Student ID: ___________

CS-537: The Broken Final Exam (Fall 2004)

Please Read All Questions Carefully!

There are ten (10) 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
Part I: Graded Questions		$(12 \times 10) + (4 \times 5) \rightarrow 140$
Part II: Free Points		$(1 \times 10) \rightarrow 10$
Total		150

Student ID: ____________

Part I: Graded Questions

The Broken Operating System.

In this exam, we examine the three major topics that we covered during the semester: scheduling and concurrency, memory management, and file systems. However, in each case, it seems like something is a little wrong with the system that we are investigating: the OS is broken.

Hence, it is your job during this exam to figure out what is wrong with the given system and then to fix it. Of course, the specific questions describe in more detail exactly what you have to do, so **please read the questions carefully**. If you don't understand something, please feel free to ask.

1. **WORTH 10 points**. We have the following segment table:

Segment	**Base**	**Bounds**	**Protection**
0	0x1000	0x100	Read
1	0x2000	0x200	Read/Write
2	0x5000	0x500	Read/Write

The segment number is taken from the top two bits of each 32-bit virtual address, whereas the rest of each address is just the offset into the segment.

We then observe this series of accesses and resulting translations, each of which seems to be wrong in some way. Your job: in what way is each translation wrong? In other words, what should each access have done in comparison to what it did do? (note: each may be wrong in a different way)

(a) Load 0x00000010 → Segmentation violation
should have worked (in segment 0, translates to 0x00001010).

(b) Load 0x40000300 → Loaded word from physical address 0x00002300
should be out of bounds (segment 1 is only 0x200 in size, 0x300 is out of bounds)

(c) Load 0x80000300 → Loaded word from physical address 0x00002300
is in segment 2 and thus base address should be 0x5000

(d) Store 0x00000050 → Stored word to physical address 0x00001050
Protection violation: read only segment

(e) Load 0xC0000010 → Loaded word from physical address 0x00000010
Segmentation violation: tries to access segment 3 which is not defined

2. **WORTH 10 points**. The Broken Dynamic Relocation system (BDR) places jobs in memory incorrectly, sometimes (note: there is no paging or segmentation, just simple base and bounds). Assume to begin with that processes are placed in physical memory as follows, and that memory is only 100 bytes in size: Process A is loaded at address 0 and needs 10 bytes of memory, and Process B is loaded at address 30 and needs 20 bytes of memory. Which of the following are broken decisions?

(a) Place new process C (needs 30 bytes of memory) at address 0: **Works** or **Broken** √?

(b) Place new process C (needs 20 bytes of memory) at address 10: **Works** √ or **Broken**?

(c) Place new process C (needs 30 bytes of memory) at address 40: **Works** or **Broken** √?

(d) Place new process C (needs 30 bytes of memory) at address 80: **Works** or **Broken** √?

(e) Place new process C (needs 10 bytes of memory) at address 90: **Works** √ or **Broken**?

3. **WORTH 10 points**. In this question, we suspect there is some Broken Synchronization. The following multi-threaded code snippet is supposed to add ’amount’ to the global variable ’balance’ (as usual). For each of the following, indicate whether the code snippet has a **race condition**, **works correctly**, or is **broken** in some other way (but not a race condition).

(a)
```
void update (int amount) {
      int tmp = amount;
      lock();
      tmp = tmp + balance;             [Race], Works, Broken?
      unlock();                        // race to update balance
      balance = tmp;
}
```

(b)
```
void update (int amount) {
      int tmp = amount;
      lock();
      tmp = tmp + amount;              Race, Works, [Broken]?
      balance = tmp;                   // adds 2x amount
      unlock();
}
```

(c)
```
void update (int amount) {
      lock();
      balance = balance + amount;      Race, Works, [Broken]?
      lock();                          // double lock
}
```

(d)
```
void update (int amount) {
      int *tmp = &balance;
      lock();
      *tmp = *tmp + amount;            Race, [Works], Broken?
      unlock();                        // just works baby!
}
```

(e)
```
void update (int amount) {
      int tmp = balance;
      lock();
      tmp = tmp + amount;              Race, Works, [Broken]?
      unlock();                        // doesn’t update balance
}
```

Student ID: ____________

4. **WORTH 10 points**. The Broken Memory Allocator doesn't even know if it implements Best Fit, Worst Fit, or First Fit allocation. Assume you start with a free list that is comprised of the following three non-contiguous blocks: block A (10 bytes free), block B (30 bytes free), and block C (20 bytes free). In the following examples, determine whether best fit, worst fit, or first fit was used.

   (a) Allocate 15: taken from block B, allocate 11: taken from block B, allocate 4: taken from block A
   Best fit, **Worst fit**, [✓ **First fit**]?

   (b) Allocate 15: taken from block C, allocate 5: taken from block C
   [✓ **Best fit**], **Worst fit**, **First fit**?

   (c) Allocate 5: taken from block B, allocate 5: taken from block B
   Best fit, [✓ **Worst fit**], **First fit**?

   (d) Allocate 10: taken from block A, allocate 20: taken from block B, allocate 5: taken from block B
   Best fit, **Worst fit**, [✓ **First fit**]?

   (e) Allocate 8: taken from block A, allocate 2: taken from block A, allocate 5: taken from block B
   Best fit, **Worst fit**, [✓ **First fit**]?

5. **WORTH 10 points**. The Broken System deadlocks a lot, so someone has to try to write a deadlock detector. The detector goes through the system and tries to determine whether a deadlock exists at a given time. It can examine the state of each thread in the system, and see which resources the thread holds. For each of the following, determine whether a **deadlock** exists, if everything is **OK**, or if there is some **other problem** with the Broken system:

   (a) Thread A holds lock1, lock2, and is waiting for lock3; Thread B holds lock3, and is waiting for lock1:
   ... [✓ **Deadlock**], **OK**, **Other**?

   (b) Thread A holds lock1, lock2, and is waiting for lock3; Thread B holds lock3, and is waiting for lock4; Thread C holds lock1: **Deadlock**, **OK**, [✓ **Other**]?

   (c) Thread A holds lock1, lock2, and is waiting for lock3; Thread B holds lock3: **Deadlock**, [✓ **OK**], **Other**?

   (d) Thread A holds lock1, lock2, and is waiting for lock3; Thread B holds lock3, and is waiting for lock4; Thread C holds lock4, and is waiting for lock1: [✓ **Deadlock**], **OK**, **Other**?

   (e) Thread A holds lock1, lock2, and is waiting for lock3; no other threads are running:
   ... **Deadlock**, **OK**, [✓ **Other**]?

6. **WORTH 10 points**. The Broken Shortest Job First (BSJF) scheduler sometimes behaves like a completely different scheduler. In fact, we aren't sure if it is behaving like it should (SJF), like a round-robin scheduler (RR), or like something else altogether. Assume that in this question, 4 jobs arrive at the same time, 2 of which have an individual run-time of 10 seconds, and 2 of which have an individual run-time of 20 seconds. Given the following measured characteristics, label whether BSJF is behaving like SJF, RR, or something else.

   (a) The measured average **response time** is 150 milliseconds: **SJF**, [✓ **RR**], **Other**?

   (b) The measured average **response time** is about 28 seconds: **SJF**, **RR**, [✓ **Other**]?

   (c) The measured average **turnaround time** is about 33 seconds: [✓ **SJF**], **RR**, **Other**?

   (d) The measured average **turnaround time** is about 60 seconds: **SJF**, **RR**, [✓ **Other**]?

   (e) The measured average **cumulative wait time** is about 18 seconds: [✓ **SJF**], **RR**, **Other**?

7. **WORTH 5 points**. Nobody is sure what the TLB is used for in this Broken system. **Circle** the following entries if they represent proper use of a TLB:

   (a) To cache frequently accessed data
   (b) To cache frequently used address translations √
   (c) To speed up the time to load data from memory √
   (d) To enforce protection across pages √
   (e) To help determine whether a page is resident in memory or not √

8. **WORTH 10 points**. Assume the following producer/consumer code written within a monitor:

```
producer () {
    if (used == MAX_BUFFERS)
        condition.wait();
    // produce a buffer
    used++;
    condition.signal();
}

consumer () {
    if (used == 0)
        condition.wait();
    // consume a buffer
    used--;
    condition.signal();
}
```

Assume that the system uses **Mesa** style semantics.

Fix the **TWO MAJOR THINGS** wrong with this code so that it works.

if should be while.

need two condition variables not one.

9. **WORTH 10 points**. The page tables this OS are (surprise!) Broken in some way. Please fix them!

(a) 32-bit Virtual address, 4 KB page size, which needs to be translated to a 32-bit physical address, but the linear page table has 2^{16} entries. **How many entries should it have?**
2^{20}

(b) 32-bit Virtual address, 4 KB page size, which needs to be translated to a 32-bit physical address, but each page table entry has space for 2 bytes. **How big should each entry be?**
Need 20 bits (at least) for PPN.

(c) 32-bit Virtual address, 1 MB page size, which needs to be translated to a 16-bit physical address, but the linear page table has 2^{16} entries. **How many entries should it have?**
The page offset is too big here for the given physical address size.

(d) 32-bit Virtual address, 1 MB page size, which needs to be translated to a 22-bit physical address, but each page table entry has space for 2 bytes. **How big should each entry be?**
The entry needs just 2 bits for the PPN

(e) 32-bit Virtual address, 1 MB page size, which needs to be translated to a 20-bit physical address, but the linear page table has 2^{16} entries. **How many entries should it have?**
There is only one physical page in this system. But you could say the page table still needs 2^{12} *entries.*

10. **WORTH 5 points**. The Broken File System uses a **linked allocation** scheme, where each block of the file tells you where the next block is. **Circle** those of the following which are true when using a linked allocation scheme.

    (a) Sequentially overwriting all of the blocks of a file takes about the same time as overwriting the last block $\surd$

    (b) Each read of a random block of a file requires only about one I/O

    (c) Computing the size of a file takes about as much time as reading the entire file $\surd$

    (d) Reading an entire file sequentially takes about the same time as reading just the last block of the file $\surd$

    (e) The size of the inode is smaller than in a more typical Unix file system inode $\surd$

11. **WORTH 10 points**. The Broken Multi-Level Feedback Scheduler doesn't seem to be working as expected. In our BMLFQ, we have many prioritized queues. Jobs in a higher-priority queue always run before jobs in a lower-priority queue. In each of the following examples, circle whether the scheduler is susceptable to **gaming** by the user, whether **starvation** can occur, or whether the scheduler **works just fine**. **NOTE: that you can circle both gaming and starvation if appropriate.**

    (a) This scheduler starts new jobs at the topmost priority. When the job has used up its quantum, it is moved downward in priority. If the job instead does an I/O before the quantum is up, when it runs again, it is restarted at the same priority with a new quantum. Jobs within a queue are scheduled in round-robin fashion. **Gaming**$\surd$**, Starvation**$\surd$**, Works Fine**?

    (b) The scheduler starts new jobs at the topmost priority. When the job has used up its quantum (regardless of I/Os issued during the quantum), it moves downward in priority. Jobs within a queue are scheduled in round-robin fashion. **Gaming, Starvation**$\surd$**, Works Fine**?

    (c) The scheduler starts new jobs at the topmost priority. When the job has used up its quantum (regardless of I/Os issued during the quantum), it moves downward in priority. The scheduler periodically moves a job up one queue. Jobs within a queue are scheduled in round-robin fashion.
    ... **Gaming, Starvation, Works Fine**$\surd$?

    (d) The scheduler starts new jobs at the topmost priority. When the job has used up its quantum (regardless of I/Os issued during the quantum), it moves downward in priority. The scheduler periodically moves a job up to the topmost queue. Jobs within a queue are scheduled in round-robin fashion.
    ... **Gaming, Starvation, Works Fine**$\surd$?

    (e) The scheduler starts new jobs at the topmost priority. When the job has used up its quantum (regardless of I/Os issued during the quantum), it moves downward in priority. The scheduler periodically moves a job up one queue. Jobs within a queue are scheduled in random order. **Gaming, Starvation, Works Fine**$\surd$?

12. **WORTH 5 points**. A process has associate state with it, which must be saved and restored when a context switch takes place. In the Broken OS, though, too much state is saved and restored. Which of the following state **does not** need to be saved and restored during a context switch?

    (a) The general purpose (integer) registers

    (b) The floating point registers

    (c) The contents of the TLB $\surd$

    (d) The contents of the processes's address space $\surd$

    (e) The program counter

13. **WORTH 5 points**. Broken FFS makes some funny decisions in its design and implementation. Which of the following BFFS decisions differ from classic FFS? **CIRCLE** those decisions that are **DIFFERENT from FFS**.

    (a) BFFS places directories in the same cylinder group as their parent

    (b) BFFS uses pathnames to group related items on disk

    (c) BFFS tries to place files that were created at nearly the same time near one another on disk √

    (d) BFFS places the inode of a given file near its data blocks, except for large files

    (e) BFFS divides the disk into different groups; each group spans one disk platter's surface. √

14. **WORTH 10 points**. We have the following Broken RAID system (BRAID). BRAID sometimes doesn't fill in the redundant parts of the disk correctly. **Circle** which parity/redundancy bits BRAID got wrong below, or indicate that there is no problem:

    (a) BRAID Level 4:

Disk 0	Disk 1	Disk 2	Disk 3
0	0	0	0
0	1	1	[1]
0	1	0	1
1	1	1	1

    (b) BRAID Level 5:

Disk 0	Disk 1	Disk 2	Disk 3
0	0	0	0
0	1	0	1
0	[0]	0	1
[0]	1	1	1

    (c) BRAID Level 1 (Mirroring):

Disk 0	Disk 1	Disk 2	Disk 3
0	0	0	0
0	0	1	1
1	1	[0	1]
1	1	1	1

    (d) BRAID Level 0:

Disk 0	Disk 1	Disk 2	Disk 3
0	0	0	0
0	1	1	1
0	1	0	1
1	1	1	1

15. **WORTH 10 points**. We are now looking at a system that implements Broken LRU (BLRU). In the following access sequences, state either how BLRU messed up as compared to traditional LRU, or if it did the right thing. Assume that memory that can hold exactly 2 pages.

    (a) Access A (miss), access B (miss), access C (miss), access B (miss):
    LRU, or [✓ **Not LRU**]?

    (b) Access A (miss), access B (miss), access A (miss):
    [✓ **LRU**], or **Not LRU**?

    (c) Access A (miss), access B (miss), access C (miss), access B (hit), access C (hit):
    [✓ **LRU**], or **Not LRU**?

    (d) Access A (miss), access B (miss), access A (hit), access A (hit), access B (hit), access C (miss), access B (miss):
    LRU, or [✓ **Not LRU**]?

    (e) Access A (miss), access B (miss), access C (miss), access A (miss), access B (miss), access C (miss):
    [✓ **LRU**], or **Not LRU**?

16. **WORTH 10 points**. We have a journaling file system called BJFS – The Broken Journaling File System. BJFS tries to keep all of its meta-data consistent through the use of journaling. In the following examples, BJFS is trying to update the file system when an application has appended a single block to a file. It goes something like this:

    (a) 1 - write **begin transaction** to journal

    (b) 2 - write **journal descriptor** to journal (describing the update)

    (c) 3 - write **data bitmap, inode, data block** to journal

    (d) 4 - write **end transaction** to journal

    (e) 5 - write **data bitmap, inode, data block** to their final in-place locations.

    (f) 6 - free transaction in journal

    However, BJFS has to **wait** for various I/Os to complete in order to work correctly. For example, if BJFS issues I/Os 1 and 2, waits for them to complete, and then issues I/O 3, it knows that I/Os 1 and 2 have completed before 3.

    The following are some suggestions as to where to place such waits. Your job is to figure out where the suggested placements work. Write down whether BJFS is either **too slow** (because it spends too much time waiting for I/O to complete), **broken** (because it doesn't actually guarantee that meta-data is kept consistent), or **just fine**:

    (a) Wait between 3 and 4:
    Too slow, [✓ **Broken**], or **Just Fine**?

    (b) Wait between 3 and 4, Wait between 5 and 6:
    Too slow, [✓ **Broken**], or **Just Fine**?

    (c) Wait between 3 and 4, Wait between 4 and 5, Wait between 5 and 6:
    Too slow, **Broken**, or [✓ **Just Fine**]?

    (d) Wait between 1 and 2, Wait between 3 and 4, Wait between 4 and 5, Wait between 5 and 6:
    [✓ **Too slow**], **Broken**, or **Just Fine**?

    (e) Wait between 3 and 4, Wait between 4 and 5
    Too slow, [✓ **Broken**], or **Just Fine**?