CS 140 Final Examination
Spring Quarter, 2018
Solutions

You have 3 hours (180 minutes) for this examination; the number of points for each question indicates roughly how many minutes you should spend on 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 printed versions of the files `inode.c`, `directory.c`, and `filesys.c` from your Pintos Project 4 solution. Other than these items, 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.

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!)

____
(email id, for sending score)

Problem	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	Total
Score												
Max	12	4	6	10	9	20	16	20	10	30	30	167

Problem 1 (12 points)

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

(a) If a system has lots of runnable threads, then thrashing isn't a problem: while one thread waits for a page fault, another thread can run, so all of the processors stay fully utilized.

False: if there isn't enough memory, then the next thread to run will also generate a page fault, and the next one after that, until eventually every thread is waiting for page faults. Running more threads actually makes the problem worse, because more threads will consume more memory.

(b) A multi-core implementation of locks must use both busy-waiting and interrupt disabling.

True: busy-waiting is needed to lock out other cores, and interrupt disabling is needed to prevent timer interrupts, which could cause another thread to run on the current core.

(c) Independent threads rarely exist in modern computer systems.

True: an independent thread does not share state with any other thread, but almost all real threads share state, such as a console, files in the file system, etc.

(d) In a stack-based storage allocator, there is no fragmentation.

True: in stack-based storage allocation, memory is freed in the reverse order from allocation, so all of the free space is in a single large block just after the current end of the stack.

(e) The dispatcher for a core decides on the scheduling priority for each thread that runs on that core.

False: the dispatcher is responsible for choosing threads to run in a way that is consistent with the current priorities, but it does not choose the priorities; they are computed elsewhere in the system.

(f) If a memory allocator returns fixed-sized regions, then there will be no external fragmentation

True: any object can fit in any "hole". Of course, there may be internal allocation if the fixed size doesn't match the actual needs of objects.

Problem 2 (4 points)

In Pintos, there are some physical memory locations that the operating system can access using multiple different virtual addresses (``aliasing''). Describe one such example: what is the value, and how many different virtual addresses can be used to access it? Describe each virtual address as precisely as possible.

Answer: the kernel can access any memory location belonging to the current thread (i.e., accessible by the thread when it is running in user mode) that is currently in memory in two ways:

- **Using the virtual address of the value (between 0 and PHYS_BASE).**
- **By computing the physical address of the value and then adding that to PHYS_BASE (all of physical memory is mapped into contiguous kernel virtual addresses starting at PHYS_BASE).**

Problem 3 (6 points)

In the list of operations below, circle the ones that are (virtually) atomic:

Write a single-word variable
ATOMIC

Create a new file with a particular name in Pintos
ATOMIC

Copy a 1 KB block of memory within a single page
NOT ATOMIC

Copy a 1 KB block of memory where the source and/or destination span(s) a page boundary
NOT ATOMIC

Acquire a lock
ATOMIC

Copy the contents of one file to another file
NOT ATOMIC

Problem 4 (10 points)

Suppose that an application attempts to open the file `x/y/z`. Suppose also that:

- The application's current working directory is `/a/b/c/d`
- `/a/b/c/d/x` is a symbolic link whose contents are `/a/q`

List all of the disk blocks that must be accessed in order to open `x/y/z` and read its first block. For each block, give the file/directory it is associated with and the type of the block, such as "inode" or "first data block". You may assume that all of the directories and files needed to access `x/y/z` exist and that `/a/b/c/d/x` is the only symbolic link. Furthermore, all directories are small enough for their entries to fit in a single disk block.

1. Inode for `/a/b/c/d`
2. Data block for `/a/b/c/d` (find entry for "`x`")
3. Inode for `/a/b/c/d/x`
4. Data block for `/a/b/c/d/x` (read symbolic link value, which is "`/a/q`".
5. Inode for root directory
6. Data block for root directory (find entry for "`a`").
7. Inode for `/a`
8. Data block for `/a` (find entry for "`q`")
9. Inode for `/a/q`
10. Data block for `/a/q` (find entry for "`y`")
11. Inode for `/a/q/y`
12. Data block for `/a/q/y` (find entry for "`z`")
13. Inode for `/a/q/y/z`
14. First data block for `/a/q/y/z`.

Problem 5 (9 points)

Answer each of the following questions briefly.

(a) (3 points) Why is the "trim" command needed for flash devices?

Answer: it informs the device that the block is no longer needed. Without this command, the device cannot tell that the OS no longer needs a block, so the device cannot garbage collect it.

(b) (3 points) In multi-core processors, there is typically a separate ready queue for each core. Describe one advantage of this approach, in comparison to a single shared ready queue, and one disadvantage.

Answer: separate ready queues eliminate lock contention for a single shared queue, which will be problematic if there are more than a few cores; they also make it easy to implement core affinities. However, separate ready queues make load-balancing difficult: what happens if there are several ready threads, but they are all in the same ready queue? We'd rather not have idle cores if there are runnable threads.

(c) (3 points) Linkers operate by making multiple passes over all of the object files. Why are multiple passes necessary?

Answer: it takes one pass to collect the size of all of the sections in all of the object files. Until this is done, the linker can't figure out where each symbol will be located, and this information is needed when the linker makes its final pass through all of the object files to touch up addresses and write the executable file.

Problem 6 (20 points)

Suppose that each of the changes below is made to a file system. Indicate whether each change affects the performance, consistency, and/or durability of the file system. For each of the properties affected by a change, indicate whether the change improves or degrades that property, and briefly describe why.

(a) Starting from a file system with no cache, a block cache with synchronous writes is added.

Answer: the performance of reads will improve because of the cache; write performance won't change, since writes are still synchronous to disk. Consistency and durability will not change, since writes are still synchronous.

(b) Starting from a block cache with synchronous writes, a delayed-write mechanism is added.

Answer: the performance of writes will improve because of the delayed-write mechanism (e.g. applications won't have to wait for synchronous writes, and repeated writes to the same block will be coalesced). However, durability will be degraded, since some information could be lost in a crash. Consistency will also be degraded: if an operation affects multiple blocks, such as creating a new file, some of the changes may be reflected on disk while others are lost.

(c) Starting from a block cache with delayed writes, an ordered-write mechanism is added for the delayed writes (for both metadata and data).

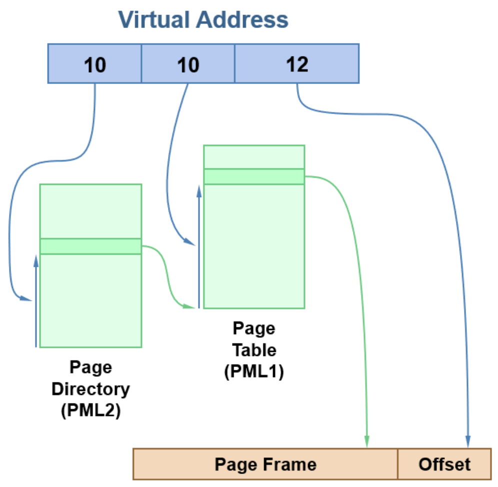
Answer: performance will drop slightly because of the ordered-write mechanism (blocks that might not have been written before the change may have to be written because of the ordering constraints). Durability will not change significantly: writes are still delayed, so information can still be lost after a crash. Consistency will be improved: the ordering mechanism will ensure that certain kinds of inconsistencies cannot occur. However, ordered writes cannot prevent all inconsistencies, so there can still be inconsistencies after crashes (typically, the ordering is chosen so that the remaining inconsistencies are relatively benign).

(d) Starting from a block cache with delayed writes, a write-ahead log is added. Both metadata and data are written to the log, and the log entry for an operation is flushed to disk before the operation modifies any blocks in the cache.

Answer: performance will drop significantly because each write now happens twice: once to the log and once to the normal location on disk. This means, for example, that write throughput will drop by about a factor of 2x. Durability will improve significantly: all operations will be durable as soon as the log information is written, which is before the operation returns to the application. Consistency will be "perfect": no inconsistencies should occur after a crash if the log is written (and replayed) properly.

Problem 7 (16 points)

This question concerns the address translation mechanism shown below:



Each page table entry includes the following bits in addition to a physical page frame number: present, read-only, protected (only kernel can access), referenced, and dirty.

(a) (4 points) What is the largest amount of physical memory that this scheme can support (express your answer as a power of 2)? What is the reason for this limit?

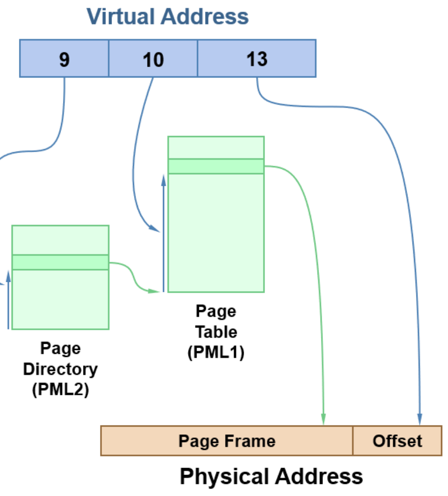
Answer: physical memory is limited by the size of the "Page Frame" field, which is limited by the size of page table entries. There are 1024 entries in each page table, and each page table occupies 4KB, so each entry must be 4 bytes (32 bits) of which 5 bits are reserved for the fields described above. This leaves 27 bits for the Page Frame; combined with the 12-bit offset, this allows for physical memory sizes up to 2^{39} bytes.

(b) (12 points) Suppose that the mechanism had to be modified to support larger physical memories. Describe two different ways you could do this; for each approach, draw the address translation mechanism for that scheme.

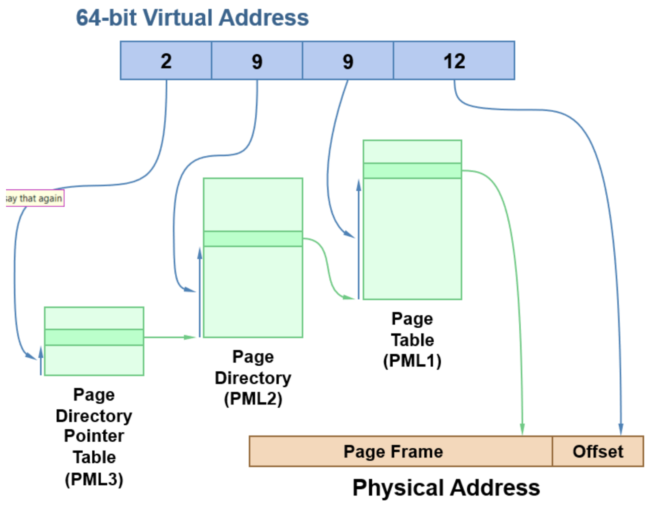
Answer: see next page.

Solution for Problem 7(b):

Approach #1: increase the page size by a factor of two, to 8192 bytes. Each page table will still contain 1024 entries, so the entries will be 8 bytes long and Page Frames can be as large as 59 bits. Note: the PML2 tables will only have 512 entries, so they will only be 4 KB in size.



Approach #2: add another level of mapping, so at most 9 bits are used to index into each page map. This allows page map entries to be 8 bytes long, which will provide more bits for the Page Frames. Note: the PML3 tables will only have 4 entries (32 bytes).



Problem 8 (20 points)

A friend of yours wrote the following function: given two accounts, it should transfer a given amount from the larger account to the smaller one, if the larger account has sufficient funds. It uses the Pintos locking mechanism to synchronize concurrent invocations.

```
1  struct account {
2      struct lock lock;
3      double balance;
4  }
5
6  void transferFromLarger(struct account *a1, struct account *a2,
7          double amount) {
8      if ((a1->balance > a2->balance && (a1->balance >= amount)) {
9              lock_acquire(&a1->lock);
10             a1->balance -= amount;
11             lock_release(&a1->lock);
12             lock_acquire(&a2->lock);
13             a2->balance += amount;
14             lock_release(&a2->lock);
15         }
16     } else if ((a2->balance > a1->balance) && (a2->balance >= amount)) {
17             lock_acquire(&a2->lock);
18             a2->balance -= amount;
19             lock_release(&a2->lock);
20             lock_acquire(&a1->lock);
21             a1->balance += amount;
22             lock_release(&a1->lock);
23         }
24     }
25 }
```

(a) (5 points) Unfortunately, this code does not always behave correctly when invoked concurrently by different threads. Describe a sequence of events that causes the code to misbehave.

Answer: The problem is that the locks aren't acquired until after the balances have been checked. Suppose that a1 has a balance of 150 and a2 has a balance of 100. Thread T1 invokes `transferFromLarger(a1, a2, 100)` and executes through line 8. Then thread T2 invokes `transferFromLarger(a1, a2, 100)` and runs to completion. Finally, T1 wakes up and finishes. This will cause a1 to end up with a negative balance.

(b) (15 points) Use the next page to rewrite `transferFromLarger` so that it always behaves correctly.

In order to fix the problems, both account locks need to be held at the same time, starting from before the balance checks until after both balances have been updated. Acquiring two different locks could potentially result in deadlock; to avoid this, the locks must be acquired in the same order.

Use this page for your solution to Problem 8(b):

```
void transferFromLarger(struct account *a1, struct account *a2,
           double amount) {
    if (a1 < a2) {
        lock_acquire(&a1->lock);
        lock_acquire(&a2->lock);
    } else {
        lock_acquire(&a2->lock);
        lock_acquire(&a1->lock);
    }
    if ((a1->balance > a2->balance && (a1->balance >= amount)) {
        a1->balance -= amount;
        a2->balance += amount;
    } else if ((a2->balance > a1->balance) && (a2->balance >= amount)) {
        a2->balance -= amount;
        a1->balance += amount;
    }
    lock_release(&a1->lock);
    lock_release(&a2->lock);
}
```

Problem 9 (10 points)

Describe a program that calls `malloc` and `free` in a way that requires at least 1200 bytes of heap space, even though the program never uses more than 1000 bytes of memory at a time. You don't need to write actual code, but your description must be clear and precise. Your program must work for any implementation of `malloc`; it can test the pointers returned by `malloc` if that helps.

One possible answer:

1. **Allocate 5 blocks of 200 bytes each.**
2. **Examine the addresses of the allocated blocks, and free the two blocks with the second-smallest and second-largest starting addresses.**
3. **Allocate a block with 400 bytes.**

Problem 10 (30 points)

In this problem you must define a mechanism for implementing an append-only log in flash memory:

- Your solution should work with the raw flash hardware described in class (no FTL). Read and write operations operate on entire pages. A write operation results in a logical AND with the current page on flash (i.e. a zero bit in the data being written causes the corresponding bit on flash to become zero, but one bits in the data being written have no effect).
- You may assume that individual write operations are atomic.
- You may assume the existence of a function that returns a clean page (all bits 1's); pages won't be returned in any particular order.
- The log will consist of variable-length entries; most will be small, but entries can be larger than a page. The contents of each entry are defined outside your mechanism; your job is to preserve them on flash.
- You must pack multiple small entries within a page.
- Each log entry must be written to flash immediately, so that its contents are durable.
- Crashes must not result in loss of data or inconsistency (after a crash it must be possible to read the log entries in order from beginning to end without any garbled or lost entries). You may assume the flash device itself does not fail.
- Your solution must not require existing log pages to be erased.
- You don't need to worry about deleting old log entries or truncating the log.

Describe (a) how each log entry is represented in flash storage, (b) how new log entries are created, and (c) how to read the log from start to finish after a crash. You do not need to write code, but you must describe the mechanism precisely enough for someone else to write the code without much thought.

One possible answer:

- **Each block of the log will be gradually filled in from beginning to end. If a block is partially filled, then it contains one or more valid log entries at the beginning, followed by all ones after that.**
- **Each log entry consists of a 4-byte length, followed by that many bytes of data for the entry. Log entries are not allowed to have lengths consisting of all ones (if the length is treated as a signed integer, this would be a negative value, which doesn't make sense anyway). Thus, the end of the log is indicated by a length field of all ones.**
- **To write log entries starting from a clean page, create a page-size buffer in memory and set it to entirely ones. Then write the length of the new entry into the buffer, followed by the entry contents. Write this buffer to the desired page of flash memory. To add each successive entry to the log, write the length and contents of the new entry into the buffer in memory, then write the buffer to the flash page. The first part of the flash page will get overwritten with the same contents it already holds, so nothing will change. The last part of the flash page is written with ones, so its contents won't change either. Only the portion of the page corresponding to the new log entry will change.**
- **The log will span multiple pages, which are linked together. The last eight bytes of each page hold a link containing the number of the next page in the log, or all ones if the page is currently the last page in the log. When the last page of the log fills, allocate a new page and write its page**

number into the link at the end of the previous page in the log. For the scheme to work, no page in the log may have a page number that is all ones, since that number indicates "end of log".

- **The page number of the first page of the log must be stored in a well-known place in flash memory; alternatively, the first block of the log could be in a well-known place, such as block 0.**
- **If a new log entry does not fit in the current page, then spread its contents across multiple pages. First, allocate enough additional pages in the log to hold the new entry, and set link fields to incorporate these pages into the log. Then write data into all of the pages except the one containing the length field: until the length field is written, the new entry is not considered part of the log. These pages may be written in any order. Finally, write to the page containing the length and first part of the entry. Since page writes are atomic, this atomically "commits" the entire log entry.**
- **To read back the log, find the first page of the log by looking in the well-known place described above. Then, loop through the entries one at a time, until an all-ones length is found. If the length of an entry extends past the end of the current page, then follow the link to the next page for the remainder of the log entry. A single entry can span any number of pages this way.**
- **This approach leaves one potential inconsistency. It is possible for a crash to occur while writing a log entry that spans multiple pages. The crash could occur after part of the log entry has been written, but before its length has been written. This entry will be ignored when replaying the log. However, if we want to continue appending new entries to the log after the crash, we need to find a way to "skip over" the partially-written entry, which is garbage. One way to do this is to include an extra bit in each log entry that indicates whether the entry is valid or garbage (one of the bits of the length field could be used for this, assuming there are enough bits remaining for the largest allowable log entry). When starting up, the system can read to the end of the log and see if there are any zero bits in the portion of the log after the last valid entry. If so, create a garbage entry large enough to cover all of the zeroes. Append new entries after this garbage entry. In the future, when replaying the log, the system can skip over the garbage entries. Note: you do not have to address this potential inconsistency in order to receive full credit for the question.**

Problem 11 (30 points)

Given your solution to Pintos Project 4, write a new function for `filesys.c` that moves an existing file from one directory to another. The function must have the following signature:

```
bool move_file(char *src_dir_name, char *dst_dir_name, char *name)
```

The first two arguments are the path names of the source and destination directories, and the third argument is the name within `src_dir_name` of the file to move. The return value of the function is true if it succeeded and false if there was a problem. You may use any of the functions in your Project 4 solution in your implementation of `move_file`.

```
bool move_file(char *dst_dir_name, char *src_dir_name, char *name) {
    bool success = true;
    struct dir* src_dir = dir_open(src_dir_name);
    if (src_dir == NULL) {
        return false;
    }
    struct dir* dst_dir = dir_open(dst_dir_name);
    if (dst_dir == NULL) {
        dir_close(src_dir);
        return false;
    }
    struct inode *src_inode = dir_get_inode(src_dir);
    struct inode *dst_inode = dir_get_inode(dst_dir);
    if (inode_get_inumber(src_inode) == inode_get_inumber(dst_inode)) {
        /* Moving file on top of itself; nothing to do here. */
        dir_close(src_dir);
        dir_close(dst_dir);
        return true;
    }

    /* Lock directories in order of inumber to prevent deadlock.
     * Must use inumbers, not names, since multiple names could be
     * used to refer to the same directory. */
    if (inode_get_inumber(src_inode) < inode_get_inumber(dst_inode)) {
        inode_lock(src_inode);
        inode_lock(dst_inode);
    } else {
        inode_lock(dst_inode);
        inode_lock(src_inode);
    }
    struct inode* file_to_move = NULL;
    if (!dir_lookup(src_dir, name, &file_to_move)) {
        success = false;
        goto done;
    }
    if (!dir_add(dst_dir, name, inode_get_inumber(file_to_move)) {
        success = false;
        goto done;
    }
    success = dir_remove(src_dir, name);

  done:
    if (file_to_move != NULL) {
        inode_close(file_to_move);
    }
    inode_unlock(src_inode);
    inode_close(src_inode);
    inode_unlock(dst_inode);
    inode_close(dst_inode);
    dir_close(src_dir);
    dir_close(dst_dir);
    return success;
}
```