Application Note: TLT-0742-AN-MICROEJ-NativeResourceManagement

Native Resource Management

In relation to: MICROEJ products

Features

This Application Note explains how to automate the release of native resources used by Java objects. Native resources here mainly refer to those system resources that are not directly managed by JVM. Indeed, as Java language defines the "native" keyword for method that are executed out of JVM, "native resources" refers to memory allocated during such calls (file handles, sockets, stack buffers, etc.).

Description

This Application Note assumes the reader wishes to understand the steps involved in automating the release of native resources used by Java objects that are not referred anymore.

Table of Contents

List of Figures

1 Introduction

1.1 Intended audience

The intended audience for this Application Note are developers who want to ensure that native resources used by Java objects are always released when the Java object is garbage collected.

1.2 Scope

This Application Note shows how the weak reference support provided by the B-ON library can be used to automate the release of native resources (such as buffers) when the Java objects using them are garbage collected.

1.3 Background

When a Java object is created the RAM required to hold its data is allocated from a memory heap, and the code that requested the creation is given a reference to the new object, which it holds in a field or local variable. That reference can be passed to other objects in the application, so that at any time there may be many references to an object. A Java object can only be accessed via a reference to it. When a reference is no longer required the field or local variable holding it is overwritten with a new value, either a reference to a different object or the special value `null`. If there are no longer any references to an object then the object cannot be used and the memory it occupies is automatically reclaimed by the garbage collector that forms part of the MicroJvm® virtual machine.

Imagine a scenario where a Java application creates several Java objects, each of which uses a native resource. When such a Java object is created it typically will call a native method that allocates the resource. At some later point when the Java object is no longer needed by the application it will be garbage collected. It is essential that the native resource the object was using is also released. That could be achieved by having the application invoke a special "release" method provided by the object, but this is often problematic because references to the object can be held in many places and there may be no one place that knows when the object is no longer required. A better solution is to implement a mechanism that will automatically release the resource when the object is garbage collected. This Application Note describes such a mechanism.

1.4 Prerequisites

This document assumes the reader is familiar with the process of creating a Java Platform (JPF), and with the creation of native methods using SNI.

1.5 Files supplied with this Application Note

This application note is packaged with an archive of an Eclipse project in the file `NativeResourceManagement-example.zip`, which can be imported in the normal way.

The project contains five files of particular interest:

- The file `NativeResourceManagementExample.java`, in the `src-example` folder, is a MicroEJ® Java application that demonstrates the management of native resources.
- The files `ResourceUser.java` and `ResourceUserHelper.java`, in the `src-example` folder, are a simple example of Java code that uses native resources.
- The file `ResourceManager.java`, in the `src-framework` folder, is a Java class that can help manage any kind of resource.
- The file `resource_user.c`, in the `c-src` folder, contains implementations of the native methods used by `ResourceUser.java`.

2 Design

As discussed above in the Introduction, a Java object becomes eligible for garbage collection when there are no references to it. However, the B-ON library supports the concept of a "weak reference". A weak reference is a reference that is ignored by the garbage collector when deciding whether an object is eligible for garbage collection.

B-ON provides a class called `EnqueuedWeakReference` whose instances can hold a weak reference. They also hold a (normal) reference to an instance of `ReferenceQueue`, another class provided by B-ON. When the object to which an `EnqueuedWeakReference` holds a weak reference is garbage collected, the `EnqueuedWeakReference` is added to the queue managed by the `ReferenceQueue`. The application can monitor this queue to detect when an object of interest has been garbage collected.

Our design uses a specially-written subclass of `EnqueuedWeakReference` called `ResourceReference`. In addition to holding a weak reference to the object using the resource, instances of `ResourceReference` hold a reference to a helper object that knows how to release the native resource being held. Each resource-using object has such a helper.

The arrangement is as shown in the figure below.

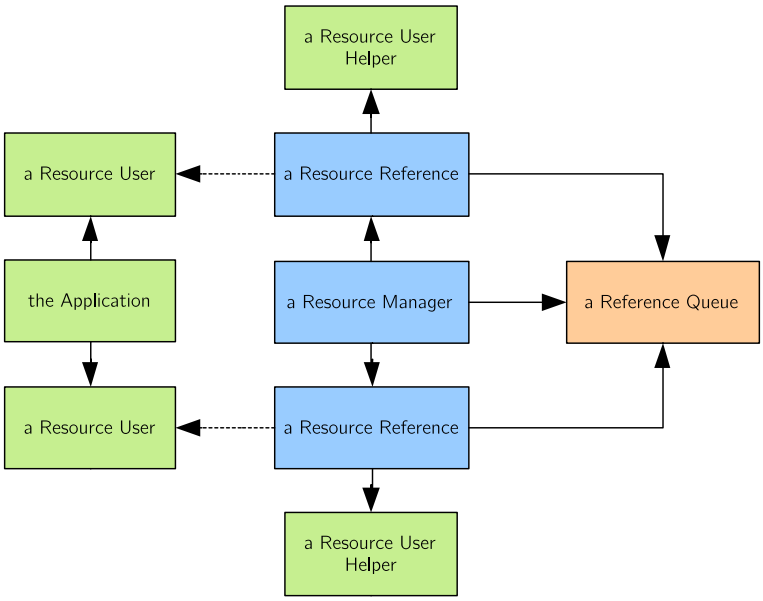

Figure 2.1. Objects prior to garbage collection

The rectangles in this diagram represent objects. The solid arrows represent normal references and the dashed arrows represent weak references. The application has created two `ResourceUser` objects, each of which is using a native resource. When it was constructed the `ResourceUser` object created a `ResourceUserHelper` object, which knows how to allocate and release its native resource, and (via the `ResourceManager`) a `ResourceReference` that holds a weak reference to it.

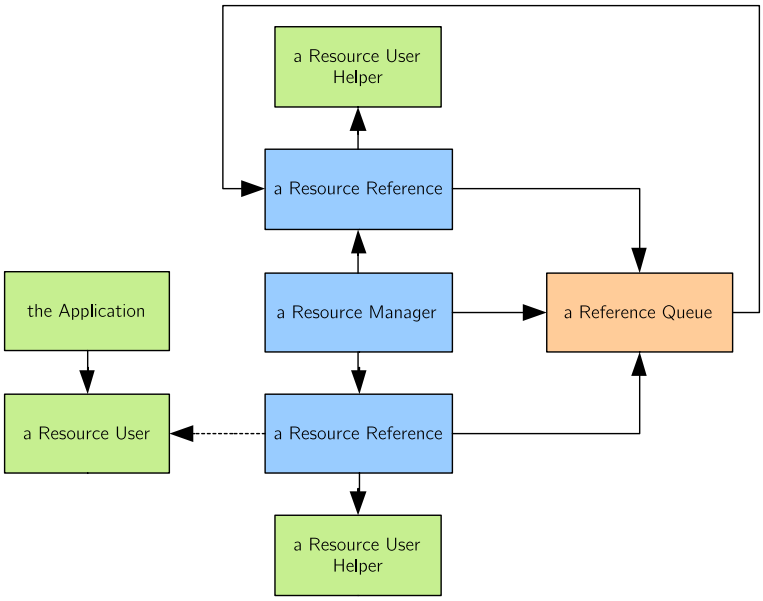

Figure 2.2. Objects after garbage collection

The application has now finished using one of the `ResourceUser` objects, and because the only remaining reference to the `ResourceUser` is a weak reference, the `ResourceUser` has been garbage collected. The `ResourceReference` that was referring to the `ResourceUser` has been added to the `ReferenceQueue`, ready for processing.

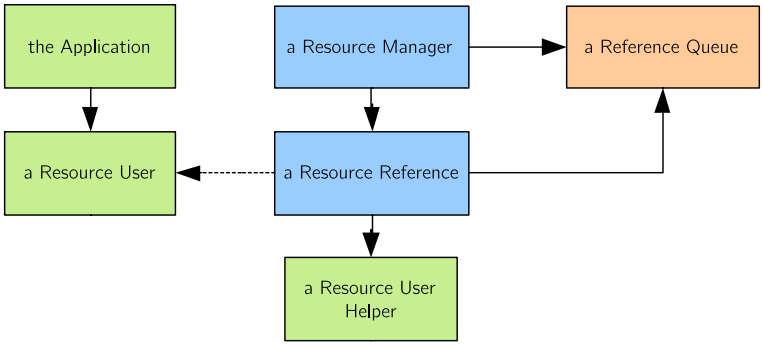

Figure 2.3. Objects after queue has been emptied

The application has taken the expired `ResourceReference` from the `ReferenceQueue` and its `ResourceUserHelper` has been used to release the native resource.

3 Implementation

We will now look at the supplied implementation, which follows the outline shown in the diagrams above.

First we will look at the code of a very simple application (in `NativeResourceManagementExample.java`). The application has no useful function but it demonstrates the principle.

```
public class NativeResourceManagementExample {

    public static void main(String[] args) throws Exception {
        createAndUseResources();
    }

    private static void createAndUseResources() throws OutOfResourcesException {
        ResourceUser resourceUser0 = new ResourceUser();
        ResourceUser resourceUser1 = new ResourceUser();
        resourceUser0.useResource();
        resourceUser1.useResource();
        try {
            new ResourceUser();
        } catch (OutOfResourcesException e) {
            // expected because only 2 resources are available
        }
        resourceUser0 = null;
        ResourceUser resourceUser2 = new ResourceUser();
        resourceUser2.useResource();
    }
}
```

Figure 3.1. Example application

The application first creates two resource-using objects and simulates their use in the application (the calls to `useResource`). It then attempts to create a third resource user. This will fail because only two native resources are available, as we will see shortly. The application then indicates that it has finished with the resource referred to by `resourceUser0` by setting it to `null`. It then makes another attempt to create a new resource user, and this succeeds because one of the two initial resources can be released.

```
#define ERROR_OUT_OF_RESOURCES -1

#define RESOURCE_LIMIT 2
static uint8_t resource_in_use[RESOURCE_LIMIT];

jint Java_com_is2t_example_ResourceUser_allocateResource() {
    jint result = ERROR_OUT_OF_RESOURCES;
    for (jint id = 0; id < RESOURCE_LIMIT; ++id) {
        if (!resource_in_use[id]) {
            result = id;
            resource_in_use[id] = 1;
            break;
        }
    }
    printf("Allocating resource %i\n", result);
    return result;;
}

void Java_com_is2t_example_ResourceUser_useResource__I(jint id) {
    printf("Using resource %i\n", id);
}

void Java_com_is2t_example_ResourceUser_releaseResource(jint id) {
    printf("Releasing resource %i\n", id);
    resource_in_use[id] = 0;
}
```

Figure 3.2. Native resource implementation

The implementation of the native resources is simple. The native resource is simply its id. There are a fixed number of resources available, and the availability of each resource is indicated by a flag in an array indexed by the id.

The result of running this application is shown below.

```
Allocating resource 0
Allocating resource 1
Using resource 0
Using resource 1
Allocating resource -1
Allocating resource -1
Allocating resource -1
Releasing resource 0
Allocating resource 0
Using resource 0
```

Figure 3.3. Generated output

The first four lines show two resources being allocated and used. The next two lines (both "`Allocating resource -1`") the result of the unsuccessful attempt to allocate the third resource. The last "`Allocating resource -1`" is generated during the successful attempt - you can see resource 0 being released and re-allocated.

The `ResourceUser` class is mainly a wrapper for the underlying C natives.

```
class ResourceUser {
    public static final int ERROR_OUT_OF_RESOURCES = -1;

    private static final ResourceManager resourceManager = new ResourceManager();

    private final int cResourceId;

    public ResourceUser() throws OutOfResourcesException {
        ResourceUserHelper helper = new ResourceUserHelper();
        resourceManager.allocateResource(this, helper);
        cResourceId = helper.getResourceId();
    }

    public int getCResourceId() {
        return cResourceId;
    }

    public void useResource() {
        useResource(cResourceId);
    }

    native static int allocateResource();
    private native static void useResource(int cResourceId);
    native static void releaseResource(int cResourceId);
}
```

Figure 3.4. ResourceUser class

The constructor first creates a helper object, and then calls the `ResourceManager`'s `allocateResource` method to allocate and track the native resource. The `ResourceUserHelper` object is passed to the `ResourceManager`, ready for use when the `ResourceUser` is garbage collected. The actual allocation is done by the helper. Notice that the helper holds the id of the resource because it will need it to release the resource - by that time the `ResourceUser` will have been garbage collected, so cannot be used to find the id.

The `ResourceUserHelper` class is very simple.

```
class ResourceUserHelper implements ResourceManager.ResourceHelper {
    int cResourceId;

    public void allocateResource() throws OutOfResourcesException {
        cResourceId = ResourceUser.allocateResource();
        if (cResourceId == ResourceUser.ERROR_OUT_OF_RESOURCES) {
            throw new ResourceManager.OutOfResourcesException();
        }
    }

    public void cleanup() {
        ResourceUser.releaseResource(cResourceId);
    }

    public int getResourceId() {
        return cResourceId;
    }
}
```

Figure 3.5. ResourceUserHelper class

The `allocateResource` method is called by the `ResourceManager` when it is asked to allocate the resource. Notice that an exception defined by the `ResourceManager` is thrown if no resource is available.

The `cleanup` method is called by the `ResourceManager` when the related `ResourceUser` object is garbage collected. It is essential that the helper object does not hold a reference to the resource-using object, because that would prevent it from being garbage collected.

The `ResourceManager` class does most of the work. Notice that it provides a simple interface (`ResourceHelper`) which all helper classes must implement. It also has a private inner class, the `ResourceReference` class, which extends `EnqueuedWeakReference` to provide behavior specific to this use of weak references.

```
public class ResourceManager {
    public static class OutOfResourcesException extends Exception {
    }

    public static interface ResourceHelper {
        void allocateResource() throws OutOfResourcesException;
        void cleanup();
    }

    private final ReferenceQueue queue = new ReferenceQueue();
    private final Vector activeReferenceList = new Vector();

    public void allocateResource(Object resourceUser, ResourceHelper helper) throws
 OutOfResourcesException {
        cleanup();
        try {
            helper.allocateResource();
        } catch (OutOfResourcesException e) {
            System.gc();
            cleanup();
            helper.allocateResource();
        }
        activeReferenceList.addElement(new ResourceReference(helper, resourceUser,
 queue));
    }

    private void cleanup() {
        ResourceReference removedReference;
        while((removedReference=(ResourceReference) queue.poll()) != null){
            activeReferenceList.removeElement(removedReference);
            removedReference.cleanup();
        }
    }

    private static class ResourceReference extends EnqueuedWeakReference {
        private final ResourceHelper helper;

        private ResourceReference(ResourceHelper helper, Object resourceUser,
 ReferenceQueue queue) {
            super(resourceUser, queue);
            this.helper = helper;
        }

        private void cleanup() {
            helper.cleanup();
        }
    }
}
```

Figure 3.6. ResourceManager class

When the `ResourceManager` is asked to allocate a resource it first frees any unused resources by calling its `cleanup` method). This will release any resources owned by `ResourceUser` objects that have been garbage collected since the last allocation request. Then it asks the helper to do the allocation. If that fails (with an `OutOfResourcesException`) it explicitly requests a garbage collection and frees any resources that become unused at a result, and asks the helper to try the allocation again. This time the exception is not caught; it is thrown to the caller because there really are no resources available. If a resource was available it creates a `ResourceReference` and adds it to its list of active references. The only purpose of this list is to prevent the `ResourceReference` objects themselves, and the helper objects they refer to, being garbage collected.

To free unused resources the `cleanup` method removes any expired `ResourceReference` objects from the `ReferenceQueue` and uses their associated helper objects to cleanup the native resources.

This design for the `ResourceManager` is appropriate when it is not important to release the underlying resources at the earliest opportunity - in this design resources are not released until another allocation is requested. An alternative design would spawn a thread in the `ResourceManager` that is blocked waiting for a `ResourceReference` to expire, and immediately releases the resource.

4 Running the example

To run the example you must already have a suitable Java Platform (JPF), possibly a "Basic" JPF created using the Java Platorm Example feature of the MicroEJ workbench.

1. Import the Eclipse project provided with this Application note. The project is in the file `NativeResourceManagement-example.zip`.

2. Create a suitable "EmbJPF" launch configuration to build the `NativeResourceManagementExample` application, and run it.

3. Copy the file `resource_user.c` from the `c-src` folder to the `src` folder of your BSP project.

4. In your C IDE, add the file `resource_user.c` to your project.

5. Ensure that the object file built by running the "EmbJPF" launch in the earlier step is available to your µVision project - the projects created using the Java Platorm Example feature assume that the Java object file is in the `xxxJPF/source/lib` folder, so either your launch should copy it there or you should reconfigure the C project to access it from wherever the launch puts it.

6. Build, deploy and run your C project.

5 Document History

Date	Revision	Description
July 2nd 2013	A	First release
October 4th 2013	B	Features paragraph improvement

Headquarters
11, rue du chemin Rouge
44373 Nantes Cedex 3
FRANCE
Phone: +33 2 40 18 04 96
www.is2t.com