Student ID: ____________

CS-736 Midterm: Live and Let Die
(Fall 2003)

Please Read All Questions Carefully!

There are twelve (12) total numbered pages

Please put your NAME on this page, and your STUDENT ID on this and all other pages

Name: ____

Student ID: ____________

This is the grading page.

	Points	Total Possible
Q1		20
Q2		20
Q3		20
Q4		20
Q5		20

1. **Virtual Machines**

Disco is an example of a virtual machine monitor (VMM), which runs underneath of an unsuspecting operating system (SGI's IRIX) in order to enhance service or provide some kind of functionality that would otherwise be unavailable. In this question, we will explore VMM technology.

1: First, describe the flow of control when an application running on a typical operating system (IRIX) running on a VMM (Disco) issues a system call (a trap). What pieces of code get invoked? (a picture is worth a thousand words, so save some ink and draw one)

A trap takes the processor into kernel mode, which in this case is occupied by Disco. Disco's trap handlers interpret that an application running at user level wants to jump into the OS, so Disco then calls into the OS, first downgrading to supervisor mode. The OS runs, executes the system call code, and tries to return from the trap with a "return from trap" instruction. This faults to Disco again, which then returns control to the user application, switching to user mode before returning control.

2: Now let's imagine we are considering a TLB miss. What typically happens on a TLB miss? (again, consider a picture of control flow).

On a TLB miss, almost the exact same control flow occurs. TLB misses are handled in software on MIPS, and so when a TLB miss occurs, the OS is usually invoked. In this case, the fault triggers Disco, which redirects it back into the OS to handle the fault and update the TLB. The TLB update instructions will trigger a fault and hop back into Disco, which will do the real update of the TLB. Finally, when the OS is done, it will once again try to return from a trap, which will hop back into Disco and then to the application, which will retry the faulting load/store. The mode switching that occurs is as above.

3: What does Disco do to speed up the operation of TLB misses? Why can't the OS itself do the same optimization? (or can it?)

Disco speeds up TLB misses by maintaining its own cache of valid address translations. That way, when it sees a miss, Disco can directly update the TLB with the needed translation, avoiding all of the hopping back and forth between Disco and the OS. In this case, the OS code to handle faults is never invoked. The OS could have a software TLB, but it would have to be faster than just looking up something in the page table to make sense. In this specific example, it wouldn't help much – all of the IRIX/Disco crossings would still take place.

4: Finally, assume that the operating system is no longer "unsuspecting" of the change – rather, assume we are building the OS with the knowledge that it will be run upon a VMM, and that the VMM can be changed in small ways as well. Describe *at least two* changes that could be made to increase the overall efficiency of such a system.

The type of thing we should be looking to change involves situations where it would be best if work was not replicated across both the OS and the VMM. For example, consider the zeroing of pages. Disco must zero each page that it hands to guest OS's (for security reasons), but OS's typically do the same thing for pages which they hand out to applications (for the same reason). Thus, an interface between the OS and the VMM in which the OS asks for zeroed pages and thus does not re-zero the page makes sense. Another example arises when the OS goes idle; at that point, an interface in which it informed the VMM that it doesn't need the processor would allow the VMM to run another guest OS. There are of course many other examples.

2. **Mirrored File Systems (FFS, LFS, RAID)**

In this question, we explore the utilization of a hybrid file system that is built on top of two existing file systems, namely FFS and LFS.

1: Before getting into details of design, describe a workload in which FFS would do much better than LFS.

FFS allocates files together that it thinks would logically be accessed together, in particular the inodes and data blocks of files that reside in the same directory. This allocation strategy holds even if the files are created at widely different times. In LFS, files are grouped by when they were created, as they get batched into segments and written to disk. Thus, if there was a workload that read files that a) were in the same directory and b) had been created over a long period of time, FFS would likely do much better, inducing fewer seeks. There are of course other examples.

2: Now, describe a workload in which LFS would do much better than FFS.

This one is easier. LFS takes small, random writes and turns them into large, sequential writes. Thus, a workload with a lot of small, random file creations or random updates to a large file would do much better on LFS than it would on FFS.

Now assume we are building a hybrid file system on top of FFS and LFS (call it FLFS). Assume we have a two-disk system (for simplicity), and that conceptually we are running FFS on one disk and LFS on the other. Your job is to design a file system that directs reads and writes to one or the other or both of these file systems.

3: How might we direct reads and writes in the system so as to get the best *performance* out of the system? What issues arise, and how might you address them?

There are lots of possible approaches here. One approach would be to direct writes to LFS, and reads to FFS, with the idea being that LFS does well for writes, and FFS does well for reads. This leads to an obvious problem, though, in that if a process tries to read a file it has just read, it is on the LFS disk, and not the FFS disk. Thus, one might use idle time and the LFS cleaner as an opportunity to move files from the LFS disk to the FFS disk. Another issue that must be addressed is the additional meta-data that FLFS needs to track, in particular any meta-data that tracks which file system (FFS or LFS) a given file is in. This meta-data would have to be kept on disk.

4: How might we direct reads and writes in the system so as to get the best *reliability* out of the system? What issues arise, and how might you address them?

Reliability in a multi-disk system is best achieved through redundancy. Thus, we'd like for data to end up in both the FFS and LFS file systems. Thus, the simple solution is just to mirror all files to both file systems. A slightly more nuanced approach would first write files to LFS, and then copy them into FFS during idle time (similar to our approach above, but with a copy instead of a move). This would enable higher performance during a burst of writes, and reasonable reliability once the copy is made.

3. **RAID-6**

In this question, we consider a new RAID level, called RAID Level 6, and we compare it to RAID Level 5. RAID-6 works as follows: instead of having just one check disk, RAID-6 has *two* check disks. Without getting into the details of how Reed-Solomon coding works, assume that if the data in a stripe changes, *both* check blocks will have to be updated too. Also, assume that the basic design allows the loss of any *two* disks without a perceived loss of data.

1: First, let's make sure we understand RAID Level 5. How does RAID-5 work (pictures are useful)?

RAID Level 5 is block-level striping plus rotated parity. See the paper for details!

2: What is the main performance problem with RAID-5, as related to "small" writes?

The problem with "small" writes, or more specifically writes to a single block in a stripe, is that they cause 4 separate I/O's to take place. Two to read the old data and old parity, and then two more to write out the new parity and new data. This reduces the effective bandwidth of RAID-5 to 1/4th of its potential. Latency is increase by a factor of two, as the two reads are issued and must complete before the two writes can be issued.

3: Now, let's understand the same issue for RAID-6. How would a "small" write work on RAID-6?

RAID-6 has an even bigger small write problem: it must read in the old block and both old parity blocks, and then write out the new block and two new parity blocks. Thus, 6 I/Os are needed.

4: What is the performance impact of RAID-6, as compared to RAID-5? When should a storage administrator decide to use it instead of RAID-5?

In a small-write intensive workload, RAID-6 induces 6 I/Os instead of 4 per small write, and thus is 50% worse than RAID-5. For this type of workload, it probably doesn't make a lot of sense. When you want something like RAID-6 is when you expect double failures as a possible occurance, perhaps in a very large disk array for example. In such a case, the extra assurance that two disks have to fail before you get into trouble might be a good idea. Also, if a workload is read intensive and space isn't a big concern, only the slight additional space cost of RAID-6 over RAID-5 might be fine, and performance would be similar.

4. **Gray boxes**

   In this question, we explore the concept of the "gray box" approach to building systems. Assume we wish to build a gray-box layer on top of the file system that gives us more control over where files get allocated on disk. Assume further that we are running on top of FFS.

   1: How does FFS decide where to place files on disk?

   FFS, as stated above, tries to collocate related files on the disk. Related in this sense is defined as files that are in the same directory, as well as their inodes. FFS also tries to spread out unrelated data, and thus keep space available in groups for files that are allocated later.

   2: How could we exploit knowledge of this algorithm in order to give applications the ability to better control which files end up near one another on disk?

   This is pretty hard. We know that files that are created in the same directory end up near one another on disk (probably). The question is thus: how can we exploit that to give applications better control? One simple approach would be as follows: create a bunch of directories, say one for each cylinder group on disk. Try to make sure each directory is in a different group – this is a bit tricky, but you could use inode numbers to get a good guess as to which group a file is in. Then, when a user wants to put two files near one another, create them first in the same group directory. The problem is that the user wants to create a file (say "/x/y/foo"), and now you've put "foo" in /group/3 (or whatever). So then simply rename() the file to it's final spot in the directory hierarchy.

   Now let's think about a gray-box approach to a lower-level of the disk system. Assume we are running on top of a RAID system, but we don't know how it's configured. Specifically, it could be RAID Level 0 (striping), RAID Level 1 (Mirroring), or RAID Level 4 (Parity-based).

   3: Design a benchmark that figures out which of these three RAID levels you are running on. Assume you can do reads and writes to the disk's address space directly (these are called "raw" reads/writes), and also assume that you know how many total disks are in the system. Remember that you can use *timing* to elicit information about what is going on.

   Start by running this simple test: issue a large number of random block reads to the RAID, varying the concurrency (number of concurrent reads) from 1 up to D (the number of disks), and timing how long they take to complete. Assume from this that we get a bandwidth B MB/s for the system. Now, run the same experiment, but for writes. Writes achieve half the bandwidth in RAID 1 (due to mirroring), and hence you can differentiate 0 from 1. Writes achieve 1/4th of the bandwidth for RAID 4, due to the small write problem. Hence, by comparing read and write performance for random blocks, we can tell if the RAID is a level 0, 1, or 4 system. There are of course other approaches.

5. **Memory Activations**

In this question, we explore the generalization of the concept of scheduler activations to other resources in the system, specifically memory. Your job is to design this *memory activations* system.

1: Before getting into memory activations, let's discuss some of the key points of scheduler activations. One of the keys to scheduler activations is the notification to the user-level thread library when something changes in the system. When do these kernel-to-library notifications occur, and why is it useful for the user-level library to know of such changes?

The basic point of scheduler activations is to let applications/user libraries know how many processors are really available, so that the upper-layers of the system can best schedule the resources that have been given to them. Thus, when the processor allocation changes, the user-level needs to be informed, so it can decide which threads to run (and which not to run). The app/lib also needs to know when a given activation blocks (either on I/O or a page fault), so it can decide to run something else. Both of these notifications enable full control of scheduling processors that have been given to a process at user level.

2: Now imagine that we wanted to apply the activation concept to memory. What kind of information should be *passed up* from the kernel to the user-level library/application?

Memory activations are a little harder to think about. They really are just more like callbacks, letting you know when the amount of memory that is available to your process has changed. Thus, whenever the kernel wants to give you more memory (in a response to a request for more), or when it wants to take memory away (a replacement), it needs to let the user-level know about it. More nuanced information passing is possible as well (e.g., info about physical pages?)

3: Similarly, in our memory activation system, what type of information should be *passed down* to the kernel from the application?

The user-level library should simply let the kernel know how much memory it needs. When it is not using some memory, it should let the kernel know that too. Also important: when the kernel tells the application that it needs to use less memory, the application needs some way to tell the OS which memory is more important to it. There are lots of different approaches one could take to that part of the interface. The details are left to the reader.

4: Why is applying the activation concept different for memory than it was for the CPU? What works better, what doesn't work as well? Discuss.

The reason memory activations are painful comes down to this: how should the application deal with the fact that it's been told that it should now use less memory? For some applications, some minimal amount of memory is needed just to make progress – what should those applications do when they are given less than this? Note this is much different from scheduler activations; therein, a process can be given 1 to N processors, and can make use of any of those amounts (given enough threads). It makes perfect sense to give a process one processor, whereas it probably doesn't make sense to give a process 1 byte of memory. Therefore, processes probably need to be able to specify some memory minimum that makes sense for them – if that much is not available, perhaps the process shouldn't be run at this time.