166166

METHOD FOR ADAPTING MIGRATING PROCESSES TO HOST MACHINES

This invention relates to a method for adapting a migrating process as it moves from one machine to another machine with potentially differing hardware configurations.

Recent years have seen a number of developments in computing science regarding how elements within a software application are treated and handled. In this context the most basic elements to be found within a software application are data and program modules. Traditional procedural programming paradigms focus on the logic of the software application, so a program is structured based on program modules. One uses the program modules by explicitly supplying to them the data on which they should operate.

More recently there has been a move towards an object-oriented paradigm. In this paradigm, programs are structured around objects, with each object representing an entity in the world being modeled by the software application. Each object manages its own data (state), which are hidden from the external world (other objects and programs). An object in a program interacts with another object by sending it a message to invoke one of its exposed program modules (method). This paradigm imposes better control and protection over internal data, and helps to structure complex applications designed and implemented by a team of programmers. An example of an object-oriented environment can be found in US 5,603,031. This discloses an environment in which new agents (essentially objects) consisting of data and program modules can be sent between machines.

While the object-oriented paradigm represents a significant advance in software engineering, the data and modules that constitute each object are static. The paradigm is still inadequate for writing programs that must evolve during execution, eg programs that need to pick up, drop, or substitute selected modules. There have been several attempts at overcoming this limitation. For example, work described in US Patents 4954941, 5175828, 5339430 and 5659751 address techniques for re-linking or re-binding selected software modules dynamically during runtime. Also Microsoft's Win32 provides for explicit mapping and un-mapping of dynamic linked libraries into the address space of a

G00002

process through the LoadLibrary and FreeLibrary calls. With this prior art, however, the prototype or specification of functions and symbols are fixed beforehand and compiled into application programs. This means that an object cannot invoke a module of another object for which the specification is not known at compile time.

Another shortcoming of both traditional procedural and object-oriented paradigms is that programs consist only of data and program modules, but not the transient information that capture the entire state of execution during runtime. This makes it difficult to interrupt a running program and to migrate it to another machine. Generally it is only possible to transfer the saved data file of completed applications. As a very simple example of this problem, while a word processing document file or a spreadsheet file may be transferred from one machine to another, it is not possible to do so while the file is currently being worked on without reverting to a saved version. Transient information is lost. In the case of a word processing file, for example, this means that any "undo editing" function cannot be used by the receiving machine on the file it has just received because the transient "undo" information is not part of the data file.

At present, such migrations have been effected from outside the program, as utilities in the computing environment. Examples include Amoeba, Charlotte, Sprite and Condor. Such utilities work by taking a snapshot of the running program (or core dump), and by resuming execution on the recipient machine from the snapshot. Unfortunately the migration utilities have no knowledge of the usage requirements and semantics of the components of the running program, and so there is no way to adapt it to the new computing environment. Consequently, the migrated program most likely cannot run in a different computing environment from the original one, such as when the machines have different devices like displays, hard disks and sound cards. Even in cases where it does, it would not run with the same level of efficiency, for example because the machines have different amounts of main memory.

This question of incompatibility between different machines is a major problem in computing networks. Elements of a process that are adapted to one machine may simply not apply to another which requires a different configuration and the problem is one of the major difficulties in developing a fully mobile computing environment. With current techniques the hardware components of the environment must be fully compatible for

processes to be transferred between them, and even if different hardware components are in theory compatible, different user set-ups and configurations can still cause difficulties.

In this specification the following terms will be used with the following meaning:

"First class entity": an object that can be manipulated directly.

"Process": a combination of data, program module(s) and current execution state.

"Execution state": the values and contents of transient parameters such as the contents of registers, frames, counters, look-up tables and the like.

According to the present invention there is provided a method for migrating a computing process from a first host to a second host, wherein said process discards data, and/or program code and/or execution states specific to the first host, and wherein said process receives data, and/or program code and/or execution states specific to said second host.

By means of this arrangement a process can adapt from the environment of the first host to the environment of the second host by losing system specific information relating to the first host, and by acquiring system specific information relating to the second host.

In a first embodiment of the invention the process discards data and/or program code and/or execution states specific to the first host prior to migration to the second host. In a preferred embodiment, prior to migration a construct is formed comprising application specific data, and/or program code and execution states of the process. This construct may be formed by a construct operation that suspends all active threads of the process and records application specific data, and/or program code and/or execution states of the process. The construct may comprise only data, program code and execution states falling within lists that are passed to the construct operation. The construct may be provided with an authorizing signature.

After it is formed, the construct may either be sent directly to the second host, or it may be sent to an intermediate memory storage means and from there to the second host when required. When the construct has been sent to the second host, a second process is run on the second host that comprises the data, program code and execution states in the construct.

In this embodiment a third process may be created containing system specific data, and/or program code and/or execution states relating to the second host, and the second process may then assimilate this third process.

In another embodiment the process may discard data, and/or program code and/or execution states specific to the first host after migration to the second host, but before receiving data, and/or program code and/or execution states specific to said second host.

In this embodiment prior to migration a construct is formed comprising data, and/or program code and/or execution states of the process. The construct may be formed by a construct operation that suspends all active threads of the process and records data, and/or program code and/or execution states of the process. The construct may be provided with an authorizing signature.

After formation the construct may be sent directly to the second host, or may by sent via an intermediary memory storage means and then from there to the second host when required. After the construct is sent to the second host a second process is created on the second host comprising the data, program code and execution states in the construct. The second process may then perform a mutate operation that suspends all active threads of the second process and discards system specific data, and/or program code and/or execution states relating to the first host. The second process may comprise only data, program code and execution states falling within lists that are passed to the mutate operation. In this embodiment a third process is created in the second host containing system specific data, and/or program code and/or execution states relating to the second host, and the second process may then assimilate the third process.

In a third embodiment of the invention the process may discard data, and/or program code and/or execution states specific to the first host after migration to the second host and after receiving data, and/or program code and/or execution states specific to the second host.

In this embodiment prior to migration a construct is formed comprising data, and/or program code and/or execution states of the process. The construct may be formed by a construct operation that suspends all active threads of the process and records data, and/or program code and/or execution states of the process. The construct may be provided with an authorizing signature.

After formation the construct may be sent directly to the second host, or may by sent via an intermediary memory storage means and then from there to the second host when required. After the construct is sent to the second host a second process is created on the second host comprising the data, program code and execution states in the construct.

A third process may be created on the second host containing system specific data, and/or program code and/or execution state relating to the second host and the second process may then assimilate this third process. Following this assimilation, the second process may perform a mutate operation that suspends all active threads of the second process and discards system specific data, and/or program code and/or execution states relating to the first host. The second process may comprise only data, program code and execution states falling within lists that are passed to the mutate operation.

In all embodiments of the invention the data, and/or program code and/or execution states discarded by the process may relate to library modules and/or input/output device drivers of the first host, and correspondingly the data, and/or program code and/or execution states received by the process may relate to library modules and/or input/output device drivers of the second host.

It will also be understood that the term "host" should be read broadly as meaning any form of computing environment ranging from a single machine of any kind, to any form of network.

Some embodiments of the invention will now be described with reference to the accompanying drawings, in which:-

Fig.1 is a general schematic model of an operating environment of a computing system,

Fig.2 schematically illustrates a process life-cycle and operations that may be performed on a process,

Fig.3 is a flow-chart illustrating the Hibernaculum Construct operation,

Fig.4 is a flow-chart illustrating the Assimilate operation, and

Fig.5 is a flow-chart illustrating the Mutate operation.

Figure 1 shows the general model of a computing system. An application program 30 comprises data 10 and program modules 20. The operating system 60, also known as

the virtual machine, executes the application 30 by carrying out the instructions in the program modules 20, which might cause the data 10 to be changed. The execution is effected by controlling the hardware of the underlying machine 70. The status of the execution, together with the data and results that the operating system 60 maintains for the application 30, form its execution state 40.

Such a model is general to any computing system. It should be noted here that the present invention starts from the realisation that all the information pertaining to the application at any time is completely captured by the data 10, program modules 20 and execution state 40, known collectively as the process 50 of the application 30.

The process 50 can have one or more threads of execution at the same time. Each thread executes the code of a single program module at any given time. Associated with the thread is a current context frame, which includes the following components:

- A set of registers
- A program counter, which contains the address of the next instruction to be executed
- Local variables of the module
- Input and output parameters of the module
- Temporary results of the module

In any module A, the thread could encounter an instruction to invoke another module B. In response, the program counter in the current frame is incremented, then a new context frame is created for the thread before it switches to executing module B. Upon completing module B, the new context frame is discarded. Following that, the thread reverts to the previous frame, and resumes execution of the original module A at the instruction indicated by the program counter, i.e., the instruction immediately after the module invocation. Since module B could invoke another module, which in turn could invoke some other module and so on, the number of frames belonging to a thread may grow and reduce with module invocations and completions. However, the current frame of a thread at any given time is always the one that was created last. For this reason, the context frames of a thread are typically stored in a stack with new frames being pushed on and popped from the top. The context frames of a thread form its

execution state, and the state of all the threads within the process 50 constitute its execution state 40 in Fig.1.

The data 10 and program modules 20 are shared among all threads. The data area is preferably implemented as a heap, though this is not essential. The locations of the data 10 and program modules 20 are summarized in a symbol table. Each entry in the table gives the name of a datum or a program module, its starting location in the address space, its size, and possibly other descriptors. Instead of having a single symbol table, each process may alternatively maintain two symbol tables, one for data alone and the other for program modules only, or the process could maintain no symbol table at all.

In a preferred embodiment of the present invention, the data and program code of a process are stored in a heap and a program area respectively and are shared by all the threads within the process. In addition the execution state of the process comprises a stack for each thread, each stack holding context frames, in turn each frame containing the registers, local variables and temporary results of a program module, as well as addresses for further module invocations and returns. Before describing an embodiment of the invention in more detail, however, it is first necessary to introduce some definitions of data types and functions that are used in the embodiment and which will be referred to further below.

In addition to conventional data types such as integers and pointer, four new data types Data, Module, Stack and Hibernaculum are defined in the present invention:

Data: A variable of this data type holds a set of data references. Members are added to and removed from the set by means of the following functions;

Int AddDatum(Data d, String dataname) inserts the data item dataname in the heap of the process as a member of d.

Int DelDatum(data d, String dataname) removes the data item dataname from d.

Module: A variable of this data type holds a set of references to program modules. Members are added to and removed from the set with the following functions;

Int AddModule(Module d, String modulename) inserts the program module modulename in the program area of the process as a member of d.

Int DelModule(Module d, String modulename) removes the program module modulename from d.

Stack: A variable of this data type holds a list of ranges of execution frames from the stack of the threads. The list may contain frame ranges from multiple threads, however no thread can have more than one range. Variables of this type are manipulated by the following functions:

Int OpenFrame(Stack d, Thread threadname) inserts into d a new range for the thread threadname, beginning with the thread's current execution frame. This function has no effect if the thread already has a range in d.

Int CloseFrame(Stack d, Thread threadname) ends the open-ended range in d that belongs to the thread threadname. This function has no effect if the thread does not currently have an open-ended range in d.

Int PopRange(Stack d, Thread threadname) removes from d the range belonging to the thread threadname.

Hibernaculum: A variable of this data type is used to hold a suspended process.

As will be explained in more detail below a process may be suspended and stored in a hibernaculum prior to being transferred from one operating environment to another operating environment and/or may be subject to evolutionary operations:

Hibernaculum Construct(Stack s, Module m, Data d): This operation creates a new process with the execution state, program table and data heap specified as input parameters. The process is immediately suspended and then returned in a hibernaculum.

The hibernaculum may be signed by the originating process as indication of its authenticity. Fig.3 is a flow-chart showing the hibernaculum construct operation.

A hibernaculum may be sent between operating environments by the following send and receive functions:

Int Send(Hibernaculum h, Target t) transmits the process contained within h to the specified target.

Hibernaculum Receive(Source s) receives from the specified source a hibernaculum containing a process.

A hibernaculum may be subject to the following evolutionary function:

Int Assimilate(Hibernaculum h, OverrideFlags f) activates the threads of the process stored within h and runs them as threads within a calling process's operating environment. Where there is a conflict between the data and/or program modules of the hibernaculum and the operating environment, the override flags specify which to preserve. Fig.4 is a flow-chart illustrating the steps of the assimilate operation.

Int Mutate(Stack s, int sflag, Module m, int mflag, Data d, int dflag) modifies the execution state, program table and data heap of the calling process. If a thread has an entry in s, only the range of execution frames specified by this entry is preserved, the other frames are discarded. Execution stacks belonging to threads without an entry in s are left untouched. In addition, program modules listed in m and data items listed in d are kept or discarded depending on the flag status. Fig.5 is a flow-chart illustrating the steps of the mutate operation.

Fig.2 illustrates very schematically how these operations may act on a process 230 (which may be loaded from an application 210 or a hibernaculum 220). The process 230 may be subject to a Construct operation 110 to create a hibernaculum, a

hibernaculum may be sent to a stream by a Send operation 120, or received from a stream by a Receive operation 130. The contents of a hibernaculum may be assimilated in the process by an Assimilate operation 140, and a process may be caused to mutate by a Mutate operation 150. The process 230 may of course also be subject to traditional operations.

An exemplary embodiment of the invention will now be described in which it is assumed that an executing process p1 is required to migrate from a first host machine t1 to a second host machine t2.

To begin with the process p1 calls the following function:

Hibernaculum h = Construct(Stack s, Module m, Data d)

where s contains all the execution stacks in the process, m contains all the application specific modules in the process (ie m excludes those modules that are system specific to the first host machine), and d contains all the application specific data (ie d excludes data that are system specific to the first host machine). This function creates a new process p2 that contains only the data, program code and execution states of the original process p1 that are specific to the application and not to the system of the first host machine. Process p2 is immediately suspended and returned within a hibernaculum h.

The original process p1 then calls the function:

Int Send(hibernaculum h, Host t2)

which transmits the process p2 contained within hibernaculum h as a stream to the new host machine t2.

At the new host machine t2 a new process p3 is created containing initial system specific data and program code relating to the new host t2. This new process p3 then immediately executes the function:

Hibernaculum h = Receive(Host t1)

which receives from the first host machine t1 the hibernaculum h containing the process p2 which includes only the application specific data, program codes and execution states.

Process p3 then calls the function:

Int Assimilate(Hibernaculum h)

to activate the threads of the process p2 stored in h and to run them as new threads within the environment of the calling process p3. The original thread in p3 then terminates, resulting in a process that has all the application specific portions of process p1, but with the system specific portions replaced to suit the requirements of the second host machine t2.

The original process p1 terminates.

In the embodiment described above the process p1 discards first host specific information prior to migration. However, the discarding may be done after migration to the second host. This is described in the following embodiment.

To begin the process p1 calls the following function:

Hibernaculum h = Construct(Stack s, Module m, Data d)

where s contains all the execution stacks in the process, m contains all modules in the process (including both system specific and application specific modules), and d contains all data in the process (including both system specific and application specific data). Thus the entire process p1 is contained within the hibernaculum as a new process p2 that is identical to p1. Process p2 is immediately suspended and returned within hbernaculum h.

The original process p1 then calls the function:

Int Send(hibernaculum h, Host t2)

which transmits the process p2 contained within the hibernaculum h as a stream to the new host machine t2 where the process p2 is then re-activated. The process p2 then calls the function:

Int Mutate(Stack s, int sflag, Module m, int mflag, Data d, int dflag)

where s contains all the execution stacks in the process, m contains all the application specific modules in the process, and d contains all the application specific data in the process, and where the flags are set to retain the contents of s, m and d. In this way all execution states, and all application specific modules and data are retained in the process p2, but all data and modules specific to the system of the first host are discarded. Process p2 is then stored within a hibernaculum h in the second host using the construct operation.

At the new host machine t2 a new process p3 is created containing initial system specific data and program code relating to the new host t2. This process p3 then calls the function:

Int Assimilate(Hibernaculum h)

to activate the threads of the process p2 stored in h and to run them as new threads within the environment of the calling process p3. The original thread in p3 then terminates, resulting in a process that has all the application specific portions of p1, but with the system specific portions replaced to suit the requirements of the second host machine t2.

It will also be understood that in a variation of this second embodiment the mutate and assimilate functions may be reversed. That is to say, following the transfer of process p2 (identical to p1) from host t1 to host t2, process p2 may assimilate process p3 (containing the t2 system specific data and code) before the mutate operation is used to discard the t1 system specific data and code.

Thus it will be seen that there is provided a method by means of which a process can migrate from one host machine to another host machine and in doing so can adapt to the system requirements and configurations of the second host machine. Such a method allows processes to be readily transferred between host machines thus substantially facilitating the creation of a mobile computing environment.

To implement the process migration system of the present invention in a Java environment, a package called snapshot is introduced. This package contains the following classes, each of which defines a data structure that is used in the migration and adaptation operations:

```
public class Hibernaculum {
        ...
}

public class State {
        ...
}

public class Module {
        ...
}

public class Data {
        ...
}

public class Machine {
        ...
}
```

In addition, the package contains a Snapshot class that defines the migration and adaptation operations:

```
public class Snapshot {
        private static native void registerNatives();
        static {
```

```
        registerNatives();
    }

    public static native Hibernaculum Construct(State s, Module m, Data d);
    public static native int Send(Hibernaculum h, OutputStream o);
    public static native Hibernaculum Receive(InputStream i);
    public static native int Assimilate(Hibernaculum h, int f);
    public static native int Mutate(State s, int sflag, Module m, int mflag, Data d, int
    dflag)

    // This class is not to be instantiated
    private Snapshot() {
    }
}
```

The methods in the Snapshot class can be invoked from application code. For example:

```
try {
        if (snapshot.Snapshot.Construct(s, m, d) != null) {
                // hibernaculum has been created
        } else {
                // failed to create hibernaculum
        }
        catch(snapShot.SnapshotException e) {
                // Failed to create hibernaculum
        }
```

The migration and adaptation operations are implemented as native codes that are added to the Java virtual machine itself, using the Java Native Interface (JNI). To do that, a Java-to-native table is first defined:

```
#define KSH "Ljava/snapshot/Hibernaculum;"
#define KSS "Ljava/snapshot/State;"
#define KSM "Ljava/snapshot/Module;"
#define KSD "Ljava/snapshot/Data;"

static JNINativeMethod snapshot_Snapshot_native_methods[] = {
        {
                "Construct",
                "("KSSKSMKSD")"KSH,
                (void*)Impl_Snapshot_Construct
        },
        {
                "Send",
                "("KSH"Ljava/io/OutputStream;)I",
                (void*)Impl_Snapshot_Send
        },
        {
                "Receive",
                "(Ljava/io/InputStream;)"KSH,
                (void*)Impl_Snapshot_Receive
        },
        {
                "Assimilate",
                "("KSH"I)I",
                (void*)Impl_Snapshot_Assimilate
        },
        {
                "Mutate"
                "("KSSKSM"I"KSD"I)I"
                (void*)Impl_Snapshot_Mutate
```

```
        },
};
```

After that, the native implementations are registered via the following function:

```
JNIEXPORT void JNICALL
Java_snapshot_Snapshot_registerNatives(JNIEnv *env, jclass cls) {
        (*env)->RegisterNatives(    env,
                                    cls,
                                    snapshot_Snapshot_native_methods,
                                    sizeof(snapshot_Snapshot_native_methods) /
                                    sizeof(JNINativeMethod)  );
}
```

Besides the above native codes, several functions are added to the Java virtual machine implementation, each of which realizes one of the migration and adaptation operations:

```
void* Impl_Snapshot_Construct(..) {
        // follow flowchart in Figure 3
        ...
}

void* Impl_Snapshot_Send(..) {
        // send given hibernaculum to specified target
        ...
}

void* Impl_Snapshot_Receive(..) {
        // receive a hibernaculum from a specified source
        ...
```

```
}

void* Impl_Snapshot_Assimilate(..) {
        // follow flowchart in Figure 4
        ...
}

void*Impl_Snapshot_Mutate(..){
        //follow flowchart in Figure 5
        ...
}
```

CLAIMS

1. A method for migrating a computing process from a first host to a second host, wherein said process discards execution states specific to the first host, and wherein said process receives execution states specific to said second host.

2. A method as claimed in claim 1 wherein said process discards execution states specific to said first host prior to migration to said second host.

3. A method as claimed in claim 2 wherein prior to migration a construct is formed comprising application specific execution states of said process.

4. A method as claimed in claim 3 wherein said construct is formed by a construct operation that suspends all active threads of said process and records application specific execution states of said process.

5. A method as claimed in claim 4 wherein said construct comprises only execution states falling within lists that are passed to said construct operation.

6. A method as claimed in any of claims 3 to 5 wherein said construct is provided with an authorizing signature.

7. A method as claimed in any of claims 3 to 6 wherein said construct is sent directly to said second host on a communication medium.

8. A method as claimed in any of claims 3 to 6 wherein said construct is sent to an intermediary memory storage means, and subsequently to said second host.

9. A method as claimed in any of claims 7 to 8 wherein a second process is run on said second host that comprises the execution states in said construct.

10. A method as claimed in claim 9 wherein in said second host a third process is created containing system specific execution states relating to said second host, and wherein said second process assimilates said third process.

11. A method as claimed in claim 1 wherein said process discards execution states specific to said first host after migration to said second host, but before receiving execution states specific to said second host.

12. A method as claimed in claim 11 wherein prior to migration a construct is formed comprising execution states of said process.

13. A method as claimed in claim 12 wherein said construct is formed by a construct operation that suspends all active threads of said process and records execution states of said process.

14. A method as claimed in any of claims 12 to 13 wherein said construct is provided with an authorizing signature.

15. A method as claimed in any of claims 12 to 14 wherein said construct is sent directly to said second host on a communication medium.

16. A method as claimed in any of claims 12 to 14 wherein said construct is sent to an intermediary memory storage means, and subsequently to said second host.

17. A method as claimed in any of claims 15 to 16 wherein a second process is created on said second host that comprises the execution states in said construct.

18. A method as claimed in claim 17 wherein said second process performs a mutate operation that suspends all active threads of said second process and discards system specific execution states relating to said first host.

19. A method as claimed in claim 18 wherein said second process comprises only execution states falling within lists that are passed to said mutate operation.

20. A method as claimed in claim 19 wherein in said second host a third process is created containing system specific execution states relating to said second host, and wherein said second process assimilates said third process.

21. A method as claimed in claim 1 wherein said process discards execution states specific to said first host after migration to said second host, and after receiving execution states specific to said second host.

22. A method as claimed in claim 21 wherein prior to migration a construct is formed comprising execution states of said process.

23. A method as claimed in claim 22 wherein said construct is formed by a construct operation that suspends all active threads of said process and records execution states of said process.

24. A method as claimed in any of claims 22 to 23 wherein said construct is provided with an authorizing signature.

25. A method as claimed in any of claims 22 to 24 wherein said construct is sent directly to said second host on a communication medium.

26. A method as claimed in any of claims 22 to 24 wherein said construct is sent to an intermediary memory storage means, and subsequently to said second host.

27. A method as claimed in any of claims 25 to 26 wherein a second process is created on said second host that comprises the execution states in said construct.

28. A method as claimed in claim 27 wherein in said second host a third process is created containing system specific execution states relating to said second host, and wherein said second process assimilates said third process.

29. A method as claimed in claim 28 wherein said second process performs a mutate operation that suspends all active threads of said second process and discards system specific execution states relating to said first host.

30. A method as claimed in claim 29 wherein said second process comprises only execution states falling within lists that are passed to said mutate operation.

31. A method as claimed in any preceding claim wherein the execution states discarded by said process relate(s) to library modules and/or input/output device drivers of said first host.

5

32. A method as claimed in any preceding claim wherein the execution states received by said process relate(s) to library modules and/or input/output device drivers of said second host.