Vrije Universiteit Brussel - Belgium

Faculty of Sciences

In Collaboration with Ecole des Mines de Nantes - France

and

Universidad de Chile - Chile

2000

Implementing a safe-threads model by modifying a JVM

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Jurgen Van Ham

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Prof. Luis Mateu (Universidad de Chile)

Contents

List of Tables

List of Figures

Listings

A word of thank

First of all I would like to thank my parents who gave me the chances and support to study.

I also want to thank 'Princess' Claudia, my Chilean friend. She gave me a lot of support during my stay in Chile. Without her it would have been a lot harder to be in this country. For all the good moments we had together, I can't thank her enough. I know that I found a very good friend in her.

For all the technical discussions and support during the implementation and the writing of this report, I thank Luis Mateu. Without him there would be even no safe-threads at all to implement.

Chapter 1

Introduction

This chapter explains briefly what this work is about without going too much into details. The intention is to give the reader a quick overview of the content without explaining the technical details.

1.1 Threads

The programming of computers evolved during the time that it exists a lot. In the beginning the only issue was efficiency. Later maintainability became an important issue. The programs became more structured during this evolution. The first step was structured programming that identified three structures: sequence, selection, iteration. The use of `goto`s became almost forbidden [4]. The use of procedures made programs better understandable in a top down approach.

The next step was object oriented programming, that tries to combine data and code to modify that data in objects. These objects can be used via an interface without knowledge of the inside.

At this moment sequential programs can not always solve the problems that today appear. If the user presses a stop button of a music player program, he expects that the program handles this request right now, not when the song has ended. This can be achieved by using threads.

With the current situation parallel computers are becoming usable outside of the very specialized areas that they were used before. To program them efficiently the task has to be split in several subtasks that can be executed in parallel.

At this moment the programmers start to need threads. This means that these threads have to be usable without causing to much trouble.

1.2 Problem

These threads allow that more instructions are executed at the same moment. This can be done with real parallel processors or with time-slices of a single processor.

When multiple threads modify at the same moment an object or an other piece of memory it is possible that some inconsistencies appear in that memory. These kind of memory accesses that may not be influenced by other threads are known as critical sections.

To prevent these problems the programmer should make sure that it is not possible that different threads modify an object at the same moment. This is not as easy as one could expect. If the programmer misses one of those cases it will cause only sometimes a problem, which means that this kind of bugs is hard to reproduce.

1.3 Possible Solution

A way to make threads more safe for the programmer is proposed in [8]. This is a model that is inspired by the Remote Method Invocation (RMI) of Java. The most important property of this model is that each thread runs in a separate virtual machine with its own variables and objects. The threads can move to another virtual machine, with the restriction that they can not enter a virtual machine that contains an active thread.

This model can be implemented in different ways. The different implementations will all have their own advantages and disadvantages. The important part was to implement the model so one can find out if it is usable to write programs and to do experiments that can lead to improvements of the model.

A prototype of this model was implemented during this thesis by modifying Kaffe, a clean-room open source Java Virtual Machine (JVM).

While implementing this prototype some problems showed up that were not expected when developing the model. These problems are not solved but some solutions are proposed at the end of this work. Also it is shown that with some programmer effort the problems that the safe-threads try to avoid, can be created again.

Chapter 2

Concurrent programming

High level programming languages like Java[5] try to hide as much as possible the underlying machine to the programmer. For instance the garbage collector can be seen as a tool that gives objects an infinite lifetime. It also resolves the problems associated with freeing memory to the programmer.

A harder to use feature of Java at this moment are the threads. Some authors discourage strongly the use of threads if not absolutely needed. Like in [10]. This is because of the problems that can happen when two threads are accessing at the same piece of memory and at least one of those threads is modifying the memory.

2.1 The need for concurrency

At this moment concurrency is needed for graphical user interfaces. For example in a database application the user starts a very time-consuming query. If the user thinks that this takes too long, he wants to be able to stop that query. Then he can try to get the required information in an other way. If a user wants to stop a query, it has to stop before it has ended by itself before it is stopped. If he presses a “Stop” button he wants that the query stops now, even if the computer is busy with handling the query.

Other possible applications are games where some graphical objects on the screen seem to be independent from others. For example some flying birds, while a car drives somewhere on the screen, and a waterfall. All this movements are not related very much. If the car stops the birds do not stop in the air.

These examples show that concurrency can be useful for a programmer even when the problem is not the time of computing. If programmers will write such applications they will consider to use *threads*. A thread is a single, sequential flow of execution that shares it memory with other threads. The word is only used when multiple threads are considered.

Recent languages, like Java, support multiple threads but the use of threads in those language can lead to a lot of unexpected problems.

2.2 Evolution in hardware

There is also another reason to use threads: the search for more computing power that until now lead to clock frequencies up to 1 GHz.

Until now the majority of the computers are close to a Single Instruction Single Data (SISD) design. Some extensions give sometimes Single Instruction Multiple Data (SIMD) possibilities but this is only useful in specific applications. The SISD is easy to program but the speed is limited by the limits of the processor. Multiple Instruction Multiple Data (MIMD) also called parallel computers are becoming a interesting solution. Even with the problems that MIMD architectures cause for software developers, these computers are becoming interesting because of their computing power.

To get more performance by using more processors, the task should be divided in more sub tasks that can be executed in parallel on different processors.

A problem with more processors is that there are different possible configurations, that require different approaches to use the computer in an efficient way.

A few examples:

- shared memory
- latency in communication
- number of processors

A parallel computer and a network of computers are conceptually very close to each other. A cheap way to build a parallel computer is to connect computers through a fast network. This will result in a distributed memory computer. Because of the hardware problems that raise when many processors need to share the same memory, distributed memory is the best choice with multiple processors.

To write a program for each different configuration would be very unpractical and expensive. This means that it would be very useful to find a single system to program all of them.

At this moment threads are used to allow concurrency in a single computer. They allow the programmer to write code that can execute different tasks at the same moment. This concurrency is simulated via time slicing but it will appear to the user like executed simultaneously.

2.3 Why are threads unsafe?

2.3.1 What is unsafe?

In [8] features are called unsafe if they generate a cost that is too high when not used properly. This cost can be caused by a long debugging time, or by a low reliability of the application.

An example of an unsafe feature is the `delete` in C++ that can be very hard to use if a block of memory is referenced by many parts of the program. This has been solved by garbage collection in programming languages like Java. The cost of this increased safety is lower performance.

2.3.2 The unsafety in threads

The problem with threads is that a programmer can easily forget to protect *critical sections*. These critical sections occur when a shared resource is manipulated by multiple threads. This means that a critical section requires synchronization between threads to avoid an inconsistent state caused by concurrent manipulation of that resource. This synchronization can be handled for example via monitors or semaphores. However it is in general very difficult to detect all the critical sections, especially in bigger programs. So a programmer can easily forget to protect some of them. An unprotected critical section is extremely hard to find because it behaves very unpredictablely and can appear only in rare cases. This results in bugs that are not reproducible.

2.4 Previous work on the safety of threads

In the past some work has been done to hide the distributed memory from the programmer and make it appear as shared memory. This will make threads easier to use, but it does not solve problems like the critical sections at all. Not many research has been done on improving the safety of threads. The work of Brinch-Hansen and Hoare (see Section 3.6) is incompatible with pointers. Java uses only pointers except for primitive types.

Some work can be used for building a more safe implementation of threads. Netclos [7] and the Remote Method Invocation (RMI) in Java are some mechanisms used that can lead to a safer thread system.

There exist now three models used for parallel programming. The model is the programmers view of the system. This defines how a concurrent program has to be written.

A physical implementation is how the hardware is build. This can be different from the model. If a model is used that is very different from the hardware this can also be implemented, although it will not always be efficient.

2.4.1 Multiple processors, shared memory

In this model the programmer gets the illusion of a single shared memory. All the processors work in that shared memory. An example of this model are the POSIX threads in a Unix system. This model can even be implemented in a multiprocessor with distributed memory, but the implementation will be very complex and inefficient. This model is easier to program than the next one. Since the programmer will not see that objects can be distributed over more computers.

2.4.2 Multiple processors, distributed memory

This model is completely different since it assumes only communication via messages between different sequential processes. This can be done for example through communication via sockets, even between different computers. This is harder to use since the programmer will need to create some protocol for the different processes to communicate. This is more complex than calling a method as in the next model.

2.4.3 Remote objects

This contains elements from the two previous models. Each computer can have multiple threads in its own memory. These threads can reference objects in the local memory. But it is also possible to have references to objects in memory of another computer. Those remote objects can only be manipulated via an interface of methods that the programmer has to define. If one of those methods are invoked, it is executed in the remote machine.

This last model is at this moment used in architectures like CORBA[12] and the Java specific RMI[9] to build distributed applications over a network.

In these two architectures remote objects are used via an interface. In RMI a Java interface is used while in Corba a special interface description language (IDL) is used.

These can also be used in a shared memory multiprocessor instead of a network. It will even save some time by avoiding the network latencies.

2.5 How to build a model with safety as an objective?

The three previous models were not made with the safety requirements that have been defined before. But the ideas can be used to make a safer system for parallel programming. It is desirable to choose one model and stay with it for all hardware implementations, since rewriting an application for another model will be a time consuming process.

This means that the prefered model should be usable in an efficient way on most physical implementations. The model of remote object can be used in hardware with shared memory in an efficient way. This makes the remote object the most interesting model.

The remote object model can be made for example by running multiple JVMs in a single computer and using RMI for the communication between those JVMs. But this will cost some performance because of the following reasons:

- context switches
- interprocess communication

This makes the remote methods a lot more expensive to use than a local method. This makes the shared memory model more efficient than the remote objects model. But the shared memory model is difficult to debug since the programmer has to find all *critical sections*. The remote objects

```

class UpCounter
  {
  static final UPPERLIMIT=5;
  private int count=0;
  public void inc() // NOT synchronized
    {
    if (count<UPPERLIMIT)
       count++;
    }
  }
```

Listing 2.1: Critical section example

time	thread A	thread B	result
0			count==4
1	count < 5		true
2		count < 5	true
3	count++		count == 5
4		count ++	count == 6

Table 2.1: Trace of a critical section problem

model has also critical sections but since global variables are not shared, there will be less. This does not make them less important.

2.5.1 Critical Sections

The problem with concurrency are the critical sections. These are pieces of code that manipulate an object that is shared between different threads. If more threads make an object change at the same moment, it can make the object inconsistent. This can be seen in the example in Listing 12 which is intended as a sequential program but used in parallel. The trace of the problem is shown in Table 2.1 with two threads.

The problem that Table 2.1 shows can be explained as follows. The counter has the value 4, in a situation with two threads called A and B. A executes the method `inc()` and executes the test. Then B executes the same test before A executes the increment of the `count` variable. This means that both threads can increment the counter and make the counter have the value 6 which is not supposed to be possible. This kind of bugs is very hard to find in a big program.

This kind of problems will appear in a random way, since the situation described above is not very easy to reach. Therefore finding these problems is very time consuming and not always successful. In almost any case a programmer will use libraries, which results in more chances to create such problems. It is very optimistic to expect that all libraries have their critical sections protected, since many libraries have been developed for sequential programs, so they use global[1] variables that can cause critical sections problems since different threads will be allowed to access the same objects.

2.6 Possible solution

A possible way to make threads more safe for programmers is proposed in the next chapter. This is based on the original paper[8] about safe-threads. But with some examples the model will be more clear for a programmer that wants to use the safe-threads.

[1]static variable in Java

Chapter 3

The Safe-threads Model

This chapter will explain the safe-threads model. The semantics of this model will be explained by some examples that were used to test the implementation during the development. A more detailed explanation of some classes and methods is given in the next chapters.

3.1 Static Variables

Object Oriented Languages like Java[5] have variables that belong to a class instead of an object. They are defined with the modifier `static`. These variables are also referred as class variables.

If multiple threads are used in Java, the static variables are shared between the threads because the classes are also shared. This can lead to many critical sections in the most unexpected places.

3.2 Logical Java Virtual Machine

An LJVM is a conceptual JVM that can have only one active thread and has its own static variables. Different LJVMs can share a single JVM but each LJVM has its own separate static variables.

The LJVMs can be seen as a distributed system where different JVMs are combined via Remote Method Invocation (RMI). With the extra restriction that each LJVM can only execute one thread at a time.

To give a demonstration of these separation of static variables the following program is used. This program was used during the development of the safe-threads implementation, and is modified to work in the most recent version.

```
/* FILE: MyLjvm.java */
public interface MyLjvmInterface extends Sharable {
    public void resetVal();
    public int  incVal();
}

/* FILE: MyLjvm.java */
public class MyLjvm extends LJVMBase
    implements MyLjvmInterface {
    private static int a=0;       // STATIC !

    public void resetVal() {
        a=0;
    }
    public int incVal() {
       a++;
       return a;
    }
}

/* FILE: StaticDemo.java */
class StaticDemo {
    public void run(String arg[]) {
          MyLjvmInterface theLjvm1;
          MyLjvmInterface theLjvm2;
          int l1,l2;
          theLjvm1=new MyLjvm();
          theLjvm2=new MyLjvm();
          theLjvm1.resetVal();
          theLjvm2.resetVal();

              theLjvm1.incVal();
          l1=theLjvm1.incVal();
          l2=theLjvm2.incVal();
35            System.out.println(
36             "the static variable in each LJVM");
37            System.out.println(" theLjvm1.a == "+l1);
38            System.out.println(" theLjvm2.a == "+l2);
39        }
40 }
```

Listing 3.1: Static Variables Demo

```
%java Safe StaticDemo
values of the static variable in each LJVM
 theLjvm1.a == 2
 theLjvm2.a == 1
```

Listing 3.2: Output of StaticDemo

The output in Listing 5 of the previous program (Listing 41) would be two times a 3 with a normal Java semantics. But in a system with the safe-threads the result will different be for each LJVM, resulting in $\{2, 1\}$ because the static variable of the `MyLJVM` class (defined in line 10) is not shared between different LJVMs.

In this example it works like the variable `MyLjvm.a` would not be static but an instance variable. A static variable will be shared by all objects that belong to the same LJVM. The example shows that the use of an LJVM is very similar to other objects. If a method in `MyLJVM` would create objects those objects would be part of the LJVM that created them and share static variables in that LJVM. In that situation the keyword `static` has more significant result than in the small example.

The output listing shows also the different way to start a program with safe-threads compared to a program with Java semantics. The `Safe` will make all the needed work to activate the safe-threads. After the word "Safe" a class name and arguments follow like for a normal Java program. In StaticDemo.java a method run (line 23) is defined instead of `main`, this makes another difference with the normal system where always a static method is defined.

In an LJVM it is possible to have references to objects in another LJVM. These objects are called *foreign objects* The variables that reference to a foreign object can only have an interface that extends `java.lang.Sharable` or `java.lang.Object` as a type.

An interface does not have instance variable. This will make it impossible to concurrently access the data of a *sharable* object. The only way to access instance variables of a foreign object is through a method call that is mutual exclusive since only one thread can be active in an LJVM.

The variables `theLjvm1` and `theLjvm2` in the `run` method of class `StaticDemo` in Listing 41 are an interface type because the objects they refer to are in their own LJVM.

3.3 Sharable Objects

Instances from a class that implements the new `java.lang.Sharable` interface (sometimes called `Sharable`). The instances of such a class are the only objects that can be referenced in another LJVM than the one they belong to. An object referenced in another LJVM is called a *foreign* object.

Sharable classes need to implement the interface `Sharable` or an interface that extends from it. Because a foreign object can only be referenced by such an interface. These methods of an interface that extends from the `Sharable` interface are called sharable methods. The instance variables of an object in another LJVM are not accessible, because the only correct way to have a reference to foreign objects is via an interface that extends from the `java.lang.Sharable` or a variable of the type `java.lang.Object`.

3.4 Calling a foreign method

Since a variable can reference a foreign object it is also possible to send methods to that foreign object. This is done in a way similar to RMI. The method is executed in the LJVM that contains the object. The invariant stating that in an LJVM only one thread can be active makes that the method call has to wait until the LJVM is free. This can lead to an infinite lock, if the other thread, that is waited for, does not leave the LJVM of the called object. Finally, this can cause a deadlock where all safe-threads are waiting on each other.

This problem is easier to find that an unprotected critical section. So it is an improvement of the current thread model.

It is only possible to call methods of a foreign object if those methods belong to an interface that extends `java.lang.Sharable`. To call a method in a foreign object, that method has to be part of the sharable interface that is used to reference the object. An interface that does not extend the `java.lang.Sharable` can not reference a foreign object.

The arguments of a foreign method call are handled similarly to what happens during a remote call in RMI. The primitive types are not handled in a special way. The sharable objects are not copied. The objects that are not sharable are deep copied in a way that is explained in Subsection 3.4.1.

The result of this copied arguments can be seen in the example of Listing 26. In this example, an argument of type `java.awt.Point` is passed to a method of an object in another LJVM. The argument is not sharable meaning that it has to be copied to have a local copy in the other LJVM. All modifications in the LJVM that executes the method will not affect the actual parameter in the LJVM that called the method. The result of this can be seen in Listing 6. This is the same behavior as when invoking a method in a remote object with RMI.

3.4.1 Copying arguments

This copying of arguments works with a depth first copy of all the objects in actual parameters that have to be copied. Each object that is copied is stored in a hash-table with the original reference as the key and the copy as the associated value. A hash table is used because this type of dictionary gives a high performance, while the memory it uses is only used for a short time. Before copying an object it is checked if the copy does not already exists in that hash-table to prevent multiple copies of the same object, since this would result in a different structure than the original. This problem is shown in Figure 3.1 where the objects A' and B' appear after a copy of an array without use of this hash-table.

This hash-table is made before the first argument is handled and destroyed after copying the last one. This allows to copy complex structures like for example a circular list, or actual parameters that refer direct or indirectly to the same object. This can take some time but it is faster than a normal serialization and deserialization because there is no network transport in between like it would be the case with RMI.

```
/* FILE: MyLjvmCopyInterface.java */
public interface MyLjvmCopyInterface extends Sharable {
    public void foreignMethod(java.awt.Point a);
}

/* FILE: MyLjvmCopy */
class MyLjvmCopy extends LJVMBase
    implements MyLjvmCopyInterface {
    public void foreignMethod(java.awt.Point a) {
        a.move(5,5);
        System.out.println("In 'foreign' LJVM  :  "+a);
    }
}

/* FILE: CopyDemo.java */
class CopyDemo {
   public void run(String arg[]) {
        MyLjvmCopyInterface theLjvm;
        java.awt.Point p=new java.awt.Point(10,10);
        theLjvm=new MyLjvmCopy();
        System.out.println("Before Foreign Call P="+p);
        theLjvm.foreignMethod(p);
        System.out.println("After Foreign Call  P="+p);
     }
}
```

Listing 3.3: Copying Argument Demo

```
%java Safe CopyDemo
Before Foreign Call P=java.awt.Point [10,10]
In 'foreign' LJVM  :  java.awt.Point [5,5]
After Foreign Call  P=java.awt.Point [10,10]
```

Listing 3.4: Output of CopyDemo

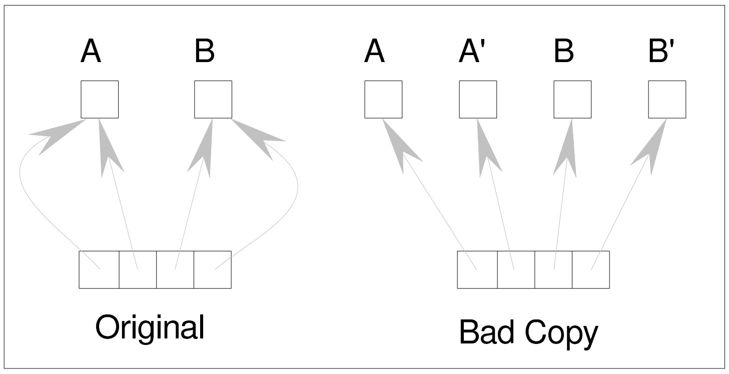

Figure 3.1: Copying complex structures

3.4.2 Type of the formal parameters

If a formal parameter receives a shared object it should be an interface type that is extended from the `java.lang.Sharable` and the interface should also be implemented by the class of the actual parameter, or the formal parameter can be a `java.lang.Object` as well.

This is because foreign references should be interfaces to prevent access to the instance variables. The class `java.lang.Object` is the most generic class therefore it is also allowed.

If the formal parameter is a class instead of an interface it would give access to the instance variables of foreign object. To prevent this, two different solutions are possible: The most strict one would be to forbid this with an exception at runtime or an error at compile time if a compiler can detect this. A less strict way would be to copy the object in that case to the LJVM that will execute the method. This can lead to some unexpected results.

Which of those solutions is the best is at this moment not decided. The prototype can be some guide to see the implications of both options. Both solutions are possible by a small modification in the source code of the modified Kaffe (see Section 6.6).

3.5 LJVMs spreaded over multiple JVMs

It is possible to distribute LJVMs over multiple JVMs in a similar way as RMI. This means not that an LJVM would be split in different parts, only that the LJVMs are no longer in the same JVM. This can be done for efficiency reasons where the power of more processors can be combined. The safe-threads model is very close to RMI and therefore both can be combined without big problems. It was not an objective for the prototype, but to make RMI work will not be very hard as described in section A.5. Of course it is important to make sure that only one thread can be active inside an LJVM at the same moment.

3.6 Monitors in general

The idea of a monitor was originally used for operating systems to allow multiple programs to use shared resources, for example storage devices.

In [1] Brinch-Hansen introduces the concept of a *critical region*[1] that modifies a certain shared variable. A critical region can only be executed by one process (or thread) at the same moment. In [6] Hoare introduces a similar concept that he calls a *monitor*. This monitor is like an object that has private variables and can contain only a single active thread.

In the concept of monitors it is possible that a thread becomes inactive, to allow an other thread to become active. Both authors use a different approach. Hoare uses a lower level approach which only signals events via conditional variables. While Brinch-Hansen proposes a more high level approach that checks a boolean expression. This results in less efficiency, because all these expressions have to be evaluated each time a thread gives the lock away. While the approach of Hoare can be more specific by specifying which threads are awakened.

In [2] Brinch-Hansen compares the language Java with these ideas that are more than 20 years old. Java uses a signals like in the Hoare approach but with a single condition variable, this means that all waiting threads are awakened on an event. In Java exists, besides the `notifyAll`, a `notify` method that works in a first in first out (FIFO) way.

This results in a combination of the disadvantages of both propositions since the programmer has to make the checks for the boolean condition

[1]This is similar to the critical section, but it is explicitly defined by the programmer

explicitly like in the Hoare approach and it has the inefficiency of wakening all waiting threads as in the Brinch-Hansen approach.

3.7 An LJVM as a Monitor

The LJVM acts like a monitor, this means that only one safe-thread can be active in an LJVM, as written before in this chapter. A safe-thread can give up the monitor to another safe-thread by using a `waitCond` call similar to the wait of Java.

3.8 A more practical example

With the safe-threads each LJVM has its own `System.in` and `System.out` that are initially forwarded via a stub/skeleton mechanism to a common input and output as described in Section 7.5. This can be changed to another configuration. This example is based on the one in [8].

It is possible to connect the `System.in` to the `System.out` of another LJVM to create a very similar situation as made via pipes in Unix.

The Unix program `grep` filters a stream so that only lines that contain a pattern can pass the program. This stream can be a given file.

The Unix program `wc` prints the number of bytes, words and lines from a stream. The name is an abbreviation of "word count".

A powerful methodology in Unix consists in solving complex problems by connecting simple commands through pipes, as in the next example:

```
grep hello notes | wc
```

This line will result in reading the file "notes" sending only the lines that contain "hello" to the wc program that will print the number of bytes, words and lines that could pass the grep filter.

The program `grep` shares no variables with the program `wc` the only relation they have is that the output of `grep` is used as input for `wc`.

This situation can be made with the safe-threads in an easy way since an LJVM does not share anything with another LJVM unless the programmer makes an explicit connection between them through shared objects that can also be exchanged via a registry (see subsection 7.4.1) or via method-call of an LJVM interface.

```
interface GrepLjvmInterface extends Sharable {
    public void setInfile(String infile);
    public void setOutPipe(PrintStreamInterface out);
    public void start();
}

interface WcLjvmInterface extends Sharable {
    public void setInPipe(InputStreamInterface in);
    public void start();
}

interface IOPipeInterface extends Sharable {
}

class IOPipe
    implements IOPipeInterface,
               InputStreamInterface,
               PrintStreamInterface {
    // not implemented here
}

class GrepLjvm
    extends LJVMBase
    implements GrepLjvmInterface {
    java.io.InputStream inFile;
    public void setInfile(String infile) {
        inFile=new java.io.FileInputStream(infile);
    }
    public void setOutPipe(PrintStreamInterface out) {
        System.setOut(out);
    }
    public start() {
        Grep grep=new Grep(inFile,System.out);
        new GrepSafeThread(grep).start();
    }
}
```

```

class WcLjvm
    extends LJVMBase
    implements WcLjvmInterface {
    public void setInPipe(InputStreamInterface in) {
        System.setIn(in);
    }
    public start() {
        Wc wc=new Wc(inFile,System.out);
        new WcSafeThread(wc).start();
    }
}

public class PipeDemo {
    public void run (String arg[]) {
        GrepLjvmInterface grep =new GrepLjvm();
        WcLjvmInterface wc=new WcLjvm();
        IOPipeInterface pipe=new IOPipe();
        grep.setInFile("notes");
        grep.setOutPipe(pipe);
        wc.setInPipe(pipe);
        grep.start();
        wc.start();
    }
}
```

Listing 3.5: Piping Example

In Listing 62 is shown that it is possible to combine the sequential classes `Grep` and `Wc` by using different LJVMs. The example uses a separate LJVM for the grep and the wc this means that even if they use a shared variable they will not have any influence on each other.

It is supposed that some extra classes exist in the example but their implementation is not needed to see how pipes can be easily implemented in the safe-threads model. The needed extra classes are a sequential Wc and Grep and the classes used to start the safe-threads in the two LJVMs.

3.9 Registry

In this model LJVMs can share their Sharable objects by using a central registry in a way that is similar to RMI. The registry is accessed via an object of the class `SafeRegistry`. This 'safe' refers to the name of the model, it does not suggests that this registry is safer than the one of the RMI system.

Chapter 4

Implementation of the model in Java

In this chapter it is explained how a language can be modified to support the safe-threads model. This does not mean that safe-threads would be only implementable in that language. This is just a way to use the safe-threads. It will be possible to implement the safe-threads model in other languages as well. The syntax of such a modified language will be the same as the original version of the language but the semantics are different, therefore the new dialect will not be compatible[1] with the original version of the language.

4.1 An Implementation of safe-threads model

Since Java is at this moment a popular language that is easy to use, it will be used as the start for implementing the model. This will save a lot of time that otherwise would be needed to build a complete language.

The safe-threads model can be implemented in different ways.

- preprocessing the source code
- modifying the byte code
- modifying the JVM

Each approach has its own advantages and disadvantages. These will be very briefly discussed.

[1]In the prototype is a compatible semantics available in the null-LJVM

4.1.1 Preprocessor

The big advantage of this approach is that it is compatible with any Java Virtual Machine (JVM). Since most platforms have fast Just-In-Time compilers the generated code will be executed very efficiently. A lot of effort has been made to make this JIT compilers efficient.

This also respects the idea of a single type of JVM that will run all Java programs. This respects also the idea of running a Java-program everywhere, instead of needing a special virtual machine. The disadvantage is that this approach has some difficult parts. The preprocessor needs to make the same type analysis as the compiler that creates byte code from Java sources. The preprocessor is at this moment also being implemented, see [11].

4.1.2 Byte code modification

The byte code is an intermediate format that can be compared with machine code, but it is intended to work with a JVM instead of an 'normal' microprocessor. By changing this byte code a program can be changed, to use the safe-threads semantics.

This approach relies on tools like the Javassist[3] toolkit. However the approach has some difficulties. The next solution will be the one used to build the prototype.

4.1.3 Modifying the JVM

This approach might seem the most difficult but some things are easier when the byte code interpreter is changed compared to the two previous approaches. Because of Kaffe[13], that is an open source project, it is possible to modify its behavior instead of writing a new JVM from scratch. Kaffe can be compiled in three different ways.

- interpreter
- JIT compiler
- JIT3 compiler

The interpreter code is not very difficult to understand. Since it is written in C, it is portable, compared to more efficient assembler implementations. The Just in Time (JIT) compilers of Kaffe are more machine dependent than the interpreter and needs more specific knowledge to be modified in an efficient way. This makes that modifying only the interpreter is the easiest solution to build a prototype. The JIT is not modified and will not work at all after the modifications.

Since Kaffe has a more open license than the Java implementation of Sun, this modified virtual machine can also be distributed! This means that people can download this modified machine and experiment with the safe-threads themselves.

It has to be considered to be a prototype instead of a production tool, since interpreters are slow, and especially the Kaffe interpreter is not written to be very efficient. For example methods and fields are looked up via a sequential search algorithm. For a production tool it would be useful to modify a JIT compiler since this would result in a much faster implementation.

4.1.4 Combination

It is possible to combine two or more of the previous possibilities. This might give an easier implementation since some modifications can be handled easier in a preprocessor, others in byte code modifications. However it will be a less elegant solution because of the different steps that are used.

4.2 Modified Kaffe

4.2.1 Compatibility

The safe-threads are implemented by modifying Kaffe in a way that this system can also be used together with the already existing system of threads. The advantage of this approach is that existing programs can be used in this modified Kaffe. An important example is the compiler program that creates byte code from the source. Since this will allow to use the modified Kaffe with the compiler that was made for the normal Java semantics.

In the rest of this text the expression “modified Kaffe” will be used when comparing to the “original Kaffe” that is compatible with Java.

4.2.2 Changes

The Kaffe virtual machine was changed in the way that the safe-threads must work, since only modifying the class library would not be enough. The programmer who uses the modified Kaffe will only see the extended class library. However some of the semantics of the language is changed in all LJVMs except in a null-LJVM that has kept the original semantics and threading for compatibility reasons. This null-LJVM is implicitly created at the moment that the modified Kaffe is started.

The original semantics is not a problem since the null-LJVM can be hidden from the programmer by using a wrapper that creates an LJVM with the semantics of the safe-threads. The LJVM that is created by the wrapper is called *Initial-LJVM* because that will be the initial LJVM from the point of view of the programmer, the null-LJVM has different semantics so it will be hidden for the programmer to avoid the complication of different semantics between the null-LJVM and the other LJVMs.

The programs that require original Java semantics will not use the wrapper and use only the null-LJVM.

Chapter 5

Programmers View

In this chapter it is explained what is changed in this model from the programmers view is changed, compared to the unmodified Kaffe. It will not explain the implementation since two other chapter will handle this. One chapter will explain the modification to the Kaffe Java Virtual Machine (Kaffe JVM) and another chapter will explain the changes to the class library.

5.1 Changes to the Kaffe JVM

5.1.1 LJVMs in the Kaffe JVM

The Kaffe JVM has been changed in a way that each object will belong to an LJVM. This means that each time an object is created it will belong to the LJVM that created the object with the exception of new LJVMs that will belong to themselves.

This introduces to difference between local and foreign objects. Objects that belong to another LJVM are considered as *foreign objects*.

5.1.2 LJVM specific static variables

Each LJVM will have its own static variables for each class. This means that a static variable is only global in an LJVM and no longer will be shared by all LJVMs that exist in a single JVM.

5.1.3 Typecasting checks LJVM-rules

If an object is only typecasted inside its own LJVM the rules are not changed. If an object that belongs to another LJVM is typecasted some extra restrictions are checked.

This typecast is only possible to a variable that can reference a foreign object. This is only possible possible by a variable of one of the following types.

- sharable interface that is implemented by the class of the object
- java.lang.Object

```
LJVM ljvm1=foreignLJVM;        // correct
Object ljvm2=foreignLJVM;      // correct
LJVMBase ljvm3=foreignLJVM;    // Not correct !!
```

Listing 5.1: Examples of this extra check

The sharable interfaces are interfaces that extend from `java.lang.Sharable`.

5.1.4 Foreign calls

When a foreign method is called, the arguments that reference to objects that are not sharable will be copied. This will avoid that another LJVM gets a reference to a foreign object that can not be shared.

The formal parameters that receive a sharable object should be able to contain a foreign references. This restriction is the same as the one for the type cast. If this rule is not respected two possible solutions are possible, these are explained in section 6.6.

5.2 Library changes

The class library is extended with some extra classes and interfaces. The following are the most important to the programmer. (The package `java.lang.safeutil` contains some extra classes that will be explained in a separate chapter.)

- Safe

- java.lang.Sharable
- java.lang.LJVM
- java.lang.LJVMBase
- java.lang.SafeThread
- java.lang.safeutil.SafeRegistry

5.2.1 Safe

This class is used to start the safe-threads version of Kaffe. It will handle the needed initializations to allow standard I/O to the programmer. It has already been used in section 3.2 for activating the safe-threads system for the user.

The programmer has to provide a class that has the method `void run(String args[])` as the entry-point to start the program.

5.2.2 Sharable

This is one of the most important interfaces. This is used to mark objects that can be shared between different LJVMs. Objects that are sharable can be referenced even if they are in another LJVM. This can be compared with the `java.rmi.Remote` interface in the RMI-system.

This interface can be extended to interfaces that contain methods. These methods can be used to allow method calls to objects in another LJVM. The methods of such an interface are called *Sharable methods*.

5.2.3 LJVM

This interface can be used to reference to a programmer defined LJVM. It will be extended by the programmer to an interface that will be implemented by a subclass of `LJVMBase`. This allows access to the methods of a foreign LJVM.

5.2.4 LJVMBase

This class can be extended to allow the creation of a specialized LJVM by the programmer. It contains also some static methods that are useful for the programmer.

1. static SafeThread currentSafeThread()
2. static LJVMBase currentLJVM()
3. static void waitCond()

The first two give the current SafeThread or LJVM as their name suggests. The last one can be used to release the monitor.

5.2.5 SafeThread

This is the class used to create a thread. The normal `Thread` class should not be used by the programmer.

5.2.6 SafeRegistry

To share objects that are sharable between different LJVMs a global registry is used. The `SafeRegistry` has the following methods that have the same functionality as in RMI-registry:

- String[] list()
- Sharable lookup(String name)
- void bind(String name, Sharable obj)
- void rebind(String name, Sharable obj)
- void unbind(String name)

The registry is used by creating an object of this class. No extra initialization by the programmer is needed. This object can be used in a similar way as the registry that RMI provides.

The following exceptions can be thrown by the methods of this registry. They are equivalent as the one thrown by the RMI registry.

- AlreadyBoundException
- NotBoundException

The name SafeRegistry does not mean that the registry is more safe than the RMI registry, but it refers to the model of safe-threads.

5.2.7 Illegal Accesses

In this implementation it is possible to reference a foreign LJVM by a variable of a class type. If this variable is used to access that LJVM one of the following exceptions will be thrown.

- IllegalForeignFieldAccessException
- IllegalForeignMethodCallException

Other implementations of the safe-threads model should be able to prevent assignment of the following form `MyLJVM m=new MyLJVM()` and will not need these exceptions at all.

Chapter 6

Modifications of the Kaffe interpreter

The model is implemented in an interpreter by changing the way that opcodes are executed. In this chapter it will be explained which instructions were modified and why the modifications are needed. The Kaffe Java Virtual Machine is modified to support the concept of safe-threads in all LJVMs except the null-LJVM. This decision has been made for compatibility with current normal programs running in the same environment. Also, objects from the null-LJVM are not protected against access from other LJVMs to avoid problems with the internal classes of Kaffe like String.

The implementation needs to make sure that the following invariants are respected:

- A foreign object can be known only through its sharable interface. This is similar to RMI where a remote object can only be known through its remote interface. This is implemented by introducing runtime checks in the instructions that can violate this invariant.

- There can be only one active thread running in a single LJVM, this has no equivalent in RMI. It is implemented by making each LJVM a Hoare monitor.

- Each LJVM must have its own set of static variables for each class which is used in that LJVM. This is implemented by associating a dictionary to each LJVM that refers to that static variable for each class.

The efficiency of the implementation is also an issue. The use of an interpreter is not efficient but this can been seen as a first step to build a JIT compiler after this prototype.

- All LJVMs run on the same machine address space.
- The code implementing a class which is used by several LJVMs must be shared, while their static variables should be separated.
- Pointers to foreign objects must be represented directly by its address. RMI uses stubs for this.
- Foreign calls must be executed by the same Java thread that makes the call. RMI needs an other thread because threads can not move to another address-space.

6.1 LJVM id

Each object gets a number that identifies to which LJVM it belongs. This is not visible for the programmer directly, but it is a fundamental change. This number will be mapped to a reference to a Java object that represents the LJVM for the programmer. This number also is the index in an array that contains for each LJVM a dictionary to find the static fields for the classes used in that LJVM.

6.1.1 Null-LJVM

The null-LJVM is the LJVM that is entered at the moment that Kaffe starts. The internal id is 0, so it is called null-LJVM. It is not represented by an object in Kaffe but by a null if the current LJVM is requested in that LJVM.

6.1.2 Initial LJVM

The null-LJVM is different from what is called in this text the *initial* LJVM. The initial LJVM is an LJVM created by the `Safe` class to run applications designed to be safe, and to hide the null-LJVM from the programmer because the threads running in the null-LJVM are not safe-threads.

6.2 Type-casting an object

The byte code instruction `CHECKCAST` has been extended with some extra checks in the case that an object from another LJVM is checked and that object does not belong to the null-LJVM.

The following checks are all performed first before the normal casting rules are checked.

- object is Sharable
- target class is one of these:
  - interface that extends `java.lang.Sharable`
  - `java.lang.Object`

This means that in the examples in Listing 4 the two first will throw an exception. Even if the foreignPoint should not be even possible to get, it will be forbidden.

```
Point p=(Point)foreignPoint;
LJVMBase b=(LJVMBase)foreignLJVM;
LJVM b=(LJVM)foreignLJVM;
```

Listing 6.1: type cast examples

The first will fail because Point is not Sharable. The second fails because LJVMBase is a class, not an interface nor `java.langObject`. The third is the only correct one since `java.lang.LJVM` is an interface that extends from `java.lang.Sharable`.

If the object belong to the null-LJVM only the normal rules from Kaffe are checked.

6.3 Object Creation

Objects and arrays are created by the byte code instructions `NEW`, `NEWARRAY`, `ANEWARRAY` and `MULTIANEWARRAY`. These instructions have been modified. This affects code like: `Object o=new java.awt.Point();`

When an object is created it belongs to the LJVM where it was created. There is an exception:

When a class that extends the `java.lang.LJVMBase` class is instantiated a new LJVM id will be given to that object. This results that an LJVM belongs to itself.

6.4 Static fields

At byte code level the static fields are accessed by the instructions `GETSTATIC` and `PUTSTATIC`. The implementation of these instructions has been altered.

Each LJVM uses different static variables for a class. This implies that the class constructor must be executed multiple times, one time for each LJVM.

The class constructor will be executed before the first time one of the following instructions are executed:

- accessing a static variable
- invoking a static method
- instantiating a class
- executing the class constructor of a subclass

These rules are always respected in the original Kaffe since the class-loader is normally executed when the class is loaded.

If these rules are not respected the code in Listing 11 will not have the same result as in the unmodified Kaffe. For example, the value of c is supposed to be 1 (y=0; a=B.y++; c=B.y++) but can become 0 (a=B.y++; y=0; c=B.y++) if the two first rules are both ignored.

This execution of the class constructor generates some problems that are solved by considering the null-LJVM always as the current LJVM when a class is loaded.

6.5 LJVM specific static fields

Each LJVM has its own static variables for a class. This is implemented by a two step lookup in dictionaries (see Figure 6.1).

The internal LJVMid is an index into via an array (L in the figure) to get an LJVM specific dictionary ($CS1$,$CS2$ in the figure). The key in that dictionary is a pointer of to class (class C in the figure). While the value

```
class A {
  static int a=B.y++;
  int c =B.y++;
  }

class B {
  static int y=0;
  }

// what is the value of c ?
```

Listing 6.2: When invoke the class constructors?

points to a static area ($SA1$ and $SA2$ in the figure) that contains all static fields for a class in an LJVM.

When a static field is accessed, the memory of the static data of that class will be looked up as explained here after. First by binding the right dictionary for an LJVM. This can be done by looking up the value for key '3' in the array L if '3' is the internal LJVMid of the LJVM that accesses a static field of the class C. This results in a reference to a dictionary. In that dictionary the class C is used to look up the static area of the class in the dictionary $CS1$. After this, the offset of the field gives access to the memory of the variable.

This is more complex than the normal way (see Figure 6.2) without LJVMs. The field already contains the address to access its data instead of an offset. The mapping H in the figure is only executed at the moment a class is loaded.

Since static fields are specific to an LJVM in the safe-threads model, these extra steps are needed, each time a static field is accessed if classes are shared between LJVMs.

6.6 Method calls

The only way for a thread to reach another LJVM is via a method call of an interface. The method calls are the most important part of the safe-threads model. If a method is called in a foreign object some extra work is done.

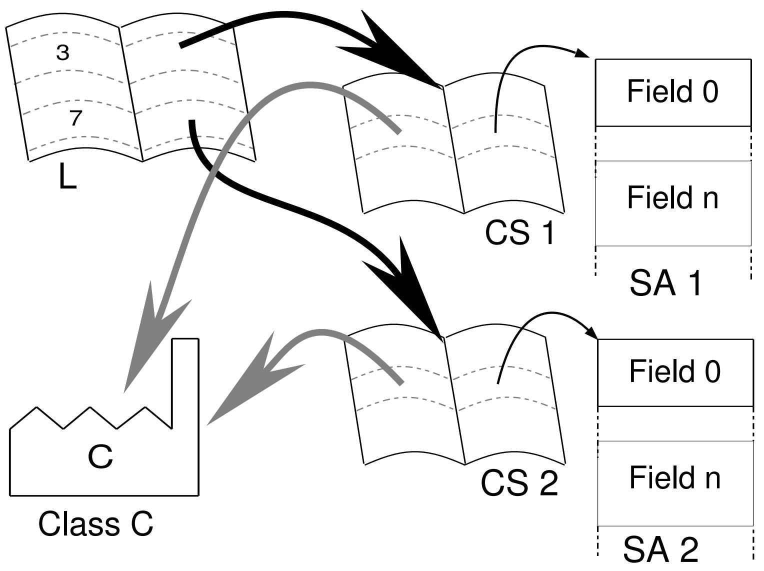

Figure 6.1: Modified static field lookup, separate static variables for each LJVM

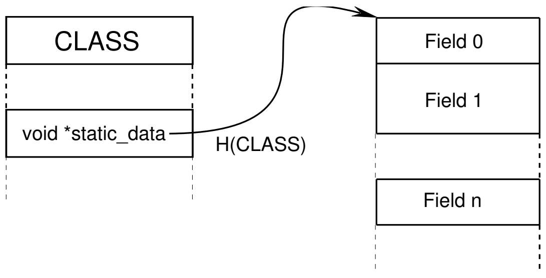

Figure 6.2: Unmodified static field lookup, static variables belong to a class

The parameters are handled with respect to their type:

- primitive types
- objects from a class that does not implement the `java.lang.Sharable`
- objects from a class that implements the `java.lang.Sharable`

As long as all the parameters have primitive types there is no problem, since their values can not be changed. But if some arguments refer to an object, things can become more complicated.

Objects that are not sharable[1] are deep-copied to the remote LJVM. If they are sharable they can be passed by reference if the formal parameter that receives them is an interface or a variable of the type `java.lang.Object`. If the receiving formal parameter has a class type, two different approaches are possible: it can be copied like an object that is not-sharable, or an exception can be thrown. Throwing an exception is a very strict policy that can prevent unexpected effects, since modifications in a possible copy can be lost.

Both options are supported in the prototype, it can be chosen by defining or not defining `STRICT_LJVM` in the file kaffe-1.0.5/kaffe/kaffevm/ljvm.h before compiling the modified Kaffe.

It is important that a reference to an object can not just be passed as a real object, since this would give a foreign LJVM access to the instance variables and therefore allow concurrent access to these variables. The modified Kaffe will not allow access to the instance variables, but by using only a sharable interface for foreign objects, the illegal access to instance variables can be detected during the compilation.

An interface does not have instance variables and has therefore not this problem. Even the class `java.lang.Object`, can later not be casted to another class. The modified `CHECKCAST` byte code instruction that will check at runtime if a type-cast is allowed. It will prevent to assign foreign objects to a variable of a class type different from `java.lang.Object`.

6.6.1 Restriction of foreign accesses

The model allows only references to foreign objects through variables having the type `java.lang.Object` or an interface type extending

[1] their class does not implement `java.lang.Sharable`

`java.lang.Sharable`. Unfortunately it is not possible to enforce these restrictions in all cases with only modifying the JVM. The problem that appears is when a subclass `java.lang.LJVMBase` is instantiated.

Because the class is known at compile time, there will be no `CHECKCAST` byte code instruction to check if that object can be referred by the variable that will refer to that LJVM. (see the example in Listing 14)

A solution in this implementation is that other byte code instructions are modified to prevent an illegal access to those objects:

1. GETFIELD
2. PUTFIELD
3. INVOKEVIRTUAL
4. INVOKEINTERFACE

The first three instructions will throw always an exception when they try to access a foreign object whose type extends from LJVMBase. The last one will check if the interface is extended from `java.lang.Sharable` if the object is in a foreign LJVM.

The result of these restrictions is that if a variable that is not allowed to contain a foreign reference contains such a reference, that variable can not be used to access the instance variables of the object or to invoke any methods at all. This results in a reference that can only be assigned to another variable that can be of the correct type.

In the example of Listing 14 the assignment to `m` in aMethod will be possible (it should not, but this can not be prevented), the method call and the assignment to the instance variable will both throw an exception. Because the variable `m` should not contain such a reference. That variable should be of an interface type that extends from `java.lang.Sharable`.

Another possible solution to this problem would be to forbid that LJVM objects are created with `new` and provide a factory for constructing LJVMs. This has two negative effects.

- less intuitive to the programmer
- reflection needed to allow easy construction

```
class MyLJVM extends LJVMBase {
        public int a;
        public void method() {
          // a method body
        }
}

void aMethod() {
 MyLJVM m = new MyLJVM();
 m.method();  //  IllegalForeignMethodCallException
 m.a = 5;     //  IllegalForeignFieldAccessException
}
```

Listing 6.3: Restricted access to foreign objects

6.7 Java Native Interface

The Java Native Interface (JNI) is modified to have the minimal support to make the prototype work with the native methods that are added to the class library. However it is not seen as an objective of this work to make JNI working in the modified Kaffe, since the JNI is not aware of different LJVMs (see also Section A.8).

Chapter 7

Modifications in the Kaffe Libraries

The class libraries had to be enriched and adapted to support the safe-thread model. Most of them are in a package `java.lang.safeutil`. Some of the elements are already discussed in the programmers view in chapter 5 and do not need extra explanation here.

7.1 List of modifications

This is a complete list of extentions and modifications.

7.1.1 Classes and Interfaces

The classes and interface marked with an asterisk are classes existing in standard Java but are modified in this prototype. The unmarked classes and interfaces are completely new.

- Safe
- java.lang.System *
- java.io.PrintStreamInterface
- java.io.PrintStream *
- java.io.InputStreamInterface
- java.io.InputStream *

- java.lang.Sharable
- java.lang.LJVM
- java.lang.LJVMBase
- java.lang.SafeThread
- java.lang.safeutil.InitialLJVM
- java.lang.safeutil.InitialLJVMInterface
- java.lang.safeutil.InputStreamForward
- java.lang.safeutil.InputStreamSkeleton
- java.lang.safeutil.InputStreamStub
- java.lang.safeutil.PrintStreamForward
- java.lang.safeutil.PrintStreamSkeleton
- java.lang.safeutil.PrintStreamStub
- java.lang.safeutil.SafeRegistry
- java.lang.safeutil.SafeRegistryInterface
- java.lang.safeutil.SafeRegistryServer

7.1.2 Exceptions

The following exceptions have been added to the class library.

- java.lang.safeutil.ForeignException
- java.lang.safeutil.AlreadyBoundException
- java.lang.safeutil.NotBoundException
- java.lang.safeutil.IllegalForeignFieldAccessException
- java.lang.safeutil.IllegalForeignMethodCallException

7.2 Sharable

This is an interface use to mark objects that can be accessed by foreign LJVMS. See also subsection 5.2.2.

7.3 LJVM and LJVMBase

These are already explained in Subsections 5.2.3 and 5.2.4.

In this implementation the LJVMBase has the following extra methods, that are not intended for the normal programmer.

- native static Sharable getGlobal(int key)
- native static void setGlobal(int key, Sharable obj)
- int getLJVMid()

The first two allow access to a low level registry this is explained in subsection A.9 The method `getLJVMid` might be useful for debugging the modified Kaffe, see section A.1 for a description.

7.4 Global registry

The communication between different LJVMs is made with two layers. The low level implementation is written in native code. This is specific to this implementation. This low level registry works with numerical keys that are called *gates*, for a more detailed description see section A.9.

Based on this low level implementation is a high level registry built that is more user friendly.

7.4.1 SafeRegistry

The `SafeRegistry` mimics the functionality of the RMI registry. This is not a coincidence since the safe-threads model and the RMI are from the programmers point of view very similar. The name SafeRegistry does not mean that the registry is more safe than the RMI registry, but it refers to the model of safe-threads. It is also safer than the gate system, that is used to implement the `SafeRegistry` in this implementation.

The difference in the `SafeRegistry` with the RMI registry is that the names can be strings instead of URLs. This registry will be even safer to use if the keys are chosen in a structured way like URLs in RMI. The use of structured keys is strongly advised but not mandatory as in RMI.

7.4.2 Implementation of SafeRegistry

Each time an object of this class is instantiated, it tries to find an object of the type `java.lang.safeutil.SafeRegistryServer` at gate 11071302[1]. If that server-object is not found it is created and published at that gate. This means that only one instance of that server is shared by all instances of the `SafeRegistry` class. The gate is specified in the interface `java.lang.safeutil.SafeRegistryInterface` as a static final. The programmer should not use that gate for other purposes.

The `SafeRegistryServer` contains the real registry while the `SafeRegistry` objects forward their calls to that global server. Because the server exists only in one LJVM, the objects that are stored in the registry have to be sharable to prevent that unsharable objects are copied.

The SafeRegistry will be a little slower than the low level gate system, but it is easier to work with strings than with numbers as keys. The lower performance will not be a problem since it is not heavily used and has no network delay as in the RMI-registry.

7.5 I/O system via a stub/skeleton construction

A problem with the LJVM system is that only one LJVM has the standard I/O via the static variables `in`,`out` and `err` in `java.lang.System` since a class has separated static variables for each LJVM. A dirty way to solve this problem would be that the `System` class would share its static variables among all LJVMs and be therefore an exception to the rule. However this would not be desirable because this would not allow a kind of piping between the LJVMs as proposed in Listing 62 and [8].

In the chosen approach each LJVM has its own `in`, `out` and `err` static variables, that use the same input or output, but the programmer can then change this I/O to other objects.

The best way to achieve this, is by sharing the I/O from the initial LJVM via a stub/skeleton mechanism that forwards the I/O function from any LJVM to the null-LJVM. The `java.lang.System` class is modified in a

[1]This number forms the date of a war

way that the following static variables are from a different type: `System.in` is changed from InputStream to InputStreamInterface and `System.out` and `System.err` are changed to a `PrintStreamInterface`. This allows to programmers to change the I/O for each LJVM separately (see Figure 7.1).

The programmer uses a stub that appears as a `PrintStream` object. That stub forwards all actions to a skeleton that forwards the calls to a real `PrintStream` object. The result is that the programmer can forget about the stub and the skeleton, and use the stub as if he was using the real input or output.

7.5.1 Design

Two classes are derived from PrintStreamForward: `PrintStreamStub` and `PrintStreamSkeleton`. The only difference is their constructor. The skeleton publishes itself on a specified gate and forwards to a specified PrintStream-Interface. The stub forwards to a published PrintStreamInterface, this can be the skeleton. The stub can be omitted but this would make things less symmetric, since at the side of the stub would only the foreign skeleton be used. Figure 7.1 shows that only the communication between the stub and the skeleton crosses the LJVMs (big grey arrow). This results in a situation where the programmer will only know about the stub.

It is possible to implement this in an easier way by making PrintStream an interface and let System.out point to the System.out of the null-LJVM but this would be less transparent for the programmer. It is not possible to make new PrintStream objects if it is changed to an interface. To reference to foreign objects it has to be a sharable interface. Because of this complication the stub/skeleton system has been chosen. It allows compatibility with the 'unmodified Kaffe' this will be easier for the programmer and allows to reuse some existing part of legacy code.

7.5.2 PrintStream

The class `java.io.PrintStream` implements now the PrintStreamInterface. But the `java.lang.safeutil.PrintStreamForward` also implements this interface.

The skeleton should be created in the first LJVM and only once for each service. This can be automated. For both `System.out` and `System.err` a skeleton is needed. The skeleton for `System.out` is published on gate 1 and

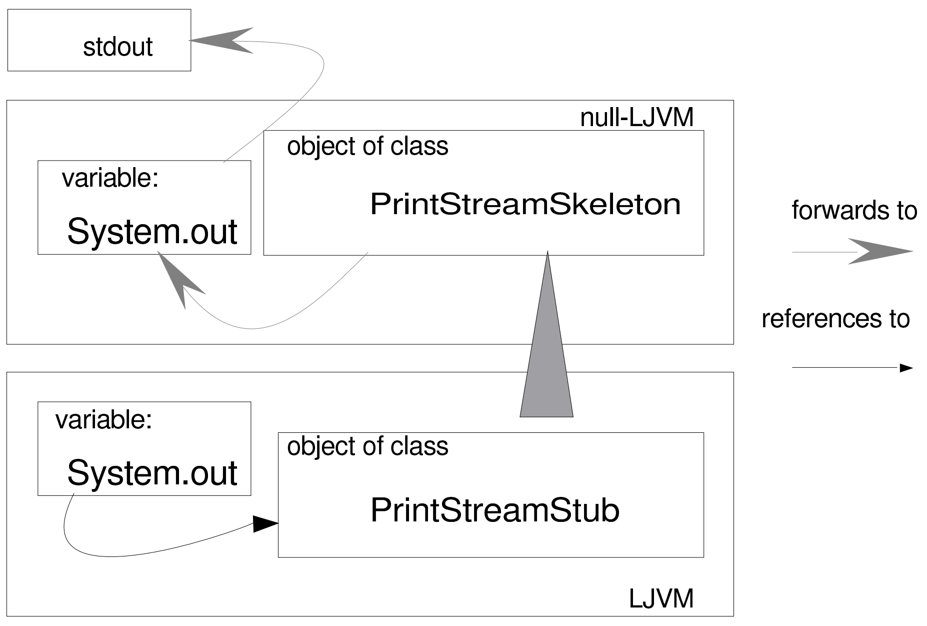

Figure 7.1: stub and skeleton

the skeleton for `System.err` is published on gate 2 this is just a convention but the stub need to check on the correct gate.

7.5.3 InputStream

The Inputstream is similar to the PrintStream. The InputStreamSkeleton is published on gate 0 where each InputStreamStub can find it.

7.6 System

The `java.lang.System` is modified to create stubs for the static variables `System.in`, `System.out` and `System.err` if used in any LJVM except the null-LJVM. This allows an easy way for I/O in any LJVM. The skeletons for those streams has to be created in the null-LJVM, this is hidden from the programmer as well by the `Safe` class, see Section 7.7.

7.7 Safe

This class is used to start the safe-threads model as shown in section 3.2 and subsection 5.2.1. To start a safe-threads program the following syntax is used:
`java Safe MyClass arg1 arg2 .. argn`

This will start the method `run` in the class `MyClass`. This method "`void run(String args[])`" is not a static method like the `main` in the unmodified Kaffe.

The program is started in new created LJVM that is created by a class `java.lang.safeutil.InitialLJVM`. This will hide the null-ljvm from the programmer.

The string arguments from the `main` method of `Safe` are passed to the contructor of the `InitialLJVM` class. Then all the arguments are stored in that LJVM-object.

The next method that the `Safe` wrapper calls is the `startup` method that creates via reflection an object of the type specified in the first of the command line. Finally the method `run` of that class is called and the strings that were passed via the constructor are now passed to the run method. This system makes the 'static' unnecessary.

Another important task of the `Safe` class is to create the Skeletons for the `in`, `out` and `err` streams of `java.lang.System` class in the null-LJVM. This allows the programmer to use these streams in each LJVM in the expected way. It is possible in each LJVM to redirect a stream, this will not affect the other LJVMs.

7.8 Exceptions

The safe-threads prototype has some extra exceptions that are in the package `java.lang.safeutil`. All these exceptions have the baseclass `java.lang.safeutil.ForeignException`. These exceptions can be divided with respect to their purpose.

1. SafeRegistry
2. Illegal Foreing Access

7.8.1 SafeRegistry

The `SafeRegistry` methods can throw 2 different exceptions that are equivalent to those that the RMI registry throws. These exceptions are part of the package `java.lang.safeutil`

- AlreadyBoundException
- NotBoundException

7.8.2 Illegal Foreign Access

These exceptions are specific to this implementation.

- IllegalForeignFieldAccessException
- IllegalForeignMethodCallException

It is an invariant of the model that foreign objects can only be referenced by variables that have as type an interface that extends `java.lang.Sharable` or the class `java.lang.Object`. It is not always possible in this implementation to prevent, at compile time, the programmer to reference foreign objects via a variable of a class type and therefore, this invariant is checked also at runtime. The exceptions mentioned above will be thrown when the programmer tries to access intstance variables or to invoke methods of an LJVMBase object of a different LJVM. This problem was also explained in subsection 6.6.1. Other implementations of the safe-threads model (eg. a preprocessor) should prevent these problems before runtime.

Chapter 8

Open problems

In this chapter, it will be explained that some problems that still exist in the safe-threads model. It does not mean that safe-threads are not usable at all. It is just an indication that the perfect model, if this exists, for concurrency is not found yet. Another possibility is that safe-threads need to be used in combination with some patterns that still need to be developed.

8.1 Finalization

Finalization is the execution of a method called `finalize` when the garbage collector has detected that an object is unreachable. This is executed in a separate thread. Even if that method makes the object reachable again, the `finalize` method will only be executed at most one time during the lifetime of an object.

8.1.1 Why is this a problem?

In the safe-threads model the finalizer should not be executed concurrently with any other thread in the LJVM that owns the finalizing object. Therefore the finalizer thread should ask the LJVM monitor before executing the finalizer.

The problem is that there are LJVMs that are client only and they will never release the monitor. An example of such an LJVM is the initial LJVM. The finalizer thread will dead-lock the first time it finalizes an object belonging to a client only LJVM.

A similar problem can also appear in the unmodified Kaffe if a finalizer

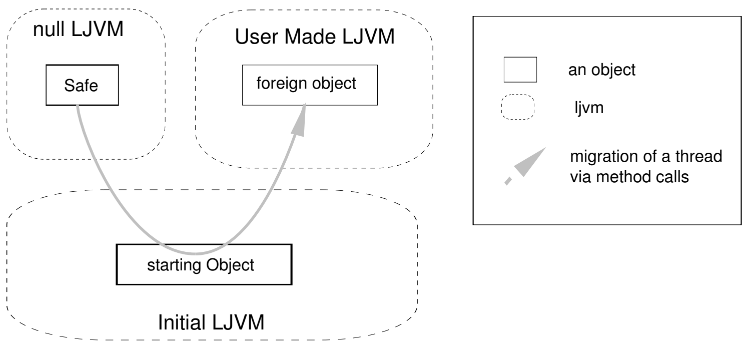

Figure 8.1: Thread migration during nested method invocations

contains an infinite loop. The problem is not the safe-threads model but more the finalization itself.

This makes the problem of finalization becoming important. At this moment different solutions are possible but none of them is really satifactory.

1. a finalization thread that is not aware of LJVMs
2. execute the finalization when objects are allocated
3. execute the finalization when an LJVM is left
4. do not execute finalizers at all

8.1.2 The finalization thread is not aware of LJVMs

In this solution the finalizers are executed concurrently with the active thread in an LJVM. This will sacrifice the safety of the model, since the `finalize` method can alter the state of the LJVM in an unexpected way. It might be the easiest solution to implement but it is a possible danger, since the finalizer might affect other objects in the LJVM.

This because an object ready to be finalized can reference objects that are reachable by the LJVM. This means that two threads can access them concurrently.

8.1.3 Finalization when objects are allocated

This means at the moment memory is allocated the finalization can be executed. This is the solution that can be used in a single thread system.

This might look better but it is not very different from the first solution. In the meaning that the finalizer can change an LJVM without the programmer being aware of it. It is also not always obvious that an object is allocated, for example when concatenating two `java.lang.String`s. It even adds the problem that in an LJVM are a lot of objects that waiting to be finalized but at that moment the thread can be in another LJVM via a foreign method so the memory can not be freed. It is however compliant with the approach in a sequential language. If only one thread is active, the right moment for garbage collection would be when memory is needed. The thread that allocates an object is stopped during the finalization, therefore only one thread is active. This would respect the model, but the finalization thread can still create unexpected effects.

8.1.4 When an LJVM is left

This solution would execute the finalization when an LJVM does no longer contain an active thread .

Some LJVMs might never be left because the only way to leave an LJVM is via returning the foreign call that let the thread enter the LJVM. For example when the InitialLJVM is left the program stops, this means that in the initial LJVM garbage collection is impossible if enough objects need a finalization that will not be executed, and the memory can not be released.

8.1.5 Importance of this problem

A garbage collector resolves the problem of freeing memory at the right moment. This can be considered as important for memory management, as safe-threads are for multithreaded programming. So it would not be a good idea to sacrifice garbage collection for safe-threads. The finalization can be sacrified because it can be seen as an optimization. It is not even guarranteed that a finalizer is executed in Java.

This same problem will also show up in other possible implementations that where suggested in section 4.1. This means that this needs to be covered by the safe-threads model in a consistent way. The problems appeared in this implementation since it is the first that is implemented this far, to reveal this important conflict with finalization.

8.2 Reflection

Our implementation does not implement a safe way to use reflection because it was not an objective of this work. Therefore in this implementation it might be possible to concurrently access data in objects via reflection.

8.3 Not always safe

If a programmer wants to create an unsafe program it is still possible. It needs a little creativity but it is possible to reconstruct the unsafety that safe-threads try to avoid. The example in Listing 20 shows such an unsafe program.

```java
public interface VarInterface extends Sharable {
  public int get();
  public void set(int nval);
}

public class Var implements VarInterface {
  int value;
  public int get() {
    return value;
  }
  public void set(int nval) {
    value=nval;
  }
}

void evilMethod(VarInterface a) {
  if (a.get()<5)    // compare with: if (x<5)
    a.set(a.get()+1);  //                x=x+1;
}
```

Listing 8.1: Unsafe Example

A critical section problem can appear when the evilMethod is executed by thread A and thread B at the same moment with the same argument and the two threads A and B are sheduled as shown in the Table 8.1.

Time	thread A	thread B	result
0			a.value==4
1	if (a.get()<5)		true
2		if (a.get()<5)	true
3	a.set(a.get()+1);		a.value==5
4		a.set(a.get()+1);	a.value==6

Table 8.1: Trace of a created critical sectionproblem

The value of `a.value` will be 6 when started with a value of 4. This is because the action is splitted in too small parts. If a method `increase` would be part of the class `Var` the test could be executed in such a way that no other thread could interfere in the wrong moment.

Notice that the program is very similar to Listing 12, which is an example to show how sequential code can give problems in a concurrent execution. The class `Var` is a conceptual variable, by sharing an instance of that class the original problem is reconstructed.

This example shows that the use of safe-threads will not solve all problems and protect a programmer from all problems that can be caused by threads. Since it is possible to reconstruct the problems that safe-threads want to prevent.

8.3.1 Other similar examples

The previous example is comparable to a programmer creating `goto` constructions in languages like Java. See Listing 18 this construction. This shows even if a language does not support the `goto`, a programmer can generate "spaghetti-code".

```
int go_to=1; // goto is a keyword
for (;;) {
  switch(go_to) {
    case 1:
      // some code
      go_to=3;
      break;
    case 2:
      // some code
```

```
        go_to=1;
        break;
      case 3:
        // some code
        go_to=2;
        break;
    }
}
```

Listing 8.2: Java Goto

It is even possible to have dangling references in a language that supports garbage collection like Java, by using array indexes as pointers. That trick has been used to teach data structures like trees in languages without pointers or references. The example in Listing 25 shows a possible implementation.

```
static final int MAGIC=9876543;
Point[] heap=new Point[1000];
for (int i=0;i<1000;i++)
        heap[i]=new Point(MAGIC,MAGIC);
// to get a reference
int p1=0; while (heap[p1].x!=MAGIC) p1++;


int p2=p1; // make an alias
heap[p2].x=10;
heap[p2].y=3;

//free p1
heap[p1].x=MAGIC;
// p2 is a now dangling reference

...

System.out.println(
        '[' + heap[p2].x +
        ',' + heap[p2].y
```

```
            );
// p2 has been changed ???
```

Listing 8.3: Dangling References

These two examples show that a programmer can create problems, even if a language is intended to prevent that specific problem.

Chapter 9

Conclusions and future work

9.1 Conclusions

From the conceptual point of view, the applications are conceived in the model of safe-threads as a set of sequential processes (the LJVMs) with independent object spaces, but interacting through remote method calls. The main advantage of this model is that it insures the mutual exclusion of the safe-threads while accessing objects, increasing the reliability of concurrent applications and diminishing debugging time.

This thesis has served to prove that it is possible to implement the model of safe-threads in a single Unix process, which is multiplexed to allocate and execute all the LJVMs with their own insulated object spaces. To prove it, we have modified Kaffe, an interpreter for Java, to support the safe-threads model. The main characteristics of this implementation are:

- all the safe-threads share the same address space
- pointers to shared objects are represented directly by their address, without the need of adding stubs or skeletons as in RMI or CORBA
- method calls of objects belonging to another LJVM are executed by the same safe-thread that executes the call, not needing a special thread to execute remote calls like in RMI or CORBA

And therefore from the implementation point of view the applications conceived in the model of safe-threads are closer to traditional (unsafe) multi-threaded applications than to applications distributed on several heavy processes communicating through the expensive RMI.

These results promise that in a production quality implementation of the model of safe-threads the application will be close in performance to traditional multi-threaded applications but more reliable than the latter.

Our implementation shows up also the main overhead of safe-threads over the traditional threads:

- in foreign calls, arguments being local objects must be deep-copied to avoid sharing them. Standard threads do not need this, because they can share the same objects concurrently.
- to access a static variable it is necessary to look it up in a hash-table, while in standard threads it can be accessed very efficiently by its address.

We have not quantified this overhead because any experiment would be meaningless due to the inefficient implementation of the Kaffe interpreter. This will be possible when a truly compiled implementation of the model will be available.

This thesis has served also to prove that it is possible to change the semantics of Java to support the model of safe-threads without any change to the syntax of the language, or even to the compiler. It is enough to change the way that the byte code is interpreted.

During this work we have discovered where is has been difficult to ensure that LJVMs access only local objects and sequentially:

1. while accessing instance variables belonging to LJVMBase objects (see section 6.6.1)
2. in finalizers (see Section 8.1)
3. when inspecting objects through reflection

The first point can be solved efficiently by adding a simple analyzer of the byte code before the execution. The two other points are not exclusive problems of the concurrent programming, they are even present in strictly sequential languages.

The modification of the interpreter of Kaffe was, even though it was a lot of work, a straight-forward work. This experience is a necessary first step to a more complex project: The modification of a JIT compiler in Kaffe.

The successful implementation of the interpreter makes the JIT compiler a promising future work.

Finally, our implementation will be used to to discover the design patterns that should be used with safe-threads to build robust programs.

9.2 Future work

To allow a programmer to write robust programs, he will need to use some design patterns with the safe-threads model. These patterns have to insure that a problem as described in section 8.3 will not occur. An important work will be to discover such design patterns and when to apply them.

To allow the use of real world applications it is important to build an implementation that is based on a compiler. For example by modifying a compiler of Kaffe or by building a preprocessor that translates a safe-threads program to a standard Java program.

An important work for the research on the safe-threads model is to write applications that use this model. This will allow to evaluate the model by comparing the development time with the time it would require to build it with traditional threads.

Bibliography

[1] P. Brinch-Hansen. Structured multiprogramming. In *Communications of the ACM*, volume 15, pages 574–577, July 1972.

[2] P. Brinch-Hansen. Java's insecure parallelism. In *ACM Sigplan Notices*, volume 34, pages 38–45, April 1999.

[3] Shigeru Chiba. Javassist — a reflection-based programming wizard for java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998. A very early sketch of the design of Javassist.

[4] Edsger W. Dijkstra. Go to statement considered harmful. In *Communications of the ACM*, pages 147–148, March 1968.

[5] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1.0 edition, August 1996.

[6] C.A.R. Hoare. Monitors: An operating system structuring concept. In *Communications of the ACM*, volume 17, pages 549–557. Association for Computing Machinery, Inc, October 1974.

[7] Lothar Hotz and Michael Trowe. Netclos - parallel programming in common lisp.

[8] Luis Mateu and José Miguel Piquer. Safe-threads: a new model for object-oriented multi-threaded languages. In *XIX Conference of the Chilean Society of Computer Science*, November 1999.

[9] Sun Microsystems. Rmi specification. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html.

[10] Hans Muller and Kathy Walrath. Threads and swing. Sunworld Web Magazine.

[11] Daniel J. Romero Nava. Pre-procesador para safe threads. Master's thesis, DCC, Universidad de Chile, 2000. "Work In Progress".

[12] Object Management Group. *The Common Object Request Broker Architecture and Specification*, Farmingham, MA., July 1995.

[13] Transvirtual. Kaffe. http://www.kaffe.org.

Appendix A

Internal hard core information

This chapter contains information specific to this implementation, so none of this is guaranteed to work with other implementations, or future versions. However this information can be useful for maintaining the modified Kaffe or porting the changes to a later release of Kaffe. This implementation is based on Kaffe 1.0.5, while recently a new version Kaffe was released.

A.1 Internal LJVM id

In the class `java.lang.LJVMBase` is a native method getLJVMid() that returns the internal ID of that LJVM. This internal ID is probably only useful for debugging. Since it is not an essential part of the safe-threads model, it is not a good idea to use that id to compare LJVMs. It is better to compare the representing LJVMBase instead. The id will be even recycled if possible since it is used to access internal tables that should be kept as small as reasonably possible.

A table contains all information for each LJVM if this table is full it will be extended. When a new LJVM is created a free slot in that table is searched. If no free slot exists the table will be extended.

The use of a separate table to contain LJVM information instead of storing the information in the LJVMBase object allows some extra modularity. It is also useful for the null-LJVM that does not have an LJVM object.

A.2 Creating LJVMs without LJVMBase

The modified Kaffe will create a new LJVM if a class that inherits directly or indirectly from `java.lang.LJVMBase`. This can be modified by changing the macro `LJVMBASE_PATH` in the file `kaffe/kaffevm/ljvm.h`. This would be useful if the model would be changed to allow LJVMs to objects from a class that does not inherit from LJVMBase.

A.3 null-LJVM

The null-LJVM that is active when Kaffe starts, does not have an object that represents it. This LJVM has the internal identifier 0 and is different compared to others. All LJVMs can access objects from the null-LJVM as if the objects are in the current LJVM. The methods from objects of that LJVM can also be invoked even if the object is not `Sharable` this done to solve problems of compatibility.

A.4 Finalizing an LJVM

In the unmodified Kaffe it is allowed to make an object reachable again during finalization. This is not completely true for the modified Kaffe. Since the tables for an LJVM are freed immediately after executing the `finalize` of an LJVM object. This will not imply much problems since finalizers are not used very much, a finalizer that makes an object reachable again is even more rare. And this finalizer should defined for an LJVM object. The finalization causes some other problems as well, these are explained in Section 8.1

A.5 RMI

To make a working RMI the invariants of the model should be enforced. This means that only one thread can be active in an LJVM. The internals of the RMI system will probably not need much changes.

kaffe-1.0.5/kaffe/kaffevm/gcFuncs.c
kaffe-1.0.5/kaffe/kaffevm/ljvm.c
kaffe-1.0.5/kaffe/kaffevm/static_ljvm.c
kaffe-1.0.5/kaffe/kaffevm/intrp/argcopy.c
kaffe-1.0.5/kaffe/kaffevm/intrp/machine.c

Table A.1: Files that support JVH_DEBUG

A.6 Debugging

In the files listed in Table A.1 it is possible to define the macro `JVH_DEBUG` before compiling the Kaffe program. This will result that some extra debug info is printed to stderr.

A.7 Locks on an LJVM

When a thread needs to enter into an LJVM it must get the lock first. This can be seen as if each method that is accessible from another LJVM has the structure shown in Listing 7.

```
void methodHeader(arg1 .. argn) {
  LJVM thisLjvm=LJVMBase.currentLJVM();
  synchronized(thisLjvm) {
    // the real methodbody
  }
}
```

Listing A.1: Locks on an LJVM

This wrapping is done by the modified virtual machine so the programmer does not see all of this. It is also a little more efficient than adding this wrapper to the java program, since the call to the native method currentLJVM is not done via the Java Native Interface (JNI) but by call a to a C function directly.

A.8 Java Native Interface

This allows the mixing C code with Kaffe. The interface is made to work in a way that it supports what is needed for the prototype. If a C program calls a java method then the method will be executed in the null-LJVM. This might not be the perfect solution but JNI does not support the LJVMs so it was needed to find a workable solution.

It would be useless to try to support safe-threads in C, because C program will always be able to access all data. Because of this consideration the JNI was just modified to support what is needed to make the prototype work.

A.9 Low level registry

To allow an easy implementation a lowlevel registry is build with numerical keys. This is easy to implement in native code. Only one gate would be sufficient to build a high level registry. However it is easy to provide a numerical key, which can be useful. The two methods, `LJVMBase.getGlobal` and `LJVMBase.setGlobal`, provide the basic functionality to share objects between different LJVMs. These two methods work with an internal hashtable in the virtual machine that stores the objects with a numerical key. A new `setGlobal` overwrites the previous value and storing a `null` removes the association from the table. As long as an object is in the table it will not be removed by the garbage-collector.

It is only possible to store objects that are sharable, therefore the programmer can not get a reference to a foreign object that is not sharable in another LJVM. This is not allowed since it would break the model.

These numerical keys can be seen as a *gate* to a shared piece of memory. Since the key is an integer will be huge amount gates[1].

These two methods will be used to build a more high level string based registry that is similar to the registry of the RMI system. This is implemented in the `java.lang.safeutil.SafeRegistry` as described in 7.4.1.

A small overview of the used gates at this moment can be seen in Table A.2. It is important that a gate is not used for different purposes at the same moment since the `setGlobal` will overwrite the previous stored object.

A normal programmer should not use this low level registry. This because it can have unexpected effects if a gate is used for different purposes. An even more important reason to not use the gates is that they are implementation specific.

[1] $2^{32} > 4 * 10^9$

0	System.in stub/skeleton
1	System.out stub/skeleton
2	System.err stub/skeleton
11071302	the SafeRegistry system

Table A.2: Global table usage

Index