



Multitasking Guide

Sun Java™ Wireless Client Software, Version 2.2
Java Platform, Micro Edition

Copyright © 2008 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

THIS PRODUCT CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF SUN MICROSYSTEMS, INC. USE, DISCLOSURE OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF SUN MICROSYSTEMS, INC.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, Solaris, HotSpot, J2ME, J2SE, J2EE, Java Developer Connection, Java Community Process, JCP, Javadoc, JDK, JavaCall, Java Card, phoneME and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. or its subsidiaries in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

OpenGL is a registered trademark of Silicon Graphics, Inc.

The PostScript logo is a trademark or registered trademark of Adobe Systems, Incorporated, which may be registered in certain jurisdictions.

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2008 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, États-Unis. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plusieurs des brevets américains listés à l'adresse <http://www.sun.com/patents> et un ou plusieurs brevets supplémentaires ou les applications de brevet en attente aux États-Unis et dans d'autres pays.

CE PRODUIT CONTIENT DES INFORMATIONS CONFIDENTIELLES ET DES SECRETS COMMERCIAUX DE SUN MICROSYSTEMS, INC. SON UTILISATION, SA DIVULGATION ET SA REPRODUCTION SONT INTERDITES SANS L'AUTORISATION EXPRESSE, ECRITE ET PREALABLE DE SUN MICROSYSTEMS, INC.

Droits du gouvernement des États-Unis - logiciel commercial. Les droits des utilisateur du gouvernement des États-Unis sont soumis aux termes de la licence standard Sun Microsystems et aux conditions appliquées de la FAR et de ces compléments.

Cette distribution peut inclure des éléments développés par des tiers.

Sun, Sun Microsystems, le logo Sun, Java, Solaris, HotSpot, J2ME, J2SE, J2EE, Java Developer Connection, Java Community Process, JCP, Javadoc, JDK, JavaCall, Java Card, phoneME et le logo Java Coffee Cup sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc., ou ses filiales, aux États-Unis et dans d'autres pays.

UNIX est une marque déposée aux États-Unis et dans d'autres pays et sous licence exclusive de X/Open Company, Ltd.

Intel est une marque déposée de Intel Corporation ou de sa filiale aux États-Unis et dans d'autres pays.

OpenGL est une marque déposée de Silicon Graphics, Inc.

Le logo PostScript est une marque de fabrique ou une marque déposée de Adobe Systems, Incorporated, laquelle pourrait à déposée dans certaines juridictions.

Les produits qui font l'objet de ce manuel d'entretien et les informations qu'il contient sont régis par la législation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des États-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régi par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DÉCLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISÉE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITÉ MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIÈRE OU A L'ABSENCE DE CONTREFAÇON.

Contents

Preface	ix
1. Introduction	1
Multitasking	2
Robustness	3
Mechanisms Compared With Policies	3
2. Multitasking Safety	5
Multitask Safety and Multithread Safety	6
Global and Static Data	7
Singletons	8
Multitasking Safety Example	9
Multithread Safety	11
Multitask Safety	13
Establishing Per-Task Context	15
3. Managing Native Resources	21
Resource Management Mechanisms	22
Reservation	22
Limit	23
Revocation	24

Default Resource Allocation Policies	24
Customization of Resource Allocation Policies	25

4. Other Multitasking Issues 29

Switching the Foreground MIDlet	29
Default Policy	30
Alternative Policies and Their Implementations	30
Scheduling the CPU	30
Default CPU Scheduling Policy	31
Alternative Policies and Their Implementations	31
Interrupting the User	32
Default User Notification Policies	32

Glossary 33

Index 37

Tables

TABLE 3-1 Constant Definitions for the Resource Management Policy 26

Code Examples

- [CODE EXAMPLE 2-1](#) Native API for a Microwave Oven 9
- [CODE EXAMPLE 2-2](#) Typical usage of the microwave 10
- [CODE EXAMPLE 2-3](#) Simple Java API for the Microwave Oven 10
- [CODE EXAMPLE 2-4](#) Introducing a Locking Mechanism for Thread Safety 11
- [CODE EXAMPLE 2-5](#) Using the Locking Mechanism 12
- [CODE EXAMPLE 2-6](#) Migrating the Initialization Variable to Native Code (Doesn't Work) 13
- [CODE EXAMPLE 2-7](#) Migrating Initialization to Native Code 14
- [CODE EXAMPLE 2-8](#) Keeping State in Java Code 16
- [CODE EXAMPLE 2-9](#) Implementing the Native `n_cook()` Method 17

Preface

The *Multitasking Guide* highlights multitasking programming issues in the Sun Java™ Wireless Client software. It describes how to make code safe for the multitasking environment of the Java Wireless Client software. There is a special section about resource management. The *Multitasking Guide* also describes some multitasking policies implemented in the Java Wireless Client software and discusses possible alternatives.

Note – Sun Microsystems has simplified the naming schemes for the various Java platforms. Java Platform, Enterprise Edition (Java EE) was formerly Java 2 Platform, Enterprise Edition (J2EE™). Java Platform, Standard Edition (Java SE) was formerly Java 2 Platform, Standard Edition (J2SE™), and Java Platform, Micro Edition (Java ME) was formerly Java 2 Platform, Micro Edition (J2ME™).

References in this guide to specific documents, specifications, and products that were released when the old naming scheme was in use retain their original names. General references in this guide to Java platforms use the new, simplified naming scheme.

Before You Read This Guide

Readers using this guide must be familiar with the *MIDP 2.1 Specification*. The specification is available from <http://www.jcp.org/>. It is also useful if the reader is familiar with the Java Wireless Client software code.

Note – Note - Sun is not responsible for the availability of web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials available through such sites.

How This Guide Is Organized

This book contains the following chapters:

[Chapter 1](#) provides an overview of the implementation and pitfalls of the Java Wireless Client software's multitasking environment.

[Chapter 2](#) describes how source code can be made safe for multitasking.

[Chapter 3](#) covers strategies and implementations for managing device resources.

[Chapter 4](#) includes a brief overview of the multitasking policies and alternatives in the Java Wireless Client software.

Related Documentation

The following documentation is included with this release of the Java Wireless Client software:

Application	Title
All	<i>Release Notes</i>
Building	<i>Build Guide</i>
Porting Issues and Overview	<i>Architecture and Design Guide</i>
Porting Procedures and Guidelines	<i>Porting Guide</i>
Running SJWC Software and Using Tools	<i>Tools Guide</i>
Multitasking Integration and Policies	<i>Multitasking Guide</i>
Using Adaptive User Interface Technology (skins)	<i>Skin Author's Guide to Adaptive User Interface Technology</i>

Application	Title
Configuration and Testing Tools	<i>Tools Guide</i>
Viewing reference documentation created by the Javadoc™ tool	<i>Java API Reference</i>
Viewing reference documentation created by the Doxygen tool	<i>Native API Reference</i>

Typographic Conventions Used in This Guide

Typeface	Meaning	Examples
Courier AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. % You have mail.
Bold AaBbCc123	What you type, when contrasted with on-screen computer output	% su Password:
<i>Italic</i> <i>AaBbCc123</i>	Important parts of a code sample Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be super user to do this.
	Command-line variable; replace with a real name or value	To delete a file, type <code>rm filename</code> .

Accessing Sun Documentation Online

The Java Developer Connection™ program web site enables you to access Java platform technical documentation at <http://java.sun.com/>.

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. Provide feedback to Sun at <http://java.sun.com/docs/forms/sendusmail.html>.

Introduction

Customers use mobile phones and other handheld devices for many tasks, such as making phone calls, taking photographs, playing games, organizing contact information, keeping a calendar of events, and accessing web sites. It is natural for them to want to do more than one of these tasks at a time. For example, a customer might not want to shut the address book to receive a flight-delay alert. The Java Wireless Client software, which can concurrently run multiple MIDlets, meets this need.

Device manufacturers and service providers also benefit from being able to run multiple MIDlets concurrently. This functionality enables them to write built-in (resident) applications, such as address books, in the Java programming language. Applications written in the Java programming language (Java applications) are easier to write and maintain, more portable, and easier to customize than those written in native code. The Java Wireless Client software meets this need too.

In addition, the Java Wireless Client software enables MIDlet writers to take advantage of multitasking without changing their code. MIDlet suites that run in the Java Wireless Client software's multitasking environment are no different from any other MIDlets: They have no new APIs and they have the same life cycle. The MIDP 2.1 Specification advises all MIDlet writers about managing resources when the MIDlet is paused and resumed. In a multitasking environment, a MIDlet might compete with other MIDlets for resources, so it is more important than ever for MIDlet writers to follow that advice and to use resources carefully.

Multitasking

The Connected Limited Device Configuration HotSpot™ Implementation can run multiple Java applications within a single operating system (OS) process. Historically, a CLDC virtual machine (VM) could run one Java application at a time, and each virtual machine typically required its own OS process. Running more than one Java application meant running more than one OS process. This could use too many resources on some small devices. The multitasking feature enables a single virtual machine, in a single OS process, to function as multiple virtual machines. From the standpoint of each Java application, it is running in a separate virtual machine.

When a MIDlet exits or encounters an error, it always leaves the virtual machine in a consistent state. This works because each MIDlet is isolated from other MIDlets that might also be running at the same time. If the MIDlets were not isolated from each other, an error in one MIDlet might be visible to other MIDlets. This might result in deadlock or crashes caused by corrupted data structures.

Multitasking, then, runs multiple logical virtual machines within a single OS process. The CLDC HotSpot Implementation enables this by providing fundamental mechanisms in the virtual machine. Mechanisms include threads, the isolation of objects in the Java runtime environment (Java objects) belonging to different programs, and safe termination. See the *CLDC HotSpot Implementation Virtual Machine White Paper* at http://java.sun.com/j2me/docs/pdf/CLDC-HI_whitepaper-February_2005.pdf for more information.

The OS mechanisms are important and necessary, but they aren't sufficient to provide a complete application environment. A complete system needs mechanisms for managing the application life cycle (starting, stopping, and switching between applications) and for enabling the user to control the application life cycle. It also needs mechanisms for managing competition for limited shared resources, such as memory and network sockets, among multiple applications. The Java Wireless Client software provides all of these additional mechanisms.

The Java Wireless Client software is a high-performance, feature-rich, deployment-ready implementation of Java Technology for the Wireless Industry that takes advantage of the multitasking functionality in the CLDC HotSpot Implementation.

The logical virtual machines in the Java Wireless Client software are called *tasks*. Each application has its own task. The Application Management System (AMS) runs at all times in its own task. This architecture provides improved performance and robustness.

Robustness

In addition to providing the multitasking that users want, tasks have the following benefits:

- **Fault containment** - If a Java application crashes, then any problems caused by this crash are limited to the task. Applications running in other tasks are unaffected.
- **Clean termination** - When a Java application exits, it leaves the virtual machine in a consistent state. In the past, some have tried to run multiple Java applications in a single process by running the applications in separate threads within a single virtual machine. When applications share a virtual machine, it is impossible to guarantee that an application can exit cleanly, because objects might be left in an inconsistent state. In contrast, tasks can and do terminate cleanly.

Note that in MIDP, a Java application is a MIDlet suite. A MIDlet suite is one or more MIDlets in the same Java Archive (JAR) file. Each MIDlet suite, then, runs in its own task. The user interacts with one of the MIDlets in the MIDlet suite.

Mechanisms Compared With Policies

Mechanisms are actions that the system carries out, such as starting, pausing, and stopping a MIDlet. Policies are the system-wide decisions that device manufacturers and service providers make about when and how to use the available mechanisms. Policies take generic mechanisms and combine them into a predictable set of system behaviors.

The Java Wireless Client software provides both the necessary mechanisms for a complete application environment, such as mechanisms for managing competition for limited shared resources, and a default set of policies. For example, memory is a limited shared resource for which the Java Wireless Client software provides mechanisms and a few default policies. The mechanism is the allocation of heap memory. One default policy is to allow the applications to compete for available heap memory.

The default policies are useful, but device manufacturers and service providers are likely to need to tailor policies to their particular platforms. Devices of different capabilities might use different policies. Devices with different user interface styles might use different policies so that their Java applications interact with users in ways that match the native applications interactions. Because policy decisions are often intertwined with user interface style and user model decisions, the Java Wireless Client software enables the development of alternative policies.

Because policies are so device dependent, this book does not recommend specific policy combinations. As you determine the policies for your device, keep in mind that policies interact with each other and not all combinations of policies make sense. For example, if you have a policy to allow a MIDlet to access a sound device while it is in the background, it does not make sense to also have a policy to suspend a MIDlet's execution when it is in the background.

Multitasking Safety

The Java Wireless Client software provides the ability to run multiple MIDlets concurrently in a single OS process. From the standpoint of the OS, there is one process and one Java virtual machine. However, from the standpoint of a Java application, it appears as if it is running in its own, independent virtual machine, isolated from other Java applications.

These apparently independent virtual machines are called tasks. Each MIDlet runs in its own task. When a new MIDlet is started, a new task is created for it. When a MIDlet exits, its task is destroyed. The Application Management Software (AMS) runs in a dedicated task, and it is running the entire time the Java Wireless Client software is active.

Although tasks cannot interact (they cannot access each other's objects, for example), they do share native code, process resources, and external resources. This sharing leads to some implementation issues, both in native code and in Java programming language code (Java code). The Java Wireless Client software takes these issues into account. Those who want to integrate existing libraries, which might not have been written with multitasking in mind, also need to be aware of the issues so that they can add the source code correctly.

Code integrated into the Java Wireless Client software must be *multitask safe*. That is, it must maintain the independence of one task from another. Because tasks run in a single operating system process, the OS process can run native code for any task. To keep tasks from interfering with each other and with each other's data, the code must be made aware of the task on whose behalf it is being called and possibly allocating resources. When the task context is established, the code is considered multitask safe.

This chapter points out issues that might arise in ensuring native code is multitask safe. It also provides techniques for making the code multitasking safe in different situations.

The following list summarizes the multitasking safety issues to consider when you update or add native code for your port:

- Multitask safety and multithread safety
- Native global or static data
- Singletons

Multitask Safety and Multithread Safety

Many systems today are *multithreaded*, which requires code that runs in these systems to be *thread safe* or *multithread safe*. For example, POSIX Threads (Pthreads) enables multiple native threads to run in the same OS process. Each native thread has access to native memory, so any data structures in native memory must be protected from concurrent access. This is typically accomplished through the use of locks to provide mutual exclusion.

Similarly, multiple Java platform threads can run in the same Java virtual machine. Each Java platform thread has access to the objects in the Java virtual machine, so these objects must also be protected from concurrent access. Java code typically accomplishes this through the use of `synchronized` code blocks or higher-level constructs.

In the Java Wireless Client software, each task has one or more threads. Each of these threads has concurrent access to the objects in that task, and so the same multithread safety issues occur in Java Wireless Client software as in conventional Java virtual machines. However, a thread in one task has access only to objects within its task, and it has no access to objects in any other task. The multitasking nature of Java Wireless Client software thus has no impact on the multithreaded safety of applications.

All of the tasks in Java Wireless Client software run in a single OS process, and therefore they all have access to the same native memory and data structures. Threads from different tasks are scheduled arbitrarily. Therefore, native methods that read or update native data structures must be prepared to deal with operations from different tasks being interleaved in an arbitrary order. Furthermore, because different tasks are generally running different applications, these different applications are likely to place quite different demands on the underlying native system.

For example, certain native functions (such as file storage) must be maintained on a per-application basis. In a single-tasking system, only one application is running, and so all file access is on behalf of that one application. In a multitasking system, several applications are running, and so the file access code can no longer assume that just one application exists. Instead, it must be aware of the possibility of multiple applications, so that one application doesn't accidentally operate on another application's files. Code that is aware of this possibility is *multitask safe*.

While operations from different tasks can occur in an arbitrary order, no possibility exists of actual concurrency among native methods. One native thread exists in the Java Wireless Client software. When it is running a native method, no possibility exists of a context switch that causes another native method to be run at the same time. Each native method runs to completion and returns before the next native method can begin. For this reason, every native method is a critical section, and it is usually not necessary to perform any OS-level locking (such as with Pthreads mutexes) in native methods. It is only necessary to use OS-level locking if other native threads are active in the system while the Java Wireless Client software is running.

Thus, while native code that runs within Java Wireless Client software must be *multitask safe*, it need not be *multithread safe*.

Global and Static Data

In a single-tasking system, it is common for native code to use global or static variables. This works because only one application is running at a time. Global data is implicitly associated with the currently running application. In a multi-tasking system, the multiple applications likely conflict over global data. Therefore, to make your code multitask safe, you might need to rearrange your native data to be allocated on a per-task basis instead of globally.

To do this, you must allocate native data dynamically instead of statically. In addition, you need to associate each piece of native data with a particular task.

To manage native data on a per-task basis, store a pointer to the native data into an `int` field in a Java object. Associating the native resource with a single Java object implicitly provides multitasking safety because each Java object resides in exactly one task. It is also a good object-oriented approach.

Create a class to represent the native resource. Give the class a private field to hold the pointer to the resource, for example:

```
private int nativePtr;
```

Maintain the following invariants:

- A value of zero means a NULL pointer

- A nonzero value means a valid native pointer

In native code, when you allocate memory, use KNI field access to store the pointer in the private field. When you free the native memory, use KNI field access to store 0 in the field. Using the K Native Interface (KNI) field access avoids race conditions.

Have operations that use the native pointer use KNI field access for consistency. Have those operations check for a `NULL` value before they use the pointer, and throw an appropriate exception if the field does not have a pointer. An appropriate exception is a `NullPointerException`.

Singletons

CLDC HotSpot Implementation 2.2 isolates the logical virtual machines, but there is one situation to consider if you add or change Java code: Singletons. Singleton classes are often used when only one instance of a class should exist. Sometimes the class itself is used as a singleton, and no instance of it is ever created. The way to handle singleton depends on whether you mean it to be used on a per-application (that is, per-task) basis or whether you mean it to be used by the entire system.

For example, in Java Wireless Client software, an event queue is a per-task singleton, because each task has its own event queue. Per-task singletons are not a problem as they are handled by the VM. Static state is automatically replicated on a per-task basis. On the other hand, the singleton that holds the foreground display is a system-global singleton. Only one `Display` can be in the foreground in the entire system. All other display instances must be in the background. System-global singletons require additional work.

To handle a system-global singleton, consider either maintaining the singleton's state in a single task (such as the AMS task) and communicate the updates through messages, or migrating key pieces of information into native memory, with access to them arbitrated through native methods.

If you maintain the singleton's state in a single task and update it using events, be aware that events are asynchronous messages. Organize the updates so that operations are tolerant of being executed out of order. Although messages are generally processed in the order received, messages from different tasks might not be processed in the order they are sent. Also organize the updates so that requestors can proceed asynchronously. Asynchronous messages do not have any acknowledgement by default, so the sender cannot know when the message is processed or whether it is processed successfully.

You might find that you cannot organize the singleton's maintenance in this way, because its state must be updated synchronously and atomically. Maintaining the foreground state is an example of this type of singleton. In this case, migrate a key piece of state into native memory and handle updates through calls to native methods.

Multitasking Safety Example

Consider a simple though somewhat contrived example of controlling an external device, a microwave oven. Assume that this device has a native API defined in the header file shown in [CODE EXAMPLE 2-1](#).

CODE EXAMPLE 2-1 Native API for a Microwave Oven

```
/* mw.h */

/*
 * Status codes passed to cook callback. The cooking might have been
 * interrupted, for instance, if the user pressed the STOP button or
 * opened the oven door.
 */
typedef enum {
    MW_DONE,          /* cooking finished normally */
    MW_INTERRUPTED,  /* cooking interrupted */
} MWSTATUS;

/* Callback for the cook operation. */
typedef void (*MWCB)(MWSTATUS status, void *context);

/* Initializes the microwave oven. Must be called exactly once. */
extern void mw_init(void);

/* Sets the cook time for the next cook operation, in seconds. */
extern void mw_settime(int nsec);

/*
 * Sets the cook power for the next cook operation, an integer
 * in the range [1-100].
 */
extern void mw_setpower(int power);

/*
 * Initiates the cooking operation. When the cooking finishes,
 * the callback is called with the status code. The context pointer
 * is passed to the callback for its own use, unmodified by the
 * library.
 */
```

```
extern void mw_cook(MWCB callback, void *context);
```

Typical usage of this API is shown in [CODE EXAMPLE 2-2](#).

CODE EXAMPLE 2-2 Typical usage of the microwave

```
void cb_popcorn(MWSTATUS status, void *context) {
    if (status == MW_INTERRUPTED) {
        /* tell the user that the popcorn isn't finished */
    } else {
        ...
    }
}

void cook_popcorn() {
    mw_init();
    mw_settime(180);
    mw_setpower(100);
    mw_cook(cb_popcorn, NULL);
}
```

For the sake of discussion, assume that this native library is extremely simplistic. If the `mw_init()` function is called twice, the system breaks. Or, if `mw_cook()` is called while the microwave is cooking, the system breaks.

A straightforward binding of Java programming language APIs (Java APIs) for the microwave library might be as shown in [CODE EXAMPLE 2-3](#).

CODE EXAMPLE 2-3 Simple Java API for the Microwave Oven

```
package javax.microwave.oven;

public class Microwave {
    public static native void init();
    public static native void setTime(int nsecs);
    public static native void setPower(int power);
    public static native int cook();
    private Microwave() { } /* prevent instance creation */
}
```

The implementation of these native methods is straightforward and is not shown here. See the *K Native Interface (KNI) Specification, Version 1.0* for further information about writing native methods.

To make this a nice Java API, assume that instead of being callback-based, the `cook()` method blocks the calling thread until the operation completes or is interrupted. This can be accomplished using `SNI_BlockThread`, and the native callback can set a flag to cause `JVMSPi_CheckEvents` to call `SNI_UnblockThread`. See the *CLDC HotSpot Implementation Porting Guide* for more information.

Multithread Safety

The most obvious problem with the interface defined in [CODE EXAMPLE 2-3](#) is that it is not thread-safe. A single Java platform thread (Java thread) calling the APIs can certainly use it effectively, but if another thread attempts to use the API, things almost certainly break. For example, one thread might call `setTime()`. Another thread might be scheduled and issue a call to `setTime()` with a different value. When the first thread calls `cook()`, the cook time used is actually the cook time set by the second thread. When the second thread calls `cook()`, the cooking operation initiated by the first thread might still be going on, which gives rise to an error.

A reasonable way to make this interface thread safe is to introduce mutual exclusion, so that one thread can be assured that no other threads are interfering with its operation. This ensures that intermediate state (such as cook time) is not altered between the setup calls and the `cook()` call. It also ensures that only one `cook()` operation can be processed at once, a restriction imposed by the underlying native API.

Because exclusion is required around multiple calls to this interface, making the methods synchronized is insufficient. Therefore, introduce a `lock()` and `unlock()` protocol that clients are required to use around their calls to other methods. Each native method is wrapped with a Java method that checks for the proper lock state before proceeding to call the native method.

CODE EXAMPLE 2-4 Introducing a Locking Mechanism for Thread Safety

```
public class Microwave {
    private static boolean initialized = false;
    private static Thread owner = null;

    public static synchronized void lock()
        throws InterruptedException {
        while (owner != null) {
            wait();
        }

        owner = Thread.currentThread();

        if (!initialized) {
            init();
            initialized = true;
        }
    }

    public static synchronized unlock() {
        owner = null;
        notifyAll();
    }

    public static synchronized setPower(int power) {
```

```

        if (owner != Thread.currentThread()) {
            throw new IllegalStateException();
        }
        n_setPower(power);
    }

    public static synchronized setTime(int nsecs) {
        if (owner != Thread.currentThread()) {
            throw new IllegalStateException();
        }
        n_setTime(nsecs);
    }

    public static synchronized int cook() {
        if (owner != Thread.currentThread()) {
            throw new IllegalStateException();
        }
        return n_cook();
    }

    private static native void init();
    private static native void n_setTime(int nsecs);
    private static native void n_setPower(int power);
    private static native int n_cook();
    private Microwave() { } /* prevent instance creation */
}

```

Expected usage of this new API from Java platform programs is shown in [CODE EXAMPLE 2-5](#).

CODE EXAMPLE 2-5 Using the Locking Mechanism

```

Microwave.lock();
try {
    Microwave.setTime(180);
    Microwave.setPower(100);
    if (Microwave.cook() == ...) {
        ...
    } else {
        ...
    }
} finally {
    Microwave.unlock();
}

```

This API is now multithread safe. However, it is not multitask safe. The reason is that the thread safety properties are achieved using mechanisms that belong to the `Microwave` class. These include static variables (`initialized`, `owner`) of the `Microwave` class. Thread synchronization and wait and notify operations use the monitor lock owned by the class. All of these mechanisms are visible to the threads of a single task, and thus they provide safety among the multiple threads of a single task.

However, in a multitasking environment, the class static variables and monitor lock are *replicated* in each task. Thus, if a thread in task A is performing an operation on the `Microwave` class, the `initialized` variable is `true` and the `owner` variable contains a reference to the thread performing the operation. However, in task B, the `initialized` variable still has the value `false` and the `owner` variable is `null`. If a thread in task B attempts an operation, it takes the lock (even though a thread in task A “owns” the lock in task A), it calls `init()` for the first time (from its point of view). This results in a second call to the native `mw_init()` function, which is an error. The thread in task B then calls some of the setup functions and then calls `cook()`, possibly before the cook operation initiated from task A completes, which is another error.

Multitask Safety

Use the microwave library and the Java programming language interface to illustrate how a library must be changed to be multitask safe.

The first and simplest case is class data (static variables) that is semantically global but is replicated into each task. Sometimes the singleton pattern is used in this fashion. Inspect each singleton Java class to determine whether it needs to be a singleton only from the point of view of Java threads that have access to it, or whether it needs to be a system-wide singleton. In the former case, ordinary singleton classes are sufficient. Even though they are replicated within each task, they can be *per-task singletons* because each Java thread sees only one instance of the singleton class. In the latter case, for *global singletons*, the singleton characteristic must apply to the entire system. The typical way to accomplish this is to migrate the relevant data from Java code into native code.

In our example, the global singleton data is a `static boolean` variable `initialized` that indicates whether the microwave library has been initialized. This can simply be migrated to an `int` variable in native code. Once this variable is in native code, it is visible to all tasks using native methods. You can write native methods to set and get this value and take action as appropriate. This technique is illustrated in [CODE EXAMPLE 2-6](#).

CODE EXAMPLE 2-6 Migrating the Initialization Variable to Native Code (Doesn't Work)

```
// Microwave.java

static native boolean getInitState();
```

```

static native void setInitState(boolean init);

public static synchronized void lock()
    throws InterruptedException {
    ...
    if (!getInitState()) {

        init();
        setInitState(true);
    }
}

```

Unfortunately, the code in [CODE EXAMPLE 2-6](#) does not work. The reason is that threads from different tasks can be switched at any time. Suppose that a thread in task A calls `getInitState()`, which returns `false`. The thread decides that the microwave library needs to be initialized, and it starts to execute the code in the `if` block. Now suppose the system switches to run a thread in task B that is also about to run this code. Task B's thread also calls `getInitState()`, which also returns `false`, and so this thread also decides to execute the code in the `if` block. This results in two calls to the native `mw_init()` function, which is an error.

From the Java programming language point of view, the tasks are completely isolated from each other, and therefore threads from different tasks cannot interact. This means that there is no Java programming language construct that can provide mutual exclusion between threads that are in different tasks. Therefore, some other kind of mechanism at the native method level is necessary, because native methods operate outside the constraints of the Java programming language.

Instead of the flag test and set logic being in Java code, it must also be migrated to native code, along with the flag itself. In this case, the Java code can call `init()` every time and it can rely on the logic in native code to ensure that the `mw_init()` function is called only once, as shown in [CODE EXAMPLE 2-7](#).

CODE EXAMPLE 2-7 Migrating Initialization to Native Code

```

/* microwave.c */

static int initialized = 0;

KNIEXPORT KNI_RETURNTYPE_VOID
Java_javax_microwave_oven_Microwave_init(void)
{
    if (initialized == 0) {
        initialized = 1;
        mw_init();
    }
    KNI_ReturnVoid();
}

```

Note that no mutual exclusion is necessary in native methods. The CLDC HotSpot Implementation has a single thread that runs all Java threads and all native methods. This thread can run at most one native method at a time. The system cannot context-switch to another Java thread while it is in the midst of a native method. Therefore, in the CLDC HotSpot Implementation, all native methods are essentially critical sections. This makes it possible to write native code without many concurrency considerations.

Note – Your system might employ multiple native threads. If this is the case, you might need to employ native-level mutual exclusion facilities such as Pthreads mutexes.

While it is convenient to treat each native method as a critical section, one must ensure that data access and updates are done within a single native method. If any logic about updating native state is performed by Java code, such as mixing Java and native methods, or by making multiple native method calls, this creates race conditions. Running any Java code provides an opportunity for Java threads to be context switched, and conditions established by code before the context switch might be invalid after the context switch. The example from [CODE EXAMPLE 2-6](#) suffers from exactly this problem.

Establishing Per-Task Context

Migrating Java platform data to native can work well for global singletons, but it doesn't work for other situations. In the case of the cook operation, the data is built using a sequence of setup calls to `mw_settime()` and `mw_setpower()`, followed by a call to `mw_cook()` that initiates the operation using the parameters previously set up. The library implicitly stores this information in its internal static data. Thus, it's effectively global.

But no global data exists because the data really belongs to the thread that's setting up to initiate the cooking operation. In this case, the data needs to be migrated upward to be closer to the calling thread. The setup data doesn't need to be sent to the microwave library until immediately prior to the call to the `mw_cook()` function.

The `lock()` and `unlock()` static methods were added to the `Microwave` class to protect the context that was being built implicitly in library static data by the `setTime()` and `setPower()` methods. This locking protocol effectively provides mutual exclusion around library static data. This works in a single-tasking environment, but it fails in a multitasking environment, as described earlier.

If the setup methods simply set values into Java variables, and these values are set into the library immediately before the call to `mw_cook()`, a locking protocol that makes the setup calls and the cook call atomic is no longer necessary. This technique doesn't work in a multitasking environment anyway. Therefore, it must be removed. The setup context logically belongs to the calling thread. Multiple threads could lock, wait, and notify over static data, but this too is no longer necessary if each thread is permitted to operate on its own instance of a `Microwave` object.

After making these changes, the resulting Java code is shown in [CODE EXAMPLE 2-8](#).

CODE EXAMPLE 2-8 Keeping State in Java Code

```
public class Microwave {

    int nsecs;
    int power;
    private int nativePtr;

    public void setTime(int nsecs) {
        // error checking elided for brevity
        this.nsecs = nsecs;
    }

    public void setPower(int power) {
        // error checking elided for brevity
        this.power = power;
    }

    public int cook() {
        init();
        return n_cook();
    }

    private native void init();
    private native void n_cook();
    public Microwave() { } /* allow instance creation */
    private native void finalize();
}
```

The responsibility of the `n_cook()` native method is now much greater. It must block the calling thread until any other cooking operation has completed. It must retrieve the field values from the object and set them into the library. Finally, it must block until this cooking operation has completed. Note that to avoid race conditions, all of this must be performed by a single native method.

For the sake of simplicity, ignore the situation where another cooking operation might already be in progress, and ignore the logic for block and unblocking the calling thread.

Note also the use of a technique for allocating a native context object and storing its pointer in a Java object field. All data used in this native method is relative to the Java object on which it's called. This automatically provides multitask safety. An object belongs exclusively to a single task, so operations that are interleaved or that occur simultaneously in different tasks cannot interfere with each other.

CODE EXAMPLE 2-9 Implementing the Native `n_cook()` Method

```
/* microwave.c */

KNIEXPORT KNI_RETURNTYPE_INT
Java_javax_microwave_oven_Microwave_n_cook(void)
{
    int nsecs;
    int power;
    MWSTATUS *statusp;
    int retval;

    jfieldID nsecsFieldID;
    jfieldID powerFieldID;
    jfieldID nativePtrFieldID;

    KNI_StartHandles(2);
    KNI_DeclareHandle(thisObj);
    KNI_DeclareHandle(microwaveClass);

    KNI_GetThisPointer(thisObj);
    KNI_GetObjectClass(thisObj, microwaveClass);

    nsecsFieldID = KNI_GetFieldID(microwaveClass, "nsecs", "I");
    powerFieldID = KNI_GetFieldID(microwaveClass, "power", "I");
    nativePtrFieldID =
        KNI_GetFieldID(microwaveClass, "nativePtr", "I");

    if (/* this is the first invocation */) {
        statusp = (MWSTATUS *)malloc(sizeof(MWSTATUS));

        nsecs = KNI_GetIntField(thisObj, nsecsFieldID);
        power = KNI_GetIntField(thisObj, powerFieldID);
        KNI_SetIntField(thisObj, nativePtrFieldID, (jint)statusp);

        mw_settime(nsecs);
        mw_setpower(power);
        mw_cook(callback, statusp);
        SNI_BlockThread();
        retval = -1; /* return value ignored if caller is blocked */
    } else {
        /* this is a reinvocation after having been awakened */
        statusp =
            (MWSTATUS *)KNI_GetIntField(thisObj, nativePtrFieldID);
    }
}
```

```

        retval = (int)(*statusp);
        KNI_SetIntField(thisObj, nativePtrFieldID, 0);
        free(statusp);
    }

    KNI_EndHandles();
    KNI_ReturnInt(retval);
}

void
callback(MWstatus status, void *context)
{
    MWSTATUS *statusp = (MWSTATUS *)context;
    *statusp = status;

    /*
     * Search the list of block threads and find the right
     * one to unblock with SNI_UnblockThread(). See the CLDC HI
     * Porting Guide for further information.
     */
}

KNIEXPORT KNI_RETURNTYPE_INT
Java_javax_microwave_oven_Microwave_finalize(void)
{
    jfieldID nativePtrFieldID;
    MWSTATUS *statusp;

    KNI_StartHandles(2);
    KNI_DeclareHandle(thisObj);
    KNI_DeclareHandle(microwaveClass);

    KNI_GetThisPointer(thisObj);
    KNI_GetObjectClass(thisObj, microwaveClass);
    nativePtrFieldID =
        KNI_GetFieldID(microwaveClass, "nativePtr", "I");
    statusp = (MWSTATUS *)KNI_GetIntField(thisObj, nativePtrFieldID);

    if (statusp != NULL) {
        free(statusp);
        KNI_SetIntField(thisObj, nativePtrFieldID, 0);
    }
}

```

These examples show how multitask safety can be achieved by judicious migration of data from Java code into native code (for global singletons) and from native code into Java code (for context-specific data). Some libraries have explicit context objects, with all operations relative to that context object.

For cases like that, a reliable technique is to put the native context pointer into a `nativePtr int` field in the Java object. In other cases, such as the microwave library, the library maintains implicit context in its own static data. For these cases, it is necessary for the code to create its own context. This context can be stored as fields in the Java object itself, or a data structure can be allocated on the native heap and a pointer to this structure placed into a `nativePtr int` field. Ensuring that the context for all operations is relative to a Java object automatically provides multitask safety.

Managing Native Resources

The typical device is constrained in the amount of resources, such as memory or sockets, that it has available. When the Java Wireless Client software is running on a device, it is often provided with a fixed set of resources that it cannot exceed. In a single-tasking environment, a single MIDlet has access to all of the available resources. However, in a multitasking environment, multiple MIDlets might have to compete among themselves for this fixed set of resources.

For example, assume that the Java Wireless Client software has 600 kilobytes of heap memory, and that MIDlet A requires 400 kilobytes and MIDlet B requires 300 kilobytes. The user can run either MIDlet A alone or MIDlet B alone, and each works fine. However, if MIDlet A is running and the user starts MIDlet B, MIDlet B might receive an out-of-memory error.

MIDlets sometimes allocate only some of the memory they need at startup, but allocate more memory when a certain operation is performed. For example, suppose MIDlet A is running and consumes 350 kilobytes, and the user starts MIDlet B. At startup, MIDlet B might consume only 100 kilobytes and might appear to work fine. However, when the user attempts a particular operation using MIDlet B, it might allocate the additional 200 kilobytes, causing an out-of-memory error during its operation. Alternatively, if the user switches to MIDlet A and MIDlet A allocates more memory, then MIDlet A might receive the out-of-memory error.

This is a difficult problem because each MIDlet might test successfully with the Java Wireless Client software and work fine when run individually. However, when run at the same time, the MIDlets could appear to be unstable, receiving out-of-memory errors at unpredictable times.

The Java Wireless Client software solves this problem by providing a set of resource management mechanisms that can be used to control how resources are allocated. The Java Wireless Client software also provides a set of resource management policies that determine how the system behaves under certain conditions. These policies can be customized to tailor the behavior of the system for a particular deployment. Two example policies are also provided as a starting point for customization. Finally, the implementation of resource mechanisms and policies resides with the resource manager.

Resource Management Mechanisms

Java Wireless Client software has three resource management mechanisms: reservation, limit, and revocation.

Reservation

The reservation mechanism sets aside a certain amount of a resource for a MIDlet and keeps it available for that MIDlet and that MIDlet alone, regardless of whether the MIDlet is using it at the moment. The reserved resource is never granted to another MIDlet. If another MIDlet attempts to allocate the resource, it might fail, even if the first MIDlet is not using all of its reservation.

Assume the heap memory available is 600 kilobytes, MIDlet A has a reservation of 400 kilobytes, but is currently using 250 kilobytes. 350 kilobytes of heap is unused. However, only 200 kilobytes is actually available, because 150 kilobytes of the unused heap is still reserved for MIDlet A. If MIDlet B starts and attempts to allocate 300 kilobytes of heap, it receives an out-of-memory error.

The reservation mechanism improves predictability and helps to prevent data corruption.

MIDlets allocate and free resources (particularly memory) throughout their operation. During a resource shortage, any allocation attempt might fail. These failures might occur at an arbitrary time, even in the midst of an operation. If MIDlet is provided with a reservation and is designed never to exceed this reservation, then its allocation attempts will never fail. Instead, the resources for the reservation are allocated at the time the MIDlet is started.

If there is a resource shortage, the failure will occur at startup time. Once the MIDlet has been started, it is guaranteed not to fail because of a resource shortage. This improves predictability because resource allocation failure occur only at startup time, not at arbitrary times while the MIDlet is running.

Many MIDlets are not designed to deal with failure in the midst of an operation. If such a MIDlet is updating one of its data structures when an allocation failure occurs, the data structure might end up in an inconsistent or corrupted state. Providing MIDlets with a resource reservation will prevent failures from occurring at arbitrary times, thus reducing the possibility of data structure corruption.

Limit

The limit mechanism allows a MIDlet to allocate up to a certain amount of some resource, but no more. If the MIDlet attempts to allocate more resources than its limit, the attempt fails, even if resources are actually available on the system. Conversely, fewer actual resources might be available than a MIDlet's limit might indicate. A MIDlet might attempt to allocate resources beneath its limit, but the attempt still fails if insufficient actual resources are available on the system.

For example, assume that 600 kilobytes total heap is available and that MIDlet A is using 400 kilobytes. MIDlet B has a limit of 300 kilobytes of heap. It receives an out-of-memory error after allocating 200 kilobytes, because that's all the heap available, even though it is under its limit. Suppose, however, that MIDlet B has a limit of 150 kilobytes. It receives an error after allocating 150 kilobytes, even though 50 kilobytes of heap is still available.

The limit mechanism is useful to prevent MIDlets from attempting to use all available resources and thereby disrupting the operation of the system.

Reservations and limits work in concert to control competition among MIDlets that are using a pool of resources, such as heap memory or network sockets. It is typical for a running MIDlet to have both a reservation and a limit in effect at the same time. Note that it only makes sense for the limit to be greater than or equal to the reservation. A limit differs from a reservation in that a reservation guarantees that the reserved amount of the resource is always available, but it doesn't prevent the MIDlet from attempting to exceed the reservation amount. A limit is quite complementary: It prevents the MIDlet from using more than a some amount of a resource, but it doesn't guarantee that the amount is available.

When resources are explicitly released by a MIDlet, they might or might not become available to other MIDlets. It depends on the reservation policy for the resources.

Revocation

The revocation mechanism lets the system take a resource from one MIDlet to give the resource to another MIDlet. This second MIDlet is sometimes said to have *preempted* the resource from the first MIDlet. Resources are revoked without receiving any request from the MIDlet. The MIDlet might or might not be informed explicitly that the revocation is taking place, depending upon the type of resource being revoked.

After allocation by a MIDlet, many resources are deallocated explicitly by the MIDlet. For example, opening a socket allocates it, and closing the socket deallocates it, after which it can be reused by another MIDlet. If a socket is revoked, the underlying native resources are reclaimed, but the Java `Socket` object still exists. In this case, any subsequent operations on this `Socket` object results in an I/O error. The Java Wireless Client software doesn't revoke sockets.

Revocation is not always so drastic. Sometimes it can occur almost transparently to the application. The device's screen is a resource. When a MIDlet is put into the background, its access to the screen is revoked so that another MIDlet can paint to the screen. In this case, the only effect on the MIDlet might be that its `hideNotify()` method is called and that it no longer receives calls to the `paint()` method.

Another resource that can be revoked transparently is the CPU. When the CPU is revoked from a MIDlet, its threads simply stop running. When the MIDlet regains the CPU, its threads pick up where they left off.

Default Resource Allocation Policies

The Java Wireless Client software provides two preconfigured resource allocation policies: open and fixed. The open policy allows open competition among applications for resources. The fixed policy provides a fixed amount of resources for each application. Under the fixed policy, the reservation and limit for each resource are set to the same value. Setting the reservation and limit to the same value effectively creates a fixed partitioning of resources.

The policies have tradeoffs. Neither is clearly better than the other. Using the open resource policy might lead to unpredictable behavior. An application could at any time receive an error that a resource is not available. Users might find this behavior confusing. Under the fixed policy, if the amount of available resources is less than the reservation amount for a new MIDlet, the system refuses to start any more MIDlets. This occurs even if the actual amount of unused resources is sufficient for the new MIDlet, meaning valuable resources might go unused. However, this

behavior might be less confusing to the user than the failures that could happen with the open resource policy, because these failures are predictable. They always happen when the user tries to start the application.

The default policy is open for competition. To use a fixed-partition resource policy, you must set a build-time flag `USE_FIXED=true`. See the *Sun Java Wireless Client Build Guide* for more information.

Customization of Resource Allocation Policies

Most implementations need to tailor resource allocation policies to each resource instead of specifying a blanket open or fixed policy for all resources. For example, on a particular platform, heap memory might be relatively plentiful, and so choosing a fixed policy for memory might be appropriate to gain predictability, possibly at the expense of some waste. However, another resource, such as network sockets, might be too scarce to tolerate any waste, even at the cost of some unpredictability. An open policy is more appropriate for these resources.

It is also possible to customize the reservations and limits to be somewhere between the extremes specified by the fixed and open policies. For example, a particular implementation might want to guarantee 250 kilobytes of heap memory for each MIDlet, but it doesn't necessarily want to constrain a large MIDlet from consuming as much heap memory as it wants. In this case, specify a reservation of 250 kilobytes and specify no limit on heap memory.

The resource allocation policies are specified in XML files in the following directories in the source tree:

```
src/configuration/configuration_xml/platform
```

These directories contain several XML files that specify various constants used in for all aspects of the system. For more information about these files and how to modify constants, see the Managing Properties and Constants chapter of the *Porting Guide*. The two XML files that specify the resource policies are `constants_fixed.xml` and `constants_open.xml`. The `constants_fixed.xml` file is used if the build flag `USE_FIXED=true`, otherwise the `constants_open.xml` file is used.

The best way to modify the resource allocation policy is to ensure that `USE_FIXED=false` (which is the default) and to modify the `constants_open.xml` file. Some of the constants in this file are described in [TABLE 3-1](#). To specify no reservation or limit, use the value `-1`. For example, the default value of `SUITE_MEMORY_LIMIT` is `-1`, which sets no limit on the amount of memory that a MIDlet suite can allocate.

Each resource typically has five constants that define the policy. A pair of constants defines the reservation and the limit for each MIDlet suite. These constants have the suffixes `SUITE_RESERVED` and `SUITE_LIMIT`, respectively. Another pair of constants defines the reservation and limit for the AMS task. These constants have the suffixes `AMS_RESERVED` and `AMS_LIMIT`. Finally, the fifth constant specifies the global limit for the entire system. The sum of all allocations by all MIDlet suites and the AMS cannot exceed this limit. This constant has the suffix `GLOBAL_LIMIT`.

For certain resources, it is important to choose the AMS reservation carefully. For example, the AMS needs to open various MIDlet files in order to change application settings. If the AMS open file reservation is not set high enough, and running MIDlets have consumed all available file handles, the normal operation of the AMS will be disrupted.

TABLE 3-1 Constant Definitions for the Resource Management Policy

Name	Description
<code>MAX_ISOLATES</code>	Maximum number of tasks (isolates) allowed.
<code>SUITE_MEMORY_RESERVED</code>	Heap memory reserved for each MIDlet suite, in kilobytes
<code>SUITE_MEMORY_LIMIT</code>	Heap memory limit for each MIDlet suite, in kilobytes
<code>TCP_CLI_GLOBAL_LIMIT</code>	TCP client socket limit for the entire system
<code>TCP_CLI_AMS_RESERVED</code>	TCP client socket reservation for the AMS
<code>TCP_CLI_AMS_LIMIT</code>	TCP client socket limit for the AMS
<code>TCP_CLI_SUITE_RESERVED</code>	TCP client socket reservation for each MIDlet suite
<code>TCP_CLI_SUITE_LIMIT</code>	TCP client socket limit for each MIDlet suite

Five constants determine the policy for TCP client sockets: the global limit, the reservation and limit for the AMS, and the reservation and limit for each MIDlet suite. This pattern of five constants is repeated for several different resources, including the following:

- TCP server sockets
- UDP datagram sockets
- Open file handles
- Audio channels
- Mutable images
- Immutable images

For each of these resource types, you can specify independently the global limit and the reservations and limits for the AMS and for each MIDlet suite.

Other Multitasking Issues

Multitasking raises some issues related to the behavior of the Java Wireless Client software. This chapter describes the issues, how they are handled by the Java Wireless Client software, and possible alternatives.

Most of the issues discussed in this chapter relate to the implementation of the Java platform AMS. The same issues must be addressed by the Native AMS.

Switching the Foreground MIDlet

In a system that cannot run concurrent MIDlets, each MIDlet effectively runs alone and it potentially has access to all the resources available on the system. The entire runtime environment is shut down when the MIDlet exits. Multitasking environments work differently.

With multitasking, a MIDlet is not alone on the system. The MIDlet runs in its own environment (its *task*), but at most one MIDlet can have access to the device's display and to the device's input mechanisms, such as keypad and pointer events. No MIDlets have access to the device's display and input mechanisms if the user is interacting with a native application. Managing which MIDlet has access to the device display and input mechanisms necessitated the introduction of the *foreground* and *background* states:

- A foreground MIDlet can draw on the screen and receive keystrokes (and pointer events, if the device has pointer input).
- A background MIDlet cannot draw to the device display, and the system ignores any graphics operations it emits. A background MIDlet cannot receive any keypad or pointer events.

Default Policy

The Java Wireless Client software supplies a default policy for how a MIDlet gains the foreground. It enables the user to switch to an application list screen at any time by pressing a *hot key*. By default, the hot key is the Home key found on many platforms. The application list screen shows a list of all running MIDlets. When the user brings up the application list, the MIDlet that had been in the foreground is moved into the background. The user scrolls through the application list and chooses the MIDlet to bring to the foreground.

Native code can request the application list to be shown by sending a `SELECT_FOREGROUND_EVENT` into the system. See the code in the file `win32app_export.c` for an example of how this event is generated in response to the pressing of the Home key. Use the same technique of sending a `SELECT_FOREGROUND_EVENT` to show the application list in response to some other external event.

Alternative Policies and Their Implementations

Policies for switching foreground applications are highly dependent on a device's existing user interface. If your device already enables a user to switch between native applications in a particular way, use the same or similar policies for the Java Wireless Client software. Users expect consumer products, such as small devices, to be predictable, easy to learn, and easy to use. Making the policies of your Java Wireless Client software as similar as possible to the native policies users already know and use speeds acceptance of the new functionality. See the *MIDP 2.0 Style Guide* (Addison-Wesley, 2003) for more information.

Scheduling the CPU

The CPU is a shared resource that requires separate policy decisions. The foreground application has exclusive access to the display and the input mechanisms, but you can have a policy that enables it to share the CPU with the background applications. Your policy depends at least in part on the capabilities of your device.

Default CPU Scheduling Policy

By default, the Java Wireless Client software permits background applications to have some CPU time. A background application can also interrupt the foreground application under some circumstances.

Specifically, the CPU scheduling policy is fair share, which gives a higher proportion of the CPU to the foreground application. Background applications can still run, but their threads are scheduled less frequently.

This policy is implemented in the `setForegroundMIDlet()` method of the `MIDletProxyList` class. The MIDlet being put into the background is given minimal priority by calling the `minPriority()` method. The MIDlet being brought into the foreground is given normal priority by calling the `normalPriority()` method.

Alternative Policies and Their Implementations

One alternative to giving limited CPU access to background applications is to give the CPU exclusively to the foreground application. This might provide better interactive performance on systems with slow CPUs. It can also be good for some applications, such as a game that displays graphics continually and requires constant user input. This type of application probably needs to be frozen when in background. The user loses the game if the game is placed into the background and still allowed to use the CPU.

On the other hand, a device that freezes background MIDlets cannot run service or daemon applications that do not interact with the user at all. An example of such an application is a MIDlet that listens on the network for information about critical system software updates.

Another alternative to the default fair-use policy is an open-for-competition policy, that gives all applications equal priority, whether they are in the foreground or background.

To implement an open-for-competition policy for the CPU, modify the contents of the `minPriority()` and `normalPriority()` methods of the `MIDletProxyUtils` class so that they do nothing, instead of changing the tasks' priorities or suspending or resuming them.

As you consider CPU policies, keep in mind that they interact with other policies for background MIDlets. For example, the policy interacts with your decisions on whether a background application is informed of its state change through a call to the `pauseApp` method, or whether it has access to shared resources such as sound. A uniform, correct answer does not exist.

Interrupting the User

Although a good user interface permits a user to interrupt applications at any time, it does not do the same to the user. Instead, it interrupts the user only when necessary. Unnecessary feedback, confirmation messages, and error messages that require a user response are distracting. See the *MIDP 2.0 Style Guide* (Addison-Wesley, 2003) for more information.

Default User Notification Policies

The default policies of the Java Wireless Client software interrupt the user only when necessary. For example, the system does not allow a background application to interrupt the user and regain the foreground of its own accord. If a background application requests the display by trying to set the current screen with a screen other than an alert, the foreground application is not affected. If a background application requests the display by trying to set the current screen to an alert, the foreground application remains in place, but an icon is placed on the status bar to let the user know that a background application needs attention.

The system does not interrupt the user for notification of most failures. For example, if a MIDlet fails because of an internal problem, such as receiving an out-of-memory error, the application's task exits, but the other application tasks are unaffected. The system does not interrupt the user with a notification that the application has exited. It simply removes the failed application from its list of running applications.

The Push system of the Java Wireless Client software, too, only interrupts the user when a pushed message arrives for an application that is neither running, nor allowed to be automatically started. If the application's user preferences require the user to give permission to start an application in response to the arrival of a pushed message or the firing of a Push alarm, then the Push system asks the user for permission. The user can also set the application's user preferences to permit the system to automatically start the application in response to the Push system. In this case, the Push system does not ask the user for permission.

Glossary

- API** Application Programming Interface. A set of classes used by programmers to write applications, which provide standard methods and interfaces and eliminate the need for programmers to reinvent commonly used code.
- AMS** Application Management Service. The system functionality that completes tasks such as installing applications, updating applications, and switching foregrounds.
- Application list** The screen that lists all of the installed applications. The user gets to this screen by pressing the `Apps` soft key on the home screen. The application list uses text color to show which applications are running. It also provides a system menu that enables the user to perform application management tasks on the highlighted application.
- Background** An application state in which the application does not receive events from its input stream and its displayable is not rendered to the screen.
- CDC** Connected Device Configuration. A Java ME platform configuration for devices, it requires a minimum of 2 megabytes of memory and a network connection that is always on.
- CLDC** Connected Limited Device Configuration. A Java ME platform configuration for devices with less than 512 kilobytes of RAM and an intermittent (limited) network connection, it uses a stripped-down Java virtual machine called the KVM, as well as several minimalist Java platform APIs for application services.
- Configuration** Defines the minimum Java runtime environment (for example, the combination of a Java virtual machine and a core set of Java platform APIs) for a family of Java ME platform devices.
- Foreground** The application state in which the application is rendered to the device display and the input stream is passed to it.
- Foreground switching** Changing which application is in the foreground by shifting the focus from one application to another.
- GCF** Generic Connection Framework. A part of CLDC, it improves network connectivity for wireless devices.

Home screen The main screen of the application manager. This is the screen the user sees after they exit an application.

HTTP HyperText Transfer Protocol. The most commonly used Internet protocol, based on TCP/IP, which is used to fetch documents and other hypertext objects from remote hosts.

HTTPS Secure HyperText Transfer Protocol. A protocol for transferring encrypted hypertext data using Secure Socket Layer (SSL) technology.

JAD file Java Application Descriptor file. A file provided in a MIDlet suite that contains attributes used by application management software (AMS) to manage the MIDlet's life cycle, as well as other application-specific attributes used by the MIDlet suite itself.

JAR file Java Archive file. A platform-independent file format that aggregates many files into one. Multiple applications written in the Java programming language and their required components (.class files, images, sounds, and other resource files) can be bundled in a JAR file and provided as part of a MIDlet suite.

Java Community Process™ (JCP™) program

Java Community Process program. An open organization of international developers and licensees who develop and revise Java platform specifications, reference implementations, and technology compatibility kits using a formal submission and approval process.

Java ME platform Java Platform, Micro Edition. A group of specifications and technologies that pertain to running the Java platform on small devices, such as cell phones, pagers, PDAs, and set-top boxes. More specifically, the Java ME platform consists of a configuration (such as CLDC or CDC) and a profile (such as MIDP or Personal Basis Profile) tailored to a specific class of device.

Java Specification Request (JSR)

A proposal for developing new Java platform technology, which is reviewed, developed, and finalized into a formal specification by the JCP program.

Java Virtual Machine A software “execution engine” that safely and compatibly executes the byte codes in Java class files on a microprocessor.

KVM A Java virtual machine designed to run in small devices, such as cell phones and pagers. The CLDC configuration is designed to run in a KVM.

LCD Liquid Crystal Display. A common kind of screen display often used in small devices.

LCDUI Liquid Crystal Display User Interface. A user interface toolkit for interacting with LCD screens in small devices. More generally, a shorthand way of referring to the MIDP user interface APIs.

MIDlet An application written for MIDP.

MIDlet suite	A way of packaging one or more midlets for easy distribution and use. Each MIDlet suite contains a Java application descriptor file (.jad), which lists the class names and files names for each MIDlet, and a Java Archive file (.jar), which contains the class files and resource files for each MIDlet.
MIDP	Mobile Information Device Profile. A specification for a Java ME platform profile, running on top of a CLDC configuration, which provides APIs for application life cycle, user interface, networking, and persistent storage in small devices.
Obfuscation	A technique used to complicate code by making it harder to understand when it is de-compiled. Obfuscation makes it harder to reverse-engineer applications and therefore, steal them.
Optional Package	A set of Java ME platform APIs that provides additional functionality by extending the runtime capabilities of an existing configuration and profile.
PNG	Portable Network Graphics. An image format commonly used with MIDP that can be compressed, transmitted, and stored without losing image quality.
Preemption	Taking a resource, such as the foreground, from another application.
Preverification	Due to limited memory and processing power on small devices, the process of verifying Java technology classes is split into two parts. The first part is preverification and done off-device using the preverify tool. The second part, which is verification, is done on the device at runtime.
Profile	A set of APIs added to a configuration to support specific uses of a mobile device. Along with its underlying configuration, a profile defines a complete and self-contained application environment.
Provisioning	A mechanism for providing services, data, or both to a mobile device over a network.
Push Registry	The list of inbound connections, across which entities can push data, maintained by the Java Wireless Client software. Each item in the list contains the URL (protocol, host, and port) for the connection, the entity permitted to push data through the connection, and the application that receives the connection.
RMI	Remote Method Invocation. A feature of Java SE technology that enables Java technology objects running in one virtual machine to seamlessly invoke objects running in another virtual machine.
RMS	Record Management System. A simple record-oriented database that enables a MIDlet to persistently store information and retrieve it later. MIDlets can also use the RMS to share data.
SMS	Short Message Service. A protocol allowing transmission of short text-based messages over a wireless network.

- SOAP** Simple Object Access Protocol. An XML-based protocol that allows objects of any type to communicate in a distributed environment, it is most commonly used to develop web services.
- SSL** Secure Sockets Layer. A protocol for transmitting data over the Internet using encryption and authentication, including the use of digital certificates and both public and private keys.

Sun Java Device Test

- Suite** A set of Java programming language tests developed specifically for the wireless marketplace, providing targeted, standardized testing for CLDC and MIDP on small and handheld devices.
- SVM** Single Virtual Machine. A mode of the Java Wireless Client software, it can run only one MIDlet at a time.
- task** At the platform level, each separate application that runs within a single Java virtual machine is called a task. The API used to instantiate each task is a stripped-down version of the Isolate API defined in JSR 121. See the *CLDC HotSpot Implementation Architecture Guide* for more information.
- TCP/IP** Transmission Control Protocol/Internet Protocol. A fundamental Internet protocol that provides for reliable delivery of streams of data from one host to another.
- WAE** Wireless Application Environment. It provides an application framework for small devices, by leveraging other technologies such as WAP, WTP, and WSP.
- WAP** Wireless Application Protocol. A protocol for transmitting data between a server and a client (such as a cell phone) over a wireless network. WAP in the wireless world is analogous to HTTP in the World Wide Web.
- WMA** Wireless Messaging API. A set of classes for sending and receiving Short Message Service messages.
- (x) button** The button the user presses to end a task. On a real device this is the End key. On Windows it is the End key and sometimes the power key on the phone skin.

Index

C

CPU scheduling, 30

D

data

- global, 7
- static, 7

F

foreground MIDlet
switching, 29

L

limit, 23

M

mechanisms
 compared with policies, 3
multitask safety, 13
multitasking, 2
multitasking safety, 5, 6
multithread safety, 6, 11

N

native concurrency, 7
native resources, 21

P

policies
 compared with mechanisms, 3

R

reservation, 22
resource allocation policy
 customization, 25
 default, 24
revocation, 24

S

singletons, 8

T

task, 2
 context, 15
tasks, 2, 5

U

user notification, 32

V

virtual machine
 logical, 2

