Student ID: ___________

CS-537: Getting into the Flow (Fall 2008)

Solutions

Please Read All Questions Carefully!

There are fourteen (14) total numbered pages.

Please put your Name (below) and student ID (top left) on this page

Please put your student ID (but NOT YOUR NAME) on every other page.

Name: ____

Student ID: ____________

Grading Page

	Points	Total Possible
1		10
2		10
3		10
4		10
5		10
6		10
7		10
8		10
9		10
10		10
11		$style\ points$
Total		$100 + style\ points$

“The quality of the imagination is to flow and not to freeze”
Ralph Waldo Emerson

In this exam, your goal is simple: to trace the **flow** of what is happening in some of the systems and code sequences we have studied. For example, when we have a file system that uses *memory* for caching data, a *disk* for its persistent storage, and a *user buffer* in the application’s address space to read data into, we might use these labels:

```
1. user buffer
2. OS cache
3. disk
```

If a question then asked, “what is the flow of data when an application first reads data from disk?” you might answer, “3, 2, 1” because the data is read from the disk into the OS cache and then finally copied into the user buffer. If the block is read again, the flow is simpler: “2, 1”. Why? Because the data is fetched from the cache and avoids the disk altogether.

Note that some answers might have no flow at all, i.e., the empty flow. For those questions, you can just write down “empty” or “nada” or “zilch” or “–” or “$\emptyset$” or any other of your favorite symbols for an empty flow.

In this exam, most of your answers will be flows of this nature. Of course, each question asks for slightly different flows and their meaning is specific to each question. Thus, please **read each question carefully!**

Good luck! It has been a fun semester (for me, at least). Don’t ruin it now.

1. **Scheduling Flows.** The first flow you will concoct is the flow of jobs through a system. We have the following jobs: A, which runs for 10 seconds, B which runs for 20, and C which runs for 5. Assume the following labels for our flows:

```
1. A runs
2. A ends
3. B runs
4. B ends
5. C runs
6. C ends
```

Let's now answer some questions about what happens under different scheduling disciplines. If, for example, you wished to indicate job A ran to completion, you would write down: "1, 2" (to indicate A runs for a while, and then A ends). Of course, scheduling all three jobs will result in longer flows. Also note that while 1, 3, and 5 may be used multiple times in a flow, 2, 4, and 6 should only be used once to indicate when a particular job is all done.

(a) Assume shortest-job-first scheduling (SJF). What is the flow that occurs assuming we wait for all three jobs to be complete?

5, 6, 1, 2, 3, 4

(b) Assume round-robin scheduling with a short (1 second) time slice. What is the flow that occurs assuming we wait for all three jobs to be complete?

1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5, 6, 1, 3, 1, 3, 1, 3, 1, 3, 1, 3, 2, 3, 4

(c) Assume round-robin scheduling with a very long time slice (longer than 30 seconds). What is the flow that occurs assuming we wait for all three jobs to be complete?

1, 2, 3, 4, 5, 6

(d) Assume a biased round-robin scheduling that gives A a time slice of 2 seconds, B a time slice of 4, and C a time slice of 1 second each time they run. What is the flow that occurs assuming we wait for all three jobs to be complete?

1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 2, 3, 4, 5, 6

(e) Assume a multi-level feedback queue scheduler like the one we discussed in class; any round-robin scheduling that occurs has a time-slice of 1 second. What is the flow that occurs assuming we wait for all three jobs to be complete?

I accepted most any answer given that the question was underspecified. Something like round-robin would be fine.

2. **Semaphore Flows:** Assume we want to build a semaphore out of locks and condition variables. We have some basic code pieces lying around:

```
 1. while (s <= 0)
 2. while (s >= 0)
 3. while (s > 0)
 4. while (s < 0)
 5. while (s == 0)
 6. s--;
 7. s++;
 8. cond_wait(&cond, &lock);
 9. cond_signal(&cond);
10. mutex_lock(&lock);
11. mutex_unlock(&lock);
```

Your job is to implement some different semaphore variants with these code chunks. For example, a rather useless code sequence would be "6, 7", which would decrement s by 1 ("s–") and then increment it by 1 ("s++"). But this is how you specify a code sequence with flows.

Assume you are implementing a **binary semaphore**. Thus, you have to implement two routines: `sem_wait()` and `sem_post()`.

(a) What is the minimal flow of operations you need to implement `sem_wait()`?
10

(b) What is the minimal flow of operations you need to implement `sem_post()`?
11

Now assume you need to build a generalized semaphore. The semaphore is initialized with the following code:

```
sem_init(int value) {
    s = value;
}
```

where "s" is the variable that holds the value of the semaphore.

(c) What is the minimal flow of operations you need to implement `sem_wait()`?
10, 1, 8, 6, 11

(d) What is the minimal flow of operations you need to implement `sem_post()`?
10, 7, 9, 11

3. **Monitor Flows:** In this problem, we consider the producer/consumer problem written in a Monitor with condition variables. Consider this code:

```
int data[MAX];
int fill = 0; int use  = 0; int num = 0;
cond_t empty, wait;

void produce(int element) {
    while (num == MAX)          // p1
        empty.wait();           // p2
    data[fill] = element;       // p3
    fill = (fill + 1) % MAX;    // p4
    num++;                      // p5
    full.signal();              // p6
}
int consume() {
    while (num == 0)            // c1
        full.wait();            // c2
    int tmp = data[use];        // c3
    use = (use + 1) % MAX;      // c4
    num--;                      // c5
    empty.signal();             // c6
    return tmp;                 // c7
}
```

Assume there are two threads: one producer and one consumer, and that the producer repeatedly calls the produce() method and that the consumer repeatedly calls the consume() method. Assume also that a context switch only happens when one of the threads calls the wait() primitive on a condition variable. Assume that MAX=1. Also assume the producer runs first.

(a) Write down the flow (e.g., "p1, p3, ...") starting with the producer and ending when the consumer successfully consumes two data items.

Call "p1, p3, p4, p5, p6, p1, p2" A. Call "c1, c3, c4, c5, c6, c7, c1, c2" B. Answer: A, B, A, B.

(b) Do the same, but assume that MAX=2.

Call "p1, p3, p4, p5, p6" A. Call "c1, c3, c4, c5, c6, c7" B. Answer: A, A, p1, p2, B, B.

(c) Now assume that every time a signal occurs, a context switch occurs as well (if there is a runnable thread). Assume MAX=1 again. Assuming the producer runs first, write down the flow (e.g., "p1, p3, ...") starting with the producer and ending when the consumer successfully consumes two data items.

The question should have more carefully specified the semantics of signal here (e.g., assume that the signal transfers the lock to one waiting thread). In such a case, answer: p1, p3, p4, p5, p6, c1, c3, c4, c5, c6, p1, p3, p4, p5, p6, c7, c1, c2, c3, c4, c5, c6.

(d) Do the same, but assume that MAX=2.

Same as the last answer.

(e) Finally, assume there are two producers and two consumers, but only a single condition variable (not "empty" and "full" but a single condition variable "c"). Write down a flow that demonstrates why having a single condition variable doesn't work for the producer/consumer problem. You can assume context switches occur wherever you need them to.

Assume two producers, and one consumer. Answer: c1, c2, p1(producer 0 or just 0), p3(0), p4(0), p5(0), p1(producer 1), p2(1), p6(0), p1(1), p2(1). At this point the producer 1 was woken where one should have woken the consumer.

4. **Deadlock Flows:** In this problem, we consider flows that might lead to deadlock. Assume we have two threads, and they of course may be arbitrarily interleaved (depending on what the scheduler does). Assume that the two threads will repeatedly run the following sequences of code, respectively:

```
          Thread 1                                 Thread 2
     1. lock(&mutex1);                       a. lock(&mutex2);
     2. lock(&mutex2);                       b. lock(&mutex1);
     3. // critical section                  c. // critical section
     4. unlock(&mutex2);                     d. unlock(&mutex1);
     5. unlock(&mutex1);                     e. unlock(&mutex2);
```

(a) Write down a flow that leads to deadlock:

1, a, b, 2 or 1, 2, a, b, or a, b, 1, 2

(b) Write down a flow that leads to deadlock that has the fewest possible context switches:

1, a, b, 2 or a, 1, 2, b

(c) Write down a flow that leads to deadlock that has the most possible context switches and that does not re-use a flow element (e.g., 1 ... 5, or a ... e) more than once and where thread 1 starts just before “1” and thread 2 starts just before “a”:

1, a, 2, b, or a, 1, b, 2

(d) Write down a flow that has one thread in the critical section and the other waiting on a lock and no deadlock, starting thread 1 just before “1” and thread 2 just before “a” again:

1, 2, 3, a or a, b, c, 1

(e) Write down a flow that has one thread holding both locks and the other holding just one:

Empty

5. **FFS Flows:** This question is about the Berkeley Fast File System, FFS. Remember, the first file system that realized it was built on a mechanical disk? The question focuses on the flow of data to disk, i.e., where does FFS place files?

   Assume we have 10 cylinder groups (or block groups) on disk. As you might recall, each group has four basic components: an inode bitmap (Bi), a data bitmap (Bd), some inodes (I), and some data blocks (D).

   This question focuses on which data structures are touched when certain operations occur. For example, if an application read a 1-block file in the second block group, the flow would be: "2I, 2D", because to read that file, the application first has to read its inode and then its data block, both of which are in the second block group.

   Assume for these questions that all files and directories are 1 block in size unless otherwise specified. Also assume the root directory is found in block group 1.

   (a) There is an existing file "/foo" in block group 3. An application wishes to read it from disk. What is the flow of reads from disk needed to satisfy this request? (Remember to start at the root)

   1i, 1d (read root dir), 3i, 3d (read foo's inode, data)

   (b) Now an empty file "/bar" is created in the same directory as foo (i.e., the root directory). What flow describes the writes to disk needed to create "/bar"? (Note: the exact ordering does not matter in this case.)

   1i, 1d (update root dir), 3bi, 3i (allocate inode for bar)

   (c) Now assume that someone runs "ls" on the root directory and wants to obtain the size of each file or directory within. What is the flow of reads from disk?

   1i, 1d, 3i, 3i (assuming foo and bar each require one inode read, otherwise just one '3i')

   (d) Finally, assume another file is getting created in the root directory, "/big". Assume it is a really big file. As it keeps growing and growing and eventually nearly filling the disk, what flow might describe the its sequence of writes to disk?

   1i, 1d, (for the root directory), 3i (bigs inode), 3bd, 3d (a data block in group 3), 4bd, 4d (and in group 4), 5bd, 5d, ... (and maybe some 3i's in there to account for the inode of big getting more and more pointers)

6. **LFS Flows:** This question is about the log-structured file system, LFS. Remember, the file system that treats the disk kind of like a tape?

   LFS has a bunch of data structures, including some new ones that typical file systems like FFS don't have. Here is a list of some important ones:

```
1. inode
2. data block
3. segment summary block
4. inode map
5. checkpoint region
```

   In this question, you will write down which data structures are accessed **from the disk** to complete a particular operation. Assume you start with nothing in memory; thus, you must read everything you need to read in order to complete the request.

   (a) Given the inode number of a file, which structures must you read in order to calculate the location of the inode on disk?

   5, 4

   (b) Given the inode number of a file, which structures must you read in order to read the inode from disk?

   5, 4, 1

   (c) Given the inode number of a file, which structures must you read in order to read the first data block of the file from disk?

   5, 4, 1, 2

   (d) Given the inode of a directory containing a file, which data structures must you read in order to read the first data block of the file?

   5, 4, 1 (dir), 2 (dir), 1 (file), 2 (file)

   (e) Given a data block on disk, which data structures must you read in order to determine whether the block is live or not?

   3, 5, 4, 1

7. **NFS Flows:** This question is about flows in NFS, Sun's Network File System. Remember NFS, the block-based protocol with a focus on simple crash recovery? There are just two machines we are considering, a client and a server. Here are the possible components of a flow:

```
1. application buffer
2. client memory
3. client disk
4. network
5. server memory
6. server disk
```

(a) An application (on the client) opens a file F which contains a data block D. What is the flow of D through the system?

Empty.

(b) An application (on the client) reads block D from file F for the first time. What is the flow of D through the system?

6, 5, 4, 2, 1

(c) The application reads D from F again, right after reading it the first time. What is the flow of D?

2, 1

(d) The application waits a *long* time, and then reads D from F again. What is the flow of D now? Does this read take longer than the previous re-read? (Why or why not?)

2, 1. The client will recheck the validity of the cache (perhaps), but it will just read the blocks again.

(e) The application now writes a block $D2$ to an existing file G. What is the flow of block $D2$?

1, 2

(f) The application, after writing $D2$ to file G, closes the file. What is the flow of $D2$?

2, 4, 5, 6

8. **AFS Flows:** Now we are going to do the same question, but for AFS (the Andrew File System). Remember AFS, the system from Carnegie-Mellon that makes good use of local disks?

```
1. application buffer
2. client memory
3. client disk
4. network
5. server memory
6. server disk
```

(a) An application (on the client) opens a file F which contains a data block D. What is the flow of D through the system?

6, 5, 4, 2, 3

(b) An application (on the client) reads block D from file F for the first time. What is the flow of D through the system?

(3), 2, 1 (3 is only needed if data somehow is not in memory cache when accessed.

(c) The application reads D from F again, right after reading it the first time. What is the flow of D?

2, 1

(d) The application waits a *long* time, and then reads D from F again. What is the flow of D now? Does this read take longer than the previous re-read? (Why or why not?)

2, 1. No longer, thanks to callbacks

(e) The application now writes a block $D2$ to an existing file G. What is the flow of block $D2$?

1, 2, 3

(f) The application, after writing $D2$ to file G, closes the file. What is the flow of $D2$?

(3), 2, 4, 5, 6

9. **RAID Flows.** This question is about flows of I/Os in a RAID system. In RAIDs, some I/Os can happen in parallel, whereas some happen in sequence. To indicate two I/Os (to blocks 0 and 1, for example) in a flow can happen at the same time, we write: "(0 1)". To indicate two I/Os must happen in sequence (i.e., one before the other), we would use this notation: "0, 1". These flows can be built into larger chains; for example, consider the sequence "(0 1), (2 3)", which would indicate I/Os to blocks 0 and 1 could be issued in parallel, followed by I/Os to 2 and 3 in parallel.

   We can also indicate read and write operations in a flow with "r" and "w". Thus, "(r0 r1), (w2 w3)" is used to indicate we are reading blocks 0 and 1 in parallel, and, when that is finished, writing blocks 2 and 3 in parallel.

   Assume we have the following RAID-4, with a single parity disk:

```
   0     1     2     3     P0
   4     5     6     7     P1
   8     9    10    11     P2
  12    13    14    15     P3
    ... (and so forth) ...
```

   (a) Assume we must **read** blocks 0, 5, 10, and 15 as fast as we can. What is the flow of these blocks?

   (r0 r5 r10 r15)

   (b) Assume we must **read** blocks 100, 109, 126, and 85 as fast as we can. What is the flow of these blocks?

   (r100 r109 r126), (r85)

   (c) Assume we must **read** blocks 0, 4, 8, and 12. What is the flow of these blocks?

   r0, r4, r8, r12 (or all at once in one big read)

   (d) Now assume we will be writing. Assume we must **write** block 5. What is the I/O flow?

   (r5 rp1), (r5 wp1)

   (e) Assume we must **write** blocks 5 and 10. What is the I/O flow?

   (r5 rp1), (r10 rp2), (w5 wp1), (w10 wp2)

   (f) Now assume that we changed from a RAID-4 to a RAID-5 system with rotating parity. How can we arrange the data and parity blocks across the disks so as to make the writes to 5 and 10 go as fast as possible? **(draw a picture)**

```
   0     1     2     3     P0*
   5*    6     7    P1      4
  11    10*   P2*    8      9
  15    P3    12    13     14
    ... (and so forth) ...
```

   Allows the most parallelism across p0, 5, p2, and 10.

10. **Virtual Machine Monitor Flows:** We have an application accessing memory running on an OS with a software-managed TLB. Assume, for simplicity, a simple linear page table kept in physical memory. Here are some different hardware and software things that can happen during memory access:

```
0. the hardware checks the TLB to see if the VPN-to-PPN translation is present
1. the hardware issues a load to a physical address
2. the OS code at the start of the TLB miss handler runs
3. the OS code at the end of the TLB miss handler runs (return from trap)
4. the OS code that accesses the page table to lookup a translation runs
5. the OS code that updates the TLB with a new mapping runs
6. the OS code that updates the page table with a new VPN-to-PPN mapping runs
7. a disk request is initiated
8. a disk request completes
```

At first, assume we are **not** running on a VMM, i.e., this is just an application running on top of the OS.

(a) Write down the flow that occurs when a user application encounters a **TLB hit**:

0, 1

(b) Write down the flow that occurs when a user application encounters a **TLB miss** to a valid page that is **present** in memory:

0, 2, 4, 5, 3, and then retry 0, 1

(c) Write down the flow that occurs when a user application encounters a **TLB miss** to a valid page that is **not present** in memory:

0, 2, 4, 7, 8, 6, 5, 3 and then retry

Now assume that the OS is running on a virtual machine monitor. There are a few more possible events for these flows (for simplicity, just these three):

```
 9. the VMM code at the start of the VMM TLB miss handler runs
10. the VMM code at the end of the VMM TLB miss handler runs
11. the VMM code that updates the TLB with a new mapping runs
```

(d) Write down the flow that occurs when a user application encounters a **TLB hit**:

0, 1

(e) Write down the flow that occurs when a user application encounters a **TLB miss** to a valid page that is **present** in memory:

0, 9, 2, 4, 5, 11, 3, 10, retry

11. **Last Question:** What is the one most important feature that an operating system absolutely should have but in your experience it does not? Note: this is not a flow question. Rather, it just seeks your opinion. **Do this question last!**

    Lots of possibilities...