CS-537: Midterm Exam (Spring 2009)

The Future of Processors, Operating Systems, and You

Please Read All Questions Carefully!

There are 15 total numbered pages.

Please put your NAME and student ID on THIS page, and JUST YOUR student ID (but NOT YOUR NAME) on every other page. Why are you doing this? So I can grade the exam anonymously, of course.

Name and Student ID: _____

Student ID: _____

Grading Page

	Points	Total Possible
Q1		10
Q2		10
Q3		10
Q4		10
Q5		10
Q6		10
Q7		10
Q8		10
Q9		10
Q10		10
Total		100

The KiloChip (kchip) and kilOS

This final examines the operating system issues around a new technology advance: the multicore CPU. In fact, we will be looking at a series of questions about a future multicore chip with 1000 processors. Because it has 1000 processors, the designers have cleverly called it the "Kilochip" or "kchip" for short.

The processors on the kchip all share a single memory and I/O subsystem, and in this way the kchip is similar to current systems; there are just more processors. Because it is in the future, though, there is more of everything, including memory (terabytes) and a new type of long-term storage device (a new solid-state device called a "flisk"). We'll be hearing more about those later.

We'll now systematically re-examine each of the components of an operating system with this new hardware architecture in mind. This new OS for the kchip is appropriately called "kilOS" (pronounced "kill us", although you shouldn't take that personally!). It is a little different than a typical OS, as we will see.

Important: Some questions build on previous ones. Thus, you should not skip around (or at least, you should first read through a question before not answering it).

1 k-Scheduling, System kilo-Calls, kilo-Interrupts

1. On a normal system, a *time-sharing* scheduler is used to run one job for a little while and then switch to another, running each job for a *time slice*. How long should a time slice be? (justify)

2. The designers of kilOS decided that with so many processors, time-sharing of a single CPU wasn't needed. Instead, kilOS places each process/thread on a dedicated processor and lets it run to completion; among competing users, it divides up the CPUs among them fairly (e.g., with two users, each would get half of the available processors). When will this strategy work well? When will it work poorly?

3. On a normal system, applications make a *system call* to invoke OS services. What happens during a system call that is different than a typical procedure call?

4. In kilOS, OS code runs on a different, dedicated set of processors (say 100 of the 1000); these are called "kernel processors". Applications run on the remaining chips, called "application processors". To communicate with a piece of the OS running on a kernel processor, applications issue "k-calls". A k-call sends a message to the desired OS service running on a different processor on the kchip and then waits for the reply from the service. What about this arrangement (of dedicated application processors and kernel processors) works better than a more typical system? What works worse?

5. On a normal system, a *timer interrupt* is a critical hardware feature for the OS. Why is this?

6. In kilOS, there are no timer interrupts, as the kchip designers wanted to simplify the hardware and knew that the general scheduling approach was to run jobs to completion on application processors. In what situations would kilOS suffer from a lack of timer interrupts?

2 kilo-Threads

The kilOS/kchip system also supports multi-threaded applications and the requisite synchronization primitives. Let's examine them in more detail. Remember that each thread runs on a dedicated application processor (i.e., no time sharing).

1. On a normal system, sometimes a thread library will use a simple *spin lock* to implement mutual exclusion. Why do spin locks sometimes perform poorly on a single processor?

2. On kilOS, why would spin locks perform well between threads running on different application processors?

3. A simple spin lock can be implemented with hardware support, say in the form of a test-and-set instruction. Write the code for such a lock – just the lock code (skip unlock); assume the lock_t data structure has a single integer named "flag" in it which you can use to indicate the status of the lock).

```
typedef struct __lock_t {
  int flag; // init to 0
} lock_t;

void lock(lock_t *lock) {









}
```

4. On kilOS/kchip, even though spin locks generally perform "well", they are not ideal. What are some other negative properties of simple spin locks such as these?

3 kilo-Test and kilo-Set

The kilOS thread team decides that using normal test-and-set is not a good approach, because in talking to the kchip team, they learn that the test-and-set instruction is quite expensive (as compared to a normal load from memory, for example). Thus, they instead use this code to implement a lock:

```
typedef struct __lock_t {
  int flag; // init to 0
} lock_t;

void lock(lock_t *lock) {
  do {
    while (lock->flag) // unprotected lock check
      ; // spin
  } while (TestAndSet(&lock->flag, 1)); // actual atomic locking
}

void unlock(lock_t *lock) {
  lock->flag = 0;
}
```

1. Does this lock work correctly? (why or why not?)

2. When does it perform better than a simple spin lock built with test-and-set? When does it perform worse?

The designers decide they feel unsafe about the unlock code. So they change it to this:

```
void unlock(lock_t *lock) {
  TestAndSet(&lock->flag, 0);
}
```

3. Does this unlock work correctly? (why or why not?)

4. Does this unlock perform better, worse, or the same as the earlier implementation?

4 kilo-Condition Variables

Condition variables are useful in building certain kinds of synchronization between threads. Thus, kilOS has these as well.

1. In a typical system, describe how condition variables are different than locks.

2. In a typical system, describe how condition variables are similar to locks. In particular, in what cases would these primitives (locks, condition variables) interact with the system scheduler?

3. In kilOS, condition variables are implemented slightly differently than in most systems. Specifically, when wait() is called on a particular condition variable, it just spins waiting for the condition to become true instead of putting the caller to sleep. Is this simple implementation OK for kilOS? (why or why not?)

5 Virtualizing k-memory

The kchip system has a huge amount of memory. One thing that kilOS must do is thus manage that memory. The kilOS design goes for simplicity again: just a single *base* register holds the physical address of the currently-running address space, and is added to the virtual address generated by the running program to compute the final physical address.

1. What are the strengths and weaknesses of the approach?

After some initial results showed that simple relocation was not very flexible, the kilOS team decided to build a new system with paging. The paging support was very simple, though: a bunch of 1MB pages with linear page tables and a software-managed TLB.

2. Describe what would happen on a TLB miss in such a system. What structures (hardware and software) are accessed?

3. The use of large pages and linear page tables was controversial within the kilOS team. Given that these systems will all ship with large amounts of memory, please justify this decision.

6 kilo File Systems

The kchip system builders decided disks were bad and a new medium call flisks were better. Flisks provide excellent random-access performance, as you can access any block in roughly uniform time (it takes T time units per block). Flisks also provide a special high-speed access mode when you read 100 contiguous blocks; such an operation is only 2 times slower than accessing a single block (2T). Writing of 100 blocks takes the usual time, though: 100 time units (100T).

1. What aspects of the Berkeley Fast File System would be useful on a flisk? (explain)

2. What aspects of FFS would *not* be needed anymore? (explain)

3. What aspects of the Log-structured File System, which writes to disk in large segments, would be useful on a flisk? (explain)

4. What aspects of LFS would *not* be needed anymore? (explain)

7 kilo-Caching

Memory caching plays a big role in any file system, even when run on flisks. Further, there is a vast amount of memory available on our kchip system, and thus copious amounts of space for caching.

1. Assuming we are reading a file of one block in size on a typical FFS-like file system, and the file is called `/1/2/3/4.txt` and contains exactly 1 file system block. How many reads will it take (including data and metadata) to read `4.txt` from the flisk? (assume that directories are only 1 block in size as well).

2. With a vast amount of caching, no block will ever be read from flisk more than once. Thus, if after reading `/1/2/3/4.txt` we decide to read `/1/2/3/5.txt` too. How many extra reads will take place?

3. In a system with a vast amount of memory, does any flisk I/O ever take place? In what circumstances?

8 kilo-Journaling

Unfortunately kilOS has bugs and still crashes occasionally. For file system updates, this causes a problem: file system structures may be left inconsistent or data may be lost.

1. On a typical file system without journaling (such as FFS), describe a sequence of events that would lead to data loss (but no file system inconsistency).

2. On a typical file system without journaling (such as FFS), describe a sequence of events that would lead to file system metadata inconsistency.

3. On a typical file system without journaling (such as FFS), describe a sequence of events that leads to garbage data being part of a file.

4. kilOS's version of FFS uses journaling to handle file system crashes gracefully and efficiently. Describe how journaling works, and how it can avoid all the problems that occur above.

9 kilo-RAID

Being a smart design squad, the kilOS team decided to build RAID support into their OS. Specifically, they decided to include a *software RAID*, which is basically a layer within the OS that implements RAID on top of many devices without any extra hardware support. Specifically, the file system just uses two routines, RAIDRead() and RAIDWrite(), to write to persistent storage; these routines then do the hard work of mapping the requests to the underlying flisk storage devices.

For this problem, assume that each flisk has 1 million 4KB blocks, and that each request to RAIDRead() and RAIDWrite() is to read or write one of those blocks (respectively). Assume further that we have a mirrored system, with only two flisks. Finally, assume that to read or write a particular flisk, you have to issue a call to fliskRead() or fliskWrite(). These are defined as follows:

```
int fliskRead(int flisk, char *buffer, int offset);
int fliskWrite(int flisk, char *buffer, int offset);
```

fliskRead() takes a flisk number (0 or 1 in this case) and reads the data at block offset "offset" into the memory location pointed to by "buffer". fliskWrite() is similar except it writes the data pointed to by "buffer" to the flisk. Each returns 0 upon success and -1 upon failure; errors can occur and thus your code should be prepared to handle them.

1. Write the code for RAIDWrite(). The routine should only return 0 (success) if the data was successfully committed to both disks; -1 otherwise.

```
int RAIDWrite(char *buffer, int offset) {









}
```

2. Write the code for RAIDRead(). The routine should return 0 if it was able to read the data successfully; -1 otherwise.

```
int RAIDRead(char *buffer, int offset) {









}
```

Now you have to write one last piece of code for the software RAID. It should simply scan through all of the blocks within the RAID and make sure that each mirrored copy agrees with each other (that the two block copies are identical). If two blocks are identical, the routine should set the corresponding slot in the "broken" array to 0; if they are not, it should set it to 1 (this will be used by a subsequent repair routine).

3. Write the code for RAIDScan(). Return 0 if and only if there are no failures detected; -1 otherwise.

```
int RAIDScan(int *broken) {









}
```

Assume we now have a RAID-4 (parity-based RAID with a single parity disk). Assume further that we have 3 flisks, two for data and one for parity. Write the code that reads a block. Note that it should be able to handle the case where the data disk fails and thus reconstruct the data from the other data and parity information. You can assume the presence of an xor() function as needed.

4. Write the code for RAIDRead() for RAID-4. The routine should return 0 if it was able to read the data successfully; -1 otherwise.

```
int RAIDRead(char *buffer, int offset) {









}
```

10 kilo-Distributed File Systems

Finally, the kchip/kilOS system makes for an awesome distributed file server. In a distributed file system such as NFS or AFS, *cache consistency* is a big issue.

1. Describe the cache consistency problem. Use examples if you need to.

2. Describe how NFS handles cache consistency. Why does NFS write dirty data from the client to the server when an application calls close()?

3. Describe how AFS handles cache consistency. Why is AFS considered more "scalable"?

4. Given that our kilOS/kchip system is used in this setting, and has lots and lots of memory, will NFS *server* performance improve much in this new world? (why or why not?)