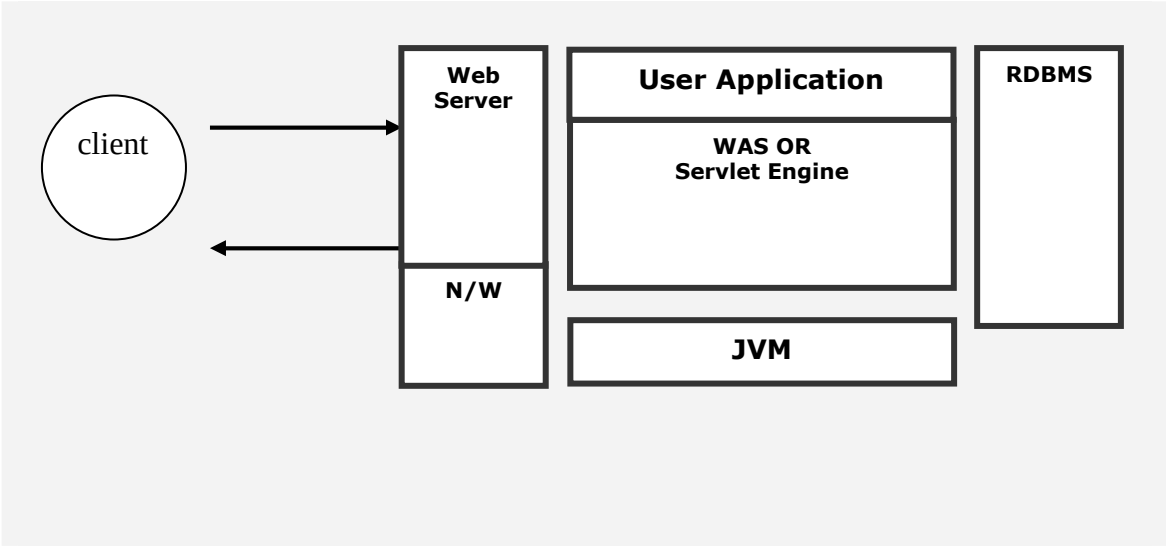
How to find bottleneck in J2EE application

작성자 : 자바스터디 네트워크 조대협(bcho@bea.com)

J2ee application 을 운영하다보면, 시스템이 극도로 느려지거나, 멈춰버리는 현상이 생기고는 한데, 분명히 개발하면서 테스트할때는 문제가 없었는데, 왜 이런일이 생기고, 어떻게 대처해야하는지에 대해서 알아보도록 하자.

일반적으로 J2ee application 을 서비스 하기 위해서는 아래와 같은 구조를 가지게 된다.



< 그림 1. 일반적은 J2ee application 의 구조 >

J2ee application 의 동작에 필요한 구성 요소를 나눠보면 위와 같이 Network, User Application (이하 User AP), WAS 또는 Servlet Engine(이하 통칭해서 WAS),JVM 과 RDBMS 이렇게 크게 다섯가지 조각으로 나뉘어 진다. 물론 JCA 를 이용해서 Legacy 와 연결할 수 도 있고, RMI/CORBA 를 이용하여 다른 Architecture 를 구현할 수 는 있으나, 이 강좌는 어디까지나 일반론을 설명하고자 하는것임으로 범위에서는 제외하겠다.

1. Hang up 과 slow down 현상의 정의...

먼저 용어를 정의하도록 하자.. 시스템이 느려지거나 멈추는 현상에 대해서 아래와 같이 용어를 정의하자

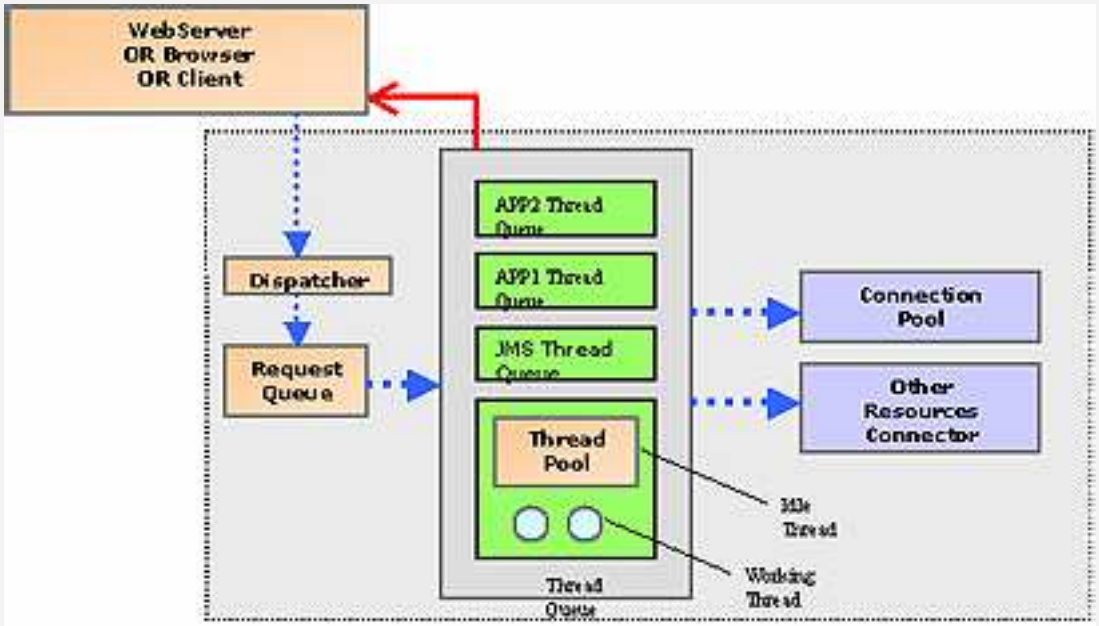
- Hang up : Server Instance 는 실행되고 있느나, 아무런 응답이 없는 상황 (멈춤 상태)
- Slowdown : Server Instance 의 response time 이 아주 급격히 떨어지는 상태 (느려짐)

이 Hangup 과 slowdown 현상은, 대부분이 그림 1 에서 설명한 다섯가지 요소중 하나 이상의 병목으로 인해서 발생한다. 즉, 이 병목 구간을 발견하고, 그 구간을 제거하면 정상적으로 시스템을 운영할 수 있게 되는것이다.

2. Slow down analysis in WAS & User AP

1. WAS 의 기본 구조

이 병목 구간을 발견하는 방법에 앞서서, 먼저 WAS 시스템의 기본적인 내부 구조를 이해할 필요가 있다.



* Thread Pooling

< 그림 2. WAS 시스템의 구조 >

<그림 2>는 일반적인 WAS 의 구조이다.

WAS 는 Client 로 부터 request 를 받아서, 그 Request 의 내용을 분석한다 JMS 인지, HTTP , RMI request 인지를 분석한후 (Dispatcher) 그 내용을 Queue 에 저장한다.

Queue 에 저장된 내용은 WAS 에서 Request 를 처리할 수 있는 Working Thread 들이 있을때 각각의 Thread 들이 Queue 에서 Request 를 하나씩 꺼내서 각각의 Thread 들이 그 기능을 수행한다.

여기서 우리가 주의깊게 봐야하는것은 Thread pooling 이라는 것인데. Thread 를 미리 만들어놓고, Pool 에 저장한체로 필요할때 그 Thread 를 꺼내서 사용하는 방식이다. (Connection Pooling 과 같은 원리이다.)

WAS 에서는 일반적으로 업무의 성격마다 이 Thread Pool 을 나누어서 만들어놓고 사용을 한다. 즉 예를 들어서 금융 시스템의 예금 시스템과 보험 시스템이 하나의 WAS 에서 동작할때, 각각의 업무를 Thread Pool 을 나누어서 분리하는 방식이다. 이 방식을 사용하면 업무의 부하에 따라서 각 Thread Pool 의 Thread 수를 조정할 수 있으며, 만약에 Thread 가 모자르거나 deadlock 등의 장애사항이 생기더라도 그것은 하나의 Thread Pool 에만 국한되는 이야기이기 때문에, 다른 업무에 영향을 거의 주지 않는다.

2. Thread Dump 를 통한 WAS 의 병목 구간 분석

위에서 살펴봤듯이, WAS 는 기본적으로 Thread 기반으로 작동하는 Application 이다. 우리가 hang up 이나 slow down 은 대부분의 경우가 WAS 에서 현상이 보이는 경우가 많다. (WAS 에 원인이 있다는 소리가 아니라.. 다른 요인에 의해서라도 WAS 의 처리 속도가 느려질 수 있다는 이야기다.)

먼저 이 WAS 가 내부적으로 어떤 Application 을 진행하고 있는지를 알아내면 병목 구간에 한층 쉽게 접근할 수 가 있다.

1) What is thread dump?

이를 위해서 JVM 에서는 Java Thread Dump 라는 것을 제공한다.
Thread Dump 란 Java Application 이 작동하는 순간의 X-Ray 사진, snapshot 을 의미한다.
즉 이 Thread dump 를 연속해서 수번을 추출한다면, 그 당시에 Application 이 어떻게 동작하고 진행되고 있는가를 살펴볼 수 있다.

Thread dump 를 추출하기 위해서는

- Unix 에서는 kill –3 pid
- Windows 계열에서는 Ctrl + break

를 누르면 stdout 으로 thread dump 가 추출 된다.

Thread dump 는 Application 을 구성하고 있는 현재 구동중인 모든 Thread 들의 각각의 상태를 출력해준다.

<그림 3-2. Thread dump >

그림 3-2 는 전체 Thread dump 중에서 하나의 Thread 를 나타내는 그림이다. Thread Dump 에서 각각의 Thread 는 Thread 의 이름과, Thread 의 ID, 그리고 Thread 의 상태와, 현재 이 Thread 가 실행하고 있는 Prorgam 의 스택을 보여준다.

- Thread name

각 쓰레드의 이름을 나타낸다.

※ WAS 나 Servlet Engine 에 따라서는 이 이름에 Thread Queue 이름등을 배정하는 경우가 있다.

- Thread ID

쓰레드의 System ID 를 나타낸다. 이 강좌 나중에 이 ID 를 이용해서 각 Thread 별 CPU 사용률을 추적할 수 있다.

- Thread Status

매우 중요한 값중의 하나로 각 Thread 의 현재 상태를 나타낸다.
일반적으로 Thread 가 사용되고 있지 않을때는 wait 를 , 그리고
사용중일때는 runnable 을 나타내는게 일반적이다.
그외에, IO Wait 나 synchronized 등에 걸려 있을때 Wait for monitor entry, MW (OS 별 JVM 별로 틀림) 등의 상태로 나타난다.

- Program stack of thread
현재 해당 Thread 가 어느 Class 의 어느 Method 를 수행하고 있는지를 나타낸다. 이 정보를 통해서 현재 WAS 가 어떤 작업을 하고 있는지를 유추할 수 있다.

2) How to analysis thread dump

앞에서 Thread dump 가 무엇이고, 어떤 정보를 가지고 있는지에 대해서 알아봤다. 그러면 이 Thread dump 를 어떻게 분석을 하고, System 의 Bottle neck 을 찾아내는데 이용할지를 살펴보기로 하자.

System 이 Hangup 이나 slowdown 에 걸렸을때, 먼저 Thread dump 를 추출해야 한다. 이때 한개가 아니라 3~5 초 간격으로 5 개 정도의 dump 를 추출한다. 추출한 여러개의 dump 를 연결하면 각 Thread 가 시간별로 어떻게 변하고 있는지를 판별할 수 있다.

먼저 각각의 Thread 덤프를 비교해서 보면, 각각의 Thread 는 적어도
1~2 개의 덤프내에서 연속된 모습을 보여서는 안되는게 정상이다.
일반적으로 Application 은 내부적으로 매우 고속으로 처리되기 때문에,
하나의 Method 에 1 초이상 머물러 있는다는것은 거의 있을 수 없는 일이다.
Thread Dump 의 분석은 각각의 Thread 가 시간이 지남에 따라서 진행되지 않고 멈춰 있는 Thread 를 찾는데서 부터 시작된다.

예를 들어서 설명해보자 . 아래 <그림 3-3>의 프로그램을 보면
MY_THREAD_RUN()이라는 메소드에서 부터
MethodA()→MethodB()→MethodC() 를 차례로 호출하는 형태이다.

```
MYTHREAD_RUN(){ ←·
  call methodA();
}

methodA(){
 //doSomething(); ←·
 call methodB();
}
```

```
methodB(){
 //call methodC(); ←◆
}

methodC(){
  // doSomething ←◇
  for(;;){// infinite loop } ←◇
}
```

<그림 3-3. sample code >

이 프로그램을 수행하는 중에 처음부터 Thread Dump 를 추출 하면 대강 다음과 같은 형태일것임을 예상할 수 있다. 함수 호출에 따라서 Stack 이 출력되다가 ◇의 단계 즉 MethodC 에서 무한루프에 빠지게 되면 더이상 프로그램이 진행이 없기 때문에 똑같은 덤프를 유지하게 된다.

```
"MYTHREAD" runnable ← ·
at …MYTHREAD_RUN()
:

"MYTHREAD" runnable ←·
at …methodA()
at …MYTHREAD_RUN()
:

"MYTHREAD" runnable ←◆
at …methodB()
at …methodA()
at …MYTHREAD_RUN()
:

"MYTHREAD" runnable ←◇
at …methodC()
at …methodB()
at …methodA()
at …MYTHREAD_RUN()
:

"MYTHREAD" runnable ← 이 덤프 모양이 반복됨
at …methodC()
at …methodB()
at …methodA()
at …MYTHREAD_RUN()
:
```

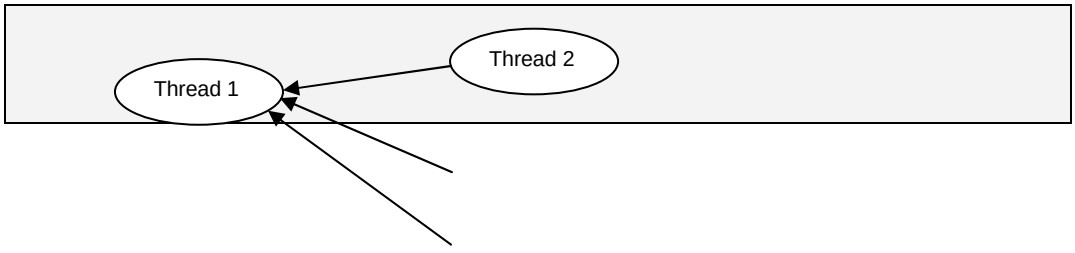
우리는 이 덤프만을 보고 methodC 에서 무엇인가 잘못되었음을 유추할 수 있고, methodC 의 소스코드를 분석함으로써 문제를 해결할 수 있다.

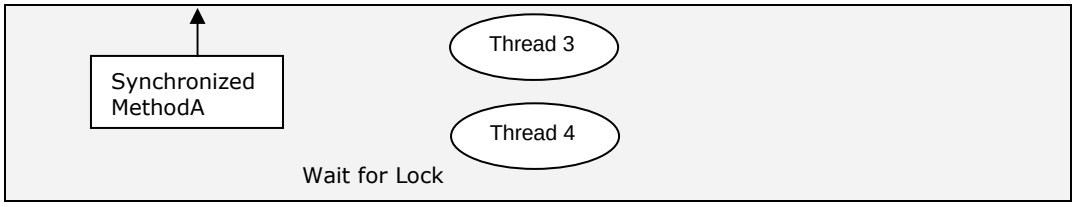
그렇다면 이제부터 Slow down 과 Hang up 현상을 유발하는 일반적인 유형과 그 Thread Dump 의 모양에 대해서 알아보도록 하자.

- **CASE 1. lock contention**

  잘 알다싶이 Java 는 Multi Threading 을 지원한다. 이 경우 공유영역을 보호하기 위해서 Synchronized method 를 이용하는데, 우리가 흔히들 말하는 Locking 이 이것이다.

  예를 들어 Thread1 이 Synchronized 된 Method A 의 Lock 을 잡고 있는 경우, 다른 쓰레드들은 그 Method 를 수행하기 위해서, 앞에서 Lock 을 잡은 쓰레드가 그 Lock 을 반환하기를 기다려야한다. 그림 3-4

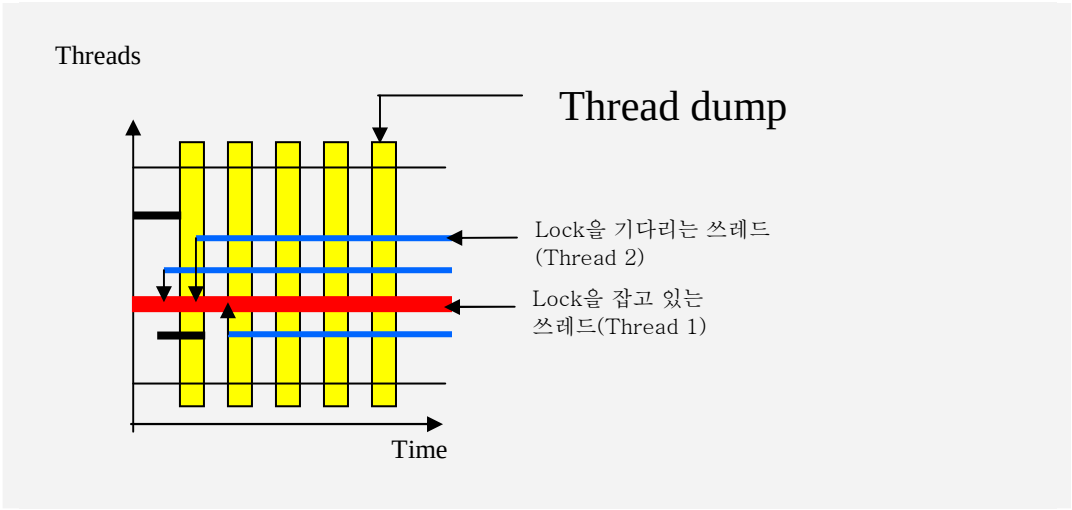




<그림 3-4. Thread 간에 Lock 을 기다리는 형태 >

만약에 이 Thread 1 의 MethodA 의 수행시간이 아주 길다면 Thread 2,3,4 는 마냥 이 수행을 아주 오랜 시간 기다려야하고, 다음 2 번이 Lock 을 잡는다고 해도 3,4 번 Thread 들은 1 번과 2 번 쓰레드가 끝난 시간 만큼의 시간을 누적해서 기다려야 하기때문에, 수행시간이 매우 느려지는 현상을 겪게 된다.

이처럼 여러개의 Thread 가 하나의 Lock 을 동시에 획득하려고 하는 상황을 Lock Contention 이라고 한다.



<그림 3-5. Lock Contention 상황에서 Thread 의 시간에 따른 진행 상태 >

이런 상황에서 Thread Dump 를 추출해보면 <그림 3-6> 과 같은 형태를 띠게 된다.

```
"ExecuteThread: '12' for queue: 'weblogic.kernel.Default'" daemon prio=10 tid=0x0055ae20
nid=23 lwp_id=3722788 waiting for monitor entry [0x2fb6e000..0x2fb6d530]
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:258)
        - waiting to lock <0x38819a18> (a sun.misc.Launcher$AppClassLoader)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:292)
        - locked <0x38db9ea8> (a weblogic.utils.classloaders.GenericClassLoader)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:292)
        - locked <0x38f520f8> (a weblogic.utils.classloaders.GenericClassLoader)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:292)
        - locked <0x3941af18> (a weblogic.utils.classloaders.GenericClassLoader)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:292)
        - locked <0x3941b6b0> (a weblogic.utils.classloaders.ChangeAwareClassLoader)
```

```
        at java.lang.ClassLoader.loadClass(ClassLoader.java:255)
                                        :
        at org.apache.xerces.jaxp.SAXParserFactoryImpl.newSAXParser(Unknown Source)
        at org.apache.axis.utils.XMLUtils.getSAXParser(XMLUtils.java:252)
        - locked <0x329fcf50> (a java.lang.Class)


"ExecuteThread: '13' for queue: 'weblogic.kernel.Default'" daemon prio=10 tid=0x0055bde0
nid=24 lwp_id=3722789 waiting for monitor entry [0x2faec000..0x2faec530]
        at org.apache.axis.utils.XMLUtils.getSAXParser(XMLUtils.java:247)
        - waiting to lock <0x329fcf50> (a java.lang.Class)
        at
org.apache.axis.encoding.DeserializationContextImpl.parse(DeserializationContextImpl.java:239
"ExecuteThread: '14' for queue: 'weblogic.kernel.Default'" daemon prio=10 tid=0x0061d680
nid=25 lwp_id=3722790 waiting for monitor entry [0x2fa6b000..0x2fa6b530]
        at org.apache.axis.utils.XMLUtils.releaseSAXParser(XMLUtils.java:283)
        - waiting to lock <0x329fcf50> (a java.lang.Class)
        at
org.apache.axis.encoding.DeserializationContextImpl.parse(DeserializationContextImpl.java:254)
"ExecuteThread: '15' for queue: 'weblogic.kernel.Default'" daemon prio=10 tid=0x0061dc20
nid=26 lwp_id=3722791 waiting for monitor entry [0x2f9ea000..0x2f9ea530]
        at org.apache.axis.utils.XMLUtils.releaseSAXParser(XMLUtils.java:283)
        - waiting to lock <0x329fcf50> (a java.lang.Class)
        at
```

<그림 3-6. Lock Contention 에서 Lock 을 기다리고 있는 상황의 Thread Dump>

그림 3-6 의 덤프를 보면 12 번 Thread 가 org.apache.axis.utils.XMLUtils.getSAXParser 에서 Lock 을 잡고 있는것을 볼 수 있고, 36,15,14 번 쓰레드들이 이 Lock 을 기다리고 있는것을 볼 수 있다.

해결방안

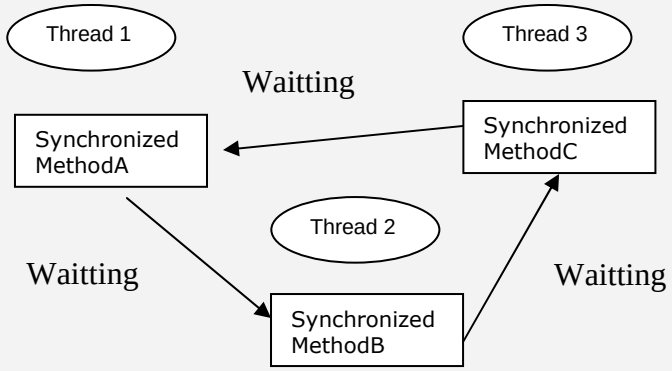
이런 Lock Contention 상황은 Multi Threading 환경에서는 어쩔 수 없이 발생하는 상황이기는 하지만, 이것이 문제가 되는 경우는 Synchronized Method 의 실행 시간이 불필요하게 길거나, 불필요한 Synchronized 문을 사용했을 때 발생한다.

그래서 이 문제를 해결하기 위해 불필요한 Sychronized Method 의 사용을 자제하고, Synchronized block 안에서의 최적화된 Algorithm 을 사용하는것이 필요하다.

- CASE 2. dead lock

이 Locking 으로 인해서 발생할 수 있는 또 다른 문제는 dead Lock 현상이다. 두개 이상의 쓰레드가 서로 Lock 을 잡고 기다리는 "환형대기조건" 이 성립되었을때, 서로 Lock 이 풀리지 않고 무한정 대기하는 현상을 이야기 한다.

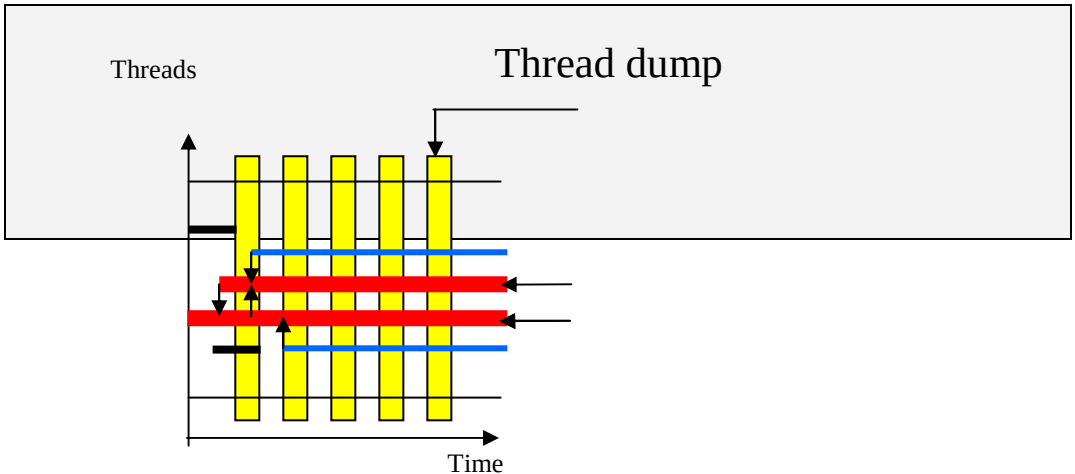
그림 3-7 을 보면 Thread1 은 MethodA 를 수행하고 있는데, sychronized methodB 를 호출하기전에 Thread2 가 methodB 가 끝나기를 기다리고 있다. 마찬가지로 Therad2 는 Thread3 가 수행하고 있는 methodC 가 끝나기를 기다리고 있고, methodC 에서는 Thread1 에서 수행하고 있는 methodA 가 끝나기를 기다리고 있다. 즉 각각의 메소드들이 서로 끝나기를 기다리고 있는 "환형 대기조건" 이기 때문에 이 Application 의 3 개의 쓰레드들은 프로그램이 진행이 되지 않게 된다.

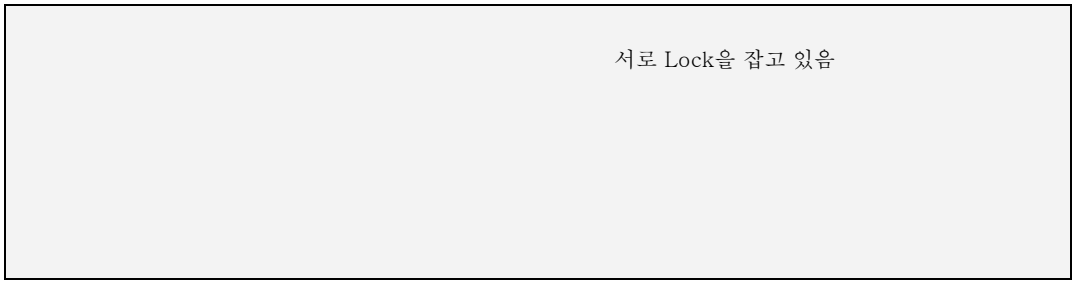



```
sychronized methodA(){
 call methodB();
}

sychronized methodB(){
 call methodC();
}

sychronized methodC(){
 call methodA();
}
```

< 그림 3-7. 환형 대기조건에 의한 deadlock >

이러한 "환형대기조건"에 의한 deadlock 은 Thread Dump 를 추출해보면 <그림 3-8> 과 같은 패턴을 띠우고 있으며 시간이 지나도 풀리지 않는다.





<그림 3-8. Deadlock 이 걸렸을때 시간 진행에 따른 Thread 의 상태>

```
"ExecuteThread-6" (TID:0x30098180, sys_thread_t:0x39658e50, state:MW, native
ID:0xf10) prio=5
at oracle.jdbc.driver.OracleStatement.close(OracleStatement.java(Compiled
Code))
at
weblogic.jdbc.common.internal.ConnectionEnv.cleanup(ConnectionEnv.java(Compiled
Code))
at
weblogic.jdbc.common.internal.ConnectionPool.release(ConnectionPool.java(Compiled
Code))
at weblogic.jdbcbase.pool.Connection.close(Connection.java(Compiled Code))
at sis.ao.svao0400R.svao0400RQ.mainProcess(svao0400RQ.java(Compiled Code))
at sis.ao.svao0400R.svao0400RQ.doGet(svao0400RQ.java(Compiled Code))
at javax.servlet.http.HttpServlet.service(HttpServlet.java(Compiled Code))
at javax.servlet.http.HttpServlet.service(HttpServlet.java(Compiled Code))
at
weblogic.servlet.internal.ServletStubImpl.invokeServlet(ServletStubImpl.java(Compiled
Code))
                                :
"ExecuteThread-8" (TID:0x30098090, sys_thread_t:0x396eb890, state:MW, native
ID:0x1112) prio=5
at oracle.jdbc.driver.OracleConnection.commit(OracleConnection.java(Compiled
Code))
at weblogic.jdbcbase.pool.Connection.commit(Connection.java(Compiled Code))
at sis.as.svas0900E.svas0902ES.mainProcess(svas0902ES.java(Compiled Code))
at sis.as.svas0900E.svas0902ES.doGet(svas0902ES.java(Compiled Code))
at javax.servlet.http.HttpServlet.service(HttpServlet.java(Compiled Code))
at javax.servlet.http.HttpServlet.service(HttpServlet.java(Compiled Code))
at
weblogic.servlet.internal.ServletStubImpl.invokeServlet(ServletStubImpl.java(Compiled
Code))
                                : (중략)
sys_mon_t:0x39d75b38 infl_mon_t: 0x39d6e288:
   oracle.jdbc.driver.OracleConnection@310BC380/310BC388: owner
"ExecuteThread-8" (0x396eb890) 1 entry ← 1)
        Waiting to enter:
           "ExecuteThread-10" (0x3977e2d0)
           "ExecuteThread-6" (0x39658e50)
sys_mon_t:0x39d75bb8 infl_mon_t: 0x39d6e2a8: \
   oracle.jdbc.driver.OracleStatement@33AA1BD0/33AA1BD8: owner
"ExecuteThread-6" (0x39658e50) 1 entry ← 2)
        Waiting to enter:
           "ExecuteThread-8" (0x396eb890
```

※ IBM AIX 4.3.3 JVM 1.3.0

IBM AIX 의 JVM 의 경우에는 Lock 의 정보가 각 Thread 의 Stack 에 나타나는 것이 아니라. 별도로 Thread 별 Locking 정보를 따로 나타내주기 때문에.. Lock 정보를 별도로 확인해야한다.

<그림 3-9. Deadlock 이 걸린 IBM AIX Thread Dump >

DeadLock 의 검출은 Locking Condition 을 비교함으로써 검출할 수 있으며, 최신 JVM (Sun 1.4 이상등)에서는 Thread Dump 추출시 만약 Deadlock 이 있다면 해당 Deadlock 을 찾아주는 기능을 가지고 있다.

그림 3-9 를 보자. IBM AIX 의 Thread 덤프이다. 1)항목을 보면 현재 8 번 쓰레드가 잡고 있는 Lock 은 10 번과 6 번 쓰레드가 기다리고 있음을 알 수 있으며. Lock 은 OracleConnection 에 의해서 잡혀있음을 확인할 수 있다. (아래)

```
sys_mon_t:0x39d75b38 infl_mon_t: 0x39d6e288:
   oracle.jdbc.driver.OracleConnection@310BC380/310BC388: owner
"ExecuteThread-8" (0x396eb890) 1 entry ← 1)
        Waiting to enter:
           "ExecuteThread-10" (0x3977e2d0)
           "ExecuteThread-6" (0x39658e50)
```

그림 3-9 의 2)번 항목을 보면 이 6 번쓰레드는 OracleStatement 에서 Lock 을 잡고 있는데, 이 Lock 을 8 번 쓰레드가 기다리고 있다. (아래)

```
sys_mon_t:0x39d75bb8 infl_mon_t: 0x39d6e2a8: \
   oracle.jdbc.driver.OracleStatement@33AA1BD0/33AA1BD8: owner
"ExecuteThread-6" (0x39658e50) 1 entry ← 2)
        Waiting to enter:
           "ExecuteThread-8" (0x396eb890
```

결과적으로 6 번과 8 번은 서로 Lock 을 기다리고 있는 상황이 되어 Lock 이 풀리지 않는 Dead Lock 상황이 된다.

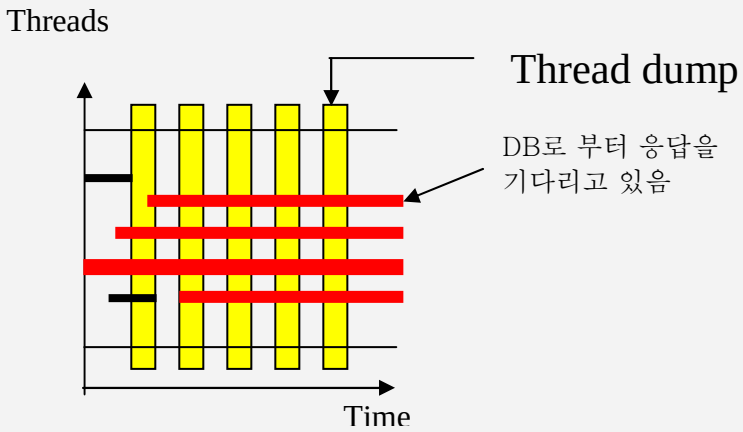
해결 방안

Dead Lock 의 해결 방안은 서로 Lock 을 보고 있는것을 다른 Locking Object 를 사용하거나 Lock 의 방향 (Synchronized call 의 방향)을 바꿔줌으로써 해결할 수 있다.

User Application 에서 발생한 경우에는 sychronized method 의 호출 순서를 "환형대기조건"이 생기지 않도록 바꾸도록 하고.
위와 같이 Vendor 들에서 제공되는 코드에서 문제가 생긴경우에는 Vendor 에 패치를 요청하도록 하여 해결한다.

- CASE 3. wait for IO response

다음으로 많이 볼 수 있는 패턴은 각각의 Thread 들이 IO Response 를 기다리는데... IO 작업에서 response 가 느리게 와서 시스템 처리속도가 느려지는 경우가 있다.



<그림 3-10. Thread 들이 IO Wait 를 할때 시간에 따른 Thread Dump 상황 >

```
"ExecuteThread: '42' for queue: 'default'" daemon prio=5 tid=0x3504b0 nid=0x34 runnable [0x9607e000..0x9607fc68]
        at java.net.SocketInputStream.socketRead(Native Method)
        at java.net.SocketInputStream.read(SocketInputStream.java:85)
        at oracle.net.ns.Packet.receive(Unknown Source)
        at oracle.net.ns.NetInputStream.getNextPacket(Unknown Source)
        at oracle.net.ns.NetInputStream.read(Unknown Source)
        at oracle.net.ns.NetInputStream.read(Unknown Source)
        at oracle.net.ns.NetInputStream.read(Unknown Source)
        at oracle.jdbc.ttc7.MAREngine.unmarshalUB1(MAREngine.java:730)
        at oracle.jdbc.ttc7.MAREngine.unmarshalSB1(MAREngine.java:702)
        at oracle.jdbc.ttc7.Oall7.receive(Oall7.java:373)
        at oracle.jdbc.ttc7.TTC7Protocol.doOall7(TTC7Protocol.java:1427)
        at oracle.jdbc.ttc7.TTC7Protocol.fetch(TTC7Protocol.java:911)
        at oracle.jdbc.driver.OracleStatement.doExecuteQuery(OracleStatement.java:1948)
        at oracle.jdbc.driver.OracleStatement.doExecuteWithTimeout(OracleStatement.java:2137)
        at oracle.jdbc.driver.OraclePreparedStatement.executeUpdate(OraclePreparedStatement.java:404)
        at oracle.jdbc.driver.OraclePreparedStatement.executeQuery(OraclePreparedStatement.java:344)
        at weblogic.jdbc.pool.PreparedStatement.executeQuery(PreparedStatement.java:51)
        at weblogic.jdbc.rmi.internal.PreparedStatementImpl.executeQuery(PreparedStatementImpl.java:56)
        at weblogic.jdbc.rmi.SerialPreparedStatement.executeQuery(SerialPreparedStatement.java:42)
        at  com.XXXX 생략
        at .....
```

<그림 3-11. Thread 가 DB Query Wait 에 걸려 있는 stack >

그림 3-10 상황은 DB 에 Query 를 보내고 response 를 Thread 들이 기다리고 있는 상황이다. AP 에서 JDBC 를 통해서 Query 를 보냈는데, Response 가 오지 않으면 계속 기다리게 있게 되고, 다른 Thread 가 같은 DB 로 보냈는데. Response 가 오지 않고, 이런것들이 중복되서 결국은 사용할 수 없는 Thread 가 없게 되서 새로운 request 를 처리하지 못하게 되고, 기존에 Query 를 보낸 내용도 응답을 받지 못하는 상황이다.

<그림 3-11>은 각각의 Thread 의 stack dump 인데, 그 내용을 보면 ExecuteQuery 를 수행한후에, Oracle 로 부터 데이타를 read 하기 위해 대기 하는것을 확인할 수 있다.

이런 현상은 주로 DB 사용시 대용량 Query(시간이 많이 걸리는)를 보냈거나, DB 에 lock 들에 의해서 발생하는 경우가 많으며, 그외에도 Socket 이나 File 을 사용하는 AP 의 경우 IO 문제로 인해서 발생하는 경우가 많다.

해결 방안
이 문제의 해결 방안은 Thread Dump 에서 문제가 되는 부분을 발견한후에, 해당 소스코드나 시스템등에 접근하여 문제를 해결해야한다.

- CASE 4. high CPU usage
  시스템이 느려지거나 거의 멈추었을때, 우리가 초기에 해볼 수 있는 조치가 무엇인가 하면, 현재 시스템의 CPU 사용률을 체크해볼 필요가 있다.

  해당 시스템의 CPU 사용률이 높은 경우, 문제가 되는경우가 종종 있는데. 이런 문제에서는 CPU 를 많이 사용하는 모듈을 찾아내는것이 관건이다.

  이를 위해서 먼저 top 이나 glance(HP UX)를 통해서 CPU 를 많이 점유하고 있는 Process 를 판독한다. 만약 WAS 이외의 다른 process 가 CPU 를 많이 사용하고 있다면, 그 Process 의 CPU 과사용 원인을 해결해야 한다. CPU 사용률이외에도, Disk IO 양이 많지는 않은지.. WAS 의 JVM Process 가 Swap out (DISK 로 SWAP 되는 현상) 이 없는지 살펴보도록 한다. JVM Process 가 Swapping 이 되면 실행속도가 엄청나게 느려지기 때문에, JVM Process 는 Swap out 되어버리면 안된다.

일단 CPU 를 과 사용하는 원인이 WAS Process 임을 발견했으면 프로그램상에 어떤 Logic 이 CPU 를 과점유하는지를 찾아내야한다.

방식을 정리해보면 다음과 같다.
WAS Process 의 Thread 별 CPU 사용률을 체크한다. ← 1)
Thread Dump 를 추출한다. ←2)
1)과, 2)에서 추출한 정보를 Mapping 하여, 2)의 Thread Dump 상에서 CPU 를 과점유하는 Thread 를 찾아내어 해당 Thread 의 Stack 을 분석하여 CPU 과점유하는 원인을 찾아낸다.

대부분 이런 요인을 분석해보면 다음과 같은 원인이 많다.

- 과도한 String 연산으로 인해서 CPU 사용률이 높아지는 경우
  잘 알고 있다시피 String 연산은 CPU 를 아주 많이 사용한다. String 과 Loop 문(for,while 등)의 사용은 CPU 부하를 유발하는 경우가 많기 때문에 가급적이면 String Buffer 를 사용하도록 하자.

- 과도한 RMI Cal
  RMI 호출은 Java Object 를 Serialize 하고 Deserialize 하는 과정을 수반하는데, 이는 CPU 를 많이 사용하는 작업이기 때문에 사용에 주의를 요한다. 특별히 RMI 를 따로 코딩하지 않더라도 EJB 를 호출하는것이 Remote Call 일때는 기본적으로 RMI 호출을 사용하게 되고, 부하량이 많을때 이 부분이 주로 병목의 원인이 되곤한다. 특히 JSP/Servlet→EJB 호출되는것이 같은 System 의 같은 JVM Process 안이라도 WAS 별로 별도의 설정을 해주지 않으면 RMI Call 을 이용하는 형태로 구성이 되기 때문에, 이에 대한 배려가 필요하다.

※ WebLogic 에서 Call By Reference 를 위한 호출 방법 정의

참고로 WebLogic 의 경우에는 Servlet/JSP→EJB 를 호출하는 방식을 Local Call 을 이용하기 위해서는 같은 ear 파일내에 패키징해야하고, EJB 의 weblogic-ejb-jar.xml 에 enable-call-by-reference 를 true 로 설정해줘야한다. (8.1 이상)

자세한 내용은
http://www.j2eestudy.co.kr/lecture/lecture_read.jsp?table=j2ee&db=lecture0201_1&id=1&searchBy=subject&searchKey=deploy&block=0&page=0
를 참고하시 바란다.

- JNDI lookup

JNDI lookup 은 Server 의 JNDI 에 Binding 된 Object 를 읽어오는 과정이다. 이 과정은 위에서 설명한 RMI call 로 수행이되는데. 특히 EJB 를 호출하기 위해서 Home 과 Remote Interface 를 lookup 하는 과장에서 종종 CPU 를 과점유하는 형태를 관찰 할 수 있다.

그래서 JNDI lookup 의 경우에는 EJB Home Interface 를 Caller Side(JSP/Servlet 또는 poor Java client)등에서 Caching 해놓고 사용하는 것을 권장한다. 단. 이경우에는 EJB 의 redeploy 기능을 제약받을 수 있다.

다음은 각 OS 별로 CPU 사용률이 높은 Thread 를 검출해내는 방법이다.

○ Solaris 에서 CPU 사용률이 높은 Thread 를 검출하는 방법

1.prstat 명령을 이용해서 java process 의 LWP(Light Weight process) CPU 사용률을 구한다.

% prstat –L –p [WeblogicPID] 1 1

```
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/LWPID
 26148 bwcho     143M   58M sleep   58    0   0:00.00 0.0% java/11
 26148 bwcho     143M   58M sleep   58    0   0:00.00 0.0% java/10
 26148 bwcho     143M   58M sleep   58    0   0:00.00 0.0% java/9
 26148 bwcho     143M   58M sleep   58    0   0:00.00 0.0% java/8
 26148 bwcho     143M   58M sleep   58    0   0:00.00 0.0% java/7
 26148 bwcho     143M   58M sleep   59    0   0:00.00 0.0% java/6
 26148 bwcho     143M   58M sleep   59    0   0:00.00 0.0% java/5
 26148 bwcho     143M   58M sleep   59    0   0:00.00 0.0% java/4
 26148 bwcho     143M   58M sleep   58    0   0:00.08 0.0% java/3
 26148 bwcho     143M   58M sleep   58    0   0:00.00 0.0% java/2
 26148 bwcho     143M   58M sleep   58    0   0:00.14 0.0% java/1
```

2. pstack 명령어를 이용해서 native thread 와 LWP 간의 id mapping 을 알아낸다.
(※ 전에 먼저 java process 가 lwp 로 돌아야되는데, startWebLogic.sh 에 LD_LIBRARY_PATH 에 /usr/lib/lwp 가 포함되어야 한다.)

% pstack [WebLogicPID]

```
 -----------------  lwp# 8 / thread# 24  --------------------
 ff29b3dc lwp_sema_wait (f2481e30)
 ff359818 _park    (f2481e30, ff37e000, 0, f2481d78, 25020, f1c81d78) + 114
 ff3594f0 _swtch   (f2481d78, 0, ff37e000, 5, 1000, e) + 424
 ff358004 cond_wait (4356, 36dfb0, ff37e000, 36dfc8, f2481d78, 0) + e4
 fe514f44 __1cNObjectMonitorEwait6MxlpnGThread__v_ (fe76fb0c, 36dfc8,
36dfb0, fe
```

3. 1 에서 얻은 LWP ID 를 pstack log 를 통해서 분석해보면 어느 Thread 에 mapping 되는지를 확인할 수 있다.
여기서는 LWP 8 이 Thread 24 과 mapping 이 되고 있음을 볼 수 있다.

kill -3 [WebLogicPID]를 해서 ThreadDump 를 얻어낸다.

```
"ExecuteThread: '11' for queue: 'default'" daemon prio=5 tid=0x36d630 nid=0x18
w
aiting on monitor [0xf2481000..0xf24819e0]
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:415)
        at weblogic.kernel.ExecuteThread.waitForRequest(ExecuteThread.java:114)
        at weblogic.kernel.ExecuteThread.run(ExecuteThread.java:138)
:
```

Thread dump 에서 nid 라는 것이 있는데, 2 에서 얻어낸 thread id 를 16 진수로 바꾸면 이값이 nid 값과 같다. 즉 2 에서 얻어낸 thread 24 는 16 진수로 0x18 이기 때문에, thread dump 에서 nid 가 0x18 인 쓰레드를 찾아서 어떤 작업을하고 있는지를 찾아내면 된다.

○ AIX Unix 에서 CPU 사용률이 높은 Thread 검출해 내기

1. ps 명령을 이용하여, WebLogic Process 의 각 시스템 thread 의사용률을 구한다.

% ps -mp [WeblogicPID] -0 THREAD

```
USER    PID   PPID     TID ST  CP PRI SC    WCHAN       F     TT BND COMMAND
   usera 250076 217266       - A   38  60 72       *   242011  pts/0   - /wwsl/sharedInstalls/aix/jdk130/...
      -     -     - 315593 Z    0  97  1       -   c00007     -   - -
      -     -     - 344305 S    0  60  1 f1000089c020e200   400400     -   - -
      -     -     - 499769 S    0  60  1 f1000089c0213a00   400400     -   - -
      -     -     - 540699 S    0  60  1 f100008790008440  8410400     -   - -
      -     -     - 544789 S    0  60  1 f100008790008540  8410400     -   - -
      -     -     - 548883 S    0  60  1 f100008790008640  8410400     -   - -
      -     -     - 552979 S    0  60  1 f100008790008740  8410400     -   - -
      -     -     - 565283 Z    0  60  1       -   c00007     -   - -
      -     -     - 585783 S    0  60  1 f100008790008f40  8410400     -   - -
      -     -     - 589865 Z    0  80  1       -   c00007     -   - -
      -     -     - 593959 S    1  60  1 f100008790009140  8410400     -   - -
      -     -     - 610365 S    0  60  1 f100008790009540  8410400     -   - -
      -     -     - 614453 S    0  60  1 f100008790009640  8410400     -   - -
      -     -     - 618547 S    0  60  1 f100008790009740  8410400     -   - -
      -     -     - 622645 S    0  60  1 f100008790009840  8410400     -   - -
      -     -     - 626743 S    0  60  1 f100008790009940  8410400     -   - -
      -     -     - 630841 S    0  60  1 f100008790009a40  8410400     -   - -
      -     -     - 634941 S    0  60  1 f100008790009b40  8410400     -   - -
      -     -     - 639037 S    0  60  1 f100008790009c40  8410400     -   - -
```

```
-    -    - 643135 S   0  60  1 f100008790009d40  8410400     -  - -
-    -    - 647233 S   0  60  1 f100008790009e40  8410400     -  - -
-    -    - 651331 S   0  60  1 f100008790009f40  8410400     -  - -
-    -    - 655429 S   0  60  1 f10000879000a040  8410400     -  - -
-    -    - 659527 S   0  60  1 f10000879000a140  8410400     -  - -
-    -    - 663625 S   0  60  1 f10000879000a240  8410400     -  - -
-    -    - 667723 S  37  78  1 f1000089c020f150   400400     -  - -
-    -    - 671821 S   0  60  1 f10000879000a440  8410400     -  -
```

여기서 CP 가 가장 높은 부분을 찾는다. 이 시스템 쓰레드가 CPU 를 가장 많이 점유하고 있는 시스템 쓰레드이다. (여기서는 66723 이다.)

2. dbx 명령을 이용해서 1.에서 찾은 시스템 쓰레드의 Java Thread ID 를 얻어온다.

1) % dbx -a [WebLogicPID]
2) dbx 에서 "thread" 명령을 치면 Thread ID 를 Listing 할 수 있다.

```
thread  state-k     wchan    state-u    k-tid   mode held scope function
 .....

 $t15    wait      0xf10000879000a140 blocked   659527     k   no   sys  _event_sleep
 $t16    wait      0xf10000879000a240 blocked   663625     k   no   sys  _event_sleep
 $t17    run                running   667723     k   no   sys  JVM_Send
 $t18    wait      0xf10000879000a440 blocked   671821     k   no   sys  _event_sleep
 $t19    wait                running   675919     k   no   sys  poll
 $t20    wait      0xf10000879000a640 blocked   680017     k   no   sys  _event_sleep
```

k-tid 항목에서 1 에서 찾은 Thread ID 를 찾고, 그 k-tid 에 해당하는 thread id 를 찾는다. (여기서는 $t17 이 된다.)

3) dbx 에서 $t17 번 쓰레드의 Java Thread ID 를 얻는다.
dbx 에서 "th info 17" 이라고 치면 $t17 번 쓰레드의 정보를 얻어온다.

```
(dbx) th info 17
    thread  state-k     wchan    state-u    k-tid   mode held scope function
    $t17    run                 running   667723     k   no   sys  JVM_Send

        general:
          pthread addr = 0x3ea55c68         size         = 0x244
          vp addr      = 0x3e69e5e0         size         = 0x2a8
          thread errno = 2
          start pc     = 0x300408b0
          joinable     = no
          pthread_t    = 1011
        scheduler:
          kernel       =
          user         = 1 (other)
        event :
          event        = 0x0
          cancel       = enabled, deferred, not pending
        stack storage:
          base         = 0x3ea15000         size         = 0x40000
          limit        = 0x3ea55c68
          sp           = 0x3ea55054
```

pthread_t 항목에서 Java Thread ID 를 얻는다. 여기서는 1011 이된다.

3. Java Thread Dump 에서 2 에서 얻어온 Java Thread ID 를 이용해서 해당 Java Thread 를 찾아서 Java Stack 을 보고 CPU 를 많이 사용하는 원인을 찾아낸다.

1) kill -3 [WebLogicPID]

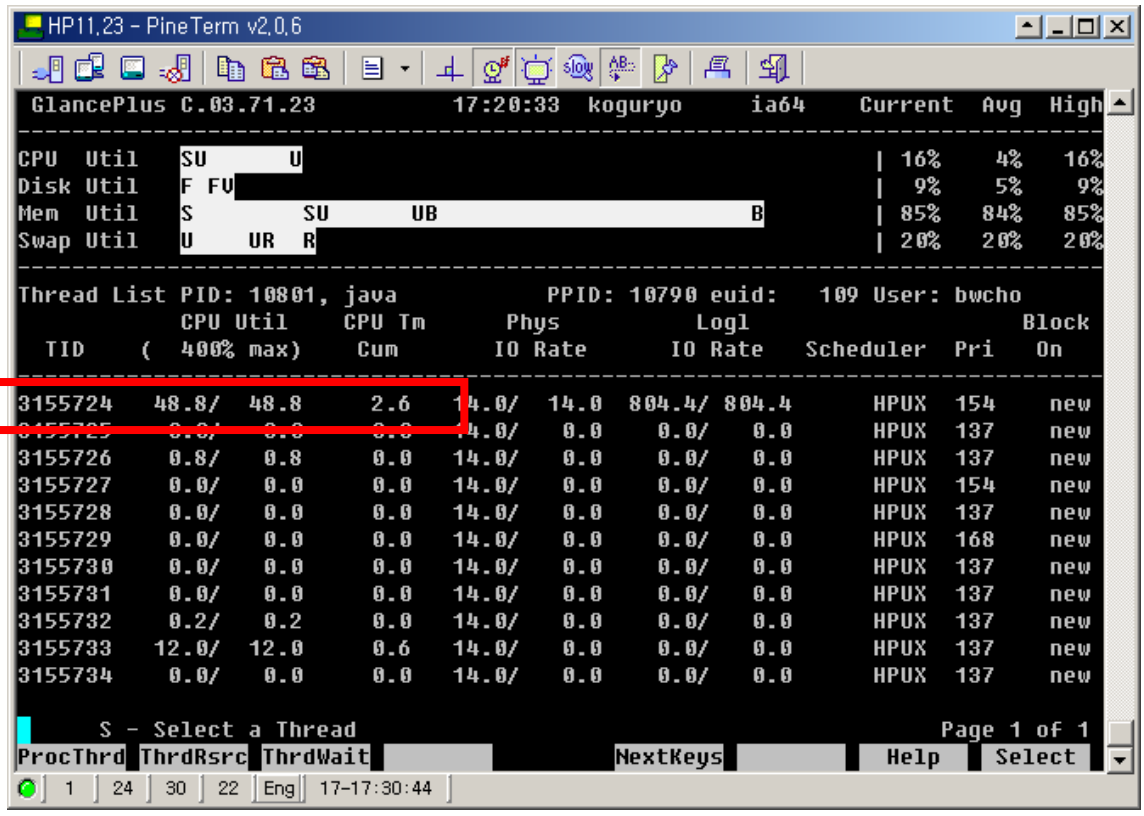
2) Thread dump 를 보면 native ID 라는 항목이 있는데, 2.에서 찾은 Java Thread ID 와 이 항목이 일치하는 Execute Thread 를 찾으면 된다.

```
"ExecuteThread: '11' for queue: 'default'" (TID:0x31cf86d8, sys_thread_t:0x3e5ea108, state:R,
native ID:0x1011) prio=5
        at java.net.SocketOutputStream.socketWrite(Native Method)
        at java.net.SocketOutputStream.write(SocketOutputStream.java(Compiled Code))
        at weblogic.servlet.internal.ChunkUtils.writeChunkTransfer(ChunkUtils.java(Compiled
Code))
        at weblogic.servlet.internal.ChunkUtils.writeChunks(ChunkUtils.java(Compiled Code))
        at weblogic.servlet.internal.ChunkOutput.flush(ChunkOutput.java(Compiled Code))
        at weblogic.servlet.internal.ChunkOutput.checkForFlush(ChunkOutput.java(Compiled
Code))
        at weblogic.servlet.internal.ChunkOutput.write(ChunkOutput.java(Compiled Code))
        at weblogic.servlet.internal.ChunkOutput.write(ChunkOutput.java(Compiled Code))
        at
weblogic.servlet.internal.ChunkOutputWrapper.write(ChunkOutputWrapper.java(Compiled
Code))
        at weblogic.servlet.internal.ChunkWriter.write(ChunkWriter.java(Compiled Code))
        at java.io.Writer.write(Writer.java(Compiled Code))
        at java.io.PrintWriter.write(PrintWriter.java(Compiled Code))
        at java.io.PrintWriter.write(PrintWriter.java(Compiled Code))
        at java.io.PrintWriter.print(PrintWriter.java(Compiled Code))
        at java.io.PrintWriter.println(PrintWriter.java(Compiled Code))
        at examples.servlets.HelloWorldServlet.service(HelloWorldServlet.java(Compiled Code))
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
        at
weblogic.servlet.internal.ServletStubImpl$ServletInvocationAction.run(ServletStubImpl.java:
1058)
        at weblogic.servlet.internal.ServletStubImpl.invokeServlet(ServletStubImpl.java:401)
        at weblogic.servlet.internal.ServletStubImpl.invokeServlet(ServletStubImpl.java:306)
        at
weblogic.servlet.internal.WebAppServletContext$ServletInvocationAction.run(WebAppServle
tContext.java:5445)
        at
weblogic.security.service.SecurityServiceManager.runAs(SecurityServiceManager.java(Comp
iled Code))
        at
weblogic.servlet.internal.WebAppServletContext.invokeServlet(WebAppServletContext.java:
3105)
        at weblogic.servlet.internal.ServletRequestImpl.execute(ServletRequestImpl.java:2588)
        at weblogic.kernel.ExecuteThread.execute(ExecuteThread.java(Compiled Code))
        at weblogic.kernel.ExecuteThread.run(ExecuteThread.java:189)
```

○ HP Unix 에서 CPU 사용률이 높은 Thread 검출해내기

1 먼저 JVM 이 Hotspot mode 로 작동하고 있어야 한다. (classic 모드가 아니어야 한다.)
옵션을 주지 않았으면 Hotspot 모드가 default 이다.

2. glance 를 실행해서 'G'를 누르고 WAS 의 PID 를 입력한다.
각 Thread 의 CPU 사용률이 실시간으로 모니터링이 되는데.



여기서 CPU 사용률이 높은 Thread 의 TID 를 구한다.

3. kill –3 을 이용해서 Thread dump 를 추출해서. 2 에서 구한 TID 와 Thread Dump 상의 lwp_id 가 일치하는 Thread 를 찾으면 된다.

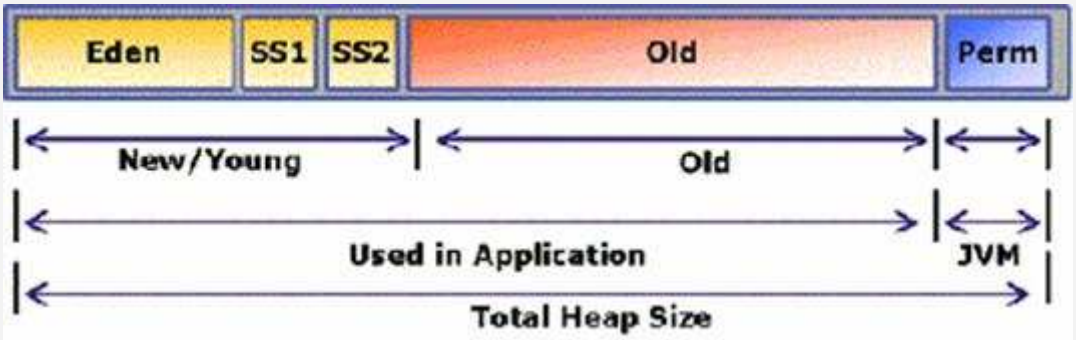
지금까지 Thread Dump 를 이용하는 방법을 간단하게 살펴보았다. 이 방법을 이용하면 WAS 와 그 위에서 작동하는 Appllication 의 Slow down 이나 hangup 의 원인을 대부분 분석해낼 수 있으나, Thread Dump 는 어디까지나 분석을 위한 단순한 정보이다… Thread Dump 의 내용이 Slow down 이나 hang up 의 원인이 될수도 있으나, 반대로 다른 원인이 존재하여 그 결과로 Thread Dump 와 같은 Stack 이 나올 수 있기 때문에, 여러 원인을 동시에 살펴보면서 분석할 수 있는 능력이 필요하다.

3. Slow down in JVM

WAS 의 성능에 큰 영향을 주는것중의 하나가 JVM 이다.

JVM 의 튜닝 여부에 따라서 WAS 상에서 작동하는 Ap 의 성능을 크게는 20~30% 까지 향상시킬 수 있는데, 우리가 지금 살펴보고 있는 slow down 과 hangup 을 일으키는 직접적인 요인이 되는것은 JVM 의 Full GC 이다.

간단하게 JVM 의 메모리 구조를 검토하고 넘어가보도록 하자.



<그림 4-1. JVM 의 메모리 구조 >

JVM 은 크게 New 영역과 Old 영역, 그리고 Perm 영역 3 가지로 분류가 된다. Perm 영역은 Class 나 Method 들이 로딩되는 영역이고 성능상의 영향을 거의 미치지 않는다.

우리가 주목해야할 부분은 객체의 생성과 저장에 관련되는 New 와 Old 영역인데, 모든 객체는 생성이 되자 마자 New 영역에 저장되고, 시간이 지남에 따라 이 객체들은 Old 영역으로 이동이 된다.

New 영역을 Clear 하는 과정을 Minor GC 라하고, Old 영역을 Clear 하는 과정은 Major GC 또는 Full GC 라 하는데, 성능상의 문제는 이 Full 영역에서 발생한다.

Minor GC 의 경우는 1 초 이내에 아주 고속으로 이뤄지는 작업이기 때문에, 신경을 쓸 필요가 없지만, Full GC 의 경우에는 시간이 매우 오래걸린다. 또한 Full GC 가 발생할 동안은 Application 이 순간적으로 멈춰 버리기 때문에 시스템이 순간적으로 Hangup 으로 보이거나 또는 Full GC 가 끝나면서 갑자기 request 가 몰려버리는 현상 때문에 종종 System 의 장애를 발생시키는 경우가 있다.

Full GC 는 통상 1 회에 3~5 초 정도가 적절하고, 보통 하루에 JVM Instance 당 5 회 이내가 적절하다고 여겨진다. (절대 값은 없다.)

Full GC 가 자주 일어나는것이 문제가 될경우에는 JVM 의 Heap 영역을 늘려주면 천천히 일어나지만 반대로 Full GC 에 소요되는 시간이 증가한다.

개당 Full GC 시간이 오래걸릴 경우에는 JVM 의 Heap 영역을 줄여주면 빨리 Full GC 가 끝나지만 반대로 Full GC 가 자주 일어난다는 단점이 있다.

그래서 이 부분에 대한 적절한 Tuning 이 필요하다.

대부분의 Full GC 로 인한 문제는 JVM 자체나 WAS 의 문제이기 보다는 그 위에서 구성된 Application 이 잘못 구성되어 메모리를 과도하게 사용하거나 오래 점유하는 경우가 있다.

예를 들어 대용량 DBMS Query 의 결과를 WAS 상의 메모리에 보관하거나 , 또는 Session 에 대량의 데이타를 넣는것들이 대표적인 예가 될 수 가 있다.

좀더 자세한 튜닝 방법에
대해서는 http://www.j2eestudy.co.kr/lecture/lecture_read.jsp?db=lecture0401_1&table=j2ee&id=1 를 참고하기 바란다.

4. Slow down analysis in DBMS

Application 이 느려지는 원인중의 많은 부분을 차지 하고 있는 것은 DBMS 의 성능 문제가 있는 경우가 많다.

흔히들 DBMS Tuning 을 받았더니 성능이 많이 향상되었다고 하는 경우가 많은데, 그건 그만큼 DB 설계를 제대로 하지 못했다는 이야기가 된다.

DBMS 자체 Tuning 에 대한 것은 이 문서와는 논외기 때문에 제외하기로 하고, DBMS 에 전송되는 각각의 SQL 문장의 실행 시간을 Trace 할 수 있는 것만으로도 많은 성능 향상을 기대할 수 있는데, 간단하게 SQL 문장을 실행시간은 아래 방법들을 이용해서 Trace 할 수 있다.

※http://eclipse.new21.org/phpBB2/viewtopic.php?printertopic=1&t=380&start=0&postdays=0&postorder=asc&vote=viewresult

※http://www.j2eestudy.co.kr/qna/bbs_read.jsp?table=j2ee&db=qna0104&id=5&searchBy=subject&searchKey=sql&block=0&page=0

5. Slow down analysis in Webserver & network

WAS 와 DBMS 앞단에는 WebServer 와 Network 이 있기 때문에 이 Layer 에서 문제가 되면 속도저하를 가지고 올 수 있다. 필자의 경험상 대부분의 slow down 이나 hangup 은 이 부분에서는 거의 일어나지 않지만 성능상에 종종 영향을 주는 Factor 가 있는데,

○ WebServer 와 Client 간의 KeepAlive

특히 WebServer 의 Keep Alive 설정이 그것이다.
WebBrowser 와 WebServer 간에는 KeepAlive 설정을 하는것이 좋은데. 그 이유는 WebBrowser 에서 하나의 HTML 페이지를 로딩하기 위해서는 Image 와 CSS 등의 여러 파일등을 로딩하는데, KeepAlive 설정이 없으면 각각의 파일을 로딩하는것에 각각의 Connection 을 open,request,download,close 를 한다. 잘들알고 있겠지만 Socket 을 open 하고 close 하는데에는 많은 자원이 소요된다. 그래서 한번 연결해놓은 Connection 을 계속 이용해서 HTTP data 를 주고 받는 설정이 KeepAlive 이다.
이 KeepAlive 설정은 웹을 이용한 서비스 제공에서 많은 성능 변화를 주기 때문에 특별한 이유가 없는한 KeepAlive 설정을 유지하기 바란다. 설정 방법은 각 WebServer 의 메뉴얼을 참고하기 바란다.

※ Apache2.0 의 Keep Alive 설정은

http://httpd.apache.org/docs-2.0/mod/core.html#keepalive 를 참고하기 바란다.
Default 가 KeepAlive 가 On 으로 되어 있다.

○ WebServer 와 WAS 간의 KeepAlive

WebServer 와 WAS 간에는 WebServer 에서 받은 request 를 forward 하기 위해서 WebServer Side 에 WAS 와 통신을 하기 위한 plug-in 이라는 모듈을 설치하게 된다. 이 역시 WebServer 와 Client 간의 통신과의 같은 원리로 KeepAlive 를 설정하게 되는데, 이 역시 성능에 영향을 줄 수 있는 부분이기 때문에 가급적이면 설정하기를 권장한다.

※ WebLogic 에서 Webserver 와의 KeepAlive 설정은

http://e-docs.bea.com/wls/docs81/plugins/plugin_params.html#1143055 을 참고하기 바란다. Default 는 KeepAlive 가 True 로 설정되어 있다

○ OS 에서 Kernel Parameter 설정

OS 의 TCP/IP Parameter 와, Thread 와 Process 등의 Kernel Parameter 설정이 운영에 있어서 영향을 미치는 경우가 있다. 이 Parameter 들은 Tuning 하기가 쉽지 않기 때문에, WAS 또는 OS Vendor 에서 제공하는 문서를 통해서 Tuning 하기 바란다.

※ WebLogic 의 OS 별 설정 정보은

http://e-docs.bea.com/platform/suppconfigs/configs81/81_over/overview.html 를 참고하기 바란다.

6. Common mistake in developing J2EE Application

지금까지 간단하게나마 J2EE Application 의 병목구간을 분석하는 부분에 대해서 알아보았다. 대부분의 병목은 Application 에서 발생하는 경우가 많은데, 이런 병목을 유발하는 Application 에 자주 발생하는 개발상의 실수를 정리해보도록 하자.

1) Java Programming

- sycnronized block

위에서도 설명 했듯이 sychronized 메소드는 lock contention 과 deadlock 등의 문제를 유발할 수 있다. 꼭 필요한 경우에는 사용을 해야하지만, 이점을 고려해서 Coding 해야한다.

- String 연산

이미 많은 개발자들이 알고 있는 내용이겠지만 String 연산 특히 String 연산중 "+" 연산은 CPU 를 매우 많이 소모하게 되고 종종 slow down 의 직접적인 원인이 되는 경우가 매우 많다.
String 보다는 가급적 StringBuffer 를 사용하기 바란다.

- Socket & file handling

Socket 이나 File Handling 은 FD (File Descriptor)를 사용하게 되는데, 이는 유한적인 자원이기 때문에 사용후에 반드시 close 명령을 이용해서 반환해야한다. 종종 close 를 하지 않아서, FD 가 모자르게 되는 경우가 많다.

2) Servlet/JSP Programming

- JSP Buffer Size

Jsp 에서는 JSP 의 출력 내용을 저장하는 buffer 사이즈를 지정할 수 있다.
<% page buffer=”12kb” %>
이 buffer size 는 출력 내용을 buffering 했다가 출력하는데, 만약에 쓰고자하는 내용이 Buffer size 보다 클 경우에는 여러번에 걸쳐서 socket write 가 일어나기 때문에 performance 에 영향을 줄 수 있으므로 가능하다면 buffersize 를 화면에 뿌리는 내용의 크기를 예측해서 지정해주는것이 바람직하다. 반대로 너무 큰 버퍼를 지정해버리면 메모리가 불필요하게 낭비 될 수 있기 때문에 이점을 주의하기 바란다.

참고로 jsp page buffer size 는 지정해주지 않는경우 default 로 8K 로 지정된다.

- member variable

Servlet/JSP 는 기본적으로 Multi Thread 로 동작하기 때문에, Servlet 과 JSP 내에서 선언된 멤버 변수들은 각 Thread 간에 공유가 된다.
그래서 이 변수들을 read/write 할경우에는 sychronized method 로 구성해야 하는데, 이 synchronized 는 속도 저하를 유발할 수 있기 때문에, member 변수로는 read 만 하는 객체를 사용하는게 좋다.

특히 Servlet 이나 JSP 에서 Data Base Connection 을 멤버 변수로 선언하여 Thread 간 공유하는 예가 있는데, 이는 별로 좋지 않은 프로그래밍 방법이고, 이런 형태의 패턴은 Servlet 이 단 하나만 실행되거나 하는것과 같은 제약된 조건 아래에서만 사용해야 한다.

- Out of Memory in file upload

JSP 에서 File upload control 을 사용하는 경우가 많다. 이 control 을 구현하는 과정에서 upload 되는 파일 내용을 몽땅 메모리에 저장했다가 업로드가 끝나면 한꺼번에 file 에 writing 하는 경우가 있는데, 이는 큰 사이즈의 파일을 업로드할때, 파일 사이즈만큼의 메모리 용량을 요구하기 때문에, 자칫하면 Out Of Memory 에러를 발생 시킬 수 있다.
File upload 는 buffer 를 만들어서 읽고, 파일에 쓰는 작업을 병행하도록 해야한다.

3) JDBC Programming

- Connection Leak

JDBC Programming 에서 가장 대표적으로 발생되는 문제가 Connection Leak 이다. Database Connection 을 사용한후에 close 않아서 생기는 문제인데,Exception 이 발생하였을때도 반드시 Connection 을 close 하도록 해줘야한다.

```
conn = getConnection();
try{
     do something
}catch(Exception e){ dosomething(); }
}finally{
  if(stmt!=null) stmt.close();
  if(conn!=null) conn.close();
}
```

< 그림 . Connection close 의 올바른 예 >

- Out of memory in Big size query result

SQL 문장을 Query 하고 나오는 resultset 을 사용할때, 모든 resultset 의 결과를 Vector 나 hashtable 등을 이용해서 메모리에 저장해놓는 경우가 있다. 이런 경우에는 평소에는 문제가 없지만, SQL Query 의 결과가 10 만건이 넘는것과 같은 대용량일때 이 모든 데이타를 메모리 상에 저장할려면 Out Of Memory 가 나온다.
Query 의 결과값을 처리할때는 ResultSet 을 직접 리턴받아서 사용하는것이 메모리 활용면에서 좀더 바람직하다.

- Close stmt & resultset

JDBC 에서 Resultset 이나 Statement 객체는 기본적으로 Connection 을 close 하게 되면 자동으로 닫히게 된다. 그러나 WAS 나 Servlet Container 의 경우에는 성능향상을 위해서 Connection Pooling 기법을 이용해서 Connection 을 관리하기 때문에 Connection Pooling 에서 Connection 을 close 하는것은 실제로 close 하는것이 아니라 Pool 에 반환하는 과정이기 때문에 해당 Connection 에 연계되어 사용되고 있는 Statement 나 ResultSet 이 닫히지 않는다.

Connection Pooling 에서 Statement 와 ResultSet 을 사용후에 닫아주지 않으면 Oracle 에서 too many open cursor 와 같은 에러가 나오게된다. (Statement 는 DB 의 Cursor 와 mapping 이 된다.)

4) EJB Programming

- When we use EJB?

EJB 는 분명 강력하고 유용한 개발 기술임에는 틀림이 없다. 그러나 EJB 의 장점과 용도를 모르고 사용하면 오히려 안쓰는것만 못한 경우가 많다.
각 EJB 모델 (Session Bean,Entity Bean)이 어떤때 유용한지를 알고 사용하고, 정확한 Transaction Model 등을 결정해서 사용해야 한다.

- Reduce JNDI look up

위에서도 설명했듯이 EJB 의 Home Interface 를 lookup 해오는 과정은 객체의 Serialization/DeSerialization 을 동반하기 때문에, 시스템 성능에 영향을 줄 수 있다. EJB Home 을 한번 look up 한후에는 Hashtable 등에 저장해서 반복해서 Remote Call(Serialization / DeSerialization)하는 것을 줄이는게 좋다.

- Do not use hot deploy in production mode

WAS Vendor 마다 WAS 운영중에 EJB 를 Deploy 할 수 있는 HotDeploy 기능을 제공한다. 그러나 이는 J2ee spec 에 있는 구현이 아니라 각 vendor 마다 개발의 편의성을 위해서 제공하는 기능이다. 운영중에 EJB 를 내렸다 올리는것은 위험하다. (Transaction 이 수행중이기 때문에) Hot Deploy 기능은 개발중에만 사용하도록 하자.

5) JVM Memory tuning

- Basic Tuning

Application 을 개발해놓고, 운영환경으로 staging 할때 별도의 JVM 튜닝을 하지 않는 경우가 많다. 튜닝이 아니더라도 최소한의 메모리 사이즈와 HotSpot VM 모델 (server/client)는 설정해줘야지 어느정도의 Application 의 성능을 보장 받을 수 있다. 최소한 메모리 사이즈와 VM 모델정도는 설정을 해주고 운영을 하도록 하자.

6. 결론

J2EE Application 의 병목구간을 확인하기 위해서는 그 문제를 발견하고 툴과 경험을 이용해서 문제의 원인을 발견하고 제거해야한다.

대부분의 WAS 또는 User Application 의 slow down 이나 hang up 은 Thread dump 를 통한 분석을 통해서 대부분 발견 및 해결을 할 수 있다.

그외에 부분 JVM 이나 WebServer,Network 등에 대해서는 별도의 경험과 Log 분석등을 알아내야하고 DB 에 의한 slow down 이나 hang up 현상은 DB 자체의 분석이 필요하다.