JanosVM User's Manual and Tutorial
Version 0.6

Flux Research Group
School of Computing
University of Utah

http://www.cs.utah.edu/flux/

March 17, 2002

Contents

Chapter 1

Introduction

The JanosVM is a Virtual Machine that can execute Java byte codes in multiple isolated contexts. The goal of JanosVM is not to implement a full Java Operating System, rather, it aims to provide the components used to implement a Java OS [2]. For example, there is basic functionality for interacting with concurrently running processes, but no IPC system is directly exported to user code. Instead, an arbiter library can be developed which maps the JanosVM primitives to a safer API, for example, a message passing system. It is hoped that this approach affords more freedom for experimentation and specialization without requiring each developer to roll their own JVM.

The goal of this document is to describe the things you need to know in order to start using and developing for JanosVM, including:

- How to install JanosVM on your system (Chapter 3).
- How to invoke the JanosVM virtual machine (Chapter 5).
- A tutorial on how to use JanosVM and how to use it effectively (Chapter 4).
- Other JanosVM support tools including one for interacting with running JVMs (Chapter 6).

Chapter 2

Overview of JanosVM

This chapter presents a high level overview of the concepts and reasoning behind much of the JanosVM. You should be able to skip directly to the installation and tutorial if you're in a hurry, however, some things will be hard to fully understand without this background.

2.1 Motivation

As more and more computers become networked together their collective power to do both good and harm grows at an incredible rate. Presently, however, the ability for users to take advantage of all the available computing power requires some amount of trust and a somewhat homogenous computing environment. There exist individual solutions for each problem, operating systems with proper security and resource controls can prevent buggy or malicious code from running amok and high level language runtimes can function consistently in heterogeneous environments. However, simply composing these systems can result in a suboptimal whole that lacks expressivity, scaling ability, performance and countless other unknowns. Of course, these problems can already be seen in everyday environments as language runtimes soak up CPU and memory to execute redundant bootstrapping code for each executed program. Therefore, we have proposed and implemented, with great results, a number of systems which replicate and enhance many of the operating system's services in the Java language runtime. Now, with these results in hand we have taken the opportunity to further refine and explore the design space with JanosVM.

2.2 History

This section will concentrate on JanosVM's pedigree, a great deal of wonderful work and experience went into the development of the system. The three main systems we'll discuss are:

- KaffeOS - The last in a long line of Java OS designs using a monolithic kernel.
- Alta - A hierarchically based Java OS similar in design to the Fluke micro-kernel.
- Migrating Threads in Mach 3.0 - An extension to Mach that performed local IPC by quickly switching the process context of a thread.

2.2.1 KaffeOS

JanosVM is a direct descendant of KaffeOS [1], a fully operational system that can run most Java code ”out of the box” with only minor modifications. The design of KaffeOS takes advantage of language based protections and, like current operating systems, uses a monolithic kernel to control access to globally shared services. The language based protections allowed the system to take advantage of direct sharing between user processes, shared heaps, and the kernel without compromising the security of the system. The monolithic kernel and the important state change of entering/leaving the kernel served to protect any globally shared data from the unexpected termination of processes and their threads. In addition, KaffeOS could exert control and accurate accounting over CPU and memory resources used by processes in the system. For the CPU, this was accomplished through a stride scheduler written for the user level threading library. In the case of memory, each process was given its own garbage collector to track usage and to separate the amount of CPU used during collection and finalization.

Unfortunately, KaffeOS isn't publicly available at this writing.

2.2.2 Alta

Alta [4] is an earlier Java-based operating system. Unlike KaffeOS and JanosVM the garbage collector was not separated into per-process components, instead Alta concentrated on class and data sharing between *spaces* (Alta's term for the process abstraction).

Alta is based on the Fluke nested process model, a model for hierarchically organizing processes in a system, whereby processes are wholly "contained" (all resources and interfaces are granted to) by their parent process. In Alta, Java processes could share data through a client-server style interprocess communication (IPC) mechanism. The most important aspect of Alta, with respect to JanosVM, was the type and object sharing. The type sharing in Alta was defined at run-time as types were referenced in the child. The dynamic nature created performance and correctness problems that JanosVM was designed to avoid.

2.2.3 Migrating Threads in Mach 3.0

Extending Mach 3.0 to use migrating threads [3] in place of the standard IPC mechanism resulted in better performance, functionality, and simplified code. While not the first work on migrating threads, it is the one implemented here at Utah, so it has had the most direct impact. Thread migration involves switching the outer context of a thread to another process in the system, an operation very similar to a thread trapping into a kernel. The benefit of this approach primarily has to do with the reduction of overhead between the client and server threads. For instance, the system only needs to change to the foreign address space, there is no full context switch to the server thread. Besides the performance improvements, migration simplifies programming by removing the rendezvous and message passing between client and server threads. Finally, the merging of the disparate threads allows for more accurate accounting and control of the operations performed by client threads.

2.2.4 Kaffe

Ultimately, the core of the JanosVM is the Kaffe Virtual Machine [5], originally developed by Transvirtual Technologies. So, we thank them and the many volunteer contributors for their work.

2.2.5 Lessons Learned

These three systems served as the basis for much of the JanosVM's design. KaffeOS's code base and overall design became the starting point, while thread migration and Alta's flexible typing provided some interesting variation. However, there were several changes in the final implementation of the JanosVM, usually in favor of making the system more resilient in the face of faults. First, the KaffeOS garbage collector was changed to drop a processes' memory instead of merging it into the kernel's heap. This removed the possibility of process memory allocated by native code leaking into the kernel, an occasional source of grief while working on KaffeOS. Next, thread migration was used to replace the transition between user processes and the kernel, thus allowing threads to switch to peer or server processes. Because the system is a single address space, there is no significant additional overhead in this change and it enables the use of a microkernel-like design. Of course, one could ignore service partitioning in favor of a single process that does everything, but this goes against the desires for fault tolerance. Finally, a modified version of the Alta type system was added to enable simple class sharing as well as type renaming. This addition allowed JanosVM to continue to use a modified version of the `java.lang` package without precluding the possibility of a true JDK compatible version.

2.3 Language Protection Basics

Narrowing the scope of supported user languages and having direct control over their execution affords a number of benefits over hardware based protections. First, type safety guarantees the integrity and security of the system, even though data and code for multiple processes exist in the same address space. Next, member access control tags like public and private allow for fine grained control over what untrusted code can see and execute. Finally, it becomes easier to manipulate the final code executed by the CPU, allowing the introduction of write barriers or other runtime checks.

A major tool leveraged by the JanosVM is the type safety inherent in the Java language. This protection makes it possible to place trusted data and code in the same address space as user code without sacrificing the security of the system. In JanosVM this is taken to an extreme by completely ignoring the interface to untrusted user code in favor of providing flexibility for one or more middleware layers. For example, creating a new process in the JanosVM does not automatically execute the `main` method of some class, instead the middleware library is expected to migrate into the new process and start a thread that will call the desired methods. This means more work for the middleware developer, but greatly increases the amount of flexibility they have in performing their tasks.

Cross process sharing of object references is another major benefit of language protected operating systems. While it is not a panacea, it does allow for more direct sharing between processes than a hardware protected OS. For example, many classes and their read only data can be directly shared with no major repercussions. In addition, system managed proxies can allow limited sharing of objects between mutually suspicious processes.

Chapter 3

Compiling and Installing JanosVM

The `INSTALL` *file in the topmost directory of the JanosVM source tree contains an abbreviated version of these instructions.*

JanosVM consists of the following programs:

`janosvm`	A script that starts up the JanosVM Virtual Machine and executes the Java OS's main class file.
`java`	Another script that starts up the JanosVM in a mode that emulates a typical Java Virtual Machine.
`kaffeh`	A tool for generating C header files from Java class files.
`jsihey`	A command line tool that can send and receive JanosVM Scripting Interface (JSI) messages. This is used to communicate with concurrently executing virtual machines.
`jsiregister`	A simple script used in conjunction with `jsihey` to register a mapping between a property for a JVM instance and a value.
`jsiunregister`	A simple script to unregister an instance or a property mapping.

3.1 Acquiring JanosVM

JanosVM is available on the World Wide Web under `http://www.cs.utah.edu/flux/janos/janosvm.html`. Currently, JanosVM is distributed only as source code.

3.2 Building the JanosVM Programs

In order to build JanosVM from source, you will need:

`x86 FreeBSD or Linux`	Kaffe works on a number of platforms, however, we have only concentrated on getting JanosVM functioning on x86 based FreeBSD and Linux. We use FreeBSD 4.3 on a Pentium III.
`gcc`	Kaffe and some of the JanosVM additions make use of `gcc` language extensions so you'll need a recent version to successfully compile everything. We use `gcc` version 2.95.3.

`awk`	Some `awk` scripts are used to automatically generate code. We use GNU `awk` 3.0.6.
GNU `make`	We make use of some `gmake` extensions so regular versions won't work. We use GNU make 3.79.1.
Disk Space	The JanosVM source will consume approximately 20 MB of disk space and a typical build tree will consume an additional 30 MB of space.

Once you have the above-listed tools, you will need to (1) unpack the JanosVM source distribution `tar` file and (2) create a build tree to contain the object files. Assuming that you did these things in your home directory, your directory setup would look something like this:

`~/janosvm-0.6/`	The JanosVM source tree, unpacked from `janosvm-0.6.src.tar.gz`.
`~/obj/`	The directory you create to be the root of your JanosVM object tree. (Of course, you can call this directory anything you like. It does not have to be called `obj`.)

3.2.1 Configuring JanosVM

Given the directory setup described in Section 3.2, you would type the following commands to configure JanosVM for your operating system:

```
cd ~/obj
../janosvm-0.6/configure
```

The `configure` script in the root of the JanosVM source tree is an ordinary `configure` script, generated by GNU `autoconf`. Type "`configure --help`" to see all of the script's command line options. In addition to all of the standard options accepted by `autoconf`-generated scripts, JanosVM's `configure` script accepts the following Kaffe and JanosVM-specific options:

`--with-staticlib`	Causes all of the Kaffe/JanosVM libraries to be statically linked, which includes the libraries that make up the JVM and the native implementations of some Java classes.
`--with-staticbin`	Same as the above, but also statically links system libraries.
`--with-staticvm`	Causes all of the libraries which make up the JVM to be statically linked, but the native implementation of some Java classes are still dynamic.
`--enable-debug`	Enable run-time debugging machinery.
`--disable-debug`	Disable assertion checking (but not -g).
`--enable-xprofiling`	Enable profiling for jitted code. You can also specify –with-profiling to generate a profile that covers both types of code.
`--enable-xdebugging`	Enable debugging symbol generation for jitted code. This is useful when debugging programs through `gdb` since you can get proper line number information when doing a back trace.
`--with-profiling`	Compile with gprof(1) profiling support.

3.2.2 Compiling JanosVM

Finally, you are ready to compile the JanosVM! Simply type the following commands to build JanosVM and its auxiliary programs:

```
cd ~/obj
make all
```

Assuming all goes well, these commands should produce all of the JanosVM programs that need to be compiled.

3.2.3 Checking and Installing JanosVM

Once the system has been compiled you'll want to run the test suite to make sure everything is working properly:

```
cd ~/obj
make check
```

There are actually two series of tests that will be run. The first is the standard kaffe test suite which verifies the ability to execute Java bytecodes and the second suite will verify the JanosVM specific extensions. Of course, if one of the tests does fail, we'd appreciate it if you'd send the `.fail`(s) file located in the `test/regression` or `test/janosvm` directories of the build tree, along with the usual system information.

Finally, if everything is working well enough you can install the system by executing:

```
cd ~/obj
make install
```

Now, you'll probably want to try out the tutorial in the following sections.

3.3 Related Software

If you're interested in running the JanosVM on a bare machine there are a few of software packages that can be used in conjunction with the JVM. First, the OSKit package provides much of the infrastructure needed to initialize and interact with a x86 based PC. A tarball and build instructions can be obtained from `http://www.cs.utah.edu/flux/oskit/`. Second, our NodeOS implementation, Moab, maps the OSKit's interfaces to an interface that JanosVM can be configured to use. Next, the Java NodeOS makes the NodeOS interfaces available to regular Java programs. And finally, there are a collection of "meta-makefiles" and instructions for automating the process of combining all the components together into a single system. These last packages can be obtained from `http://www.cs.utah.edu/flux/janos/`.

Chapter 4

Tutorial

This chapter takes you through the steps needed to run and write some simple example programs. Complete source code for each of the examples is located in the `doc/janosvm/tutorial` subdirectory of the JanosVM software distribution. It is advised that you make a copy of the directory and try out the JanosVM as you read the chapter.

4.1 A Hello World Kernel

First, compile `HelloWorld.java`:

```
buffy@ucsd tutorial> jikes -classpath <install dir>/share/kaffe/JanosVM.jar \
  HelloWorld.java
```

Note that you have to use the `JanosVM.jar` JAR file since it contains the JanosVM specific classes, as well as the basic Java classes. Next, run the class file with the following command:

```
buffy@ucsd tutorial> janosvm -config tutorial.config HelloWorld
[Ignore any warnings for now]
Hello World!
```

This will execute the `janosvm` script which starts up the main Kaffe binary with `JanosVM.jar` in the class path. The JVM then goes through some extra initialization steps:

- The JanosVM extensions become enabled when they detect their classes in `JanosVM.jar`.
- The extensions configure themselves using the configuration file specified by the `-config` option.
- The "kernel" team is created, where team is the JanosVM name for the process abstraction (think team of threads).
- The main thread migrates into the kernel team and executes the "Kernel Main" class file, `HelloWorld.class` in this case.

This method of starting up the JVM is useful for simple kernels like "Hello World" or fully capable kernels that can run for a long time and communicate with users. However, not having some mechanism to communicate with running JVMs would likely make the process of developing kernels and services extremely hard. Thus, we have developed the JanosVM Scripting Interface (JSI) in order to help mitigate these issues. The interface is modeled after AppleScript and the BeOS scripting interface so its pretty simple and can function over a number of communication mediums. To help you get comfortable with its usage we'll be using JSI to drive the JVM in the rest of the tutorial. First, though, we'll need to learn some of the jsi-specific `janosvm` command-line options (described in Section 5.1.1):

`-jsitcpport`	Specify the TCP port used to send and receive JSI messages.
`-noexit`	Causes the JVM to continue running even though the only live threads are daemons.
`-name`	Binds a name to this instance of the JVM. This name is then used to store the TCP port value in a simple registry in `~/.janosvm/registry/` that can be referenced by other programs.

With these options we can now run an instance of the JVM that won't exit immediately and doesn't need a kernel class file:

```
buffy@ucsd tutorial> janosvm -config tutorial.config -name janosvm \
  -jsitcpport 14000 -noexit &
buffy@ucsd tutorial> jsihey janosvm execute run of team 1 with \
  class HelloWorld then nothing
Hello World!
```

After starting the JVM we used the `jsihey` command to create and send a JSI message. This particular message tells team one, the kernel, to execute the HelloWorld class with no arguments. If you wanted to pass arguments to the class you'll need to add some data to the message by appending a number of identifier/value pairs to the command line. In the case of the `run` message, it can contain the `arg` and `args` identifiers which specify a single argument and separate arguments grouped together within quotes, respectively. For example:

```
buffy@ucsd tutorial> jsihey janosvm execute run of team 1 with \
  class HelloWorld args "xander anya" arg "Willow Rosenberg" then nothing
Hello xander
Hello anya
Hello Willow Rosenberg
```

Since we're done with the JVM for this section we'll send it a `shutdown` message so that it quits.

```
buffy@ucsd tutorial> jsihey janosvm shutdown
```

The `jsihey` tool can send a number of other messages also; check out its man page or Chapter 6 for more details. There is also a simple python binding for JSI in the `janosvm/libraries/unsupported/jsi/pylib` directory if you don't like shell scripts.

4.2 Creating a new Team

In the previous section we created a simple HelloWorld kernel, so now it's time to start creating our own teams and playing with them.

4.2.1 Using Java

For this example we'll be using the `HelloNewTeam.java` program, so compile and run it.

```
tservo tutorial> jikes -classpath <install dir>/share/kaffe/JanosVM.jar \
  HelloNewTeam.java
tservo tutorial> janosvm -config tutorial.config HelloNewTeam
kernel says hello
Child says hello
```

From the source code we can see that the first thing to print something is:

```
System.out.println(Team.current().getName() + " says hello");
```

Since the class is being run in the kernel the call to `Team.current()` will return the handle associated with the kernel, hence we get its name. Next we create a child with the default resource requirements and the name ”Child”.

```
TeamHandle child = Team.create("Child");
```

However, besides the GC threads there is nothing else executing within the context of this team, so we need to visit the team in order to get something done. The `child.switchTo()` call accomplishes this by switching the context of the current thread to the child team, now we can execute the same printing code and yield a different result. Next, we need to back out of the child team using `child.returnFrom()` call. Finally, we use `child.terminate()` to terminate the child team and return its resources to the JVM.

4.2.2 Using JSI

In addition to the Java interface we can also use JSI to create and terminate teams.

```
tservo tutorial> janosvm -config tutorial.config -name janosvm \
  -jsitcpport 14000 -noexit &
tservo tutorial> jsihey janosvm create team with name "Parent" \
  then print "%(result:d)\n"
2
```

The `jsihey` line tells the JVM to create a team with the name ”Parent” and print out the unique identifier associated with the team. This id is then used for all interactions with the team when using JSI. To get a listing of all the teams and their id's just execute the following:

```
tservo@satolove tutorial> jsihey janosvm list teams
Normal Reply:
  id = ‘1’
  name = ‘kernel’
  description = ‘’
  id = ‘2’
  name = ‘Parent’
  description = ‘’
```

Now that we have our new team let's run HelloNewTeam and see if its output has changed:

```
tservo tutorial> jsihey janosvm execute run of team 2 with class \
  HelloNewTeam then nothing
Parent says hello
Child says hello
```

Well it seems to work, now let's terminate the parent:

```
tservo tutorial> jsihey janosvm delete team 2 then nothing
```

That should be enough for now, shutdown the JVM and let's move on to the next section.

4.3 A Client/Server HelloWorld

The ability to create new teams means we can experiment with sharing objects via cross-team pointers. This example implements a client/server HelloWorld using the `HelloBackEnd.java` class to export an object from the kernel and the `HelloHandle.java` class to import that object into the current team. These classes make use of the `Exportable` and `Importable` system classes which manage references into other teams.

```
stewie tutorial> jikes -classpath <install dir>/share/kaffe/JanosVM.jar \
  HelloBackEnd.java
stewie tutorial> jikes -classpath <install dir>/share/kaffe/JanosVM.jar \
  HelloHandle.java
```

Now, start up the JVM and run `HelloBackEnd` in the kernel to export the server object.

```
[Terminal 1]
  stewie tutorial> janosvm -config tutorial.config -name janosvm \
    -jsitcpport 14000 -noexit

[Terminal 2]
  stewie tutorial> jsihey janosvm execute run of team 1 with class \
    HelloBackEnd then nothing
```

Let's divert our attention for a second to the `tutorial.config` file because it's working some magic required for all this to work. The major part of the config file defines which classes can be exported from a team in order to be imported by others. Basically, it lists what classes can be shared between teams. Near the bottom you might have noticed that `HelloBackEnd` is listed as one of the exports. The export is needed because any teams that import a `HelloBackEnd` object are going to do a cast from `Object` to `HelloBackEnd` so they'll need the same class. If this export wasn't there a different `HelloBackEnd` class will be loaded by each team and the cast wouldn't work, so we wouldn't be able to use the object as anything more specific than a `java.lang.Object`. The actual process of exporting the classes was already done by the kernel and the `HelloBackEnd` main, so we just need create a team that imports the classes and run `HelloHandle`:

```
[Terminal 2]
  stewie tutorial> jsihey janosvm create team with name "hello-client" \
```

```
    namespace 1:kernel:identity namespace 1:tutorial-hello:identity \
    then print "%(result:d)\n"
  2
  stewie tutorial> jsihey janosvm execute run of team 2 with class \
    HelloHandle then nothing

[Terminal 1]
  Client hello-client says ‘Hello World’
```

Cool. The first `jsihey` line created the team and imported the kernel classes and the `HelloBackEnd` class using the `namespace` arguments. The `namespace` argument is a bit complicated, it specifies what team to import the classes from (1 == the kernel), the group of classes as listed in the config file, and the team's view of the classes. There will be more detail about class groups and views later, but for now, lets shutdown the JVM and continue with the tutorial. Or, if you're adventurous, create a new team without the `namespace 1:tutorial-hello:identity` argument and see what happens.

4.4 Terminating an uncooperative Team

This is a quick example; we just want to show that you can kill a misbehaving team. We'll be using `HelloSpaz.java` for this example:

```
msmart@control tutorial> jikes -classpath <install dir>/share/kaffe/JanosVM.jar \
  HelloSpaz.java
```

The program doesn't create its own team so we'll have to start one for it.

```
[Terminal 1]
  msmart@control tutorial> janosvm -config tutorial.config -name janosvm \
    -jsitcpport 14000 -noexit

[Terminal 2]
  msmart@control msmart> jsihey janosvm create team with name "Kaos" then \
    print "%(result:d)\n"
  2
  msmart@control msmart> jsihey janosvm execute run of team 2 with class \
    HelloSpaz then nothing
  msmart@control msmart>

[Terminal 1]
  hihihihihihihihihihihi
  hihihihihihihihihihihi
  hihihihihihihihihihihi
  ...
```

Now we just delete the team and the output should be stopped:

```
[Terminal 2]
  msmart@control msmart> jsihey janosvm delete team 2
```

Well, this is the end of the tutorial. You can also look in `test/janosvm` for more code examples.

Chapter 5

Using JanosVM

JanosVM executes Java bytecodes in one of two environments:

- The Java Operating System substrate mode which causes the JVM to enable many of the internal features that make it possible to implement a high level Java OS. This is activated by running the `janosvm` script which uses the `JanosVM.jar` file.
- The Java Virtual Machine emulation mode which causes the JVM to behave like a version of Kaffe. This is activated when running the `java` script which uses the `Klasses.jar` file.

We'll concentrate on the substrate mode in this document, since that is what you will be using most of the time, however, the emulation mode can be useful for testing purposes.

5.1 The `janosvm` Virtual Machine

Starting up a JanosVM instance using the `janosvm` script requires either a Java class file and/or the `-jsiprovider` and `-noexit` options. The first method depends on the class to initialize the kernel of the Java OS and construct some communication path for the user to interact with the JVM. While the second method starts the JVM as a daemon and uses the built in JanosVM Scripting Interface (JSI) to facilitate communication with the user. The rest of the command line options are described in the next section.

5.1.1 Command Line Options

The command line options to `java` and `janosvm` are as follows:

`-help`	Give a short usage message.
`-version`	Print the version number.
`-fullversion`	Print verbose version info.
`-ss size`	Set the maximum stack size to *size*.
`-ms size`	Set the initial heap size to *size*.
`-mx size`	Set the maximum heap size to *size*.
`-as size`	Set the heap increment to *size*.
`-classpath path`	Override the default search path for class files with *path*.

`-addclasspath path`	Appends the given *path* to the current classpath setting. Multiple instances of this flag may be given.
`-D`*`property=value`*	Set *property* to the given *value*. Some properties are used by the JVM at startup. All can be fetched at runtime through the method `System.getProperty()`.
`-noverify`	Do not verify any bytecode.
`-noclassgc`	Disable garbage collection of classes.
`-jar`	Indicates that the main class file name is a JAR file.
`-v`	Enable verbose output.
`-verbosegc`	Print messages during garbage collection.
`-verbosejit`	Print messages during JIT code generation.
`-verbosemem`	Print detailed memory allocation statistics.
`-vmstats flag{,flag}`	Turn on VM statistcs. Use `-vmstats list` for a list of available flags.
`-name name`	Register this JVM instance as *name*. This is useful when the JVM is going to be the target of JSI messages since you can just use the registered name instead of a whole protocol and address specification.
`-jsiprovider name`	Specify where the JSI provider should listen. The format of *name* is a protocol name followed by a colon and then some implementation dependent address data. For example, to listen on TCP port 10000 of the current machine (sloane.sd6.gov), you would use `-jsiprovider tcp:sloane.sd6.gov/10000`. Currently, only one provider is allowed at a time.
`-jsitcpport port`	Specify a TCP based JSI provider that should listen on *port*. Basically, a fast alternative to the above.
`-jsifile file`	Setup a JSI link through files with *name* as their basename. This option can be used multiple times.
`-config file`	Use *file* for initial configuration of the JanosVM.
`-noexit`	The JanosVM continues even if only non-dameon threads are running. Useful for JSI-controlled JVMs that would otherwise exit.
`-vmdebug flag{,flag}`	Enable VM debugging. Use `-vmdebug list` to get a list of available flags.
`-nodeadlock`	Disable deadlock detection.
`-Xxprof`	Enable cross language profiling. See `FAQ/FAQ.xprofiler` for details.
`-Xxprof_syms file`	Name of the profiling symbols file. Defaults to *kaffe-jit-symbols.s*.
`-Xxprof_gmon file`	Base name for gmon files. Defaults to *xgmon.out*.
`-Xxdebug`	Generate a debugging symbols file usable by `gdb`. See `FAQ/FAQ.xdebugging` for details.

`-Xxdebug_file file`	Name of the debugging symbols file. Defaults to *xdb.as*.
`-Xfeedback`	The file name to write feedback data to.

Besides the command line options, the JVM will read a few environment variables:

`CLASSPATH`	The path used when searching for classes. In a UNIX system this is a colon separated list of paths.
`KAFFE_DEBUG`	This indicates that the startup script should execute the JVM in a debugger, typically, this is set to `gdb`.

Once the JVM has been started the user is expected to start and stop programs using the communication path implemented by the Java OS kernel or the JSI tools, like `jsihey`.

Chapter 6

Using jsihey

The `jsihey` command line tool is designed to be the lowest common denominator for sending any type of JSI message, over any type of communication medium, and dealing with the reply.

6.1 Argument Format

The arguments to `jsihey` are given in a near English syntax similar to AppleScript's `tell` and BeOS's `hey` forms. The style tries to simulate a natural conversation between the user and the target object. For example, "hey photoshop create window with file picture.jpg" is a little easier to understand than "hey photoshop window.create('picture.jpg')". In addition, the style doesn't require `jsihey` to fully understand the structure of the destination object, thus allowing it to be used for any JSI enabled application.

The overall argument format is:

- The target program's address.
- A "verb" that describes what action to take.
- A "specifier stack" that routes the message to the target object.
- An optional argument list given as identifier/value pairs.
- And, an optional reply action.

In addition, `jsihey` will print out a pseudo-BNF usage statement if no arguments are given, if you'd like a more structured description of the syntax.

6.1.1 Target Program

The target program's address is a formatted string starting with the protocol type, followed by a colon and the protocol specific address. For example, to communicate with a program running on the same machine and listening on TCP port 10000, you would use "tcp:localhost/10000". There are only a few communication implementations at the moment:

tcp	Uses TCP/IP to communicate with the target program. Format: tcp:<hostname>/<port>.

`file`	Uses a pair of files, `<prefix>.to` and `<prefix>.from`, to communicate with the target program. Format: `file:<prefix>`. Note: Unlike TCP, this protocol cannot support more than one concurrent client, so you'll need to create extra files if you wish to do this.
`plex:udp`	The `plex` protocol is used to multiplex JSI streams over a single unreliable data link. In this case, UDP, however, any unreliable communication link would suffice. Format: `plex:udp:<hostname>/<port>`. Note: This is still somewhat experimental and is only used when communicating with a JanosVM running on a bare machine.

Since the above address specification can be a little wordy, you can use a "registered" name in its place. The "registered" name is simply a directory in `$JSI_ROOT/registry` that contains a `jsi.id` file, where `$JSI_ROOT` defaults to `$HOME/.janosvm`. In the case of JanosVM, you can automatically register a name using the `-name` option or you can use the `jsiregister` script:

```
[stewie@home ~] janosvm -jsiprovider tcp:localhost/9000 -name jvminstance
[stewie@home ~] jsiregister jvminstance jsi.id tcp:localhost/9000
```

6.1.2 Message Verb

After the target program value, the rest of the arguments are used to build the request message. The first value is a "verb" that determines the action to be taken. Currenty, `jsihey` understands the following verbs:

`getsuites`	Get the set of `suites` for a given object. Each `suite` contains information about an object in the program, including what verbs it can be used with and their arguments. So, if you need help interacting with a program, this is the first place to look.
`list`	List any objects contained within another object.
`get`	Get the value of an object.
`set`	Set the value of an object.
`create`	Create a new object with the given arguments.
`delete`	Delete an object.
`execute`	Execute a method on an object with the given arguments.
`shutdown`	Shutdown the target program.

Several of the verbs, `getsuites`, `list`, and `shutdown`, can be used without additional arguments since they operate directly on the root of the program's object tree. However, most require a real object to process them properly, which brings us to the next part of the argument list and message, the "specifier stack".

6.1.3 Specifier Stack

The verb determines what action to take, but doesn't indicate which object should take that action, this is the job of the "specifier stack". The stack contains a list of specifiers that describe the path through the program's object tree to the desired destination. Each specifier in the stack is made up of an object type or name and an optional identifier, for example, "window 1". Finally, you aggregate the specifiers into a stack using the `of` keyword. For example, "button ok of window 1" would build a stack that addressed the "ok" button inside of window number one.

6.1.4 Identifier Pairs

In the case of the `create` and `execute` verbs they can take an optional list of identifier/value pairs to be used as arguments. To add a list to the message simply add `with` to the end of the specifier stack and then write the identifier name and value separated by a space or equals sign. You can add the same identifier more than once to create array. For example:

```
[stewie@home ~] jsihey jvm create team with name lois
[stewie@home ~] jsihey jvm create team with name=peter
```

The `set` verb also requires an argument, however, the identifier is implicit so only the value needs to be given after a `to` keyword. For example:

```
[stewie@home ~] jsihey raygun set level to stun
```

6.1.5 Reply Actions

Once the request message has been constructed and sent off to the target program and object, `jsihey` can react in a number of different ways to the reply message. The default response is simply to wait for a response and print out the contents of the reply message. However, you can change this by using the `then` keyword followed by one of these actions:

`print` Print out a formatted string that can use values from the reply message. The format string can contain plain text, formatting escapes, and directives which print out values contained in the reply message. An escape starts with a backslash and is followed by a t, for tab, or n, for a line feed. Each directive starts with a `%(` and is followed by the name of a value in the message, an optional range if the value occurs more than once, then a colon followed by a regular printf format conversion, and finally, the closing parentheses. For example, the format, `%(result[1]:d)` would print out an integer with the value of the second instance of `result` in the reply message. In addition, it is possible to print out a range of indeces separated by a string. This is done by using a pair of periods to separate the starting index, a separator string, and the final index. For example, `%(result[1. .5])`, would print out the second through sixth values of `result` separated by spaces. Also, the range values can be left out to imply the first and last indeces.

`forinstance` Walks over all of the instances of an identifier in the message and repeatedly executes the rest of the actions on the command line. The arguments to `forinstance` are a temporary variable name used to store the current index, the `in` keyword, the message identifier to loop over, and finally, the action to perform on each iteration.

`nothing` Wait for the reply, but don't print anything, even if the reply is an error.

`ignore`	Dont wait for the reply, return immediately. Note: This is the default action for the shutdown message since it isn't expecting a reply.

6.1.6 Interim I/O Messages

The JSI protocol supports what is known as "interim messages" that can be transmitted while an object is trying to satisfy a request. For example, a request to the stdout object can send an interim message to an object on the requesting side requesting the data to print. The option to allow objects to coordinate themselves without user input can be quite handy, however, `jsihey`'s primitive nature leads it to only support a minor amount of interaction. Therefore, `jsihey` implements three objects named "in", "out", and "err" which represent the standard I/O files. With these objects you should be able to do something like the above example using `janosvm`:

```
[lsimpson@utah ~] jsihey jvm create file with name test
Normal reply:
  result = 'ok'
[lsimpson@utah ~] jsihey jvm execute write of file test
Hello World!
^D
Normal reply:
  result = 'ok'
[lsimpson@utah ~] jsihey jvm execute read of file test
Hello World!
Normal reply:
  result = 'ok'
```

As you can see, we created a file and executed it's write method. Since no data was given to write, the JVM sent an interim message to `jsihey`'s "in" object asking for the data to write. And, a similar process occurred when we read the file, except the message was sent to the "out" object.

6.1.7 Return Values

When `jsihey` is finished processing messages it will exit with one of these values:

`0`	Everything worked fine.
`10`	The reply message was an error.
`20`	There was a communication error.
`30`	There was a usage error.
`100`	There was an internal error.

6.2 Example jsihey usage

List the root properties of the JanosVM listening on TCP port 14000 of the local machine:

```
[bubbles janosvm] jsihey tcp:localhost/14000 list
Normal reply:
  result = 'err'
```

```
  result = 'out'
  ...
```

Getting the suites of the teams object:

```
[bubbles janosvm] jsihey tcp:localhost/14000 getsuites of teams
Property:  name
  Supported verbs:  get
Usage (arguments, if any, followed by a description):
  The name of the team

Property:  description
  Supported verbs:  get, set
Usage (arguments, if any, followed by a description):
  A description of the contents of the team

...
```

Create a new team with the name “THX” in the JanosVM listening on UDP port 1138 and then print the resulting team identifier.

```
[bubbles janosvm] jsihey plex:udp:townsville/1138 create team with name "THX" \
  then print "%(result:d)"
5
```

Delete the newly created team and don’t do anything with the reply:

```
[bubbles janosvm] jsihey plex:udp:townsville/1138 delete team 5 then nothing
```

Turn on the xprofiler:

```
[bubbles janosvm] jsihey janosvm set xprofiler to on
```

Get the current value of the xprofiler:

```
[bubbles janosvm] jsihey janosvm get xprofiler
Normal reply:
  result = 'on'
```

Shutdown the JVM:

```
[bubbles janosvm] jsihey janosvm shutdown
```

List all of the agents of a covert spy organization:

```
[sloane@sd6 ~] jsihey db list agents
Normal reply:
  result = 'Arvin Sloane'
  result = 'Sydney Bristow'
  result = 'Jack Bristow'
  ...
```

Format the above list:

```
[sloane@sd6 ~] jsihey db list agents then \
  forinstance lpc in result print ''Agent #%(lpc:d): %(name[lpc]:s)\n''
Agent #1: Arvin Sloane
Agent #2: Sydney Bristow
Agent #3: Jack Bristow
...
```

Another formatting of the above list:

```
[sloane@sd6 ~] jsihey db list agents then print ''%(name[.\n.]:s)\n''
Arvin Sloane
Sydney Bristow
Jack Bristow
...
```

Executing the domination property of the world object:

```
brain@acmelabs ~> jsihey janosvm execute domination of world
Error:
  why = 'Pinky'
```

Executing the domination property of the world object using plan B:

```
brain@acmelabs ~> jsihey janosvm execute domination of world with plan B
Error:
  why = 'Pinky'
```

Chapter 7

Java OS Substrate

This chapter will concentrate on the interfaces and mechanisms for creating a Java Operating System on the JanosVM. Because a great deal of infrastructure has already been implemented, the task of creating a Java OS boils down to using these tools in a middle-ware library that consists of the following:

- The server code which manages any shared resources and/or data. Typically, these classes would run in one or more isolated contexts in order to compartmentalize the data and provide fault tolerance.
- The user visible classes which provide a safe path, through language based protection, for untrusted code to access the shared data.

7.1 Teams

The method for isolating computations and tracking their resource usage is the Team, as in, "team" of threads. Isolation is primarily provided by the semantics of the Java language, therefore, the JanosVM doesn't need to rely on any underlying OS or hardware support. However, the vast amount of infrastructure needed to execute Java code is shared and needs to be protected or duplicated for each computation. For example, at the JVM level:

- Garbage collectors are created for each team to separate the amount of time they spend marking and sweeping objects.
- The class path can be specified per-team in order to allow fine grained control.
- A separate class name space is created for each team, eliminating some intermingling of internal data structures and allowing name remapping.

In addition, there are quite a few Java objects and classes that need to be shared, these are held by the "kernel" team which is created by the system upon initialization.

Creating a team is done using the `create` static method of the `edu.utah.janosvm.sys.Team` class:

```
TeamHandle th = Team.create();

th = Team.create(``Important Server Team'');

th = Team.find(``Important Server Team'');
```

The first line creates a team with the default amount of memory, class path, and other resources, while the second does the same as well as giving it a name which can be searched for using the `find` method. Once the resources for the new team have been allocated and initialized, the system will return a `TeamHandle` object that gives limited access to the team. Notice that the system will not start up any threads or run any user code, the next section will introduce some methods for bootstrapping the new team. The `create` method can also take an array of resource specifications that override the defaults, these will be discussed later in section 7.3.

Failure to create a team can occur if the current team is already terminated or there was a failure to allocate a resource. In the case of resource failure the `create` method will throw a `ResourceSpecException` with a reference to the specification that couldn't be satisfied, unless the default was used, in which case there is no indication of what failed. In the future this should change to more accurately reflect the failure.

Besides the `TeamHandle` object, there is also a `Team` object which only exists local to the team. It is a subclass of `TeamHandle` and possesses a few methods for manipulating state pertinent to the current team. The current `Team` object can be retrieved through the `current` method in the `Team` class.

```
Team curr = Team.current();
curr.setDescription(``Insert Descriptive Text Here'');
```

Unlike many other systems, there is no strict hierarchy maintained by the system, every team is a peer of every other, including the kernel. Therefore, team's are not bound to the lifetime of their creator, nor do they inherit any resource specifications. Of course, there are methods for performing these actions, however, the JVM will not do them by default.

7.1.1 Visiting Teams

A team will isolate untrusted user computation and resource usage from the rest of the teams, however, system code requires a path for manipulating state in foreign teams. Therefore, the JanosVM allows threads to visit other teams if they need to perform some operation or communicate some data to the destination team. The migration of a thread is split into two stages (1) reserving the current and destination team so they cannot be terminated and (2) switching the context of the current thread to the reserved team. The first stage is used to protect any global structures that might be damaged when the team's threads are stopped on termination. For example, taking a shared lock should be done within the context of a reservation since the JVM will not fix up the lock if the thread is terminated. The second stage affects any operations that reference data structures hanging off the current team, for instance, creating threads and allocating memory.

Because the JanosVM functions in a single address space, the end result of a visit is the thread's ability to gain urestricted access to the resources of all the team's involved. The thread can then store references in its stack that point to objects in visited teams, copy primitive data, and perform restricted copies of references (Section 7.2) without any special mapping operations. These relaxed conditions make it extremely easy to write straightforward code for manipulating state and copying data, a desirable alternative to a more heavyweight messaging system.

The actual act of reserving a team is a little involved because we have to maintain the sanity of the internal team structures in the face of threads jumping through stack frames. For instance, Java code executing in the middle of a reservation could throw an exception through the block that reserved the team. The current solution is to record each team reservation and context switch on a secondary thread stack and then use these recordings to rollback any changes. Another solution might have been to mandate the use of a `finally` clause in all code, however, the system would still need to know when a visit had taken place so it could copy the exception to the previous team. In addition, the lack of automation in this approach places more pressure on developer, leading to more places for coding errors.

Performing a visitation can be done in the above two stage form by using the `pushReservation`/`pop` and `visit`/`leave` methods.

```
try
{
        TeamReservation tr = th.pushReservation();

        tr.visit();
        {
                /* Do something in the remote team context. */
        }
        tr.leave();

        /* Do something in the original team context. */

        tr.visit();
        {
                /* Do something in the remote team context. */
        }
        tr.leave();

        tr.pop();
}
catch(TeamEjectionException e)
{
        /*
         * A ``catastrophe'' occurred in the remote team and our thread was
         * forcefully thrown out.
         */
}
catch(DeadTeamException e)
{
        /*
         * Couldn't push a reservation since the current or remote team were
         * already dead...
         */
}
```

The first line tries to push a reservation for the team referred to by the `TeamHandle` object in the `th` variable. If both teams are still alive, the call will succeed and return a `TeamReservation` object which can be used to visit the remote team. However, if either team has been terminated the method will throw a `DeadTeamException` which should be dealt with appropriately. It should be noted that the `TeamReservation` object is not a dynamically allocated object, it is the actual object allocated on the secondary stack. So, the object must be popped off the secondary stack with the `pop` method or automatically when an exception is thrown. Next, we temporarily visit the other team two times to do some operations in its context. It is possible for the thread to panic or be forcefully ejected during its visits so any `TeamEjectionExceptions` will also need to be handled. Currently, a thread is only ejected when the `VMInstance.crash` method is called or if the JVM is unable to allocate memory for a stack back trace.

An alternative to the above visitation form is to use the all-in-one `switchTo` and `returnFrom` methods which perform as their names imply:

```
try
{
        th.switchTo();
        {
                /* Do something in the other team context. */
        }
        th.returnFrom();
}
catch(TeamEjectionException e)
{
        /*
         * A ``catastrophe'' occurred in the remote team and our thread was
         * forcefully thrown out.
         */
}
catch(DeadTeamException e)
{
        /*
         * Couldn't push a reservation since the current or remote team were
         * already dead...
         */
}
```

7.1.2 Termination

Termination of a team is done by calling the `terminate` method on the team's `TeamHandle` object. The actual process of termination involves breaking any VM-maintained cross-team pointers, stopping the team's threads, and returning its resources to the JVM. The first stage is responsible for maintaining type safety because the terminated team's memory will soon be freed and the garbage collectors in other teams need to be notified. Next, the team is "caged" to stop any inside threads from leaving or any outside threads from entering. The purpose of the "cage" is to prevent the team from living in perpetuity and to protect any shared data from being damaged when the threads are stopped. Once all the absent threads have returned to the terminated team and the visitors have left, the threads are stopped. This hard stop is possible because any shared data is protected by team reservations and any other data, damaged or not, is ignored during termination processing. Finally, the team's resources are returned to the JVM for other teams to use.

The responsibility for doing the termination work falls partially on the terminating thread, any other threads interacting with the team, and the "Kernel Reaper" thread. First, the terminating thread is responsible for breaking any Java level cross-team pointers and caging the team by marking it as "dying". Next, the first thread that notices there are no more reservations for the team marks the team as "dead" and places it on a queue managed by the "Kernel Reaper" thread. Finally, the "Kernel Reaper" does the work needed to break down the team's components and marks it as "buried".

7.1.3 Team Dependencies

Teams in the JanosVM are not totally independent entities, sometimes their lifetimes need to be bounded by some other team in the system. The need for this restricted lifetime arises from the convenience of directly sharing classes and other objects in the system. For example, `static final` values and static strings in classes, as well as several internal data structures, can be directly referenced without using proxies or team reservations. In other words, without these strict lifetime guarantees a team could have references to memory locations that were freed or reused. In addition, a Java OS might have semantic requirements concerning the team's spawned by a parent program.

Currently, dependencies are automatically generated by some resources or by using the `addDependency` method on the `Team` object. For example:

```
/*
 * Reserve the kernel so the Team.create() and Team.addDependency() calls are
 * atomic with respect to termination of either team.
 */
TeamReservation tr = kernel.pushReservation();
{
        Team parent = Team.current();
        TeamHandle th = Team.create();

        th.switchTo();
        {
                Team child = Team.current();

                /* Make the child depend on the parent team. */
                child.addDependency(parent);
        }
        th.returnFrom();
}
tr.pop();
```

7.1.4 Team Events

The creation, termination, and other important events in the lifetime of a team are tracked by the kernel as well as broadcast to any listeners through a `TeamList` object. Adding a listener is done by creating a new `TeamList`, adding a `TeamListHandler` event handler implementation, and calling the `listen` method. The kernel will then call the `event` method on the `TeamListHandler` object whenever one of the following events occurs:

`CREATED`	A new team was created.
`FINISHED`	All of the non-daemon threads in a team have exited. Note, this is only sent if you've called the `notifyOnFinish` method on the `Team` object.

`TERMINATED`	A team's `terminate` method has been called.
`BURIED`	A team's resources have been returned to the system.

The arguments to the `event` method are a `TeamReservation` for the listener's team, since the handler is executing in the context of the kernel team, and a `TeamEvent` object which describes the event and which team it concerns. Team identification is done using the unique integer identifier given to each team upon creation. This identifier is very similar to a UNIX process identifier and is expected to be unused for some amount of time after the death of the process. Thus, a `TeamEvent` can properly identify a buried team even though its objects no longer exist and the `TeamHandle`s are invalid.

7.2 Cross-Team Pointers

Sharing objects between teams is quite enticing considering the JanosVM is a single address space/language protected system, however, there are a number of issues that need to be resolved. For example, write barriers would be required to notify garbage collectors that their objects were being referenced by other teams. The approach taken to solve these problems is to use a write barrier with no side effects to deny most direct sharing between teams and a system of object proxies for all other types of sharing.

7.2.1 Write Barrier

Unlike KaffeOS, the write barrier used by the JanosVM has no side effects except to throw an `edu.utah.janosvm.sys.SegmentationFaultError` when a thread tries to write a reference from one team into another. Unfortunately, this is a bit draconian, so there are a number of exceptions:

- Pointers from dependents into their parents, although the objects they refer to are expected to be part of the root set or have a lifetime greater than the dependents.
- Pointers in the proxy objects specified in the next section. This allows the proxies to cache pointers without having to worry about triggering the write barrier.
- A number of exempt system classes. For example, the `Thread` object is implemented to store the object it is waiting on in a field, so `Thread` uses the exempt class, `ThreadObjectHolder`, to store the object.

7.2.2 Object Proxies

References between objects in peer teams cannot be allowed because one team could tie up memory by artificially maintaining references to objects. Alternatively, the JanosVM offers system maintained object proxies in the form of the `Exportable` and `Importable` base classes. The `Exportable` class serves as a base for objects that are exported to other teams in the system. Typically, a middeware library would provide exports for protected objects in the system (i.e., files or sockets) and middleware-specific shared data. As you might have inferred, the `Importable` class is then used as a base for classes that import the objects exported by other teams. For example, the `TeamHandle` class discussed earlier is an `Importable` that refers to a `TeamBackEnd` object in the kernel.

Implementing imports and exports is a relatively simple process. First, an export can be created by extending the `Exportable` class and then using an `ExportManager` to track the export and all of its importers. The `ExportManager` objects are created by the `PortAuthority` class which can be accessed through the `Team` class. For example, an object can be exported under a given identifier by calling the `exportObject` method on the current team's Team object. It is also possible to export the object under multiple identifiers resulting in multiple ExportManagers. Of course, an export doesn't always require a name so the exportAnonymousObject method can be used to export the object and obtain an ExportManager. The export can then be revoked manually by calling the revokeObject and revokeAnonymousObject methods in the Team class.

Implementing an import is done by creating a class that extends the Importable class and implements the bindTo and unbindFrom methods. These methods are used to control the direct references held by the import to the export in the remote team. In a typical implementation the Importable subclass would contain a field of the export's type which is initialized with the Exportable value passed into bindTo. The implementation for the unbindFrom method would simply null this field to break the cross team reference. The rest of the class would then be made up of wrapper functions that may or may not visit the team to execute the exports methods. Actually importing an export is done by calling the importObject method on the remote team's TeamHandle with the export's identifier. Or, executing the addImporter method on the ExportManager while visiting the remote team. Note that a badly-written Importable can violate the requirements of its usage and cause the VM to crash. Thus, only trusted code should be allowed to use and create Importable and Exportable objects.

Currently, all methods on the Importable that access the Exportable object must synchronize on the import object. This synchronize is needed to delay a terminating thread from revoking the reference while it is still in use. Unfortunately, this is a poor solution because hostile code could hold the lock on the importable indefinitely as well as requiring the developer to remember to use synchronized on all the methods. Alternatives are being investigated, however, a real solution has yet to be decided upon.

Naming Conventions

We've adopted a specific naming convention when developing Importable and Exportable classes and would like to see it used in outside developments so as to avoid misunderstandings amongst users. This convention is to append BackEnd to all Exportable classes, for instance, the TeamBackEnd. Next, since there can be many types of Importable classes importing the same Exportable class we can't enumerate all forms, so we've chosen to mandate two forms, the primary and secondary object handles. The primary handle is simply given a regular name (e.g. MessagePort) and can used to implicitly create an Exportable upon instantiation. The secondary handle has Handle appended to its name and is used to grant restricted access to the export. For example, a team could use new MessagePort to create a message port and its exportable. The team could then create a communication path by passing MessagePortHandle objects to other teams in the system. Since the MessagePortHandle is a restricted interface, it only allows messages to be queued on the interface, while the MessagePort can be used to enqueue and dequeue messages.

7.3 Resources

System primitives needed to execute and support a team are referred to as resources. These resources are specified at team creation, tracked and limited on a per-team basis by the system, and automatically released upon termination. Specifying the needs of a team is done through a resource specification, an object or collection of objects that describes the team's requirements. This specification is then passed to the resource and used to construct a client object that the team can use to access the resource. For most resources the client object is attached directly to the team and accessed by executing static methods in the resource's class.

The classes described in this section can be found in the edu.utah.janosvm.resources package in the JanosVM.jar file. Describing all the resources required by the team is done by passing an array of ResourceSpecs to the Team.create. Constructing ResourceSpecs can be done manually or by using the `current` static method available in many of the classes to copy the current team's specification. As discussed before, if the system is able to satisfy the given specs as well as any defaults needed to get the team going it will return a `TeamHandle`. Otherwise, it will throw a `ResourceSpecException` possibly containing the unsatisfied spec. For example, the following code will try, and most likely fail, to allocate a gigabyte of memory.

```
MemorySpec ms = null;

try
{
        Team.create(new ResourceSpec[] {
                ms = new MemorySpec(1024 * 1024 * 1024), // Gig of memory
        });
}
catch(ResourceSpecException e)
{
        if( e.getUnsatisfiedSpec() == ms )
                System.err.println(``Couldn't allocate a gig'');
        else
                System.err.println(``Some other resource was unsatisfied'');
}
```

7.3.1 Memory

The memory requirements of a team are specified with a `MemorySpec` object that describes the number of bytes required by the team and an optional garbage collection policy. The memory size is basically a guess that should be refined through profiling, there is currently no prescribed method for picking the number. The optional garbage collection policy is the name of a subclass of `GCPolicy` that is consulted when the collector needs to make a decision. Currently, the call-backs available to a policy are when, if ever, to start the GC and finalizer threads and when to trigger a collection.

Once the team has started it can access its memory client object through the static methods in the `MemoryResource` class. Currently, the class only provides methods for getting simple statistics on current and peak memory usage.

7.3.2 CPU

Unfortunately, the standard build of the JanosVM only includes a simple priority based scheduler that will not limit the CPU usage of a team. However, it is possible to take advantage of a proportional share scheduler in the oskit/moab/JanosVM configuration. There are plans to include a more advanced thread scheduler in future versions.

7.3.3 Name Space

The root class name space required by the Java language is serviced by the NameSpace resource. This resource tracks all classes loaded into a team and any classes that are shared amongst the teams. Basic interaction with the resource is done through the usual Java interfaces like `Class.forName` and class loaders. Unfortunately, the actual loading of classes is very sensitive to the context in which these methods are executed. In the case of `forName`, the class is loaded into the name space based on the caller's class loader and owning team. For example, if a class loaded into the root name space of the kernel uses `forName` during a visit, the requested class will be resolved in the context of the kernel. Loading a class into the current team irrespective of the caller's context can be done using the `loadTeamClass` static method in `NameSpaceResource`.

In addition to satisfying the standard requirements of the Java language the name space resource also manages any classes that are shared between teams. The ability to share classes makes it possible to consolidate memory, initialization, and information across all uses of the classes, but also introduces a great deal of complexity. Sharing in previous systems was usually accomplished through some amount of class loader magic, however, JanosVM's ultimate goal is to explore design alternatives so a new primitive called the "class group" was introduced. A "class group" refers to a group of closely related classes that can be used as an alternative to the standard class path during loading. Effectively, this allows one team to resolve names to classes residing in another team. This is accomplished by loading and resolving the group's classes in a team and then exporting them to other teams.

Grouping classes together is done partly as an optimization to avoid tracking every shared class and to reflect the property that few classes can exist in isolation. For example, the `Integer` class requires the `String` class in order to generate a string version of an `int`. Most class groups should be closed over the set of classes, in other words, any types visible to untrusted code should be satisfied by other members of the group. Closure is not a requirement for sharing classes since multiple groups may need to be combined to create a usable namespace, however, the combination of groups must be the same as those used to resolve the classes in the exporting team. Additionally, types not exposed to the user are not required to be in the group since they are always resolved in their original name space. For example, `Importables` can have a private field of their `Exportables` type without needing to include the `Exportable` in the class group. Defining the contents of a class group is done in the configuration file that is passed to the JVM on startup. Each class group is a subsection of the "namespace-resource" section and contains the list of class names in the internal forward slash form (e.g. `java/lang/String`).

The possibility of name clashes in this class sharing mechanism led to the development of the "view group" and, in fact, is another reason for putting classes into groups instead of managing them individually. The main example of name clashing has to do with JanosVM not providing a JDK compatible Java runtime, but still using the `java.lang` package. The reasons for doing this involve reducing the number of changes to the internals of Kaffe and the standard Java tools. Unfortunately, if a real java runtime was written for JanosVM then it would probably want its own version of the `java.lang` classes instead of the JanosVM specific ones. Therefore, class groups were designed to allow a team to view a set of classes through a specific mapping. Because the class group is usually a closed set, this is mostly a matter of exchanging the names used in methods and class lookup to refer to the alternatives. Again, like a class group specification, the mapping is specified in the config file as a subsection of the class group and contains the list of class names paired with the mapped name. By default, every class group has an "identity" view group which, as you might expect, makes the classes available under their builtin names.

The process of "exporting" a class group for other teams to use is done by calling the `exportClassGroup` method in `NameSpaceResource`. This method will load the classes into the current team and split them off of the team's main group into the new class group. The group can then be imported during team creation by passing a `NameSpaceSpec` with an array of `NameSpaceGroups` specifying which view group of a class group to import from a given team. Because the imported classes are directly referenced, the system will implicitly add a dependency on the exporting team. Unfortunately, you cannot dynamically import class groups from other teams or remove imported groups. These issues may be addressed in future versions by extending class loaders to allow them to import class groups instead of limiting this to the root class loader.

The following is an example of some config file entries for defining the "MyServerClasses" class group and the “BarView” view group.

```
%begin namespace-resource
  %begin classgroup MyServerClasses
    edu/utah/myserver/Foo
    edu/utah/myserver/FooHandle

    %begin viewgroup BarView
      edu/utah/myserver/Foo Bar              # Remap .../Foo -> Bar
      edu/utah/myserver/FooHandle BarHandle  # Remap .../FooHandle -> BarHandle
      %end

    %end
  %end
```

A server team can then export the classes using `exportClassGroup` and then create a new team that imports them.

```
NameSpaceResource.exportClassGroup(‘‘MyServerClasses’’);

nss = new NameSpaceSpec(new NameSpaceGroup[] {
        /* The kernel’s class group and ‘‘identity’’ view. */
        new NameSpaceGroup(kernelTeam, ‘‘kernel’’),
        /* My server class group and ‘‘BarView’’ view group. */
        new NameSpaceGroup(Team.current(), ‘‘MyServerClasses’’, ‘‘BarView’’),
});
```

Class Locals

Unlike most other systems, the JanosVM doesn’t automatically create process local static fields, instead a static always resolves to the same lvalue regardless of the current team. This approach is easier to implement and allows classes to collect and reference global information, however, it complicates some other aspects of the code. In an attempt to alleviate these problems the JanosVM provides methods for storing `Objects` with a class on a per-team basis. This functionality is accessed through the `getCurrentClassLocal` and `setCurrentClassLocal` methods in the `Team` class. The implementation simply stores an object in a team local hash table that is keyed on the class reference.

Note, because the “.class” language form of getting a `Class` object calls `Class.forName` it is advised that a static field be used to store the class object reference. Also, this interface is likely to change in favor of something similar to `java.lang.ThreadLocal`.

7.3.4 Class Path

The class path resource allows the user to specify zero or more directories or JAR files that can be used as a source for class files. The class path given to the JVM on startup determines the set available to any team and the default given to a team on creation. A more discriminating class path can be given using a `ClassPathSpec` object with an array of strings representing the desired elements of the path. The `ClassPathResource` class provides a number of static methods for querying the global class path if the path strings are not given by the user. Of course, its also possible to specify an empty class path, in which case the classes must be available through class groups imported into the team's root name space.

7.3.5 Standard I/O

The standard input and output streams available through `System.in/out/err` are maintained as a sacrifice to compatibility and ease of use . By default, all teams share the JVM's standard I/O streams. Specifying another set of streams can be done by creating a `StdioSpec` with some URI like strings. The format is `<protocol>://<path>` where the protocol and path values are as follows:

`builtin`	The JVM's "real" stdio streams. And the path is:
	`none` — No stream.
	`stdin` — Standard input.
	`stdout` — Standard output.
	`stderr` — Standard error.
`file`	A regular file and its path.
`jsi`	The JSI channel that was used to execute "run of team X."
`fd`	A file descriptor in the JVM, the fd given in the path is `dup`'d and used to build a `FILE` object. This form is used by the `StdioSpec.currentFD` method which can be used for sharing streams with child teams.

Again, for the sake of some amount of compatibility, these `public` and `static` fields in `System` have been maintained even though it results in the sharing of the same objects in all teams. Because of this sharing the input and output stream classes are written to indirect to the current team's stream objects. So, synchronizing on any of these stream objects will block all other teams and, if not bracketed by a kernel reservation, could result in memory corruption.

Bibliography

[1] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 333–346, San Diego, CA, October 2000. USENIX Association.

[2] Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Techniques for the design of Java operating systems. In *Proceedings of the USENIX 2000 Annual Technical Conference*, pages 197–210, San Diego, CA, June 2000. USENIX Association.

[3] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proc. of the Winter 1994 USENIX Conf.*, pages 97–114, January 1994.

[4] Patrick A. Tullmann. The Alta operating system. Master's thesis, University of Utah, 1999. 104 pages. Also available at http://www.cs.utah.edu/flux/papers/tullmann-thesis-base.html.

[5] Tim J. Wilkinson. Kaffe—a virtual machine to compile and interpret Java bytecodes. `http://www.kaffe.org/`.