CS249B Winter 2010 Midterm Review

Administrivia

- Time: Tuesday, Feb 9th from 6:00pm-7:30pm or 7:30pm-9:00pm.
- Location: ECON 140
- No notes, books, or electronics allowed.
- We will provide blue exam books for you at the exam. All you need is a writing utensil.

General Hints

- The questions will mostly be of short answer / essay format. In the past, the exam has typically been structured with one question per chapter, with possible sub-parts contained in each. Please take a look at the sample old midterms from CS249B and CS349 in years past on the website for a representative set of questions you might be asked.
- Brain dumps will not necessarily get you more credit. It is important to be concise with your answers and hit on the key points that the professor is expecting. Moreover, simply trying to write down everything you know without being coherent and actually answering the question will likely cause you to lose points.
- Some questions will allow you to choose which side of an argument to support, which may be in opposition to the ideas taught by Professor Cheriton. Feel free to take such a side, but be sure to have good reasons to support your claim, and address the downsides you believe the ideas presented in lecture have.
- Professor Cheriton emphasizes that everything comes in threes. This is very important for his midterms, as whenever you're asked to provide supporting arguments, you usually want to provide three. We'll be expecting three.
- Read through the questions carefully and dig through the wording to find out exactly what the question is asking. The professor likes to create a context involving people such as the Dalai Lama, so just really take a minute and understand for sure what he's asking.
- Many questions will be very open ended, but you'll still need to argue your points with the material presented in lecture, whether by using them as support or by shooting them down.
- Feel free to use bullet point form to present your arguments if you'd like. This is not an essay writing class, but please do be coherent.
- Major emphasis will be placed on the material presented in lecture, but you are still responsible for the material in the readings that was not presented in class.

Key points from the lectures and readings (not all-encompassing of the material you need to know)

Chapter 11 - Software Engineering

- High Pressure Steam engines in comparison to software engineering

    - Overly complicated steam engines blew up
    - Basically, you want to keep it simple and testable
- Engineering challenges
    - Features
        - Many bugs come from feature interactions
        - Adding a new feature is essentially adding N-1 feature interactions, which involve a whole net set of tests and costs
    - Performance
    - Cost
        - There are many different metrics for cost, including monetary, resource, time, and maintenance
        - One important measure is the Total Cost of Ownership
- Engineering is all about trade-offs – even the best solution has disadvantages
- Software development parts
    - Process, people, and practice
    - We would like to have a simple but enforceable process, so that the most important resource – the people – can be leveraged efficiently.

Chapter 12 - Auditing

- Types of bugs
    - Malfunction – usually checked by unit tests
    - State corruption – usually manifests itself later on from the source of the bug
        - i.e. memory leaks resulting in an out of memory crash down the road
        - These are a major source of software failures
    - Timing/synchronization
        - Very difficult to test and track down
- Repair code is hard
    - You would be dealing with nasty cases
    - How do you test repair code?
    - It doesn't actually fix the underlying bug/issue
- Problems with asserts
    - What do you check with asserts? Developers tend to be lazy and hand-wavy with asserts
    - You would be shipping a different binary version to your customers, with the asserts removed.
- Audit should be orthogonal to the normal functional flow in a system
- A good audit framework should be able to obviate the need for asserts
- Audit is completely integrated into the rest of the program
- Writing audit code leads software engineers to use cleaner interfaces and simpler data structures that warrant easier auditing. This can likely reduce complexity and result in fewer bugs.

Chapter 13 - Named descriptions

- Large and complicated value types can lead to unpredictable memory requirements.

    - There may be excessive overhead with operations such as equality, when you compare large values to each other
- Named descriptions allow for a more predictable and analyzable set of memory requirements
    - They also can potentially save space, which allow for more cache hits, and thus better performance.
- A complex system can be separated into three dimensions
    - Separate objects
    - Components of these objects
    - Layers of these objects
- We want to achieve Pointer equality = value equality (PEEVE)
    - Gives us bounded space usage
    - Much faster nominal operations, such as equality and comparison
    - But, this involves significant management of these values such that there will never be two different, but semantically equal objects of a certain type in existence.
- Named descriptions are built upon a dictionary that provides lookup from a "common" name to references to the object
- A named description infrastructure allows for more extensibility.

Chapter 14 - Concurrency

- The challenge of concurrency should be solved without overly relying on different mechanisms including locks, while not introducing too much inter-object dependencies
- Basically, the complexities of concurrency warrant avoiding it altogether unless absolutely necessary
- If concurrency is necessary, then use distributed parallel execution with completely sequential execution inside any given process. If this is not possible, then resort to inter-module parallelism.
- Using asynchronous interaction reduces the need for true concurrency
- Concurrency can be either horizontal (session-based) or functional (pipelined). Systems in practice typically use a combination of both.
- Ideally, you want non-blocking synchronization with SWAAT (single writer at a time, but multiple readers allowed). The attribute-oriented design supports this structure very well.
- Traditionally, we have been concerned with procedural aspects of concurrency, such as how threads behave and how they synchronize with each other, using locks, etc. Instead, we want to think about concurrency as how best to structure state to be accessed concurrently, as state is the only real issue in parallelism. Use a system database with distributed parallel execution for this purpose.
- Our lives would be much simpler with better hardware support, but since we don't have it, we have to make do with these techniques. Concurrent systems are very difficult to get right, but these techniques will help you manage the complexities.

Chapter 15 – Collections and Iterators

- Collections should not be first-class objects, being passed around your program as directly usable entities
- Use generic collection class templates to enforce type safety.
- Use the most specific type when instantiating a collection. For example, if you want to create a collection of Puppies, in C++ you would instantiate something like vector<Puppy> instead of vector<Dog>. Similarly in Java, you would create a generic Vector<Puppy> instead of Vector<Object>.
- Use invasive collections whenever feasible. They better control memory allocation and requirements, and are also more efficient in processing. With invasive collections, you won't have a memory allocation problem with operations such as adding to a collection.
- Example of using invasive collections:

```
class Dog{
        string name;
        Dog* next;
};

Dog A;
Dog B;
A.next = &B;
B.next = NULL;
```

- The downside of invasive collections is that you will need implementation access to the particular object you want to place in a container. This is certainly not always true in practice.
- For fixed-size collections, the ideal data structure is a fixed-size array-based implementation.
- Use iterators to access a collection's objects. What tends to work best is snapshot semantics, that is, freezing a copy of the state of a collection for use by an iterator, so that the collection can continue to be modified, but the iterator will see a consistent view of the collection. You can also employ version numbers for this purpose.