CS 140 Final Examination
Winter Quarter, 2012
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 two double-sided pages of notes that you have prepared ahead of time; other than that, you may not consult books, notes, or your laptop. 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 two 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	10	10	40	14	8	6	16	14	30	5	165

Problem 1 (12 points)

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

(a) If a memory allocation mechanism uses a compacting garbage collector for reclamation then it does not experience fragmentation

Answer: false. Between garbage collections free space will build up, which cannot be reclaimed until the next time the garbage collector runs. This is a form of fragmentation.

(b) The dispatcher is responsible for setting thread priorities.

Answer: false. The dispatcher is only responsible for switching threads; the scheduler determines the thread priorities, based on the system's scheduling algorithms.

(c) If a thread is CPU-bound, it makes sense to give it higher priority for disk I/O than an I/O-bound thread.

Answer: true, for the same reason that I/O-bound threads should get higher priority for the CPU. If a CPU-bound thread must wait for I/O behind I/O-bound threads, we could end up in a situation where all of the threads are waiting for I/O and the CPU is idle. Better to finish the I/O for the CPU-bound threads as quickly as possible so they can get keep the CPU busy; this increases our chances of keeping all of the system components busy at all times.

(d) Virtual addresses must be same size as physical addresses.

Answer: false. The size of physical addresses is determined by the width of page table entries, so it can be either larger or smaller than virtual addresses.

(e) Page offsets in virtual addresses must be the same size as page offsets in physical addresses.

Answer: true. The offset is passed directly from the virtual address to the physical address without any mapping or modification, so its size cannot change.

(f) Implementing processor affinity in a multiprocessor scheduler is likely to reduce the number of misses in caches such as translation lookaside buffers.

Answer: true. If a blocked process resumes execution on the same processor it was using previously, there's a good chance that some information for that process is still in caches.

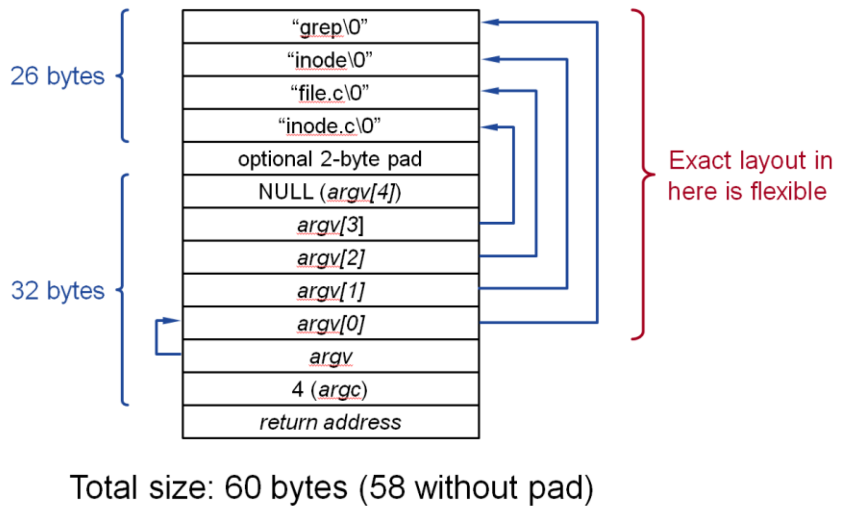
Problem 2 (10 points)

Suppose the `grep` program is invoked under Pintos with the following command line:

```
grep inode file.c inode.c
```

Using a diagram for illustration, explain what the call stack will look like when the program's `main` function is invoked. How much total memory will be needed to set up this stack?

Answer:



Problem 3 (10 points)

(a) If a normal access to memory takes 100 ns (100×10^{-9} sec) and reading a page from disk takes 10 ms (10×10^{-3} sec), what is the average memory access time (including page fault overhead) if page faults occur in 0.1% of the memory references? You may assume that there is no additional overhead due to TLB misses.

Answer: Average Time = 0.1%*10ms + 99.9%*100ns = 10µs + 99.9ns ≈ 10µs

(b) What is the maximum allowable page fault rate if performance degradation is to be no more than 10% (i.e. average memory access time ≤ 110 ns)?

Answer:

Average access time = PFR*10ms + (1-PFR)*100ns = 110ns

PFR ≈ 10^{-6}

Problem 4 (40 points)

Suppose that you are given the task of adding **hard** links to Pintos. You must implement a new system call that programs can use to create hard links:

```
bool ln(const char *target, const char *link)
```

The `target` argument contains the path name of an existing file or directory, and `link` contains the path name where a new hard link should be created. After the `ln` system call returns, `link` should refer to the same file or directory as `target`. The system call returns true for success and false for failure. Once created, hard links can be deleted using the `remove` system call that you implemented in Pintos Project 2.

(a) (15 points) Describe the changes you would need to make to Pintos to implement hard links, assuming that you are starting from your solution to Pintos Project 4. You do not need to write actual code, but your answer must be precise enough for us to understand exactly what changes are required. Be sure to describe any changes to data structures, as well as code changes.

Answer: First, we must add a new field `int ref_count` to `struct inode_disk`, which represents the number of active hard links that refer to this file.

Then we must modify the remove syscall so that after he moves a directory entry for the file it documents the `ref_count` of that file. If the `ref_count` is now 0 it is safe to actually deallocate the file.

The create syscall should be modified so that it initializes `ref_count` to 1.

To actually implement `ln` we must first check if the new link would create a circular reference and disallow attempts to do so. Then, we can add a directory entry for the base name of `link` in the directory specified by `link` and set the inode # in the directory entry to that of the file specified by `target`. Finally, we increment the `ref_count` of target.

Problem 4, cont'd

(b) (10 points) If a system crash occurs during or shortly after a call to `ln` or `remove`, it is possible that some of the on-disk structures may be left in an inconsistent state, which could impact the operation of the system after it restarts. Describe all of the possible inconsistencies that relate to hard links (i.e., information that is manipulated by the `ln` system call). You do not need to consider inconsistencies unrelated to hard links, such as the allocation of disk blocks to files or directories.

Answer: the `ref_count` of the file could be either greater than or less than the number of directory entries that refer to the file. In the first case, this could eventually lead to inode leaks; the second case could eventually to a dangling pointer (a directory entry that refers to an unallocated inode or, worse, an inode that has been reallocated for a different file).

(c) (15 points) Suppose you were asked to implement journaling in Pintos in order to repair inconsistencies caused by system crashes. When `ln` or `remove` is invoked, one or more log records will be written before each operation is carried out. During the next system restart, these log records will be processed to repair inconsistencies. Describe the log record(s) that must be written by `ln` and/or `remove` to repair the inconsistencies described in (b) above, and how the information in these records should be used during crash recovery. Assume that hard links are implemented as described in your answer to part (a) above. You do not need to write code, but make sure your answer is precise and complete (use pseudo-code if that's helpful).

Answer:

- **Before starting an `ln` syscall log the operation name ("link"), the inumber of the directory in which the link will be created, the name of the link file, the inumber of the target file (the file being linked to), and the new value for the reference count of the target file.**
- **Before starting a `remove` syscall, log the operation name ("remove"), the inumber of the directory containing the link to be removed, the name of the file being removed, the inumber of the file, and the new value for that file's `ref_count` after the remove completes.**
- **During recovery, for each "link" operation, create the link file if it doesn't already exist and update the `ref_count` of the target file. For each "remove" operation, remove the link (if it hasn't already been removed) and update the reference count of the target file. Note that these operations are idempotent!**

Problem 5 (14 points)

Suppose you are asked to design a new file system that will be used exclusively for storing and playing videos on the YouTube Web site.
(a) (4 points) Describe the access patterns that you expect to be most common in this file system.

Answer:

- **Sequential reads and writes**
- **Very large file sizes**
- **Reads much more common than writes**

(b) (4 points) You are given the choice of using flash memory instead of hard disks to store and serve the video data. How would you choose between these two alternatives?

Answer: Here are some considerations:

- **Cost: disk is much cheaper per byte than flash, and we will need a lot of storage**
- **Access patterns consist of large sequential reads and writes, so the I/O device bandwidth matters a lot more than latency (access time). This means that the slow seeks of disks will not be an issue, so the best device from an access standpoint is the one that provides the cheapest bandwidth.**
- **On the other hand, if most videos are not accessed frequently, then storage capacity will be the primary factor that determines the size of the system. It's possible that we will already need so many devices just to store all the videos that there will be plenty of access bandwidth with either disk or flash.**

(c) (6 points) How would you expect the design of this file system to differ from the organizations described in class and/or implemented in Pintos, and why?

Some possible answers:

- **Larger blocks would make sense (e.g. 1MB or more) since all files are large.**
- **Prefetch definitely makes sense, since files are almost always accessed sequentially.**
- **Caching probably doesn't make sense on a block-by-block basis: Once a block is read, it's unlikely to be read again soon, since all accesses are sequential. On the other hand, caching might make sense on a per-file basis, since there will be "hot" videos that are accessed a lot. The caching mechanism should try to identify these hot videos.**

Problem 6 (8 points)

Suppose a friend told you: "If a system has lots of runnable threads, it can use them to hide the cost of page faults." Explain whether your friend is right, wrong, or both.

Answer: both. If the working sets of all the runnable threads fit in memory, then while one thread is waiting for a page fault another thread can execute, effectively hiding the cost of page faults. However, if the working sets do not fit in memory, then the system will quickly end up in a state where all threads are waiting for page faults; as soon as a thread resumes after a page fault, it will touch another page that is not in memory and generate another page fault to wait for. The more runnable threads there are, the worst things will get.

Problem 7 (6 points)

Below is one of the attempts presented in lecture for the “Too Much Milk” problem (note: the `buyMilk` function will increment the `milk` variable):

Thread A

```
1  noteA = 1;
2  if (noteB == 0) {
3      if (milk == 0) {
4          buyMilk();
5      }
6  }
7  noteA = 0;
```

Thread B

```
1  noteB = 1;
2  if (noteA == 0) {
3      if (milk == 0) {
4          buyMilk();
5      }
6  }
7  noteB = 0;
```

Unfortunately this approach suffers from starvation. A friend suggests the following code in order to fix the starvation problem:

Thread A

```
1  noteA = 1;
2  if (noteB == 0) {
3      if (milk == 0) {
4          buyMilk();
5      }
6  }
7  noteA = 0;
```

Thread B

```
1  if (noteA == 0) {
2      noteB = 1;
3      if (milk == 0) {
4          buyMilk();
5      }
6      noteB = 0;
7  }
```

Does your friend’s approach solve the “Too Much Milk” problem? Explain your answer.

Answer: no. Consider the following execution sequence:

B1 A1 A2 A3 B2 B3 B4 A4

The result is that both threads buy milk.

Problem 8 (16 points) Short answers

(a) (3 points) If the page size is doubled, does the size of the working set for a process increase, decrease, or stay the same? Explain your answer.

Answer: it increases, due to internal fragmentation

(b) (3 points) What causes a thread's state to change from "running" to "ready"?

Possible answers:

- **When a thread voluntarily yields**
- **When a thread's time slice expires**
- **When priorities change (or a higher priority thread unblocks) so that a thread no longer has highest priority**

(c) (3 points) Briefly describe one advantage and one disadvantage of the UNIX `fork`-`exec` style of process creation, in comparison to the Windows `CreateProcess` style.

Advantages: simple; easy to personalize state of child process and add new forms of personalization.

Disadvantages: could be expensive to copy all of the state of the parent process during `fork`, just to throw almost all of it away in the subsequent `exec`; two system calls instead of one.

(d) (4 points) Describe two situations in which busy-waiting is appropriate (and indicate why busy-waiting is appropriate in each situation).

Possible answers:

- **Implementing locks on multiprocessors**
- **Very short critical sections**
- **Tightly interacting threads**
- **Ultra-low latency I/O**

(e) (3 points) Of the four conditions for deadlock, which is the one most commonly eliminated in order to prevent deadlock? Why?

Answer: circularity of resource dependencies. This is typically the easiest condition to eliminate: all that's needed is to structure the lock acquisition hierarchically.

Problem 9 (14 points)

(a) (6 points) Alice, Bob, and Carol go to a Chinese restaurant at a busy time of the day. The waiter apologetically explains that the restaurant can provide only two pairs of chopsticks (for a total of four chopsticks) to be shared among the three people.

Alice proposes that all four chopsticks be placed in an empty glass at the center of the table and that each diner should obey the following protocol:

```
while (!had_enough_to_eat()) {
    acquire_one_chopstick();       /* May block. */
    acquire_one_chopstick();       /* May block. */
    eat();
    release_one_chopstick();       /* Does not block. */
    release_one_chopstick();       /* Does not block. */
}
```

Can this dining plan lead to deadlock? Explain your answer.

Answer: no. Deadlock cannot occur because there are enough chopsticks to guarantee that at least one of the 3 diners will be able to get the 2 chopsticks that he/she needs. Once that diner finishes, he/she will release their chopsticks for someone else to use, so eventually everyone finishes.

(b) (8 points) Suppose now that instead of three diners there will be an arbitrary number, D. Furthermore, each diner may require a different number of chopsticks to eat. For example, it is possible that one of the diners is an octopus, who for some reason refuses to begin eating before acquiring eight chopsticks. The second parameter of this scenario is C, the number of chopsticks that would simultaneously satisfy the needs of all diners at the table. For example, Alice, Bob, Carol, and one octopus would result in C = 14. Each diner's eating protocol will be as displayed below:

```
int s;
int num_sticks = my_chopstick_requirement();
while (!had_enough_to_eat()) {
    for (s = 0; s < num_sticks; ++s) {
        acquire_one_chopstick();     /* May block. */
    }
    eat();
    for (s = 0; s < num_sticks; ++s) {
        release_one_chopstick();     /* Does not block. */

    }
}
```

What is the smallest number of chopsticks (in terms of D and C) needed to ensure that deadlock cannot occur? Explain your answer.

Answer: C – D + 1. This guarantees that every diner can get all but one of the chopsticks it needs (C – D), with one additional chopstick to guarantee that at least one diner gets all of the chopsticks it needs.

Problem 10 (30 points)

You have been hired to help implement RoboDeli.com, a new automated delicatessen where both the customers and the servers are robots. Like all delis, RoboDeli.com uses a ticketing system and a “Now Serving” display. You must implement the ticketing mechanism in C by defining a structure `struct deli`, plus four functions described below.

The following function is invoked once to initialize the deli object:

```
void deli_init(struct deli *deli)
```

At the time this function is invoked there are no customers and no servers in the deli.

When a new customer arrives in a deli, it invokes the function

```
int deli_get_ticket(struct deli *deli)
```

This function will return the customer's ticket number, which is the smallest positive integer that has not been returned to any previous customer (1 for the first customer, 2 for the second, and so on). No two customers must ever receive the same ticket number. At this point the customer is free to look around the deli. Eventually, each customer will invoke the function

```
deli_wait_turn(struct deli *deli, int number)
```

where `number` is the value returned to the customer when it called `deli_get_ticket`. This function must not return until there is an available server and the “Now serving” display shows a number at least as high as `number`. Once `deli_wait_turn` returns, the customer will go to the counter for service (you do not need to implement that mechanism).

When a new server arrives, or when an existing server finishes with its current customer and is ready to serve a new customer, it will invoke the function

```
deli_wait_customer(struct deli *deli)
```

This function must increment the value displayed in the “Now Serving” display. It must not return until there is a customer ready for service (i.e. invocations of `deli_wait_customer` and `deli_wait_turn` should return in pairs at about the same time). Once this function returns the server will meet the customer and fill their order (you do not need to implement that mechanism).

- You must write your solution in C using the Pintos functions for locks and condition variables:
  ```
  lock_init (struct lock *lock)
  lock_acquire(struct lock *lock)
  lock_release(struct lock *lock)
  cond_init(struct condition *cond)
  cond_wait(struct condition *cond, struct lock *lock)
  cond_signal(struct condition *cond, struct lock *lock)
  cond_broadcast(struct condition *cond, struct lock *lock)
  ```
  Use only these functions (e.g., no semaphores or other synchronization primitives).
- You may not use more than one lock in each `struct deli`.
- You may assume that someone else has written a function `set_now_serving(int number)`, which will cause `number` to be displayed in the “Now Serving” display. Invoke this function as needed in your code.
- Your solution must not use busy-waiting.
- You do not need to worry about ticket numbers overflowing.
- You can assume that customers are always available when their number comes up (you do not need to implement a mechanism to skip over nonresponsive customers).

Use the space below to write a declaration for `struct deli` and the four functions described on the previous page. You do not need to write any other code for the customers or servers.

```
// Solution to deli ticket-machine synchronization problem
// (alternate solution where servers must wait for customers).

struct deli {
    int next_ticket;
    int now_serving;
    struct lock lock;
    struct condition server_available;
}

void deli_init(struct deli *d)
{
    d->next_ticket = 1;
    d->now_serving = 0;
    display_now_serving(0);
    lock_init(&d->lock);
    cond_init(&d->server_available);
}

int deli_get_ticket(struct deli *d)
{
    int result;
    lock_acquire(&d->lock);
    result = d->next_ticket;
    d->next_ticket++;
    lock_release(&d->lock);
    return result;
}

int deli_wait_turn(struct deli *d, int number)
{
    lock_acquire(&d->lock);
    while (d->now_serving < number) {
        cond_wait(&d->server_available, &d->lock);
    }
    cond_signal(&d->customer_available, &d->lock);
    lock_release(&d->lock);
}

int deli_wait_for_customer(struct deli *d)
{
    lock_acquire(&d->lock);
    d->now_serving++;
    display_now_serving(d->now_serving);
    cond_broadcast(&d->server_available, &d->lock);
    cond_wait(&d->customer_available, &d->lock);
    lock_release(&d->lock);
}
```

Problem 11 (5 points)

Is your solution to Problem 10 fair? Explain why or why not.

Answer: for customers, the solution is almost completely fair. The use of ticket numbers inherently serializes customers, so the old ticket numbers get to go first. The one exception is that if two customers call deli_wait_turn, then two servers call deli_wait_customer in quick succession, the two customers might return from deli_wait_turn out of order. However, the older customer will immediately unblock (there is a server waiting for it), so this is still pretty fair.

For servers, the solution is not fair: if there are more servers than needed, some servers might never get a chance to wait on customers.