CSE 120 Principles of Computer Operating Systems
Fall Quarter, 2000
Final Exam Solutions

Instructor: Geoffrey M. Voelker

Name _____

Student ID _____

Attention: This exam has nine questions worth a total of 110 points, and the last one is a freebie. You have three hours to complete the questions. As with any exam, you should read through the questions first and start with those that you are most comfortable with. If you believe that you cannot answer a question without making some assumptions, state those assumptions in your answer.

Best of luck, and have a great holiday!

1	/12
2	/14
3	/12
4	/12
5	/8
6	/14
7	/10
8	/18
9	10/10
Total	/110

1. (12 pts) Potpourri: Answer yes or no, or with a single term or phrase, as appropriate. You should go through these quickly, answering with the first answer that comes to mind — it probably is the correct one. Only dwell on ones you are unsure of after finishing the other questions.

(a) Which bit in the PTE does the operating system use for approximating LRU replacement?
   Reference/Use bit

(b) The layout of disk blocks for a file using Unix inodes is always/sometimes/never contiguous on disk?
   Sometimes

(c) Does a TLB miss always/sometimes/never result in a pagein from disk?
   Sometimes

(d) In RPC, what does the server-stub call?
   The server procedure

(e) What are the two components of a virtual address used in segmented virtual memory?
   Segment # and offset

(f) In general, are there more/same/fewer inodes than directories in a file system such as Unix?
   More

(g) In what Nachos file was StartProcess implemented?
   progtest.cc

(h) What kind of pages in a process' virtual address space are usually protected as "Read Only"?
   Code pages; copy-on-write pages was also a good answer

(i) The layout of virtual pages for a virtual address space is always/sometimes/never contiguous in physical memory?
   Sometimes

(j) In what metric do RAID arrays improve performance?
   Throughput

(k) Is it possible to implement a file system inside of a file stored in another file system?
   Yes; just like we can virtualize an OS and user programs in a single process, we can virtualize a file system in a single file

(l) Was the exam was too easy/just right/too hard (this is a free point)?

2. (14 pts) Identify the following in one phrase or sentence (do more than just expand the acronym, though). Then state whether each item is related to an operating system *mechanism* or *policy*.

(a) PTE – *Page Table Entry, used to map virtual to physical addresses, stores access and protection bits (mechanism)*

(b) SCAN – *AKA Elevator, a disk scheduling algorithm that moves the disk head back and forth across the disk surface, servicing requests (policy)*

(c) Working set – *In a given time interval, the set of pages required to prevent page faults; used in the Working Set model for page replacement (policy)*

(d) LRU – *Least Recently Used, a virtual page replacement algorithm that evicts pages that were last used furthest in the past (policy)*

(e) Thrashing – *A condition where the system spends most of the time transferring pages to and from the disk and little time making progress on application computations; usually due to memory being over-subscribed or a poor page replacement algorithm (relates to policy)*

(f) Inode – *A Unix file descriptor for the layout of file blocks on the disk; it is index-based and hierarchical (mechanism)*

(g) Access control list – *a protection mechanim that describes, for a given object, the subjects and the actions allowed by each subject on the object (mechanism)*

Most people got most of these correct. The mistake that cost people the most points was forgetting to state whether each term relates to a policy or mechanism.

3. (12 pts) Operating systems frequently exploit locality to improve performance. *Briefly* describe two examples where operating systems do so, and state how locality is exploited.

1. Virtual memory and associated page replacement algorithms exploit the temporal and spatial locality of a program's accesses to its data. Programs tend to access pages that were accessed recently, enabling physical memory to be used as a cache for all of the data used by a program during its execution. Programs also tend to access data across a given page, which helps amortize the cost of bringing pages in from disk. Without such locality, virtual memory would be prohibitively slow.

2. The file buffer cache and associate replacement algorithms exploit the temporal and spatial locality of a program's accesses to file blocks. Programs tend to access recently-accessed file blocks, enabling the file buffer cache to serve a significant fraction of requests from memory instead of disk even though the file system is a couple of orders of magnitude larger than the file buffer cache. Programs also tend to access file blocks logically near previously-accessed file blocks (e.g., sequentially), enabling the system to use techniques like read-ahead to anticipate program behavior.

And there are others (e.g., TLB)...

I was looking for two separate examples, and at least a temporal or spatial motivation for how locality is exploited in each example.

4. (12 pts) One day famed student Joe Surfer had an inspiration while hanging ten at Mission Beach. He observes that most programs have most of their data at the beginning of the address space. For his homegrown SaltWater OS, he decides that he is going to implement his page tables similar to the way Unix implements inodes. He calls this page table design *Inode Page Tables*. Inode Page Tables are essentially two-level page tables with the following twist: The first half of the page table entries in the master page table directly map physical pages, and the second half of the entries map to secondary page tables as normal. Call the first half the entries *fast*, and the second half *normal*.

For the following questions, assume that addresses are 32 bits, the page size is 4 KB, and that the master and secondary page tables fit into a single page.

(a) How many virtual pages are *fast* pages?

   $4KB/4 = 1024$ PTEs
   $1024/2 = 512$ PTEs $=> 512$ or (2^9) pages are fast

(b) How many virtual pages are *normal* pages?

   The remaining 512 PTEs refer to second-level page tables. Each second-level page table has 1024 PTEs, so:
   $2^9 * 2^{10} = 2^{19}$ pages are normal

(c) What is the maximum size of an address space in bytes (use exponential notation for convenience, e.g., 2^3)?

   The size of the address space is the total number of pages times the size of each page:
   $(2^9 + 2^{19}) * 2^{12} = 2^{21} + 2^{31}$ bytes

(d) Inode Page Tables reduce the lookup time for fast pages by one memory read operation. Do you think that this is an effective optimization? Briefly explain.

   Not really. This optimization saves one memory access whenever there is a miss in the TLB. Since TLB misses are relatively infrequent, saving one memory access in the miss handler is not going to improve performance significantly. It also makes the PTE lookup a bit awkward since you have to check to see what kind of a page needs to be loaded into the TLB.

   A number of people answered that this shrank the address space too much to be useful. The address space is still > 2 GB, which is pretty large. Most programs will still easily fit inside that address space.

5. (8 pts) In lecture we said that, if the semaphore operations *Wait* and *Signal* are not executed atomically, then mutual exclusion may be violated. Assume that *Wait* and *Signal* are implemented as below:

```
void Wait (Semaphore S) {
  while (S.count <= 0) {}
  S.count = S.count - 1;
}

void Signal (Semaphore S) {
  S.count = S.count + 1;
}
```

Describe a scenario of context switches where two threads, T1 and T2, can both enter a critical section guarded by a single mutex semaphore as a result of a lack of atomicity.

Assume that the semaphore is initialized with count = 1. T1 calls Wait, executes the while loop, and breaks out because count is positive. Then a context switch occurs to T2 before T1 can decrement count. T2 also calls Wait, executes the while loop, decrements count, and returns and enters the critical section. Another context switch occurs, T1 decrements count, and also enters the critical section. Mutual exclusion is therefore violated as a result of a lack of atomicity.

6. (14 pts) Unix provides two mechanisms to link one file to another, *hard links* and *soft links*. With hard links, assume that the directory entry for the link maps the link file name to the *inode* for the file to which it is linked. With soft links, assume that the directory entry for the link maps the link file name to the *file name* of the file to which it is linked.

Consider the situation where we want to create a link "/bin/ls" to the existing file "/sbin/ls". In this case, "/bin/ls" is the link file name and "/sbin/ls" is the file to which it is linked. For the questions below, assume that, each time Unix has to retrieve information from the disk, it takes only one disk read operation.

I was looking for two things from this problem. One was an understanding of how path searching relates to the use of the on-disk data structures supporting file systems (directories and inodes). The second was understanding the difference between having a directory point to a name, which needs to be resolved again, and having it point to an inode, which can be referenced directly.

The most common error on this problem was to forget that inodes need to be read from disk, and that both files and directories have inodes that need to be read at every step.

I was lenient with respect to the master block.

(a) First, succinctly describe what steps Unix will take, in terms of the disk data structures it must read, in order to resolve the path name "/sbin/ls" so that it can read the first byte of the file.

Master block − > *inode for "/"* − > *first block of "/"* − > *inode for "sbin"* − > *first block of "sbin"* − > *inode for "ls"* − > *first block of "ls".*

(b) How many disk reads will be required to resolve the path name and read the first byte of the file "/sbin/ls"?

7

(c) Consider when we use a hard link to link "/bin/ls" to "/sbin/ls". Succinctly describe the steps required to read the first byte of "/sbin/ls" starting with the link "/bin/ls". How many disk reads will it require?

Requires the same steps and same number of reads as "/bin/ls", with "sbin" replaced by "bin". Regardless of the steps and read count, I also specifically looked to see if the read count was the same as "/sbin/ls".

(d) Consider when we use a soft link to link "/bin/ls" to "/sbin/ls". Succinctly describe the steps required to read the first byte of "/sbin/ls" starting with the link "/bin/ls". How many disk reads will it require?

First: Master block − > *inode for "/"* − > *first block of "/"* − > *inode for "bin"* − > *first block of "bin".*

This gets us to the link name "/sbin/ls". We then follow the steps in part (a) to resolve the link and access the first block of ls.

$7 + 5 = 12$ *reads.*

I specifically looked to see if the read count was more (roughly twice as much) as the counts in (a) and (c). I gave full credit for an answer that said it required twice the number of reads as part (a).

(e) Unix maintains a reference count of the number of hard links to a file, and it only removes a file's blocks on disk when this count reaches zero. If we remove the file "/sbin/ls", is the hard link still valid? Is the soft link still valid?

The hard link is still valid (the link count is non-zero, so the blocks are not removed and the pointer to the inode in the hard link is still valid), but the soft link is not (when it tries to resolve the name "/sbin/ls" the resolution will fail).

7. (10 pts) Briefly describe the most challenging bug that your group encountered in project 3, how you found it, and how long it took you to find it. Which of your group members do you think will give the best answer to this question?

This was a gimme. If you wrote something, but didn't actually describe a bug, I gave you 8 points for not following directions.

8. (18 pts) In this problem you will outline an implementation of shared memory in Nachos. Shared memory will be implemented using two new system calls:

```
RegionID SharedRegionCreate (int beginAddress, int endAddress);
      void SharedRegionAttach (RegionID region);
```

SharedRegionCreate creates a shared memory region in the caller's address space defined by the begin and end addresses. If successful, it returns a RegionID identifying the shared region. SharedRegion-Attach maps the existing shared region identified by the RegionID into the caller's address space. A region can only be created by one process, but it can be attached by an arbitrary number of processes. You can assume that each PTE has an additional flag *Shared*, which should be TRUE if that virtual page is being shared. You can also assume that a shared memory region can only be created within a defined region of a process' virtual address space.

These operations can be used as follows. A parent process uses SharedRegionCreate to create a shared memory region, and then Execs a child process and passes the returned RegionID for the shared region to the child as an argument to Exec. The child then uses SharedRegionAttach to map that region into its address space so that it can share memory with its parent. As a result, whatever data the parent places and modifies in the shared region, the child can access it (and vice versa).

For each of the following questions, answer them descriptively at a high level. The answers should be brief, and capture the essence of what needs to be implemented at each stage. You do not need to use psuedo-code.

  (a) What should SharedRegionCreate do to the caller's page table?

    It should mark all pages in the region as Shared = TRUE. When implementing this, you will also want to allocate a RegionID, update a RegionID table, etc., but this question only asked about what happens with the page table.

  (b) What should SharedRegionAttach do to the caller's page table?

    It should (1) mark all pages in the region as Shared = TRUE, and (2) set the PTE fields in the caller's page table to be equivalent to those in the page table of the process that created the region. The main thing I looked for was setting the virtual to physical page mapping to point to the shared physical page. In effect, the caller's page table should copy the PTEs from the creator's page table.

  (c) What *additional* page table operations must be performed when a shared page is paged out (evicted) from physical memory?

    The problem introduced with sharing pages is that multiple page tables have virtual to physical mappings to the same physical page. So if a shared page is paged out (evicted), all of those page tables have to be updated to reflect the eviction. In particular, the valid bits for this page in all page tables sharing the page have to be set to FALSE because an access by any process to the shared page needs to produce a page fault. At least one of the page tables has to be updated with the location of the page on disk, but all of them can be, too.

    Delaying eviction until the page is not shared does not answer the question.

    Setting Shared = FALSE is very problematic because then you don't know which pages in which page tables need to be updated when the page is paged in.

  (d) What *additional* page table operations must be performed when a shared page is paged in from backing store?

The problem here is the complement of the previous question. All PTEs referencing the shared page need to be updated to reflect the fact that the page is now in memory. In particular, the virtual to physical mapping needs to be updated, and the valid bit has to be set to TRUE; if you ignored other bits, like the dirty bit, I let it slide. In effect, the PTEs in all page tables sharing the page have to be set to the same values.

(e) How should the destructor for AddrSpace be changed to account for shared pages?

I accepted two answers. The first is to have the destructor only free the physical page if the AddrSpace is the last one referencing a shared page (preferred). The second is to have the destructor invalidate the PTEs in all address spaces that attached to the shared region (plausible, but harsh).

(f) Name three error conditions that an implementation of SharedRegionCreate and SharedRegion-Attach will have to check for.

1. SharedRegionAttach – region does not exist

2. SharedRegionAttach – unknown RegionID

3. SharedRegionCreate – invalid VA range

4. SharedRegionCreate – region already exists

5. SharedRegionAttach – region already created or attached

6. ...

There was a plethora of error conditions to choose from. Some people answered that checking that the beginAddress was valid was one error check, and checking that the endAddress was a second. You have to be more creative than this – these are essentially the same check.

9. (10 pts) You get full credit for the problems below whether you answer them or not, or what your answer is. In other words, these questions will have no impact on your grade. **Do not** bother to write answers to these questions until you are finished with the rest of the exam, if you choose to do so at all.

   (a) What topic in this course struck you as the least interesting, least relevant, and made the least impact on your education?

   (b) What topic in this course was the most interesting to you?

   (c) What operating system topic were you hoping to learn about, but we didn't cover?

   (d) What message would you give to incoming students of the next CSE 120 class I teach? I will use some of these answers in the intro lecture next time.