Host Locking

Host Locking is used to synchronize *concurrent* plan and component executions on a host. The locking takes place for all executions on the hosts managed by a single server instance. It is inherently built into all executions and by default ensures that a single host may only be locked by one plan or component at a time, thus serializing access to the host. However, this level of isolation may be controlled by the developer to a certain extent, as described below. Host locking is never used when accessing host or component instance variables using substitution language.

> Host Locking and Instance Locking are always both active and prevent deadlocks when interacting with hosts and instances respectivel y.

> A plan or component method may acquire multiple host locks during its execution, i.e. when retargeting to other hosts. Each retarget operation effectively acquires an additional host lock, which is released when the retarget operation finishes. Thus, an execution thread may block multiple hosts depending on the number of nested retarget operations, leaving them blocked for other executions. This should be taken into consideration during development, in order to avoid deadlock situations.

Isolation levels

This section describes the different isolation levels (and terms) used for locking executions.

Exclusive Locking (EXCLUSIVE)

With exclusive locking a host executing a plan or component method will be locked, such that no other execution can take place for the duration of the locking plan or component method. This means that any other plan or component trying to access the host in question will be blocked, until it is unlocked again, regardless of the isolation level requested by the latter one. Exclusive locking behaves equally for virtual and physical hosts.

It represents the default behavior, in case no other isolation levels are requested by the developer, i.e. any plan or component *not* specifying any locking behavior will be executed with this level, by default. In case a physical host is explicitly configured to be locked for any execution taking place on any of its virtual child hosts, exclusive locking actually behaves like exclusive parent locking for the afore mentioned virtual hosts.

Exclusive Parent Locking (EXCLUSIVE_PARENT)

Exclusive parent locking represents an extension of exclusive locking, also locking the physical parent host of the current execution host. Therefore, it is only different from exclusive locking in case the execution host is virtual.

Shared Locking (SHARED)

Shared locking is the lowest level of host locking, that may be selected by the developer. It allows other executions to take place at the same time for the host in question, if and only if these are shared lock executions as well. In other words, exclusively locked hosts cannot be accessed by a shared locking plan or component method at the same time. Shared locking also applies to the root physical host, if it is configured to be locked for any of its child hosts.

> A shared host lock may only be acquired when starting execution on a specific host, i.e. an already running execution (with a higher level) may not be downgraded to shared locking. This also applies if retargeting execution to a new host, which has already been accessed prior within the same execution thread.

Inherited Locking (INHERITED)

For retargeting operations it is possible to inherit the currently configured host locking mode, when accessing the new host.

Strict Host Locking

Any physical host can be configured to enforce strict host locking, such that all executions taking place on any of its child hosts require the physical host to be locked as well.

Deadlocks

With multiple plan or component executions in parallel, it may well happen, that two or more executions are trying to lock different hosts interdependently, i.e. two plans each owning a lock on host A and B trying to lock the other plan's host, respectively. This so-called deadlock situation will be detected by the system prior to acquiring the host lock in question, so that no two executions end up waiting for each other. In fact, the latter of the two executions will be aborted, thus preventing the deadlock situation detected.

> It is the obligation of the automaIT developer or architect to avoid deadlocks in the first place, e.g. by designing the plans and components using well-defined retarget strategies.

Examples

This section gives some examples for host locking in plans and components and explains their intention.

Declarative Locking

With *declarative locking* a plan or component method may be annotated with a specific locking strategy. This is primarily intended to serve as the initial locking strategy applied by the system, when the annotated plan or component method is directly executed by the user, e.g. via Web UI. In addition to that, the locking strategy annotated at component method level will also be applied, when called by other components or plans. The following example shows a plan defining shared locking as its default, mainly due to the fact that it executes non-critical statements, which may be run in parallel.

SharedPlan.xml

```
<plan path="/examples" name="SharedPlan" hostLockingMode="SHARED" ...>
  <echo message="This is an uncritical statement that may be executed in
parallel."/>
</plan>
```

Programmatic Locking

Programmatic locking allows locks to be upgraded from within a plan or component, giving the developer more fine-grained control over which statements shall be executed at which locking level. The following example shows a plan containing some critical statements, which are executed exclusively.

UpgradePlan.xml

```
<plan path="/examples" name="UpgradePlan" hostLockingMode="SHARED" ...>
  <echo message="This is an uncritical statement that may be executed in
parallel."/>

  <lockHost hostLockingMode="EXCLUSIVE_PARENT">
    <!-- This is a critical operation. -->
    <execNative cmd="rpm">
      <arg value="-i somepackage"/>
    </execNative>
  </lockHost>
</plan>
```

Retargets

Retargeted operations contained within plans and components may be annotated with different locking semantics, i.e. each retarget operation may decide which isolation level should be applied on the target host. The following plan shows two retarget operations, while the first one inherits the isolation from the surrounding plan (exclusive locking) and the second one applied shared locking to the target host.

RetargetPlan.xml

```
<plan path="/examples" name="RetargetPlan" ...>
  <echo message="This is the originating execution host."/>

  <retarget host="file-repository">
    <execNative cmd="rm">
      <arg value="/tmp/somefile"/>
    </execNative>
  </retarget>

  <retarget host="ldap-server" hostLockingMode="SHARED">
    <execNative cmd="finduser">
      <arg value="guest"/>
    </execNative>
  </retarget>
</plan>
```

Note, that the isolation level applied during a retarget operation does not enforce a downgrade of any previously existing locks, nor does it release any of them.

Deadlocking

Deadlocks occur, if two (or more) plan or component execution try to lock a host, which is already locked by the other one, and vice versa. Deadlocks are detected by the system in advance and resolved by aborting the plan or component execution in question, so system stability isn't affected. The following plan can be used as an example for deadlock demonstration, when executed against two hosts, retargeting against each other.

DeadlockPlan.xml

```
<plan path="/examples" name="RetargetPlan" ...>
  <paramList
    <param name="otherHost" default=""/>
  </paramList>
  <execNative cmd="sleep">
    <arg value="10"/>
  </execNative>

  <retarget host=":[otherHost]">
    <execNative cmd="sleep">
      <arg value="10"/>
    </execNative>
  </retarget>
</plan>
```