SOEN 387: Web-Based Enterprise Application Design

Chapter 5. Concurrency

- Concurrency is one of the most tricky aspects of software development. Whenever you have multiple processes or threads manipulating the same data, you run into concurrency problems.
- Some concurrency issues: Lost **updates** and **inconsistent** read.
- Both of these issues cause a failure of correctness (or safety), and they result in incorrect behavior that would not have occurred without two users trying to work with the same data at the same time.
- The essential problem of any concurrent programming is that it's not enough to worry about **correctness**; you also have to worry about **liveness**: how much concurrent activity can go on.

Execution Contexts

From the perspective of interacting with the outside world, two important contexts are the **request** and the **session**.

A **request** corresponds to a single call from the outside world which the software works on and for which it optionally sends back a response. During a request the processing is largely in the server's court and the client is assumed to wait for a response. Some protocols allow the client to interrupt a request before it gets a response, but this is fairly rare. More often a client may issue another request that may interfere with one it just sent. So a client may ask to place an order and then issue a separate request to cancel that order. From the client's view the two requests may be obviously connected, but depending on your protocol that may not be so obvious to the server.

A **session** is a long-running interaction between a client and a server. It may consist of a single request, but more commonly it consists of a series of requests that the user regards as a consistent logical sequence. Commonly a session will begin with a user logging in and doing various bits of work that may involve issuing queries and one or more business transactions . At the end of the session the user logs out.

- Two important terms from operating systems are **processes** and **threads**.
  - A **process** is a, usually heavyweight, execution context that provides a lot of isolation for the internal data it works on.
  - A **thread** is a lighter-weight active agent that's set up so that multiple threads can operate in a single process. People like threads because they support multiple requests within a single process—which is good utilization of resources. However, threads usually share memory, and such sharing leads to concurrency problems. Some environments allow you to control what data a thread may access, allowing you to have **isolated threads** that don't share memory.
- Since processes are properly isolated from each other, this would help reduce concurrency conflicts. Currently we don't know of any server tools that allow you to work this way. A close alternative is to start a new process for each request, which was the common mode for early Perl Web systems. People tend to avoid that now because starting processes tie up a lot of resources, but it's quite common for systems to have a process handle only one request at a time—and that can save many concurrency headaches.

When you are dealing with databases there is another important context—a **transaction**. Transactions pull together several requests that the client wants treated as if they were a single request. They can occur from the application to the database (a system transaction) or from the user to an application (a business transaction).

Isolation and Immutability

For enterprise applications two solutions are particularly important: **isolation** and **immutability**.

- Concurrency problems occur when more than one active agent, such as a process or thread, has access to the same piece of data. One way to deal with this is isolation: Partition the data so that any piece of it can only be accessed by one active agent.

- You only get concurrency problems if the data you are sharing can be modified. So one way to avoid concurrency conflicts is to recognize **immutable** data.

- we cannot make all data **immutable**, as the whole point of many systems is data modification. But by identifying some data as **immutable**, or at least **immutable** almost all the time, we can relax our concurrency concerns for it and share it widely.

- Another option is to separate applications that are only reading data, and have them use copied data sources, from which we can then relax all concurrency controls.

Optimistic and Pessimistic Concurrency Control

What happens when we have mutable data that we cannot isolate? In broad terms there are two forms of concurrency control that we can use: **optimistic** and **pessimistic**.

- Let us suppose that Martin and David both want to edit the Customer file at the same time.
- With **optimistic locking** both of them can make a copy of the file and edit it freely. If David is the first to finish, he can check in his work without trouble. The concurrency control kicks in when Martin tries to commit his changes. At this point the source code control system detects a conflict between Martin's changes and David's changes. Martin's commit is rejected and it is up to him to figure out how to deal with the situation.
- With **pessimistic locking** whoever checks out the file first prevents anyone else from editing it. So if Martin is first to check out, David can't work with the file until Martin is finished with it and commits his changes.

A good way of thinking about this is that an optimistic lock is about conflict detection while a pessimistic lock is about conflict prevention.

Both approaches have their pros and cons. The problem with the **pessimistic** lock is that it reduces concurrency. While Martin is working on a file he locks it, so everybody else has to wait.

- Optimistic locks allow people to make much better progress, because the lock is only held during the commit. The problem with them is what happens when you get a conflict. Essentially everybody after David's commit has to check out the version of the file that David checked in, figure out how to merge their changes with David's changes, and then check in a newer version.

- The essence of the choice between **optimistic** and **pessimistic** locks is the frequency and severity of conflicts. If conflicts are sufficiently rare, or if the consequences are no big deal, you should usually pick optimistic locks because they give you better concurrency and are usually easier to implement. However, if the results of a conflict are painful for users, you'll need to use a pessimistic technique instead.

Preventing Inconsistent Reads

Inconsistent read problem : Consider this situation, Martin edits the Customer class, which makes calls on the Order class. Meanwhile David edits the Order class and changes the interface. David compiles and checks in; Martin then compiles and checks in. Now the shared code is broken because Martin didn't realize that the Order class was altered underneath him.

Pessimistic locks have a well-worn way of dealing with this problem through read and write locks.

To read data you need a read (or shared) lock; to write data you need a write (or exclusive) lock.

- Many people can have read locks on the data at one time, but if anyone has a read lock nobody can get a write lock. Conversely, once somebody has a write lock, then nobody else can have any lock. With this system you can avoid inconsistent reads with pessimistic locks.

Optimistic locks usually base their conflict detection on some kind of version marker on the data. This can be a timestamp or a sequential counter. To detect lost updates the system checks the version marker of your update with the version marker of the shared data. If they're the same, the system allows the update and updates the version marker.

Deadlocks

A particular problem with pessimistic techniques is **deadlock**. Say Martin starts editing the Customer file and David starts editing the Order file. David realizes that to complete his task he needs to edit the Customer file too, but Martin has a lock on it so he has to wait. Then Martin realizes he has to edit the Order file, which David has locked. They are now deadlocked—neither can make progress until the other completes

There are various techniques you can use to deal with deadlocks. One is to have software that can detect a deadlock when it occurs. In this case you pick a victim, who has to throw away his work and his locks so the others can make progress.

Deadlock detection is very difficult . A similar approach is to give every lock a time limit. Once you hit that limit you lose your locks and your work—essentially becoming a victim.

Transaction

The primary tool for handling concurrency in enterprise applications is the transaction. transaction is a bounded sequence of work, with both start and endpoints well defined. An ATM transaction begins when the card is inserted and ends when cash is delivered, or an inadequate balance is discovered.

Software transactions are often described in terms of the **ACID** properties:

Atomicity: Each step in the sequence of actions performed within the boundaries of a transaction must complete successfully or all work must roll back. Partial completion is not a transactional concept. Thus, if Martin is transferring some money from his savings to his checking account and the server crashes after he's withdrawn the money from his savings, the system behaves as if he never did the withdrawal. Committing says both things occurred; a roll back says neither occurred. It has to be both or neither.
Consistency: A system's resources must be in a consistent, noncorrupt state at both the start and the completion of a transaction.
Isolation: The result of an individual transaction must not be visible to any other open transactions until that transaction commits successfully.
Durability: Any result of a committed transaction must be made permanent. This translates to "Must survive a crash of any sort."

Application Server Concurrency

Another form of concurrency is the process concurrency of the application server itself: How does that server handle multiple requests concurrently and how does this affect the design of the application on the server?

Explicit multithreaded programming, with locks and synchronization blocks, is complicated to do well. It's easy to introduce defects that are very hard to find—concurrency bugs are almost impossible to reproduce.

Such software is incredibly frustrating to use and debug, so our policy is to avoid the need for explicit handling of synchronization and locks as much as possible. Application developers should almost never have to deal with these explicit concurrency mechanisms.

The simplest way to handle this is to use **process-per-session**, where each session runs in its own process. Its great advantage is that the state of each process is completely isolated from the other processes so application programmers don not have to worry at all about multithreading. As far as memory isolation goes, it's almost equally effective to have each request start a new process or to have one process tied to the session that's idle between requests.

The problem with **process-per-session** is that it uses up a lot resources. To be more efficient you can pool the processes, such that each one only handles a single request at one time but can handle multiple requests from different sessions in a sequence. This approach of pooled **process-per-request** will use many fewer processes to support a given number of sessions.

Even process-per-request will need many processes running to handle a reasonable load. You can further improve throughput by having a single process run multiple threads. With this **thread-per-request** approach, each request is handled by a single thread within a process. Since threads use much fewer server resources than a process, you can handle more requests with less hardware this way, so your server is more efficient. The problem with using **thread-per-request** is that there's no isolation between the threads and any thread can touch any piece of data that it can get access to

If you use **thread-per-request**, the most important thing is to create and enter an isolated zone where application developers can mostly ignore multithreaded issues. The usual way to do this is to have the thread create new objects as it starts handling the request and to ensure that these objects aren't put anywhere (such as in a static variable) where other threads can see them. That way the objects are isolated because other threads have no way of referencing them.

In creating objects developers need to avoid: static, class-based variables or global variables because any use of these has to be synchronized.