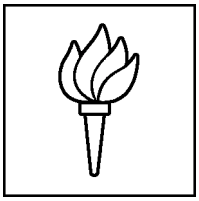

Computer Science Department

New York University

G22.2250-001 Operating Systems: Spring 2009

Final Exam

Many problems are open-ended questions. In order to receive credit you must answer the question *as precisely as possible*. You have 90 minutes to answer this quiz.

Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.

I (xx/16)	II (xx/15)	III (xx/20)	IV (xx/10)	V (xx/14)	VI (xx/5)	Total (xx/80)

Final statistics:

Score range	Num of students
$[60, 80]$	2
$[55, 60)$	3
$[50, 55)$	4
$[40, 50)$	6
$[30, 40)$	6
$[0, 30)$	4

I Basic OS knowledge

Answer the following multiple-choice questions. Circle *all* answers that apply. Each problem is worth 4 points. Each missing or wrong answer costs -2 point.

A. In a typical x86-based OS, which of the following statements are true about virtual memory?

1. The kernel is mapped into each user-level application process' address space in order for applications to access the kernel's data conveniently.
2. A user-level application cannot modify its page table entries because the PTE_U flag of its page table entries (PTEs) is cleared by the kernel.
3. In 32-bit x86 with paging turned on, a machine instruction such as "mov 0x08000000, %eax" could potentially involve 3 accesses to main memory.
4. On x86, upon a TLB miss, the kernel traverses the 2-level page table to fill in the virtual to physical address mapping in the TLB.

Ans: 3 or 2,3.
1 is incorrect. Kernel is mapped into user-level process' address space for kernel to access applications' data conveniently (and to avoid TLB flushes when transitioning from user-level to kernel level execution.)
4 is incorrect. x86's TLB is filled by the hardware instead of the kernel (software).
2 is ambigious and hence we give credits for both. A user-level application cannot modify its page table entries bacause its page table resides in kernel's portion of the address space. And the kerne's portion of the address space is not accessible by user-level processes because its corresponding page table entries have PTE_U *flag cleared.*

B. Which of the following statements are true about file systems?

1. Reading a random location in big files is usually faster in inode-based file systems than the FAT file systems.
2. For better sequential read/write performance, the file system should try to allocate consecutive blocks to a file.
3. If the journaling file system does not immediately flush the committed transaction record of each completed FS operation to disk, the FS will become inconsistent upon crash and recovery.
4. If the journaling file system does not immediately flush the committed transaction record of each completed FS operation to disk, the FS might lose the last few FS operations but will remain consistent upon crash and recovery.

Ans: 1, 2 and 4
2 is correct. When performing sequential read/writes, the achievable throughput is limited by disk throughput, e.g. 40MB/s. This is much better than performing random read/writes whose achievable throughput is limited by disk seek time. E.g. if each read is 100 bytes, then random read throughput is $\frac{100}{10*10^{-3}} = 10$*KB/sec for a disk with 10ms seek time.*

C. Which of the following statements are true about virtual machines?

1. A VMM can virtualize the x86 interface on top of any hardware architecture (e.g. the MIPS or ARM architecture)
2. Using dynamic binary translations to implement VMM slows down kernel-level function calls (`call`) and function returns (`ret`).

3. Using dynamic binary translations to implement VMM slows down user-level function calls (`call`) and function returns (`ret`).
4. The guest OS kernels cannot access the code and data of VMM (e.g. the shadow page table).

Ans: 2,4.
1 is incorrect because virtualization requires the VMM to be able to execute most of the instructions identically. If one is to simulate a x86 interface on top of MIPS, then every x86 instruction needs to be simulated using many MIPS instructions. This is called "simulation", not "virtualization".

2 is correct because dynamic binary translation cannot translate the `call` *and* `ret` *instructions identically. For direct* `call`*s, it's possible for the binary translator (BT) to directly substitute the call target's address with its translated code fragment's address at translation time, thus avoiding the cost of hash table lookups at execution time. For* `ret`*, no such optimization is possible and the translated code must call into BT to perform a hash table lookup to find the translated target's address, resulting in slow down at execution time.*

4 is incorrect because user-level processes are not subject to binary translation.

D. Which of the following statements are true?

1. A non-root user in UNIX can never launch programs that execute with root privileges.
2. Only the owner of a file can set the `setuid` bit on that file.
3. TOCTTOU bugs can only happen to `setuid` programs.
4. We can solve UNIX' security problems by having each user encrypt all his files.

Ans: 2.
In most UNIX systems such as Linux, the root cannot directly set the `setuid` *bit of programs not owned by root. However, the root can indirectly set the* `setuid` *bit by first changing its identity to be the owner of those files and then set the bit. Therefore, we give full credits to those who did not mark 2 as correct.*

3 is incorrect because TOCTTOU is a generic type of bug that can happen to any privileged programs, i.e. a normal root process can be vulnerable to TOCTTOU bugs.

II Synchronization

We often build higher-level synchronization primitives on top of lower-level ones. For example, we have seen how one can build mutexes using spin-locks during Lecture. Another useful high-level primitive is the reader/writer lock. A reader/writer lock allows multiple threads to acquire read access to shared data simultaneously, on the other hand, a thread modifying the shared data can only proceed when no other thread is accessing the data. A reader/writer lock can be implemented on top of mutexes and conditional variables.

Recall that mutex and conditional variable export the following interfaces:

- Mutex has type `mutex_t` and it implements `lock(mutex_t *m)` and `unlock(mutex_t *m)`.
- Conditional variable has type `condvar_t`. The function `cond_wait(condvar_t *cv, mutex_t *m)` releases the mutex pointed to by `m` and blocks the invoking thread until `cond_broadcast` or `cond_notify` is called. The function `cond_broadcast(condvar_t *cv)` wakes up all threads blocked in `cond_wait`. The function `cond_notify(condvar_t *cv)` wakes up one of the threads blocked in `cond_wait`.

The reader/writer lock has type `rwlock_t` and it needs to implement the following interfaces:
`read_lock(rwlock_t *rw)` //allow any number of readers to proceed if no writer has grabbed lock
`read_unlock(rwlock_t *rw)`
`write_lock(rwlock_t *rw)` //allow a writer to proceed only if no reader nor writer has grabbed lock
`write_unlock(rwlock_t *rw)`

1. [10 points]: Write C code to implement the 4 function interfaces of reader/writer lock using the mutex and condition variable. Hint: you might want to declare `rwlock_t` as follows (or you are welcome to use a different data structure for `rwlock_t`):

```
typedef struct rwlock {
  int nreaders; //number of readers who have grabbed the lock, initialized to 0
  int nwriters; //number of writers who have grabbed the lock, initialized to 0
  mutex_t m;
  condvar_t cv;
} rwlock_t;
```

```
void
read_lock(rwlock_t *rw) {
  lock(&rw->m);
  while (rw->nwriters > 0)
    cond_wait(&rw->cv,&rw->m);

  rw->nreaders++;
  unlock(&rw->m);
}

void
read_unlock(rwlock_t *rw) {
  lock(&rw->m);
  rw->nreaders--;
  cond_broadcast(&rw->cv);
  unlock(&rw->m);
}

void
write_lock(rwlock_t *rw) {
  lock(&rw->m);
  while (rw->nreaders > 0 && rw->nwriters > 0)
    cond_wait(&rw->cv, &rw->m);

  rw->nwriters++;
  unlock(&rw->m);
}

void
write_unlock(rwlock_t *rw) {
  lock(&rw->m);
  rw->nwriters--;
  cond_broadcast(&rw->cv);
  unlock(&rw->m);
}
```

The above code correctly implements the reader/writer lock. However, it is not fair in the sense that readers might starve a waiting writer: there could be arbitarily many readers coming and holding the lock while a writer is waiting for the lock. There are many ways to fix the writer starvation problem. For example, one could make new readers wait when writers have been waiting for a long time.

2. [5 points]: Why does the `cond_wait(condvar_t *cv, mutex_t *m)` function require a second argument pointing to a mutex? In other words, if you replace every line `cond_wait(cv, m);` in your reader/writer lock implementation with two lines `unlock(m); cond_wait(cv); lock(m)` is your code still correct? Explain.

Ans: `cond_wait` *function must perform the two steps (1. release mutex 2. block until cond_notify) atomically. If not, after the mutex is unlocked (*`unlock(m)`*) but before wait is called (*`cond_wait(cv)`*), the producer thread (i.e. threads doing* `read_unlock` *or* `write_unlock` *functions) might come in between and generate a signal that nobody is waiting for, resulting in the signal becoming lost. Please argue for yourself why performing unlock and wait atomically in* `cond_wait` *avoids such sleep-wakeup race.*

III File system Layout

3. [5 points]: In an i-node based file system implementation, the i-node typically stores 12 direct block pointers, one 1-indirect block pointer, one 2-indirect block pointer, and one 3-indirect block pointer. Recall that an indirect block is a disk block storing an array of disk block addresses (i.e. pointers). The pointers in a 1-indirect block point to disk blocks that store file data. The pointers in a 2-indirect (or 3-indirect) block point to other 1-indirect (or 2-indirect) blocks. Suppose the file system is configured to use a block size of 2^{10} bytes and each pointer takes up 4-byte. What is the maximum file size that can be supported in the file system? Explain your calculation.

Ans: Each 1-indirect block can address $2^{10}/4 = 2^8$ *data blocks. Each 2-indirect block can address* $2^8 * 2^8 = 2^{16}$ *data blocks. Each 3-indirect block can address* $2^8 * 2^8 * 2^8 = 2^{24}$ *data blocks. In total, the biggest file can contain at most* $12 + 2^8 + 2^{16} + 2^{24} \approx 2^{24}$ *data blocks (i.e.* $2^{24} * 2^{10} = 4GB$*).*

4. [5 points]: Ben Bitdiddle runs a program that reads 100-byte data chunks in random locations of a file. What's the maximum number of random reads can Ben's program hope to achieve in a second? (Explain. Write down your assumptions about hardware if there's any.)

Ans: The latency of reading a random location on disk is dominated by seek+rotation delay. Assume a typical seek time of 10ms, Ben can hope to perform at most 100 random reads/sec.

A number of students calculated random read throughput based on disk throughput, which is incorrect. One achieves disk throughput (e.g. 40MB/sec) only for sequential reads/writes.

5. [5 points]: Ben notices that his program gets close to the best possible random read throughput when running on small files. However, when reading from large files ($>$ 1GB), the actual random read throughput is much lower. Why? Explain with concrete performance numbers.

Ans: To read a random location in a big file ($>$ 1GB), the file system must read 3 indirect blocks in addition to the data block. For a random file data block, its indirect blocks also reside on random (or non-sequential) locations on disk. Thus, one could suffer 4 seeks for each random read in a big file, resulting in 25 reads/sec.

6. [5 points]: Please describe an alternative scheme to replace the indirect-block based mapping scheme. Your scheme should improve the performance of random reads in large files.

Ans: To improve random reads in large files, we need to be able to address all the data blocks of a big file with fewer bytes. Since the file system already tries to allocate consecutive blocks to a file (so sequential file accesses result in sequential disk reads/writes, maximizing throughput.), we can have the file system record the start and end block address of each consecutive run of blocks. For example, if a 1GB file consists of 10 consecutive regions of blocks, we only need 20 numbers to address all of 1 million ($2^{32}/2^{10}$) data blocks. The upcoming Linux ext4 file system will have this feature.

Some of you also propose to make the block size larger. This solution will increase internal fragmentation for small files. Since a large fraction of files are small (< 4KB), the increased amount of internal fragmentation is substantial.

IV File system crash recovery

The Linux journaling file system writes the content of all modified disk blocks to the log. Ben Bitdiddle finds such logging wasteful since copying the content of modified disk blocks to the log doubles the amount of disk writes for each logged file system operation.

Ben decides to implement a more efficient journaling file system. In particular, he decides to only record an operation's name and parameter in the log file instead of recording the content of all modified blocks. For example, for an operation that creates file "/d/f", the file system appends the transaction record [create "/d/f"] to the log. Ben's file system ensures that the corresponding transaction record is written to the log before the modified disk blocks are flushed to disk. Upon crash and recovery, Ben's file system re-executes the logged file system operations and truncates the log.

7. [10 points]: Ben's new logging mechanism is certainly more efficient since each transaction record is much smaller than that with Linux's logging. Is his design also correct? i.e. can it recover a file system correctly from crashes? Explain your reasoning and give concrete examples.

Ben's design is not correct. Consider the following example. The unlink("/D/f") operation involves modifying 5 blocks:

1. decrement the nlink field of "/D/f"'s i-node
2. modify direcotory D's dir block to remove a's entry
3. modify D's i-node to update mtime, length
4. modify i-node bitmap if a's i-node is now free
5. modify block bitmap to indicate a's data blocks are now free

The FS logs the operation [unlink "/D/f"] and crashes upon performing step 1,2 and before steps 3,4,5. Upon recovery, the FS attempts to re-execute the operation unlink("/D/f"). However, this operation fails upon noticing that there's no file "f" in the directory "D". Hence, the i-node freemap and block freemap become inconsistent (i.e. freed i-node and disk blocks are "lost").

Two students gave the following excellent example. Suppose two files ("/D/f1" and "/D/f2") are hard-links of each other, i.e. they have the same i-node whose nlink field is 2. Suppose the operation unlink("/D/f1") crashes after flushing modified data belonging to step 1 to disk, so the corresponding i-node's nlink field is decremented to 1. Upon recovery, the logged operation unlink("/D/f1") is re-executed, causing the nlink field to be decremented again to be zero and the i-node to be freed. This is incorrect because "/D/f2" still uses that i-node!

In summary, simply logging the operation's name is not correct because the high-level operations like create/link/unlink are not idempotent. Non-idempotent means applying an operation multiple times leads to different results than applying it once (because the state has changed). When log operations are not idempotent, the recovery process must apply them based on the same state as that before the crash in order to achieve identical results. Unfortunately, when crash happens, the disk might have already written some subset of modified data blocks. As a result, applying a non-idempotent operation on these modified data blocks yields different (and inconsistent) results.

V A Different VM interface

Ben Bitdiddle finds the VM interface provided by MemOS to applications limiting. In MemOS, applications use the `sys_page_alloc(va)` syscall to request for more memory. To handle this syscall, the MemOS kernel simply allocates a new physical page and maps it to the requested virtual address `va`.

Ben decides to adopt a different interface that gives applications greater control over how it manages its memory and address space. Ben's interface contains two system calls, `sys_phypage_alloc()` and `sys_map_va2pa(va, pa, perm)`.

The `sys_phypage_alloc()` syscall requests a physical page from the kernel and it returns a physical address to the application. The `sys_map_va2pa(va, pa, perm)` syscall asks the kernel to add a mapping from the virtual address `va` to the physical address `pa` with permission bits `perm`. It's easy to see that applications can use these two system calls in combination to achieve the equivalent effect of the original system call `sys_page_alloc`.

Ben has implemented his new interface based on Lab 3. Here's a snippet of Ben's system call implementation.

```
...
void
interrupt(register_t *reg) {
  current->p_registers = *reg;
  reg = &current->p_registers;

  switch (reg->reg_intno) {
    case INT_SYS_PHYPAGE_ALLOC:
      //page_alloc_free allocates an idle physical page
      //and records the ownership of the page in the pageinfo array
      reg->reg_eax = page_alloc_free(PO_PROC+current->p_pid);
      run(current);
    case INT_SYS_MAP_VA2PA:
      //%eax,%ebx,%ecx contains the 3 syscall parameters va, pa, perm respectively
      //pgdir_set(pgdir, va, pa|perm) modifies page table pgdir to add mapping
      //from va to page table entry pa|perm
      pgdir_set(current->p_pgdir, reg->reg_eax, reg->reg_ebx | reg->reg_ecx);
      reg->reg_eax = 0; //syscall returns 0 to indicate success
      run(current);
    case ... :
      ...
  }

}
...
```

8. [10 points]: Ben tells Alyssa Hacker about his new interface. Upon examining Ben's implementation, Alyssa declares that Ben's implementation is not secure, i.e. it does not provide proper isolation against malicious or buggy applications. Is Alyssa right? Explain and provide a plausible fix. You can describe your fix in plain English instead of C code.

Ans: Ben's implementation is not correct. A malicious user-level program can read/write arbitary memory belonging to the kernel or other processes by calling `sys_map_va2pa` *with arbitary physical addresses.*

The fix: when handling `INT_SYS_MAP_VA2PA`*, the kernel must check that* `pa` *argument is owned by the requesting process by examining the* `pageinfo` *array.*

9. [4 points]: Alyssa Hacker is very enthusiastic about Ben's new interface. By exposing physical addresses to applications, Alyssa argues, this interface could enable application-specific memory management. For example, database applications typically perform application-level caching. It's desirable to let the database application decide what physical page to swap out to disk when the system is under memory pressure (instead of having the kernel swap out pages without an application's knowledge using a fixed LRU policy). Augment Ben's interface to handle physical page de-allocation and swapping in a way that enables such application-level memory management. Describe how an example application such as the database application can use Ben's interface to better manage its memory.

Ans: When under memory pressure, the kernel could make an upcall to applications to request for the release of some memory pages. The application could pick a physical page to swap out. If the chosen page is dirty, the application writes it to disk first. Then, the application invokes a syscall like `page_dealloc(pa)` *to release the chosen page and unmap the appropriate page table entries.*

The advantage of this setup is that applications, who have the most information, can have full control of the page eviction policy. If you are interested in learning more about this approach, you can look up this research paper for more information: `http://www.news.cs.nyu.edu/~jinyang/sp09/readings/engler95exokernel.pdf`

VI G22.2250-001

10. [5 points]: We would like to hear your opinions about the class. Please tell us what you like the most/least about this class. Any suggestions for improving the class are welcome.

End of Final