Optimistic Design

CDP
Guarded Methods

• Do something based on the fact that one or more objects have particular states
  • Make a set of purchases assuming all items are available and not too expensive
  • Make a withdrawal based on the fact that the client has sufficient balance

• How do we do that?
Pessimistic Design

• A guarded method will act as follows:
  • Lock the relevant object(s) to avoid concurrent changes
  • Check that state is still OK
  • If the state is no longer as required, release the lock
    • What do we do next in this case?
  • Apply the necessary actions
  • Finally release the lock

• Useful when
  • Thread can tolerate indefinite postponement (unlike realtime)
  • You can guarantee that no thread will lock forever
Analysis

• Consider a scenario in which we lock before every access to an object. Assume:
  • The program has 1,000 objects
  • One of the other threads is accessing one of those objects 10% of the time
  • The changes made to the object invalidate our action 10% of the time
  • On average the synchronization is necessary only 0.001% of the time!
  • But we are paying for it 100% of the time while accessing each object!

• Maybe we can skip the synchronization?
  • What can go wrong?

• Murphy's law: Anything that can go wrong, will go wrong
Optimistic Design

• Attempt actions with/without partial synchronization
• Rollback (undo) actions that should not have been performed after we discover the problem
• Main issues:
  • How to discover the problem
  • How to rollback
  • What to do after the rollback (on failure)

```java
State state = currentState(); // Unguarded method
updateState(state); // Unguarded method
synchronized(this) { // Guarded block
    success = commit(state);
}
if (! success) retry(); // Unguarded method
else actionsNotModifyingState(state); // Unguarded method
```

Most of the computation (the heavy part) is outside of a guarded block
Advantages of Optimistic Design

• Reduces the time holding locks

• Reduces the number of lock operations
When to Apply

• Chance of failure is low enough
• Synchronization overhead is high enough
  • E.g. synchronization via network
• Can tell when some operations has been applied incorrectly / illegally
  • Always can go back to the last correct state
Example: Spell Checker

• In a word processor we have two threads
  • The user thread
  • The spell checker thread

• The user thread should never wait for the speller thread while it is spell checking a text

• The speller thread should take pieces of text and highlight spelling mistakes
Spell Checker Solution
User Thread

• The document is composed of a sequence of text chunks

• In order to edit text the user thread:
  • Locks the chunk containing the current editing position
  • Marks the chunk as “busy”
  • Then it unlocks the chunk

• When editing ends the user thread:
  • Locks the chunk again
  • Sets a timestamp that marks the completion of the editing
  • Marks the chunk as “not busy”
  • Then, it unlocks the chunk
Spell Checker Solution
Spell Checker Thread

• The spell checker looks for chunks that are not busy
• When it finds one it:
  • Locks the chunk
  • Verify that it is indeed “not busy” and if so, makes a replica of the text contained in the chunk and unlocks the chunk
  • Find the spelling mistakes and keeps a local records of them
  • Locks the chunk again
  • If the chunk is busy or has been updated after the first lock, the spell checker discards its results
  • Otherwise, it highlights the erroneous words and unlocks the chunk
Provisional Action

• Only “pretending” to act, delaying commitment of effects until the possibility of failure has been ruled out
• Instead of making changes to an object, act on its shadow copy
• Once in a while commit our changes by replacing relevant objects with our copies
• Most of the time need to make sure the originals had not changed at all!
• Using immutable fields makes it easy to recognize changes by comparing references rather than data
• Easier to manage for methods that only update instance variables
Provisional Action - Code

class Optimistic {
    // State class is used instead of a few fields.
    // State should be immutable for this to be efficient (and safer)
    private State currentState;

    // assumed is the state before the action
    synchronized boolean commit(State assumed, State next) {
        boolean success = (currentState == assumed);
        if (success) // no interference – safe to commit
            currentState = next; // the new state after action
        return success;
    }
}

Can a thread change the state during the commit of another?
Provisional Action - Usage

State assumed = currentState(); // Unguarded method
State next = updateState(assumed); // Unguarded method
if (! commit(assumed, next)) // Guarded method
  retry(); // Unguarded method
else
  // Unguarded method
  otherActionsDependingOnNewStateButNotChangingIt(next);
Rollback / Recovery

• Undoing the effects of each performed action
• Every action performed within an optimistic method should have an inverse action
• Keeping a log of all actions performed so far to allow rolling back (undoing) in reverse order
• Periodically, verify that the actions were applied correctly (e.g. without interference) and discard parts of the log
• Allows concurrent modifications as long as we can check that they were applied correctly
class Optimistic {
  private State currentState; // State class is now mutable
  // The state now contains extra information that helps us
  // determine if a contention occurred

  boolean synchronized commitExecution(UndoList log) {
    boolean success = verify(currentState);
    // check for indication that the operations applied correctly
    if (!success) // apply inverse of each operation in reverse order
      log.undo(currentState); // i.e. a rollback
    else log.clear(); // operations effectively committed
    return success;
  }
}
Rollback - Usage

```java
State state = currentState();  // Unguarded method
Log log = updateState(state);  // Unguarded method (or partially guarded)
if (!commitExecution(log))     // Guarded method
    retry();                  // Unguarded method
else
    // Unguarded method
    otherActionsDependingOnNewStateButNotChangingIt(next);
```
Spell Checker Thread with **Rollback**

• The spell checker looks for chunks that are not busy
• When it finds one it:
  • Locks the chunk
  • Verify that it is indeed “not busy” and if so, makes a replica of the **timestamp** in the chunk and unlocks the chunk
  • Find the spelling mistakes and highlights the erroneous words on the screen (does not change the text)
  • Locks the chunk again
  • If the chunk is busy or has been updated after the first lock, the spell checker erase the highlights on the screen
  • Otherwise, continue
• Assumes the user doesn’t mind a split-second mistakes in the spell checking while typing
Shadowing vs Rollback

• Acting on a copy is usually much easier to implement

• Shadowing does not allow concurrent modifications, even if independent
  • Can be solved by improving state granularity or providing state merge operation

• In rollback there is no guarantee of progress
  • Without necessary precautions all threads might keep rolling back each other forever
  • In shadowing at least one thread makes progress
Nesting and composing optimistic methods

- Optimistic methods can sometimes call other optimistic methods (nesting)
- An operation may contain a sequence of a few optimistic methods (composition)
- Handling in shadowing:
  - Pass the shadow state as a parameter
    - \textit{State func(State before)}
- Rollback:
  - Each method returns an undo log for its modification
  - Caller appends callee log to its own log
- Top-level method eventually commits
Detecting Problems

• In Shadowing we only need to verify that the state of the target object has not changed since we started applying our modifications
  • Provide an efficient *comparison* operator
    • Simple reference comparison on immutable objects
    • If there is too much to compare, use *versioning* or *timestamping*

• In Rollback we must have a set of well defined verifiable conditions on the object’s state
  • So that those conditions are true if and only if all actions performed on the objects so far were performed correctly and consistently
  • Complicated when both threads are modifying the state
Volatile Members

• Unlike normal fields, reading / writing goes directly to memory
• True current values vs. performance penalty
• In Java, it marks for the compiler, that the field may be accessed without synchronization
  • For volatile reference, it only copies the reference atomically, not the entire object
  • AtomicReference<Integer/Boolean/.../V> in java can do this and more
• C++’s equivalent to Java’s volatile is atomic<T>
  • More on that on C++ memory model (following weeks)
Failure Handling

• Report failure or throw exception
  • Allows the caller to decide

• Internally retry the action until it succeeds
  • Are we sure it will eventually succeed?

• Retry a number of times, or until a timeout

• Drop the action because it is no longer relevant
  • For example avoid retrying to update a user entry if the user has just been deleted
Retrying Smartly

• Retrying an optimistic method immediately is not efficient
  • It is unlikely that object state has changed to a desirable or consistent state
  • It is unlikely that thread contention has passed
• It might be beneficial to yield and allow the other threads to execute
Exponential Backoff

• When using rollback, we may enter *livelock*, by causing several threads to rollback/retry continuously

• Most operations, when executed without interference for sufficient time, will complete successfully
  • Known as obstruction-freedom

• Use exponential *backoff*
  • After the first failure sleep for a bounded but random number of time units
  • After each consecutive failure, enlarge random bound exponentially