CS 140 Final Examination
Spring Quarter, 2018

Although this exam was used for CS 140, most of its questions are still relevant for CS 111. Questions involving the Pintos operating system are out of scope for CS 111.

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.

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

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

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

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

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

Problem 2 (4 points)

This question is out-of-scope for CS 111.

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.

Problem 3 (6 points)

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

Write a single-word variable

Create a new file with a particular name in Pintos *(out of scope for CS 111)*

Copy a 1 KB block of memory within a single page

Copy a 1 KB block of memory where the source and/or destination span(s) a page boundary

Acquire a lock

Copy the contents of one file to another file

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.

Problem 5 (9 points)

Answer each of the following questions briefly.

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

(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.

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

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.

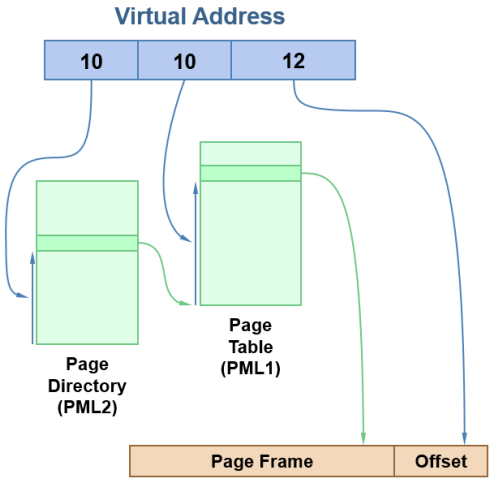
(b) Starting from a block cache with synchronous writes, a delayed-write mechanism is added.

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

(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.

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?

(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.

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     } else if ((a2->balance > a1->balance) && (a2->balance >= amount)) {
16             lock_acquire(&a2->lock);
17             a2->balance -= amount;
18             lock_release(&a2->lock);
19             lock_acquire(&a1->lock);
20             a1->balance += amount;
21             lock_release(&a1->lock);
22     }
23 }
```

(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.

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

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

```
void transferFromLarger(struct account *a1, struct account *a2,
           double amount) {
```

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.

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.

Additional working space for Problem 10

Problem 11 (30 points)

This question is out of scope for CS 111.

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, char *dst_dir, 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` of the file to move. The return value of the function is true if it succeeded and false if there was a problem. You can use any of the functions in your Project 4 solution in your implementation of `move_file`.