CSE451
Final Exam
Winter 2008

Name:

Short Answer [4 points each]

1) Recall Andrew's "solution" to the bug Nick discovered in the CSE451 file system:

```
// I just wrote some stuff into the buffer
  set_buffer_dirty(bh);

// Andrew's fix: immediately flush it to disk
  sync_dirty_buffer(bh);
```

Explain why this is not the default behavior for Linux. In other words, why doesn't `set_buffer_dirty` simply call `sync_dirty_buffer` by default? Give an example workload where this would behave badly.

Synchronous writes would result in poor performance. One reason is that many small writes to a single buffer would each result in a disk write. Another reason to favor asynchronous write-back is that it allows the desk block scheduling algorithm to make smarter choices.

2) We learned that para-virtualization is a technique in which the virtual machine monitor exposes a different virtual architecture than the underlying physical hardware architecture. Could we use para-virtualization to add a No-Execute bit (which we studied in the context of buffer overflows) to the virtual architecture (assume that the physical hardware lacks the NX bit). Why or why not?

This will not work. VMMs work efficiently by interposing on a small subset of instructions -- the privileged instructions. Ordinary memory loads (via code execution) are not privileged. Conceivably, one could try to fake this by using virtual memory tricks, but the performance impact would be enormous. This is counter to how VMMs typically work.

3) The argument below is bogus. Give one reason why.

High TLB hit rates are predicated on having large pages. You can't shrink the page size without affecting TLB hit rate

The time to fault in many small pages is large relative to a smaller number of larger pages.

4) Java's version of remote procedure call is Remote Method Invocation (RMI). Java RMI is not transparent: remote methods look and behave differently than local methods in some cases. Provide an example of how transparency is broken. Why did the Java designers make this choice?

All methods throw remote (to expose failures). Copy-by-value semantics to avoid copying large arguments unnecessarily.

5) Explain how you could use UNIX's `setuid` mechanism to enforce the principle of least privilege.

Create a user for a particular program. Assign the user a restricted set of file access privileges -- the smallest it can possibly get away with. Set the program as the owner for the executable. Enable the setuid bit.

6) Fast symbolic links [8 points]

Traditionally, UNIX symbolic links were implemented by storing the link information in a separate block (essentially, in a separate file). Propose an improved version of symbolic links that is faster and more space efficient. It is acceptable to optimize for the common case of short file names and path links. You should describe how your data structures differ from those used in the CSE451 file system.

Two options here: one store the symbolic link information in the directory entry; or store the symbolic link information in the inode. Both approaches are viable, and have been attempted.

The state associated with a symbolic link is the complete path to the linked file. For the inode solution, we can store the symbolic link information inside the data block array. Symbolic links have no "state", so we can safely use the data block pointers to contain the path information.

For directories, we would need to change the directory entry format to contain two strings: the file name, and the complete path to the linked location. We should retain "short" directory entries for ordinary files...

```
struct long_directory_entry {
    boolean isSymbolicLink;
    char name [NAME_LENGTH];
    char link_location [LINK_LENGTH];
}
```

7) Asynchronous memory checkpointing [10 points]

Some operating systems have provided a mechanism to checkpoint and restore the memory state of a process. One approach is to *synchronously* flush memory state. Under this model, all threads in the process are halted while dirty memory is flushed to disk.

By contrast, your task is to design an *asynchronous* memory checkpoint facility. Under this model, the checkpoint operation proceeds in the background while the process continues to execute in the foreground. It is important that your asynchronous checkpoint implementation *behaves identically to synchronous flushing.* That is, you should capture a snapshot of memory as it existed at a single point in time. You should minimize the impact on the foreground process (and its threads). You do not need to implement a clever storage strategy; a simple flat file should suffice.

*We can use virtual memory page protections to allow threads to continue executing while guaranteeing a consistent snapshot of memory. The intuition is that it is always safe to *read* a page of memory. But, we must prevent page writes until that page has been safely written to disk.*

First, mark all pages in the address space as read-only. Fork a background thread that flushes pages to disk. When a page has been written back, then mark the page as valid (readable and writeable).

If a thread tries to write to a read-only page, a page fault will be generated. That thread must block until the page is written back. To minimize the performance impact of writeback, the faulting page should be given the highest priority for writeback.

8) Buggy Job Service Implementation [10 points]

The code below implements a `JobService` in Java. Callers can submit jobs in the form of `Runnable` objects. Internally, the `JobService` uses a thread pool to execute the jobs in parallel. (This is very similar to Java's `ExecutorService`, for those of you who used this in project #2).

This code contains one or more problems that affect safety, liveness, or performance. Identify as many problems as possible (but be careful not to identify non-problems). Then, propose solutions to those problems.

To make things painfully obvious for the graders, please be clear whether you are talking about a problem or a solution. For example: "Problem: code is not thread-safe because... ; Solution: Add synchronized block..."

There are two major problems with this code.

1) Problem: There is little opportunity for parallelism because the jobs are run while holding the queue's lock.

Solution : shrink the worker thread's critical section to exclude task execution. Note that we must rearrange size-- to be inside the critical section.

2) Problem: It is possible for sleepers to never wakeup. This is because the condition variable can contain threads that are blocked on queue empty and queue full.

Solution: Use notifyAll instead of notify. Or, use a split condition variable on the single queue lock.