CS 111 Final Examination
Spring Quarter, 2024
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. 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.

Be sure to write only on the front sides of pages. We'll be using Gradescope to grade the exams and may not see information written on the back sides. There are extra pages at the end of the exam, if you need more space for an answer.

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	12	8	11	5	6	20	12	20	10	12	14	20	150

Problem 1 (12 points)

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

(a) The dispatcher runs as a special high-priority thread.

Answer: false. The dispatcher is not a thread: it is just a piece of operating system code that is invoked at various times (e.g. when a timer interrupt occurs or a thread blocks or unblocks) to switch the currently-running thread.

(b) Reducing the length of time slices could worsen a system's average response time.

Answer: True. Reducing the length of time slices is problematic in two ways. First, it results in more overhead for context switching, which will slow everything down. Second, it can result in a more even sharing of CPU cycles across threads, whereas it is better to let some thread have the CPU long enough to finish its current work ("run to completion").

(c) There is no fragmentation when a stack is used for storage allocation.

True: all of the free space is in a single contiguous block (below the stack pointer).

(d) If the "ordered writes" approach is used for file system crash recovery, there is no need to run `fsck` to restore consistency during reboots.

True: the "ordered writes" approach prevents the kinds of inconsistencies that would prevent the system from resuming normal operation after a crash. The inconsistencies that remain with the "ordered writes" approach result in resource leaks; these can be found and repaired later, perhaps even while the system is running.

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

True: the page offset in a physical address is taken directly from the page offset in the corresponding virtual address. If the offset in the virtual address were larger, it's unclear what to do with the extra bits; if it's smaller, it's unclear how to fill in the extra physical address bits.

(f) The linker must be aware of the page size.

True. When the linker assigns the starting address for the data segment, it must ensure that it is on a page boundary, so that code (which is read-only and potentially shared) and data (which must be read-write) do not coexist in a single page. To do this, it needs to know the page size.

Problem 2 (8 points)

(a) (4 points) Suppose that the name of each file were stored with the file on disk (for example, in its inode), not in a separate directory file. Would this make file lookups faster or slower, and why?

Answer: this would make file lookups significantly slower. In order to search a directory for a particular file, the operating system would have to read the inode for each file in the directory. In the absence of a block cache, this would require one disk I/O per file in the directory (whereas, with names stored in the directory, many file names can be checked within a single disk block). Even with a block cache, lookups will still be slower, since the OS will have to look up a different block in the cache for each file name. Furthermore, the extra blocks used in the cache for all of these names may cause other data to be ejected, resulting in more cache misses.

(b) (2 points) Give an example of how someone could infer trust in a closed-source OS such as macOS or Windows.

Answer: they could run tests on it to check its behavior. Or, they could rely on good prior experience with the OS itself or with the vendor of the OS.

(c) (2 points) Give an example of how someone could infer trust in an open-source OS (such as Linux) that would not also apply to a closed-source OS.

Answer: they could look at the code of the operating system to verify proper behavior.

Problem 3 (11 points)

(a) (4 points) In the solution discussed in lecture for implementing locks in a multiprocessor operating system, the `lock` and `unlock` methods both (a) disable interrupts and (b) busy-wait on a lock variable. Does the order of these operations matter? If so, indicate what the order must be and explain why. If not, explain why not.

Answer: yes, the order matters. Interrupts should be disabled before busy-waiting on the lock variable. The reverse order could result in a thread T getting preempted right after acquiring the spin lock but before disabling interrupts. As a result, another thread T2 could run, attempt to acquire the spin lock, and loop indefinitely. In the worst-case this could result in deadlock (if T2 has higher priority than T), but even if the system doesn't deadlock it will waste an entire time slice until the scheduler eventually allows T to run again.

Note: answers must mention this specific scenario. We can't give credit for vague statements about race conditions.

(b) (3 points) What is the difference between internal and external fragmentation?

Answer: internal fragmentation is when the wasted space is inside allocated areas; an example is when the OS allocates a whole page when an application only needs part of a page. External fragmentation is when the wasted space is between allocated areas, such as in best fit or first fit.

(c) (4 points) In order to avoid thrashing in a paging system, the kernel must keep in memory the pages that a process is actively using; this is called the "working set" of the process. All pages outside the working set can be moved to backing store. If the page size is doubled, will the size (**in bytes**) of the working set of a process increase, decrease, or stay the same? Explain your answer briefly.

Answer: the working set cannot decrease with this change. It might possibly stay the same, but it's much more likely to increase. The problem here is analagous to internal fragmentation: with larger pages, more information must be brought into memory on a page fault, but the application might not need to use the additional information, so memory won't be used as efficiently.

Problem 4 (5 points)

Circle "Yes" or "No" next to each of the statements below to indicate whether that function is performed by a linker.

(a)	Combine like sections from different object files.	**Yes**	No
(b)	Fill in the addresses in a jump table. **(The dynamic loader does this)**	Yes	**No**
(c)	Modify addresses in code sections.	**Yes**	No
(d)	Create a symbol table.	**Yes**	No
(e)	Compute the virtual address of the first byte of the data segment.	**Yes**	No

Problem 5 (6 points)

In the original DOS FAT file system, each entry in the FAT (File Allocation Table) was 16 bits in size and stored either a block pointer or a special reserved value indicating "end-of-file" or "block free".

(a) If each disk block contains 512 bytes, what is the largest disk size that this version of FAT can support (you can give the answer as a power of two, if that is more convenient)? Explain your answer.

Answer: Each disk block contains 512 bytes, which is is 2^9 bytes. Since FAT entries are 16 bits in size, there can be at most 2^{16} blocks on disk. Thus the total disk capacity is $2^{16} * 2^9$, which is 2^{25}, or 32 Mbytes).

(b) If the FAT is loaded entirely into main memory, how much memory will it take? Explain your answer.

Answer: there is one entry in the FAT for each block on disk. Thus the FAT has 2^{16} entries. Each FAT entry holds a disk block pointer which is 16 bits (2 bytes) in size. Thus the total size of the FAT is $2^{16} * 2^1$, which is 2^{17}, or 128 Kbytes.

Problem 6 (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:

```
1  struct account {
2      std::mutex mutex;
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          a1->mutex.lock();
10         a1->balance -= amount;
11         a1->mutex.unlock();
12         a2->mutex.lock();
13         a2->balance += amount;
14         a2->mutex.unlock();
15     } else if ((a2->balance > a1->balance) && (a2->balance >= amount)) {
16         a2->mutex.lock();
17         a2->balance -= amount;
18         a2->mutex.unlock();
19         a1->mutex.lock();
20         a1->balance += amount;
21         a1->mutex.unlock();
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.

Answer: the problem is that the locks aren't held long enough. One race:

- **Two threads, T1 and T2, invoke `transferFromLarger(A, B, 100)`, where A and B are two accounts. A initially contains $150 and B contains $0.**
- **T1 executes first, but blocks just before executing line 9.**
- **T2 executes next, completing the entire method: A now contains $50 and B contains $100.**
- **T1 now finishes executing. It already decided to transfer from A to B, so it continues down this path. A will end up with a negative balance.**

(b) (15 points) Use the next page to rewrite `transferFromLarger` so that it always behaves correctly. Your solution must use the same definition of `struct account` (e.g. there must be a separate `mutex` for each account).

Write your solution to Problem 6(b) here:

```
struct account {
    std::mutex mutex;
    double balance;
}

void transferFromLarger(struct account *a1, struct account *a2, double amount)
{
    /* There are two problems we have to solve. First, we need to hold
     * locks on both accounts from beginning to end. But this runs the risk
     * of deadlock if one thread invokes transferFromLarger(A, B, 100) and
     * another thread invokes transferFromLarger(B, A, 200). So, we need to
     * make sure that accounts are always locked in the same order in all
     * threads. To do this, lock the account with the lower address first.
     */
    if (a1 == a2) {
        return;
    }
    if (a1 < a2) {
        a1->mutex.lock();
        a2->mutex.lock();
    } else {
        a2->mutex.lock();
        a1->mutex.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;
    }

    /* The order of unlocks doesn't matter. */
    a1->mutex.unlock();
    a2->mutex.unlock();
}
```

Problem 7 (12 points)

The `exchange` method on `std::atomic` types was introduced in class as a primitive to use in implementing higher-level synchronization mechanisms such as locks and condition variables. For example, consider the following code:

```
std::atomic<int> x;
...
int y = x.exchange(12);
```

The `exchange` method will atomically read `x`, overwrite it with `12`, and return the old value of `x`, which is then stored in `y`.

A friend shows you the following code and claims that it will atomically exchange the values of the variables `x` and `y`:

```
std::atomic<int> x = 3;
std::atomic<int> y = 4;
...
y.exchange(x.exchange(y));
```

Suppose that two threads execute the last line of code above at the same time; assume that `x` has the value 3 and `y` has the value 4 initially, and that the variables are shared between the threads. If this code is truly atomic, then once both threads have executed the code `x` will be 3 again and `y` will be 4 (the variables will have been exchanged twice, restoring each variable to its initial value).

(a) (6 points) Is it possible that both variables could end up with the value 4? If so, explain how; if not, explain why not. You may assume that memory reads and writes are atomic.

Answer: yes, they can both end up with the value 4:

- **T1 executes the inner exchange: x is now 4, y is 4, and the result is 3.**
- **T2 executes the inner exchange: x is now 4, y is 4, and the result is 4.**
- **T1 executes the outer exchange: x is now 4 and y is 3.**
- **T2 executes the outer exchange: x is now 4 and y is 4.**

(b) (6 points) Is it possible that both variables could end up with the value 3? If so, explain how; if not, explain why not.

Answer: no. The only way x can end up with the value 3 is if one thread executes both exchanges before the other thread executes either exchange (so that y has the value 3 before the second thread executes the inner exchange). But in this case the second thread will receive 4 as the value of the inner exchange and it will store that value in y during the outer exchange. So, if x ends up with the value 3, y can only end up with 4.

Problem 8 (20 points)

Suppose that each of the changes below is made to a file system. Consider whether each change affects the performance, consistency, and/or durability of the file system. For each of the three properties indicate whether the change improves that property, degrades it, or leaves it about the same. Explain each answer briefly.

(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 both metadata and data; the ordered writes are performed synchronously.

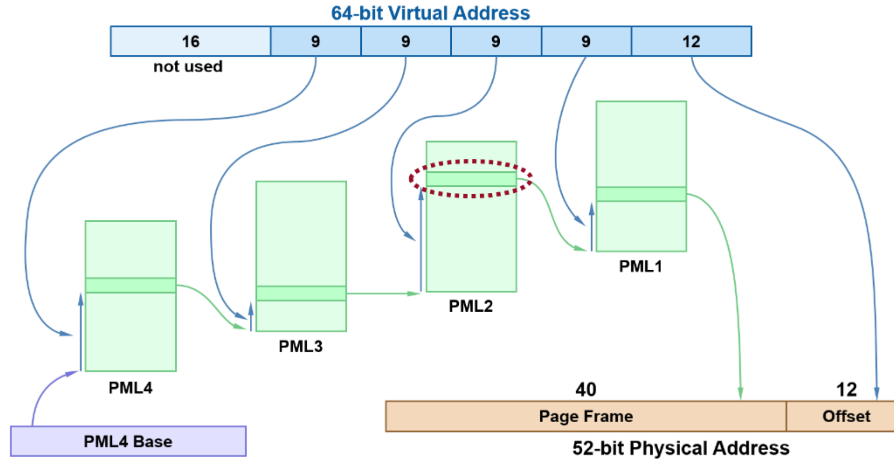
Answer: performance will drop because of the ordered-write mechanism (blocks that might not have been written before the change may have to be written because of the ordering constraints). Durability will not change significantly: writes are still delayed, so information can still be lost after a crash. 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. Only metadata changes 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 because an additional disk write (for the log entry) must be performed synchronously before each operation. Durability won't change much, because delayed writes are still used for data (it's OK if you said that it will improve a bit because some metadata is flushed to disk sooner). Consistency will improve to the point where it is now "perfect": no inconsistencies should occur after a crash if the log is written and replayed properly.

Problem 9 (10 Points)

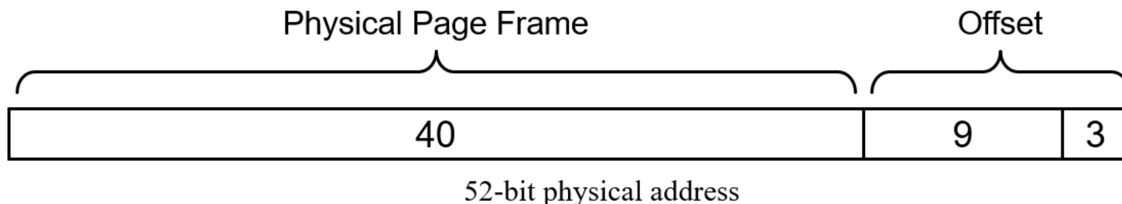
The figure below shows the address translation mechanism used for 64-bit mode in Intel x86 processors, which was discussed in lecture. It uses 4 levels of page map to translate virtual addresses to physical addresses.



As part of translating a virtual address, one entry will be read from a PML2 page map (dotted oval above). This page map entry resides in physical memory, so the physical address for that entry will need to be generated in order to read it. The empty box below represents the **physical address of the PML2 entry**. Mark up the box to indicate where every bit of the physical address comes from (if a bit is computed, explain the computation). Be as precise as possible: for example, give the size in bits of each field.

Answer: like all physical addresses, the physical address for the PML2 entry will consist of a physical page frame number (40 bits) and an offset (12 bits):

- **The page frame will come from the PML3 entry for the virtual address (it selects the page containing the PML2).**
- **The page offset indicates where the desired entry is within the PML2. We want to index into the PML2 using the second 9-bit chunk of virtual address from the right (the one with an arrow pointing into the PML2).**
- **Since PML entries are 8 bytes in size, the low-order 3 bits of the page offset will be zeroes and the high-order 9 bits will contain the 9 bits from the virtual address.**



52-bit physical address

Problem 10 (12 points)

Describe a program that calls `malloc` and `free` (or C++ `new` and `delete`) in a way that requires at least 1200 total bytes of heap space, even though the program never has more than 1000 bytes of memory allocated at once. You don't need to write actual code, but your description must be clear and precise. Your program must work for any (non-garbage-collecting) implementation of `malloc` and `free`; 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.**

(Note: you can't count on the memory allocator to allocate memory in any particular order; that's why it's necessary to check the addresses. If two non-adjacent blocks are freed, there is no way the new allocation can overlap more than one of them, so the 200 bytes for the other will not be used. This means that at least 1200 bytes of heap space will be required.)

Problem 11 (14 points)

Consider an operating system that uses a FIFO thread scheduler and supports demand paging. The OS is running on a system with a single-core CPU and two disks; one disk is used exclusively as the backing store for demand paging, while the other disk holds files. When a particular workload is running on the system, the measured utilizations of the elements (% of time each unit is doing useful work) are:

CPU	20%
Paging disk	99.7%
Disk for files	5%

For each of the following actions, indicate whether it is likely to increase the CPU utilization and explain your answer briefly.

Note: the utilizations indicate that the virtual memory system is thrashing (the system is spending all of its time reading and writing the paging disk). Thus, the only way to increase CPU utilization is to reduce the thrashing problem.

(a) Get a faster processor.

Answer: won't help. This will probably *reduce* CPU utilization: a faster CPU will take even less time to reach the next page fault, so CPU utilization will drop.

(b) Get a paging disk with more capacity.

Answer: won't help. There's no indication that the current paging disk has run out of space, and increasing its capacity won't allow it to serve page faults any faster.

(c) Start more processes executing.

Answer: this will probably make things worse, since the additional processes will need additional memory, which will make the thrashing problem even worse. It's possible this might help if one of the new processes needs almost no memory but is totally CPU-bound. Assuming its pages don't get flushed to disk, it might be able to keep the CPU busy, particular since the scheduler is FIFO.

(d) Kill some existing processes.

Answer: this should improve the CPU utilization. It will reduce the size of the system's overall working set, which should then reduce thrashing. As a result, the system will take fewer page faults and more useful work will get done.

(e) Replace the paging disk with a flash drive that has the same capacity.

Answer: this should improve the CPU utilization. It won't eliminate the thrashing problem, but the flash drive will be able to service requests much faster than the disk it replaced, so the CPU won't spend as much time waiting for page faults.

(f) Switch from a FIFO scheduler to the 4.4 BSD scheduler.

Answer: won't help. In fact, this might actually reduce CPU utilization. A FIFO scheduler is more likely to focus execution on a single process, whereas the 4.4 BSD scheduler is likely to give a variety of processes a chance to execute (e.g. a process that is generating a lot of page faults will get high priority, since it isn't using much execution time!). The more processes that execute, the larger their combined working set, and hence the greater the risk of thrashing.

(g) Install more memory.

Answer: this will help. As more memory is added, the gap between available physical memory and the size of the overall working set will reduce, resulting in fewer page faults. As a result, the CPU will spend less time waiting for page faults and more time executing. This is the best solution of all those listed.

Problem 12 (20 points)

Extend your solution for Assignment 7 (Reading Unix V6 Filesystems) by writing a C function with the following signature:

```
int inode_upgrade(const struct unixfilesystem *fs, struct inode *inp);
```

The `inp` argument points to an inode for a "small" file; the function must make whatever changes are needed to convert this file to the "large" file format in V6 Unix while retaining the file's contents. It should return 0 if it is successful and -1 if an error occurred. Your solution may use any of the code in your solution for Assignment 7, and it may also assume the existence of the following C functions:

```
int diskimg_writesector(int fd, int sectorNum, void *buf);
int block_alloc(const struct unixfilesystem *fs);
```

`diskimg_writesector` is similar to the `diskimg_readsector` function you used in Assignment 7 except that it writes to disk instead of reading. The `block_alloc` function will allocate a free block on the disk and return its block number (if there is an error it will return -1).

The `inode_upgrade` function need not write the inode back to disk (the caller will handle this), but it must write back any other changes by calling `diskimg_writesector`. You need not worry about consistency issues that could occur after an error or crash.

```
int inode_upgrade(const struct unixfilesystem *fs, struct inode *inp)
{
    int i;
    uint16_t indirect[BLOCK_NUMS_PER_INDIRECT];

    for (i = 0; i < BLOCKS_IN_INODE; i++) {
        indirect[i] = inp->i_addr[i];
        inp->i_addr[i] = 0;
    }
    for (i = BLOCKS_IN_INODE; i < BLOCK_NUMS_PER_INDIRECT; i++) {
        block_nums[i] = 0;
    }
    int inp->i_addr[0] = block_alloc(fs);
    if (inp->i_addr[0] == -1) {
        return -1;
    }
    if (diskimg_writesector(fs->fd, inp->i_addr[0], indirect) < 0) {
        return -1;
    }
    inp->i_mode |= ILARG;
    return 0;
}
```

Use this page if you need extra space for any problem on the exam

Use this page if you need extra space for any problem on the exam