DISTRIBUTED SYSTEMS

Distributed Systems – Assignment 3

Submitted: November 17, 2014
By: Devan Shah 100428864
Submitted to: Weina ma & Ying Zu

1. A significant concern in distributed systems is conflicts that might arise from unsynchronized requests to a single resource. A very simple example of this concern can be demonstrated by a class that increments a counter through a method. If this class is instantiated by another class that supports multi-threading, and increments the counter N times within its run() method one can demonstrate that when multiple threads of this class are created there are situations when the value of the total count is not what is expected. For example, let's say that the multi-threaded counter object increments count 10 times and 3 threads of this object type are launched. The expected value of count is 30 (10*3) but this will not be the case if the 3 threads are not synchronized.

   For this question create a Counter class with 2 methods *increaseCount()* and *getCount()* that correspondingly increase the value of a counter by 1 and gets the value of the counter. Now define a *CountingThread* class that instantiates a Counter object and implements the Runnable interface (i.e. supports multi-threading). In the *run()* method of *CountingThread* call *increaseCount()* several times. In the main method of *CountingThread* instantiate 3 threads of *CountingThread*. When these threads end get the value of the counter using *getCount()* and print out the result. Note: you might have to put the Counter object to sleep for a few milliseconds in the *increaseCount()* method to get significant synchronization issues. Code this and show that you have issues when you do not use method-level synchronization. Add the method-level synchronization to the code so that the expected counter sum is achieved. [10]

   Answer:

   The implementation of the Counter class is to keep an incrementing count of an integer. On Object creation of Counter the counter (incremental counter) is initialized at 0, which can be incremented by 1 every by calling the function increaseCount() available in the Counter class (or object). The Counter class also contains a getCount() that is used to get the current count value. The Counter class object is shared between all threads in this example, there for any updates to the common variables need to be synchronized to make sure that threads are updating the value correctly. Figure 1 and Figure 3 show code with no use of method-level synchronization and with method-level synchronization.

```
public void increaseCount()
{
    // increment the counter by 1
    counter++;

    /**
     * Thread sleep is used at this point to simulate slow
     * data processing and modifications of the counter
     * variable.
     */
    try
    {
        // Sleep for 55 milliseconds, this was based on spe
        Thread.sleep(55);
    }
    // Catch the exception and provide the necessary inform
    catch ( InterruptedException e ) { System.out.println (
}
```

Figure 1 Shows code snippet from of the increaseCount() function from the Counter class without the use of method-level synchronization.

Figure 1 on the left shows the increaseCount() function from the Counter class **without the use of method-level synchronization**, the reason for the thread sleep is to make sure that slow data processing was simulated accurately. With the use of 3 threads and each thread incrementing the shared Counter object the result at the end should be 30 (10*3). But without the use of method-level synchronization this is now the case. Figure 2 shows the out of the code without the use of method-level synchronization.

```
<terminated> CountingThreadInitializers (1) [Java Applic
Total Time Program Running: 550 ms.

Counter Value after Execution ----->  24
```

Figure 2 Shows the output from the run of the code without the use of method-level synchronization. The result will always be different, sometimes it may be 30.

Figure 3 on the right shows the increaseCount() function from the Counter class **with the use of method-level synchronization**, using the same thread sleep as before, I made use of the method-level synchronization to allow threads to synchronize their calls in order to correctly execute the application. There are 2 ways that the synchronization can be made to the increaseCount() function code, the synchronized keyword can be added to the method declaration or put the code that performs update to the global variable in a synchronized code block.

```
public synchronized void increaseCount()
{
    synchronized (this)
    {
        // increment the counter by 1
        counter++;

        /**
         * Thread sleep is used at this point to simulate slow
         * data processing and modifications of the counter
         * variable.
         */
        try
        {
            // Sleep for 55 milliseconds, this was based on spe
            Thread.sleep(55);
        }
        // Catch the exception and provide the necessary inform
        catch ( InterruptedException e ) { System.out.println (
    }
}
```

Figure 3 Shows code snippet from of the increaseCount() function from the Counter class with the use of method-level synchronization.

```
<terminated> CountingThreadInitializers [Java Applicat
Total Time Program Running: 1707 ms.

Counter Value after Execution ----->  30
```

Figure 4 Shows the output from the run of the code with the use of method-level synchronization. The result will always be 30. (The correct value)

Figure 4 shows the output of the code with the use of method-level synchronization. The result will always be 30, because of the synchronized blocks added to the increaseCount() functions.

The implemented CountingThread class is to be executed within a thread, it extends the Runnable class in order to utilize its methods and override the run() function. The CountingThread class will run for each thread that is invoked and increase the counter value in the Counter object reference by 10.

The implemented CountingThreadInit class contains the main function that is used to create and start the 3 counter threads. This class is also responsible of waiting for all the threads to finish and then print the results of the counter value from the Counter class object.

The source code for the programs with and without method-level synchronization are included in the submission, can be found at the following location under submission Assignment 3 - Devan Shah 100428864.zip file.

Code without method-level synchronization can be found under:
Distributed Systems - Assignment 3 - Question 1 No Synchronization/src

Code with method-level synchronization can be found under:
Distributed Systems - Assignment 3 - Question 1 With Synchronization\src

2. A file server uses caching and achieves a hit rate of 80%. File operations in the server cost 5 ms of CPU time when the server finds the request blocked in the cache, and take an additional 15 ms of I/O time otherwise. Explaining any assumptions you make, estimate the server's throughput capacity (average requests/sec) if it is:
   I. Single-threaded
   II. Two-threaded, running on a single processor;
   III. Two-threaded, running on a two-processor computer. [8]

Answer:

General Assumptions:

80% of file operation accesses cost 5 milliseconds of CPU time

The remaining 20% of file operation accesses cost 20 milliseconds of CPU time.

Takes 5 milliseconds of CPU time to find the requested block in cache

Takes additional 15 milliseconds of disk I/O time

This gives the total of 20 milliseconds for blocks not in the cache

$$Average\ request\ time\ = 5\ milliseconds * 80\% + 20\ milliseconds * 20\%$$
$$Average\ request\ time\ = 4\ milliseconds + 4\ milliseconds$$
$$\boldsymbol{Average\ request\ time\ = 8\ milliseconds}$$

I. Single-threaded

$$Throughput\ Capacity = \frac{1\ requests}{8\ milliseconds} * \frac{1000\ milliseconds}{1\ seconds}$$
$$Throughput\ Capacity = \frac{1\ requests}{8\ \cancel{milliseconds}} * \frac{1000\ \cancel{milliseconds}}{1\ seconds}$$
$$Throughput\ Capacity = \frac{1000\ requests}{8\ seconds}$$
$$\boldsymbol{Throughput\ Capacity = 125\ \frac{requests}{seconds}}$$

II. Two-threaded, running on a single processor

4 cached requests take 10 milliseconds total using two threads on a single processor.

1 un-cached request takes 15 millisecond using two threads on a single processor, this single request takes longer because overlap I/O with computation is required.

$$Throughput\ Capacity = \frac{1\ requests}{5\ milliseconds} * \frac{1000\ milliseconds}{1\ seconds}$$
$$Throughput\ Capacity = \frac{1\ requests}{5\ \cancel{milliseconds}} * \frac{1000\ \cancel{milliseconds}}{1\ seconds}$$
$$Throughput\ Capacity = \frac{1000\ requests}{5\ seconds}$$
$$\boldsymbol{Throughput\ Capacity = 200\ \frac{requests}{seconds}}$$

III. Two-threaded, running on a two-processor computer

2 cached requests take 5 milliseconds total using two threads on 2 processors

2 requests * 1000 / 5 seconds = 2000/5 = 400 requests / seconds

The disk can still only serve 20% of requests at only 66.67 requests / seconds. (The disk requests are serialised).

$$Throughput\ Capacity = 5 * \frac{1\ requests}{15\ milliseconds} * \frac{1000\ milliseconds}{1\ seconds}$$

$$Throughput\ Capacity = 5 * \frac{1\ requests}{15\ milliseconds} * \frac{1000\ milliseconds}{1\ seconds}$$

$$Throughput\ Capacity = \frac{5\ requests}{15\ \cancel{milliseconds}} * \frac{1000\ \cancel{milliseconds}}{1\ seconds}$$

$$Throughput\ Capacity = \frac{5000\ requests}{15\ seconds}$$

$$\boldsymbol{Throughput\ Capacity = 333.33\ \frac{requests}{seconds}}$$

3. A clock is reading 10:27:54:0 (hr:min:sec) when it is discovered to be 4 seconds fast. Explain why it is undesirable to set it back to the right time at that point and show (numerically) how it should be adjusted so as to be correct after 8 seconds has elapsed. [5]

Answer:

The reason that it is undesirable to set the clock back to the right time at the point where it is detected as incorrect is because applications set timestamps based on the computer clock. The impact on applications would be drastic as the timestamps would start boing back in time and would make them incorrect and not in order, which would violate the happened before clause of clocks. The way that the time difference can be corrected is by slowly adjusting the computer's software clock to match the computer's hardware clock. For example, in this situation the current software clock is reading 10:27:54:0 (hr:min:sec) but the computer hardware clock (real time) is 10:27:50:0 (hr:min:sec), therefore the incorrect software clock needs to adjusted to be correct after 8 seconds have elapsed.

The current clock standings are as follows, current ISC (Incorrect Software Clock) and HC (Hardware Clock). We now need to construct a 3rd clock that can be used to correct the incorrect software clock. Let's call this 3rd software clock TSC (Temporary Software Clock). Once this TSC clock is adjusted to hold the correct time it will replace the current incorrect clock. This gives us the following equation

$$0.\ TSC = c\ (ISC - T_{skew}) + T_{skew}$$

Where T_{skew} is 10:27:54:0 (hr:min:sec) and c is the factor that needs to be determined.

Couple of facts that we know are the following:

$$1.\ TSC = T_{skew} + 4\ secs$$
$$2.\ ISC = T_{skew} + 8\ secs$$

Equation 1 is based on the fact that the Temporary Software Clock (TSC) is the correct time, when equation 2 holds for the Incorrect Software Clock (ISC).

Using substitution and elimination we can substitute equations 1 and 2 into equation 0 to solve for c.

$$T_{skew} + 4\ secs = c\ (T_{skew} + 8\ secs - T_{skew}) + T_{skew}$$

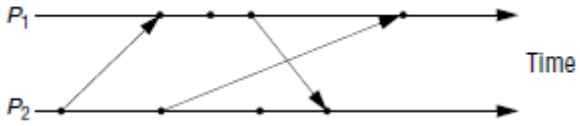
$$\cancel{T_{skew}} + 4\ secs - \cancel{T_{skew}} = c\ (\cancel{T_{skew}} + 8\ secs - \cancel{T_{skew}})$$

$$4\ secs = 8\ secs\ c$$
$$\boldsymbol{c = 0.5}$$

This gives us the following equation for correcting the time over 8 seconds:

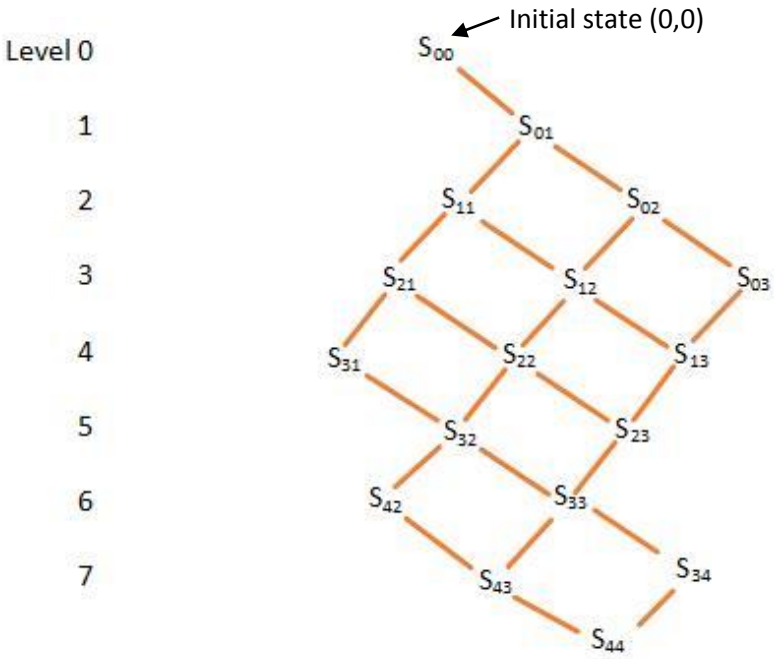
$$.\ \boldsymbol{TSC = 0.5\ (ISC - T_{skew}) + T_{skew}\ \ when\ T_{skew} \leq ISC \leq (T_{skew} + 8\ secs)}$$

4.



The figure above shows events occurring for each of two processes, p1 and p2. Arrows between processes denote message transmission. Draw and label the lattice of consistent states (p1 state, p2 state), beginning with the initial state (0,0). [15]

Answer:



S_{ij} is the global state after i events at P_1 and j events at P_2

Figure 5 Shows the lattice diagram of the consistent states (p_1 state, p_2 state)

5. In a certain system, each process typically uses a critical section many times before another process requires it. Explain why Ricart and Agrawala's multicast-based mutual exclusion algorithm is inefficient for this case, and describes how to improve its performance, Does your adaptation satisfy liveness condition ME2? [7]

Answer:

The Ricart-Agrawala multicast-based algorithm is for mutual exclusion on a distributed system. The Ricart-Agrawala algorithm in terms of each process typically using a critical section many times before another process is extremely inefficient. In this case the Ricart-Agrawala algorithm is inefficient because for a process to access a critical section the single process needs permission from all the other process to access the critical section again. The process requests access by sending a multicast request messages to the n number of process to allow access to the critical section, this is an extreme waste of time and bandwidth.

The current performance of the Ricart-Agrawala algorithm is slow as to enter the critical section one single process needs to send the request message to all the process and needs to get a reply message from the process before entering the critical section. One possible way to improve the performance of the Ricart-Agrawala algorithm is to setup a multi-step process for entering the critical section. Which allows a process to enter the critical section at any time with no need to transmit a message to all process and once the process had finished with the critical section a flag can be set identifying the critical section is ready to enter again. This can be accomplished by setting a token “TEMP_FREE” identifying that the process has just finished with the critical section. The information about the process finishing with the critical section is not sent to all the process, this will save some time and bandwidth. When another process wants to access the critical section and the token “TEMP_FREE” is found, the process can change it directly to “HOLD” and enter the critical section. This way allows for process to enter the critical section without requesting over and over to enter the critical section, the token allows to enter the critical section when it is free. (Identified by token “TEMP_FREE”).

My current adaptation does not satisfy the liveness condition ME2 of being free from deadlock and starvation. Currently there is a change to hit a deadlock when process try to enter the critical section at the same time, there is also a possibility to hit starvation when process are entering critical sections. One way to resolve this would be to replace the token being set to “TEMP_FREE” to “FREE” instead when a request for entry is received.