Homework-7

Mohan Srinivas Naga Satya Pothukuchi

NUID: 001821918

Exercise 16.1

The TaskBag is a service whose functionality is to provide a repository for 'task descriptions'. It enables clients running in several computers to carry out parts of a computation in parallel. A master process places description of subtasks of a computation in the TaskBag, and worker processes select tasks from the TaskBag and carry them out, returning descriptions of their results to the TaskBag. The master then collects the results and combines them to produce the final result. The TaskBag service provides the following operations:

- setTask allows clients to add task descriptions to the bag;
- takeTask allows clients to take task descriptions out of the bag.

A client makes the request takeTask, when a task is not available but may be available soon. Discuss the advantages and drawbacks of the following alternatives:

- The server can reply immediately, telling the client to try again later.
- The server operation (and therefore the client) must wait until a task becomes available.
- Callbacks are used.

CASE A:
When the client makes a request to "takeTask" and the server replies immediately with its status indicating unavailability, the response sets a precedent to an inefficient process of polling the server with client request and the server in carrying out extra requests. It is also potentially unfair because other clients may make their requests before the waiting client tries again.

The advantages of this process is that if any client whose workflow can be altered based on the server response of task unavailability, the client can efficiently execute its request without being blocked until a task is added to it task bag.

CASE B:

A multi-threaded server handling request from multiple clients can efficiently handle requests by communicating between multiple thread objects (representing clients) and not cause any deadlocks while serving the clients. This can be achieved using the *wait()* and *notify()* operations being implemented server's thread objects. When a client makes a request for the task even before a task exists in the task bag, the *wait()* operation can be called on the client's thread which would suspend the operation and put the thread in the wait state. Once another client makes a request to the add a task to the task bag, the thread can then notify to the wait threads that a task is now available in the bag.

The disadvantages of this operation is that waiting client can be put in waiting state until another client makes a request to the add a task and this may increase the response time to the client and can result inefficient processing of the thread or timeouts for the client response.

CASE C:

A callback from the server is when the server makes an RPC call to a method at client indicating the client of availability of a task in the bag. The advantage of this process is that the client need not wait until a task is returned and it can either continue its execution or work on a different process while waiting for a callback from the server. Similarly, a thread object need not wait in the server thread queue while waiting for an operation from another client.

The disadvantage of this process is when the client requests a callback on availability of resources, a server thread which executes the callback can be indefinitely waiting when the client doesn't respond back and keeps on using the server memory. When multiple client executes a similar workflow, a load is added upon the server's memory, and thus can lead to crashing of the server. Moreover, callbacks are a way to establish communication between two remote-processes and hence, would increase load on the kernel when compared to thread communication.

Exercise 16.2

SERIAL-EQUIVALENCE 1:

Transaction T	**Transaction U**
READ(J)	
WRITE (J, 44)	
	READ(J)
	READ(K)
	WRITE (K, 66)
READ(I)	
WRITE (I, 33)	
	WRITE(I, 55)

SERIAL-EQUIVALENCE 2:

Transaction T	**Transaction U**
	READ(J)
READ(J)	
WRITE (J, 44)	
	READ(K)
	WRITE (K, 66)
READ(I)	
WRITE (I, 33)	
	WRITE(I, 55)

SERIAL-EQUIVALENCE 3:

Transaction T	**Transaction U**
	READ(J)
READ(J)	
WRITE (J, 44)	
	READ(K)
	WRITE (K, 66)
	WRITE (I, 55)
READ(I)	
WRITE (I, 33)	

Explain the 3 different concurrency control methods briefly. Compare and contrast how they achieve concurrency control.

The three different concurrency control methods are:

- Locks
- Optimistic Concurrency Control
- Timestamp ordering

Locks:

In achieving concurrency control using Locks, the strict two-phase locking mechanism is employed to make sure there are no dirty reads or writes. In this locking scheme, the server attempts to lock any object that is about to be used by any operation of a client's transaction. If a client requests access to an object that is already locked due to another client's transaction, the request is suspended, and the client must wait until the object is unlocked.

Serial equivalence requires that all of a transaction's accesses to a particular object be serialized with respect to accesses by other transactions. All pairs of conflicting operations of two transactions should be executed in the same order. To ensure this, a transaction is not allowed any new locks after it has released a lock. Under a strict execution regime, a transaction that needs to read or write an object must be delayed until other transactions that wrote the same object have committed or aborted. To enforce this rule, any locks applied during the progress of a transaction are held until the transaction commits or aborts. The presence of the locks prevents other transactions reading or writing the objects. When a transaction commits, to ensure recoverability, the locks must be held until all the objects it updated have been written to permanent storage.

Optimistic Concurrency Control:

The optimistic concurrency control has evolved because of the disadvantages of locking mechanism. Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data. Also, to avoid cascading aborts, locks cannot be released until the end of the transaction. This may reduce significantly the potential for concurrency.

In optimistic concurrency control, transactions are allowed to proceed as though there were no possibility of conflict with other transactions until the client completes its task and issues a "closeTransaction" request. When a conflict arises, some transaction is generally aborted and will need to be restarted by the client.

A transaction in optimistic concurrency control has three phases:

- Working Phase

    - During the working phase, each transaction has a tentative version of each of the objects that it updates. This is a copy of the most recently committed version of the object.
- Validation Phase
    - When the closeTransaction request is received, the transaction is validated to establish whether or not its operations on objects conflict with operations of other transactions on the same objects.
- Update Phase
    - If a transaction is validated, all of the changes recorded in its tentative versions are made permanent. Read-only transactions can commit immediately after passing validation.

Timestamp Ordering

In concurrency control schemes based on timestamp ordering, each operation in a transaction is validated when it is carried out. If the operation cannot be validated, the transaction is aborted immediately and can then be restarted by the client. Each transaction is assigned a unique timestamp value when it starts. The timestamp defines its position in the time sequence of transactions. Requests from transactions can be totally ordered according to their timestamps.

In timestamp ordering, there is only one version of each object and restricts access to one transaction at a time. If each transaction has its own tentative version of each object it accesses, then multiple concurrent transactions can access the same object. The timestamp ordering rule is refined to ensure that each transaction accesses a consistent set of versions of the objects. It must also ensure that the tentative versions of each object are committed in the order determined by the timestamps of the transactions that made them. This is achieved by transactions waiting, when necessary, for earlier transactions to complete their writes.

Identify 5 different programming devices (e. g. Tasks, Exec Framework) used in Java for Task Execution. Then, briefly explain how you might use these to implement a basic Locks based concurrency control in a simple Web App.

Tasks

Tasks are independent activities: work that doesn't depend on the state, result, or side effects of other tasks. Independence facilitates concurrency, as independent tasks can be executed in parallel if there are adequate processing resources.

Task executing in a Threaded web server can achieve concurrency using threads locking on shared object between the tasks. The threaded server alternates between accepting incoming connections and processing outgoing connections. For a new incoming connection, the tasks are spawn in a new thread. In doing so, concurrency can be achieved as follows:

- Task processing is offloaded from the main thread, enabling the main loop to resume waiting for the next incoming connection more quickly. This enables new connections to be accepted before previous requests complete, improving responsiveness.
- Tasks can be processed in parallel, enabling multiple requests to be serviced simultaneously. This may improve throughput if there are multiple processors, or if tasks need to block for any reason such as I/O completion, lock acquisition, or resource availability.
- Task handling code must be thread safe, because it may be invoked concurrently for multiple tasks.

Executor Framework

Executor may be a simple interface, but it forms the basis for a flexible and powerful framework for asynchronous task execution that supports a wide variety of task execution policies. It provides a standard means of decoupling task submission from task execution, describing tasks with Runnable. The Executor implementations also provide lifecycle support and hooks for adding statistics gathering, application management, and monitoring.

Executor replaces the threaded implementation of web server using the unique threaded implementation. This allows the executor to achieve concurrency based on locks using the executor policies. Execution policies are a resource management tool, and the optimal policy depends on the available computing resources and your quality of service requirements. By limiting the number of concurrent tasks, you can ensure that the application does not fail due to resource exhaustion or suffer performance problems due to contention for scarce resources. Separating the specification of execution policy from task submission makes it practical to select an execution policy at deployment time that is matched to the available hardware.

Thread Pools

A thread pool, as its name suggests, manages a homogeneous pool of worker threads. A thread pool is tightly bound to a work queue holding tasks waiting to be executed. Worker threads have a simple life: request the next task from the work queue, execute it, and go back to waiting for another task. Executing tasks in pool threads has a number of advantages over the thread per task approach. Reusing an existing thread instead of creating a new one amortizes thread creation and teardown costs over multiple requests.

To achieve concurrency using thread pools, the class library provides a flexible thread pool implementation along with some useful predefined configurations. The fixedThreaded pool allows the tasks to be invoked in their own separate thread based on the number of the tasks and thus allows the locking mechanism to achieve individual locking on resources.

Delayed and Periodic Tasks

Delayed tasks are the ones which can defined as “execute the task after 10ms”, whereas the periodic tasks are the ones which can be defined as “execute the task for every 10ms”. In this implementation, thread pool can be used to schedule or delay the execution of tasks. The thread

pool class library provides an implementation for ScheduleThreadPoolExecutor which can be used to spawn, and control threads during its lifecycle.

The reason the timer used to execute the delayed and periodic tasks needs to follow an implementation from the thread pool is:

- Timer executes the tasks in a single threaded system, which decreases the efficiency with which we can execute the tasks.
- Timer also throws an unchecked exception during its run time and thus decreases the efficiency with the application can be executed.

Callable and Future

The Executor framework uses Runnable as its basic task representation. Runnable is a fairly limiting abstraction; run cannot return a value or throw checked exceptions, although it can have side effects such as writing to a log file or placing a result in a shared data structure.

Many tasks are effectively deferred computations executing a database query, fetching a resource over the network, or computing a complicated function. For these types of tasks, Callable is a better abstraction: it expects that the main entry point, call will return a value and anticipates that it might throw an exception.

Tasks are usually finite: they have a clear starting point and they eventually terminate. The lifecycle of a task executed by an Executor has four phases: created, submitted, started, and completed. Since tasks can take a long time to run, we also want to be able to cancel a task. In the Executor framework, tasks that have been submitted but not yet started can always be cancelled, and tasks that have started can sometimes be cancelled if they are responsive to interruption. Cancelling a task that has already completed has no effect. Future represents the lifecycle of a task and provides methods to test whether the task has completed or been cancelled, retrieve its result, and cancel the task.