Student ID: _____

CS-537 Exam: The Final Conflict (Spring 2002)

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: _____

Grading Page

	Points	Total Possible
1		25
2		30
3		25
4		20
5		20
6		30
Total		100

Student ID: _____

Questions

All questions are in the long answer style. There are six total questions, and then you are out of here!

1. **Be AfRAID, be very AfRAID (25 points)**

   This question explores RAID storage technology.

   a) Assume you have a RAID-4 based system, which uses parity as a redundancy mechanism. In RAID-4, something called the "small write" problem occurs. What is the small write problem, and why is it bad?

   b) RAID-5 is a slight variant of RAID-4. How does RAID-5 work?

   c) In what way does RAID-5 address some of the problems of RAID-4?

Be AfRAID, be very AfRAID (Continued)

d) One idea to improve the performance of RAID-4 and RAID-5 based systems is to *delay* the update of the parity block. For example, assume that under a write, the parity block does not need to be immediately updated, but only after some amount of time. How could this improve performance?

e) What problems are introduced by delaying updates to the parity block?

2. **A Quest for Information (30 points)**

In this question, we explore the addition of new interfaces to the OS. Specifically, we add *informing* operations, which can be called to gather information about what the OS is doing or some of its internal state.

One informing operation is called `float File_GetPercentInCache(char *file)`. When called, this routine informs the caller of the percent of the file (named by the string `file`) that is in the file system buffer cache. For example, before any process has accessed a particular file `foo.c`, none of the file will be in the cache. Thus, if `File_GetPercentInCache(''foo.c'')` is called, it will return 0. If a user reads `foo.c` in its entirety, and the file is not "too big" (i.e., bigger than the size of the buffer cache), the entire file will likely reside in the buffer cache. If `File_GetPercentInCache(''foo.c'')` is called, it will thus return 100. As blocks of the file `foo.c` get evicted from the cache, a percentage between 0 and 100 may be returned.

Let's examine the following bit of pseudo-code:

```
int cnt = 0; // global variable

void search(char *file) {
    int fd = open(file, O_RDONLY); // open file for reading
    char c;
    while (read(fd, &c, 1) > 0) { // read file, one byte at a time
        if (c == 'a')
            cnt++;
    }
    close(fd); // close up properly
}

// idea: search through all files for total number of a's
int main(int argc, char *argv[]) {
    foreach file (in a list of files obtained from argv) {
        search(file);
    }
    printf(''total number of a's in files: %d\n'', cnt);
}
```

a) Assume that the time to run this program is completely dominated by the time to read the files, i.e., that all other operations take near-zero time. Assuming that the program reads in 100 4KB files, and that the disk takes on average 10 ms to read a 4KB block, how long will it take to read all 100 files from disk when the program is run? Assume that all of the files are on-disk and not cached, and **show your work!**

A Quest for Information (Continued)

b) Now assume that you have run the program once, and all of the files are in main memory in the buffer cache. Assume that reading 4KB from main memory takes 50 microseconds. How long will it take to read in all of the files now? **Show your work!**

c) An interesting case occurs when repeatedly reading the same set of files over and over again, but when the total size of the files is *slightly bigger* than the size of the buffer cache. Assuming LRU replacement of blocks within the cache, a cache that can fit 999 4KB blocks, and assuming that the program is run twice (2 times), each time accessing 1000 4KB files (the same 1000 files are accessed during each run), how long will it take to read the 1000 files during **the second run of the program?**

A Quest for Information (Continued)

d) Finally, we get to use the informing interface. Re-write the above code to use the `File_GetPercentInCache()` interface. Your goal should be to **maximize the performance of the code**, i.e., reduce its total running time.

e) Now, again assume the scenario in part (c), i.e., that the buffer cache can fit 999 4KB blocks and does LRU replacement, the program is run twice, each time accessing the same 1000 files, each of which is 4KB in size. How long does your improved code take to read the 1000 files during **the second run of the program?**

3. **Pardon the Interruption (25 points)**

In this problem, we revisit the concepts of critical sections and locking, with a focus on using interrupt masking to synchronize access to shared data.

a) What is a critical section, and why are they important to concurrent programs?

b) One technique used to implement a lock was to turn off interrupts, say with a routine `TurnOffInterrupt()`, and then re-enable interrupts when unlocking `TurnOnInterrupts()`. Why is it dangerous to expose this functionality to user-level programs?

c) We introduce a new primitive, called `TurnOffInterrupt(float amount)`, that turns off interrupts for a bounded amount of time, namely for `amount` seconds. After that time, interrupts are automatically re-enabled in the hardware. How does this primitive solve the problem introduced by the traditional interrupt off/on lock described in part (b)?

d) Paired with this new routine is the traditional routine `TurnOnInterrupts()`, which re-enables interrupts on the processor. Does this new routine have any value? (Should users bother calling it?)

e) What aspects of modern systems would make a timed interrupt mechanism like this one difficult to use?

4. **It's All A Game (20 points)**

In this question, we explore the process of "gaming" the scheduler. In gaming, the idea is to exploit aspects of the scheduler design or the behavior of other processes so as to gain an unfair share of the CPU for your process.

a) Assume a priority-based scheduler that works as follows. A newly created process enters at the highest priority. If the process uses up its entire time quantum when it runs, it moves to the next lowest priority. If a process initiates an I/O request, it is moved back to the top priority, and its time quantum begins anew. How would you "game" this scheduler?

b) Assume a lottery-based scheduler, and that you only have a single ticket. Also assume there is a single lock in the system that most processes periodically grab, in order to examine some shared state. What could you do here in order to gain an unfair share of the CPU?

c) Assume a shortest-job first scheduler, which somehow "knows" the run-time of each process, and schedules the shortest ones first. How could you (re)write your programs so as to take advantage of this scheduler?

d) Finally, assume an "efficiency-based" scheduler that gives preference to programs that exhibit good file cache behavior. In our efficiency-based scheduler, a process that has a higher file-cache hit rate will be run more frequently by the scheduler. How could you exploit this knowledge to game the scheduler?

5. **Your Losing Your Memory (20 points)**

In this problem, you have to design a virtual memory system for TinyOS, an operating system for a handheld device. Unlike a traditional modern OS, you don't have a lot of memory to fool around with, and so saving space is important.

For this question, assume that your TinyOS is running on a handheld with 1K pages, and on this handheld in particular, there is 128KB of physical memory. Assume addresses (virtual and physical) are 20 bits long.

a) Let's say we want to use a single linear array as the page table of a process. Assume that each page table entry needs, in addition to any translation information, 2 bits for protection information, and 4 other bits for miscellaneous stuff. How much memory will the page table occupy?

b) Now let's try a different structure to track translations, called *MiniTable*. MiniTable is a linear array that is proportional to the size of *physical* memory, with one entry per *physical* page. Each entry specifies which process has the page mapped (i.e, a process ID), as well as the virtual page number of the page that maps to this physical page. How much space does a MiniTable occupy? (assume that a PID is 6 bits in length)

c) One problem with MiniTable is the **time overhead** to perform a translation. How much time does a virtual to physical translation lookup take in MiniTable, as compared to the linear array described in part (a)? Please state your assumptions.

d) Another problem with MiniTable is that it does not easily allow processes to **share** a page. Describe why this is, and how you might fix MiniTable to allow sharing.

6. **Scalable File Systems (30 points)**

   You are in charge of designing a new file system for Linux. Unfortunately, the Linux community won't accept the standard Berkeley FFS – they need more. In each of the following questions, we will try to address some of the scalability limits of FFS in our new file system, XFS.

   a) Assume the standard FFS supports an 8KB block size, and an inode has 12 direct pointers, a single pointer for indirect blocks, and a single pointer to a doubly-indirect block. Assume also that each pointer is 32 bits in size. What is the maximum file size supported in FFS?

   b) Let's say in XFS, we want to support larger files. Name at least two things we can change about FFS that will allow us to support larger files.

   c) In FFS, the number of inodes that can be allocated is fixed when the file system is first created. Describe why it is fixed (i.e., describe how inodes the allocation and deallocation of inodes are managed).

   d) In XFS, we want to have a more flexible system, that allows the number of inodes to grow over time. Describe how you could implement this.

Scalable File Systems (Continued)

e) Directory operations (such as looking up a file in a directory) take too long when there are **lots** of files in a directory in FFS. Describe how FFS (or most Unix file systems) organize data in a directory, and why a directory operation would take a long time when there are a lot of files in a directory

f) In XFS, we want to have fast directory operations, even if the directories have lots of files in them. Describe how you might solve this scalability problem. **Hint: caching the directory data is not the answer!**