Bootstrapping and Startup of an object-oriented Operating System

R. Goeckelmann, M. Schoettner, M. Wende, T. Bindhammer, and P. Schulthess

Department of Distributed Systems, University of Ulm, 89075 Ulm, Germany

goeckelmann@vs.informatik.uni-ulm.de

Abstract. The Plurix project implements an object-oriented Operating System (OS) for PC clusters. Network communication is implemented via the well-known Distributed Shared Memory (DSM) paradigm using restartable transactions and an optimistic synchronization scheme to implement memory consistency. The total OS (including kernel and drivers) reside in the persistent DSM – there is no distinction between DSM and local memory. Herewith all classes and data may be shared and can be persistent with no additional efforts. In this paper we illustrate the boostrapping of our OS identifying the need for different memory pools in the DSM and for node private contexts. Furthermore we discuss the startup process benefiting from potentially persistent device driver instances in the cluster-wide nameservice.

Keywords: Operating System, Object-Orientation, Memory-Management

1 Introduction

Plurix is an object-oriented Operating System (OS) for the PC platform totally written in Java. We abandon the hardware independence as we do not rely on a Java Virtual Machine (JVM) like JavaOS does, [1]. Herewith we gain efficiency and speed. Furthermore Plurix is not implemented on top of an existing OS discarding any overhead caused by commercially justified backward compatibility

The well-known Distributed Shared Memory (DSM) paradigm offers a natural solution for distributing data among several nodes, [3], [4]. Applications running on top of the Plurix DSM are not aware of data locations. Any reference can either point to local or remote memory blocks. During program execution the OS detects a remote memory access and automatically fetches the desired block.

A file system can be avoided by implementing orthogonal persistence, [5]. One of our research goals is to provide a persistent DSM enclosing the kernel and the compiler. Not much can be found in literature about adding persistence to DSM systems, [6]. To the best of our knowledge no other OS project combined the properties of Plurix: persistence, DSM, and object-orientation but we are aware of other object-oriented OS implementing persistence like Grasshoper, [9].

Our system is developed using an own Plurix Java Compiler (PJC) which is an integral part of the OS. All fundamental runtime structures are modeled in Java and the kernel and the PJC are constructed on top of them, [7].

The remainder of this paper is organized as follows. The properties of the Plurix environment including the compiler are presented in section two. The following section illustrates the boostrapping of the Plurix system followed by the startup process. Finally, we conclude with experiences we gained from the bootstrapping of our system and give an outlook on future work.

2 The Persistent DSM Environment

2.1 The Operating System

The current prototype runs within a single LAN segment (100 Mbit/s Ethernet). The central abstraction within our design is a Distributed Shared Memory (DSM) sharing data and code. A node runs one or several cooperative tasks which are controlled by a central loop like in the Oberon System, [2]. Tasks in the OS are automatically or manually partitioned into restartable transactions. Together with an optimistic synchronization scheme they realize the transactional consistency model of Plurix, [10].

Memory accesses (read or write) are monitored by the MMU. There may be many read-only replications of a memory page until a write to that page occurs.

The Plurix DSM is *orthogonally persistent* – any Java object reachable from a global name service root persists including classes and code. Memory persistence is initially provided by a central disk-server within the Plurix cluster.

A distributed Garbage Collection (GC) relieves programmers of the burden of memory management. The GC keeps track of all references to an object by using the *backchain concept*, [8]. Every heap object includes a backchain pointer locating the first reference pointing to it, see fig.-1. If there are several references to an object they are chained together. If the backchain is empty an object is garbage and can be collected.

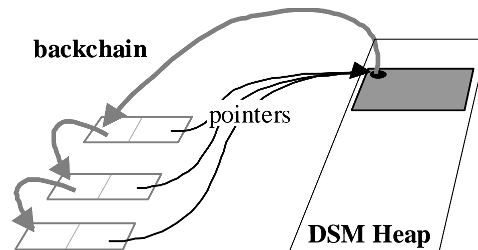

Fig. 1., The backchain concept

The current Plurix user interface continues the Oberon tradition, [2]. The viewer concept has been adopted including executable text elements and the event handling was modified towards Java event listeners, [1].

Commercial Operating Systems typically use numerous name services implemented in different components (e.g. file system, address book, ...). The persistent DSM concept facilitates the implementation of a single cluster-wide namespace.

2.2 The Plurix Java Compiler

The OS is developed with an own Java compiler translating source texts directly into machine instructions. The Plurix Java Compiler (PJC) is written in Java and bootstrapped to the Plurix world where it is used for device-driver and application development. Additionally, we have a cross compilation facility running under MS-Windows.

The compiler is tailored for the persistent DSM and automatically registers parts of the symbol table in the name service by melting the scope concept with the directory concept. Storing symbol information in the name service lays the foundation for a textual interface like known in the Oberon system, [2]. Furthermore, runtime structures and code segments are directly generated avoiding traditional object-, symbol-, library- and exe-files.

3 Bootstrapping of the Plurix OS

The main goal of the boostrapping of the Plurix OS is to store all objects (classes, code-segments and instances) in the persistent DSM, register classes in the name service including symbolic information and get the system up and running. All user objects and even the kernel reside in the DSM and benefit of the concepts sharing and persistence.

Our cross-compiler running under Microsoft Windows translates java sources directly into Intel machine instructions and runtime structures. The output is written as an bootimage on floppy disc. Together with a small loader stored in the bootsector the disc can be used to boot the Plurix OS. Alternatively a hard disc can be used but the compact size of our system allows us to perform testing with floppy discs without any limitations. Optionally we are able to burn the bootimage into the ROM of a network adapter with very short booting time (below 1 second).

The generated image will be loaded into main memory and the kernel can be executed. The kernel initialization routine checks if there is already a running DSM and if yes, tries to join this heap. Otherwise the machine opens a new DSM by relocating all objects included in the boot image into the DSM.

There are two options to solve this problem. The compiler in the bootimage can be used to recompile the total sources again and therewith allocating all classes and code-segments in the DSM. The integrated architecture of the PJC automatically registers all successfully compiled classes in the cluster-wide name service [11]. Another strategy is to include symbolic information in the bootimage and copy the memory objects in the bootimage manually in the DSM and register them in the name service.

Currently we use the second way to reduce possible error sources and to investigate problems and solutions for the bootstrapping of the OS. We have identified the need for special memory pools and new Java class attributes which are described in the following subsections. The results presented here are also required when the PJC is involved in the boostrapping process.

3.1 Memory Pools

The Plurix heap is implemented as a page based distributed shared memory and objects of different nodes may be located on the same memory page. Due to the page granularity a write access to one object of a page invalidates the total page and therewith possibly other objects. In such a case objects of other nodes may be invalidated by a write access or by false-sharing.

If an object is invalidated and the station accesses it again a page-fault occurs and the page-fault-handler requests the desired object from the DSM. Of course the code-segment of the page-fault handler must be protected against invalidation because it is required to fetch memory pages over the network. Beyond the page-fault handler also the DSM protocol and the network driver need to be protected and all objects that are transitively reachable from one of these classes or methods. We call these objects *SysObjects* (system objects) which must always be present at every node and must not be invalidated. Hence, SysObjects can not participate the transaction concept and can not reside in the normal DSM.

There are two alternatives to protect SysObjects against invalidation. Firstly, they can be placed in a memory area outside the DSM address space, or secondly they can be allocate in special areas within the DSM.

Objects residing outside the DSM can not reference any objects in the DSM, because the Backchain of such an DSM-object points to an address that is not unique (this address is not part of the DSM). If the DSM-object is relocated all references to it need to be adjusted using the backchain which is not possible if the backchain points outside the DSM address space. Therewith this first solution is not suitable for our DSM system and all objects should be stored in the DSM. The address space outside the DSM is a good candidate for IO buffers of devices. These buffers must not be shared nor be persistent and furthermore we save logical address space (the buffers for graphics adapters can be very large).

We have successfully implemented the strategy using special memory pools to protect SysObjects. Introducing several memory pools with different properties offers further performance benefits. In the following text we describe the different memory pools we use in our DSM address space for special objects:

- *ReadWrite-SystemPool*: a memory pool for read-write SysObjects. These are class-descriptors which contain static variables as much as instances of system classes. To protect these objects against false-sharing, the memory management allocates only one object per page.
- *ReadOnly-SystemPool*: this memory pool is reserved for code-segments and read-only class descriptors (the class has no class-variables) of SysObjects. These objects can not be replaced be normal DSM functions but may be replaced by special memory management functions. As all Objects in this memory pool are read-only false-sharing can not occur.
- *NonBackchain-Pool*: as described in chapter 2.1 each object contains a backward reference. This could lead to large chains, if the reference points to a base objects such as “String”. To remove a reference, the whole Backchain must be passed to search the pointer to be removed. Additionally objects can be invalidated by the Backchain if an entry is added or deleted. This is especially crucial for base classes that are referenced often. Whenever a new reference is generated, the object which contains it is inserted at the beginning of that Backchain and therefore the referenced class is invalidated. To avoid a lot of page-faults, base classes does not implement a Backchain. These classes can not be relocated and for this reason they are located in a special memory pool
- *NonTransactionPool*: some devices need a strict sequence of writing data into the device registers. If such a sequence is broken the device will hang. For this reason the corresponding device drivers should not be reset by the transaction concept. Contrary to the SysObjects these objects can be replaced normally.
- *ReadOnly-Pool*: during the startup of the system, code-segments can be allocated in a special memory area to avoid false-sharing. This is not really necessary, but it improves system performance.

In Plurix the DSM address space is statically divided into memory pools. The pools can be differentiated into the User-pools and the System-pools. The System-pools are shared within the cluster and each node allocates special objects in these shared pools. Furthermore each node has its own User-pool.

This strategy alleviates the false-sharing problem and the conflicts that would occur if all stations would try to allocate their objects in a common address-range. Current address space limitations will be overcome by the emerging IA64 architecture which we plan to investigate in future work.

3.2 The Station Object

Device drivers store their data in instances of the related device driver class. Each station needs its own instance for its private device and data sharing is normally not wanted. In this context the access to station private instances need to be investigated. They can either be accessed by passing the reference as a parameter to each method which requires access to the instance. Alternatively, such a refrence can be stored in a class variable. A well known example for the latter solution is the “System.out” class variable used for printing out messages on the console.

Using a class variable allows direct access by simply writing down the class name in the source text without thinking about parameter passing which tends to growing method signatures. Unfortunately, class variables are shared among all stations in the cluster and can not be directly used for our purpose. Moving such classes outside the DSM address space seems to solve the problem but we run into problems if referenced objects in the DSM are relocated (see also 3.1).

We solved this problem by a so called *Station-Object* which handles all station private driver and configuration instances. The class “Cluster” consists mainly of one class variable an array of “Station” instances, see source excerpt below.

```
public class Cluster {
  private Station stations[];
  ...
}
```

```
public class Station {
  NetworkAdapter net;
  ...
}
```

The instances of the class station are accessed as if they were class variables of the class “Cluster” by using the language extension “SYSTEM” (e.g. “SYSTEM.sound”).

Each station gets an unique index into the station array during boot-time. This index is stored outside the DSM address space at an location known by the compiler. The Plurix Java Compiler automatically generates the indirect access to driver instances via the “stations”-array if the keyword “SYSTEM” is used in the source text.

The class “Cluster” is declared as a SysObject, because the “stations”-array is necessary for handling page faults and requests for DSM pages and must not be invalidated. We are aware of the extension problem regarding the “stations”-array but the 32-Bit address space is sufficient for a dozens of stations where the presented solution is appropriate.

3.3 Class attributes

We have extended the Java language by two additional class attributes *attr_sys* and *attr_nonbc* to specify the memory pool to be used. Each Plurix memory block contains an additional flag entry that specifies if the block is SysObject, not-backchainable or read-only. The compiler automatically sets the read-only flag for all code-segments. The other flags are set by the compiler corresponding to the used class attributes (attr_sys and attr_nonbc). Furthermore, the compiler checks if a class has class variables. If not, the class descriptor can be marked as read-only. Whenever a new memory block is allocated, the memory management decides in which memory pool it should be placed based on the set flags.

4 Startup of Cluster Nodes

The Plurix system is started by a boot disc or by the boot ROM on the network adapter. The bootloader loads the bootimage generated by the cross-compiler into main memory switches to the protected mode and starts the primary kernel.

4.1 Starting the Primary Kernel

The primary kernel initializes the interrupt handlers and the memory management used to access or initiate the Plurix cluster. Which method the kernel uses depends on the station number, which is determined over the protocol by a special *isDSMalive* packet. If the booting station is the first node, the kernel initiates a new cluster. In this case all memory blocks of the bootimage are copied into the DSM, respectively into the memory pools according to the flags set by the compiler. During copying the memory blocks the backchain entries are build.

During the copying process all classes in the bootimage are registered in the cluster-wide nameservice. After copying all memory blocks into the DSM, the heap kernel is started.

If the station is not the first node in the cluster it requests all pages that contains SysObjects (around 21 pages with 4 KB each) from the DSM. These memory blocks are necessary to execute and startup the heap kernel.

4.2 Starting the Heap Kernel

The heap kernel itself is stored in the DSM in system pools. The station either generates this kernel by copying objects or receives the memory pages from the existing DSM heap. After all SysObjects are present, the page-fault-handler and the devices drivers for the network adapter are replaced and the new kernel starts. This method enables Plurix to start with different bootimages as long as the image contains a compatible network protocol. After the device drivers and the page-fault-handler are replaced, the kernel starts to configure the system.

4.3 Configuration

The Plurix kernel tries to configure the system by scanning the PCI bus. After detecting a device the kernel checks the nameservice for an appropriate device driver. If a device driver is found an existing instance of the driver may be reused or a new one created to start the device. All device drivers are registered in the nameservice by classID, subclassID and vendorID and a reference to the device driver class. The kernel generates a new driver instance for this station and registers the reference to this instance in the station object. After scanning the PCI bus the kernel continues by starting the Plurix loop.

Device drivers in Plurix extends the basic class Device. This class defines all methods that are necessary to handle a device in Plurix, see following source code. These are especially the methods for initialization and reinitialization.

```
abstract attr_sys attr_nonbc class Device {
        static int STOPPED = 0, STARTED = 1;

        Device next, sharedInts=null;
        int    state = STOPPED;

        abstract boolean detectDevice();
        abstract int reinitialize();
        abstract int initialize();
        abstract int unload();
        abstract int start();
        abstract int stop();
        abstract int ISR();

        int status() { return state; }
}
```

Initialize is used, if the device driver is used for the first time on this station. Due to the consistency of the Plurix heap, instances of device drivers can be present in the cluster, if the station joins the heap again. In this case, the user defined settings for this device should be used again and therefore it is not useful to generate a new instance of this device driver. Nevertheless the device have to be initialized and for this situation each device driver must implement the method reinitialize.

5 Experiences and Future Work

Our experience with object-oriented programming in the kernel is generally positive. We have developed the Plurix kernel and several drivers. Objects provide a cleaner design because of the encapsulation of data structures and the enforcement of well-defined interfaces. Furthermore, using a strongly typed language like Java eliminates several typical error classes (e.g. pointer arithmetic, type errors, index range checking) well-known from programming drivers in C.

The boostrapping of our system revealed the need for special memory pools to avoid invalidation of system-critical classes and instances. Furthermore device driver and configuration data need to be station private. Our station object implementation together with a compiler extension provides a reasonable mechanism to implement such private context in a DSM heap.

The current version of the Plurix system (including all drivers and a somple GUI) contains about 40.000 lines of source code resulting in 5.050 objects. These objects are distributed in the following categories:

- 870 SysObjects (864 read only),
- 4180 normal objects (4159 read only),
- 0 nonBC objects: (currently all nonBC objects are also SysObjects).

We tested the heap kernel version with the described memory pools on a cluster with 8 Pentium PCs connected via an 100 Mbit/s Ethernet. Furthermore we have shown a prototype at the CeBIT 2001 fair. In future we will continue our research to investigate different memory consistency models, false-sharing and pageserver implementations.

6 References

[1] "JavaOS", http://www.sun.com/javaos, 1997.

[2] N. Wirth, and J. Gutknecht, *Project Oberon – The Design of an Operating System and Compiler*, Addison-Wesley, 1992.

[3] J.L. Keedy, and D. A. Abramson, "Implementing a large virtual memory in a Distributed Computing System", in: *Proc. of the 18th Annual Hawaii International Conference on System Sciences*, 1985.

[4] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing", In *Proceedings of the International Conference on Parallel Processing*, 1988.

[5] M.P. Atkinson, and R. Morrison, "Orthogonal Persistent Object Systems", Very Large Data Bases Journal, 4(3):319-401, 1995.

[6] Morin C. and Puaut I., „A survey of recoverable Distributed Shared Memory Systems", Technical Report Nr 975, IRISA, France.

[7] M. Schoettner, O. Schirpf, M. Wende, and P. Schulthess, "Implementation Aspects of a Persistent DSM Operating System in Java", in: *Proceedings of the International Conference on Information Systems Analysis and Synthesis*, Orlando, USA, 1999.

[8] S. Traub, "Speicherverwaltung und Kollisionsbehandlung in transaktionsbasierten verteilten Betriebssystemen", PhD thesis, University of Ulm, 1996.

[9] A. Lindström, R. di Bona, A. Dearle, S. Norris, J. Rosenberg and F. Vaughan, „Persistence in the Grasshopper Kernel", Australasian Computer Science Conference, ACSC-18, pp 329-338, February 1995.

[10] M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, P. Schulthess, "Optimistic Synchronization and Transactional Consistency", to appear in: Proceedings of the 4th International Workshop on Software Distributed Shared Memory, Berlin, Germany, 2002.

[11] M. Schoettner, "Persistente Typen und Laufzeitstrukturen in einem Betriebssystem mit verteiltem virtuellen Speicher ", PhD thesis, University of Ulm, 2002.