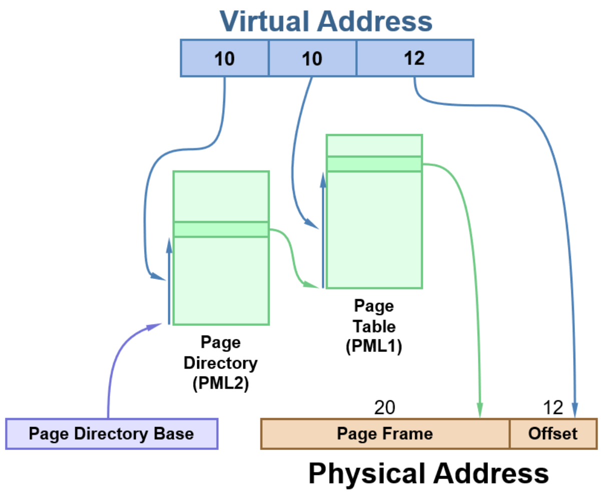
Problem 10 (3 points)

In 1-3 sentences, explain how crash recovery helps substitute some need to trust the file system.

Answer: without crash recovery, user would have to trust the file system to operate in such a way that there can never be inconsistencies after a crash. With crash recovery, users can trust the recovery mechanism to clean up problems, so they need not trust the file system to avoid all inconsistencies.

Problem 11 (24 points)

Intel has implemented several different virtual address translation mechanisms for the x86 processor. The x86-64 mechanism discussed in lecture is the latest version. The diagram below illustrates an earlier approach called x86-32:



(a) (3 points) How large are individual page map entries in this scheme (in bytes)? Explain your answer briefly.

Answer: each page map contains 212 (4096) bytes and 210 (1024) page map entries. Thus each page map entry is 22 (4) bytes in length.

(b) (3 points) How large is the Page Directory Base register, in bits? Explain your answer briefly.

Answer: we know that the Page Directory occupies one entire page of physical memory, so a 20-bit Page Frame number is sufficient to address it. Said another way, the physical address of the Page Directory contains 32 bits but the low-order 12 bits will always be zero, so there's no need to store them in the Page Directory Base.

(c) (5 points) With this mechanism, how many physical pages will be consumed by a process with one page of code, one page of data, one page of stack? Explain what each page will be used for.

Answer: 6 pages:

- **1 page for code**
- **1 page for data**
- **1 page for stack**
- **1 page for the PML1 for code and data (at the bottom of the virtual address space)**
- **1 page for the PML1 for the stack (at the top of the virtual address space)**
- **1 page for the PML2**

Problem 11, cont'd

(d) (5 points) How large would the stack have to grow before another page table would need to be created? You can express your answer as a power of 2.

Answer: 222 +1 bytes. The original PML1 holds 210 entries, each referring to a page of 212 bytes. All of this space would need to be occupied, as well as one additional byte to trigger the allocation of a second PML1.

(e) (4 points) Describe two advantages of this addressing scheme over the x86-64 scheme discussed in lecture.

Answer:

- **Page maps take up less memory, in part because page map entries are smaller and in part because there is no need for PML3s and PML4s.**
- **TLB misses can be serviced more quickly because they only need to consult 2 levels of page map.**

(f) (4 points) Describe two advantages of the x86-64 scheme over this one.

Answer:

- **The x86-64 scheme can support larger virtual address spaces, and hence larger programs, because virtual addresses are larger.**
- **The x86-64 scheme call support larger physical memories because physical addresses are larger.**

Problem 12 (40 points)

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

```
int find_indirect(const struct unixfilesystem *fs, int block_num)
```

The `block_num` argument is the number of a particular sector on disk. This function must determine whether `block_num` is used as an indirect block (but not a doubly-indirect block) in a file in the file system. If so, it will return the inumber of the file containing the block; if not it will return 0. If an error occurs, `find_indirect` will return -1.

Here are some additional details and simplifications:

- Your solution may invoke any of the functions defined for Assignment 7, and you may assume they have been implemented correctly.
- For this problem you do not need to print a message for each error condition; returning -1 is sufficient.

Answer: this question turned out to be much more difficult than we expected... sorry about that. The easiest way to solve this problem is to break it down into two pieces: (a) given an inode, check each of its indirect blocks (if any) to see if it is stored in the desired sector, and (b) iterate over all of the inodes in the file system.

Let's create a helper function that solves the first problem. We know that an inode doesn't have any indirect blocks unless its ILARG bit is set. If the file is large, then indirect blocks appear in two places. First, they can appear in the inode itself (all block pointers but the last will be indirect blocks). Second, if there is a doubly-indirect block pointer, then each of the pointers in the doubly-indirect block refers to an indirect block. Note: this solution takes advantage of the fact that unused block pointers in inodes and indirect blocks are always zero, so we can consider all available block pointers, even if they are logically beyond the end of the file. An alternative approach is to use the file size to stop scanning when we reach the last indirect block in the file (that requires a bit more code).

```
#define BLOCKS_IN_INODE 8
#define INODES_PER_BLOCK (DISKIMG_SECTOR_SIZE/sizeof(struct inode))
#define BLOCK_NUMS_PER_INDIRECT (DISKIMG_SECTOR_SIZE/sizeof(uint16_t))

// Returns 1 if block_num is an indirect block in the given inode, 0
// if it is not, and -1 if a disk error occurred.

int inode_find_indirect(const struct unixfilesystem *fs,
        struct inode *inp, int block_num)
{
    if (!(inp->i_mode & ILARG)) {
        return 0;
    }

    // See if one of the indirect block pointers stored in the inode
    // matches the desired block.
    for (i = 0; i < BLOCKS_IN_INODE-1; i++) {
        if (inp->i_addr[i] == block_num) {
            return 1;
        }
    }

    // Read in the doubly-indirect block (if there is one), then check
    // to see if the one of the indirect block numbers stored in it matches
    // the desired block.
    int doubly_ind = inp->i_addr[BLOCKS_IN_INODE-1];
    if (doubly_ind == 0) {
        return 0;
    }
    uint16_t indirect_blocks[BLOCK_NUMS_PER_INDIRECT];
    if (!diskimg_readsector(fs->dfd, doubly_ind, indirect_blocks)) {
        return -1;
    }
    for (i = 0; i < BLOCK_NUMS_PER_INDIRECT; i++) {
        if (indirect_blocks[i] == block_num) {
            return 1;
        }
    }    return 0;
}
```

To find all of the inodes, all we need to do is figure out how many inodes there are, then use inode_get to read each of them in order. Some inodes are not currently in use (this is indicated by the IALLOC bit), so we need to skip those.

```
int find_indirect(const struct unixfilesystem *fs, int block_num)
{
    int num_inodes = fs->superblock.s_isize * INODES_PER_BLOCK;
    int i;

    for (i = 1; i <= num_inodes; i++) {
        struct inode inode;
        if (inode_get(fs, i, &inode) < 0) {
            return -1;
        }
        if (!(inode.i_mode & IALLOC)) {
            // This inode isn't currently in use.
            continue;
        }
        int result = inode_find_indirect(fs, &inode, block_num);
        if (result < 0) {
            return -1;
        } else if (result > 0) {
            return i;
        }
    }
    return 0;
}
```