CSE451
Sample Exam Questions
Part 2

* Why is copy-on-write a useful implementation strategy for UNIX `fork`?

* Can you give a reason why Windows uses a FIFO paging strategy on multi-processor machines?

* Suppose we have a set of physical page frames numbered 1..N. Give an access pattern for which LRU page replacement does poorly.

* Are TLBs more or less important as we add additional layers of page tables?

* What is thrashing?

* Assume we have a simple UNIX-like file system. Describe the worst-case sequence of disk operations that are required to read the following file. Assume the file fits in a single block. ("worst case" implies that nothing is cached)

```
/home/andrew/www/someFile.txt
```

* What is a strength of software-loaded TLBs, as opposed to hardware-loaded TLBs?

* Describe the sequence of events that occur during a page fault. Assume a hardware-filled TLB.

* Suppose we were to double the page size, while leaving all other system parameters the same size. What would be the effect on the following system variables? Explain your answer (Note: "it depends" is a possible answer; you should say what "it" depends on...).

i) TLB hit rate
ii) Worst-case page table size
iii) Internal fragmentation
iv) External fragmentation

* In general, it is a bad idea to do file I/O inside a Java `synchronized` block. Why is this?

* Generally speaking, which operation is worse: doing a read() system call inside a synchronized block, or doing a write() system call inside a synchronized block?

* Why are hard links generally faster than symbolic links?

* Suppose the file path for a symbolic link was embedded in the i-node itself. Would this be faster than traditional symbolic links? Would this version of symbolic links be as fast as hard links?

* File systems often reserve some free space on disk (space they refuse to allocate to files). Why is this a good idea?

* Explain why doubling the file system block size can more than double sequential read performance. (You should assume that the on-disk layout is perfectly sequential in both cases).

* Consider the web-server we built as part of project 2. Suppose we vary the number of web pages that are requested by web clients. Create a graph that illustrates the aggregate throughput for the web server as the number of active web pages grows large. The x-axis describes the number of active web pages, and the y-axis describes the web server throughput (aggregated across all web pages). You may assume that all web pages are the same size and that each web page is accessed with the same frequency (that is, with probability 1/N). Note that precise quantities are not important here; we are mostly interested in the shape of the curve.

* File systems use a block as the minimum unit of allocation and disk I/O. A typical block size is 4 KB. Given this, how is it possible for the `read()` system call to return a single byte?

```
char buffer[1]; // one byte buffer
read (fileDescriptor,buffer,1);
```

In a homework assignment, we discussed implementation options for a pool of memory buffers. One potential problem is that the set of memory buffers can become scattered over time. In the worst case, allocating N buffers can require touching N different memory pages, even if those buffers would fit on a single page. This leads to increased memory usage and (in the worst case) thrashing.

As we discussed, we can avoid scattering buffers by using multiple linked lists. Each buffer is assigned to a linked list using a hashing strategy. Assuming a good hash function, we can guarantee that a sequence of memory operations uses the fewest possible memory pages:

```
Buffer allocateBuffer() {
    // Return a buffer from the first non-empty linked list
}

returnBuffer (Buffer b) {
    // return the buffer to a linked list based on a hash function
}
```

This problem asks you to evaluate the following three hash functions (called A, B, and C). The functions are written in C, so that we can easily extract the virtual memory address of the Buffer. As in the homework, buffers are 512 bytes and pages are 4 KB.

```
// how many bits makeup the page offset inside virtual memory addresses
#define PAGE_OFFSET_BITS 12

// A number with 20 zeroes followed by 12 ones
#define PAGE_OFFSET_MASK 0x00000fff

// the size of a memory page in bytes
#define PAGE_SIZE (1 << PAGE_OFFSET_BITS)

// Hash Function A
// The Buffer pointer contains the virtual memory address of the
// buffer's data.
unsigned int hashCode (Buffer* buffer) {
  // first, convert the Buffer pointer into an unsigned integer.  All pointer
  // addresses are positive, so this avoids any weirdness associated with
  // signed bitwise operators.  This assumes that a memory address fits in
  // an integer.

  unsigned int rawAddress = (unsigned int) buffer;

  // Right shift the address, filling with zeroes on the left side
  return  (rawAddress >> PAGE_OFFSET_BITS);
}
```

```
// Hash Function B
unsigned int hashCode (Buffer* buffer) {

  unsigned int rawAddress = (unsigned int) buffer;

  // Use a logical AND operation
  return (rawAddress & PAGE_OFFSET_MASK);
}

// Hash Function C
unsigned int hashCode(Buffer* buffer) {
  unsigned int rawAddress = (unsigned int) buffer;

  // return a random number.  Use the buffer address as the "seed",
  // which means that a given buffer always maps to the same hash value.
  srandom(rawAddress);
  return (unsigned int) random();
}
```

* What is the name of the value returned by hash function A?

* What is the name of the value returned by hash function B?

* How would you rank these three hash functions, from best (uses the least memory) to worst (uses the most memory)? You should assume that the hash table contains a large (but bounded) number of linked lists.

* Most buffer cache implementations are built using a hashtable, which is keyed by virtual block (within a file) instead of by physical block (on disk). What is an advantage of this approach?

* Do you think that a log-structured file system would exhibit more or less performance variance (standard deviation) than a normal file system?

* How does a log-structured file system find the data associated with a particular file / i-node?

* Jim Gray speculates that log-structured file systems might make a comeback. Why is this so?

* on x86 systems, why must the address at the start of a memory-mapped file region end with three zero's?

* Explain why context-switching between threads of a single process is faster than context switching between different processes (this is true even if all process's working sets reside in main memory).

* The data structures for page tables and block tables (i-nodes) are similar but different. What are the differences, and why do they exist?

* In class, we looked at multiple ways to implement a "copy" operation for files. How could you implement a very fast copy for read-only files?

* Suppose an application uses lots of files, but doesn't care what the files are named. How might you optimize the file system for this case? In particular, what name should the OS assign to new files?

* Suppose a disk's transfer rate doubles while all other factors remain constant. Does this make cylinder groups more or less appealing? Why?

* Suppose a disk's rotation speed doubles while all other factors remain constant. Does this make cylinder groups more or less appealing? Why?

* Suppose a disk's seek time decreases by 50% while all other factors remain constant. Does this make cylinder groups more or less appealing? Why?

* Give a reason why asynchronous file writes typically yield better performance than synchronous file writes.

* [Problem #9 from the midterm]

The class shown below implements a Set data structure. A set is an unordered collection of Objects that does not contain duplicates. This implementation uses a hashing strategy (similar to a hash table). Each object is mapped to a bucket using its hashCode method. Each bucket is represented by a LinkedList, which handles the case when multiple objects map to the same bucket.

```
public class SimpleSet {
    private static final int NUM_BUCKETS = 16;

    // The set is implemented with an array of buckets;
    // each bucket contains a linked list of items
    // Note: LinkedList is not internally synchronized, so we
    // must manually synchronize to maintain thread-safety
    private final LinkedList [] buckets;

    public SimpleSet() {
        buckets = new LinkedList [NUM_BUCKETS];
        for (int i = 0;i<NUM_BUCKETS;i++) {
           buckets[i] = new LinkedList();
       }
    }

    // A helper method which gives the hash bucket for the given value
    private LinkedList hashBucket (Object value) {
       return buckets[value.hashCode() % NUM_BUCKETS];
    }

    // does the set contain the given value?
    public boolean contains (Object value) {
       LinkedList bucket = hashBucket(value);
       synchronized(bucket) {
          return bucket.contains(value);
       }
    }

   // add the given value to the set
   public void add (Object value) {
      LinkedList bucket = hashBucket(value);
      synchronized(bucket) {
        // make sure we don't add the value twice!
        if (! bucket.contains(value))
           bucket.add(value);
      }
   }

   // remove the value from the set
   public Object remove(Object value) {
      LinkedList bucket = hashBucket(value);
      synchronized(bucket) {
         return bucket.remove(value);
      }
   }
}
```

9a) [4 points] Notice that the SimpleSet does not use any synchronized methods. Instead, it synchronizes on the individual hash buckets. What is the primary advantage of this approach (as compared to synchronized methods)?

9b) [10 points] The code below shows a broken implementation of an xorAdd method (xor means “exclusive or”). The goal of the method is to ensure that exactly one of the two arguments is contained in the set. However, the given implementation contains safety and/or liveness problems. Re-write this method so that it is free from concurrency errors. Your implementation should be reasonably efficient.

```
// Ensure that exactly one of these arguments is in the set.
// If xor is already true, do nothing.
// Otherwise, give preference to the first argument
// TODO: This is not thread-safe: FIXME!
public void xorAdd(Object val1,Object val2) {
    boolean contains1 = this.contains(val1);
    boolean contains2 = this.contains(val2);

    // if the xor is already true, we do nothing
    if (contains1 ^ contains2) {
        return;
    }
    // otherwise, force XOR to be true
    else {
        this.add(val1);
        this.remove(val2);
    }
  }
}
```

9c) [6 points] Describe (but do not code) an implementation for a getSize method, which returns the total number of elements in the set. Your implementation should be reasonably efficient.

* [Problem 10 from the midterm]

In project #2, you implemented a semaphore using a condition variable. This problem asks you to do the opposite.

In class, we looked at a BufferPool, which provides a thread-safe repository for a set of data buffers. When the pool is empty, any attempt to acquire a buffer will block. This blocking is implemented using a condition variable. When a buffer is returned to the pool, a single blocked thread is awoken. The source code for the BufferPool is provided below.

Your task is to re-write the BufferPool to use semaphore(s) instead of a condition variable. Your new version should behave the same as the current version. You can assume the existence of a Semaphore class, with the same interface as project #2 (shown on the next page). The Semaphore is correctly written (i.e., it uses sufficient synchronization to ensure thread-safety). Your new implementation **cannot** directly use a condition variable. In other words, the words "wait" and "notify" should not appear in your solution.

(As you learned in project #2, the Semaphore *might* be implemented with a condition variable. The implementation details of the Semaphore do not matter for this problem).

```
// This class maintains a statically-allocated pool of memory buffers
// TODO: Convert this to use a Semaphore instead of a condition variable
public class BufferPool {

   public static final int BUFFER_POOL_SIZE = 512;

   // Note: the list is not internally thread-safe.
   // So, every read and write must be synchronized
   private final List<Buffer> freeBuffers = new LinkedList<Buffer>();

   public BufferPool() {
      for (int i=0;i<BUFFER_POOL_SIZE; i++) {
         Buffer buf = new Buffer();
         freeBuffers.add(buf);
      }
   }

   public synchronized Buffer getFreeBuffer() throws InterruptedExcepetion {
      while (freeBuffers.isEmpty()) {
         wait();
      }

      assert (!freeBuffers.isEmpty());
      return freeBuffers.remove(0);
   }

   public synchronized void returnFreeBuffer(Buffer buf) {
      freeBuffers.add(buf);
      notify();
   }
}
```

```
// Interface to a Semaphore object; You can assume this is already written!
public abstract class Semaphore {
  // initialize a semaphore with some value
  public Semaphore (int initialValue) { ; }

  // initialize a sempahore with value zero
  public Semaphore () {  }

  public void down ();

  public void up ();
}
```

10b) [4 points] In class, we discussed how the buffer pool can deadlock. One proposed solution was for each thread to request its entire allocation up-front. Can you explain how this would prevent deadlock?

10c) [6 points] Could you implement this deadlock-prevention scheme with the current Semaphore? If yes, describe how you would do it. If no, describe how the Semaphore would need to change. In either case, you do not need to implement your approach.

* What does it mean for a program to be “setuid root”? Why is this considered dangerous from a security perspective?

* What is problematic about the following interface when used in conjunction with Java RMI (remote method invocation)? How would you fix it?

```
interface FileStream extends Remote {
     // read from a (possibly) remote file into a buffer
     int read (byte[] buffer, int offset, int howManyBytes)
              throws RemoteException;
}
```

* In an RPC system, which of the following pieces of software implement the “Remote” interface: the client-side caller, the client-side stub, the server-side stub, and the server-side callee?

* A virtual machine monitor provides each virtual machine with the illusion of a dedicated physical machine. How does a VMM emulate multiple virtual disks on a machine with a single physical disk?