A Kernel Running in a DSM -
Design Aspects of a Distributed Operating System

R. Goeckelmann, M. Schoettner, S. Frenz 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. Communication is achieved via shared objects in a Distributed Shared Memory (DSM). The consistency of this distributed memory is guaranteed by an optimistic synchronization scheme and restartable transactions. We contend that coupling object orientation with the DSM property allows quick system startup, simplified development of distributed applications and a type-consistent system bootstrapping procedure. The OS (including kernel and drivers) is written in Java using our proprietary Plurix Java Compiler (PJC) to translate Java source code directly into Intel machine instructions. We briefly illustrate the architecture of our DSM-based OS kernel and the resulting synergies for communication between applications and OS. We present advanced issues of memory management with respect to the DSM-kernel classes and strategies to avoid false-sharing .

1. Introduction

Typical cluster systems are built on top of traditional operating systems (OS) such as Linux, MacOS or Microsoft Windows and data is exchanged using message passing (e.g. MPI) or remote invocation (e.g. RPC, RMI) mechanism. As each node in a cluster runs its own OS with somewhat differing configurations and contexts, the migration of processes and objects is difficult for a number of reasons - it is typically unknown which libraries and resources will be available on the next node and on migration the entire context including relevant parts of the kernel state must be saved, serialized and transmitted.

As a consequence our Plurix OS has been specifically tailored for cluster operation and avoids these difficulties. The Distributed Shared Memory (DSM) accommodates all objects and offers an elegant solution for distributing and sharing data among loosely coupled nodes [1], [2]. Applications running within the Plurix DSM are unaware of the physical location of objects. A reference can either point to a local or to a remote memory block. During program execution the OS detects a remote memory access and automatically fetches the desired memory block. Plurix extends the DSM to a distributed heap storage (DHS), providing the benefit, that not only data but also the code segments of the programs are available on each node as they are shared in the DSM.

One of our major research goals is to simplify the development of distributed applications. Many DSM systems use weak consistency models to guarantee the integrity of shared data. Each programmer must explicitly manage the consistency of the data by using the offered synchronization mechanism [3]. This makes the development of applications hard and as an alternative to this Plurix uses a variant of strong consistency, denoted *transactional consistency* [4] which relieves the programmer from explicit consistency management.

Single-System-Image (SSI) computing architectures have been the mainstay of high performance computing for many years. In a system implementing the SSI concept, each user gains a global and uniform view on available resources and programs and obtains the same libraries and services on each node in the cluster. We extend the SSI concept by storing OS-kernel, run-time support and all drivers in the DSM. As a consequence we can implement a type-safe kernel interface.

2. Design of Plurix

Plurix implements SSI properties at the operating system level, using a page-based distributed shared memory. According to the SSI concept all programs and libraries must be available on all nodes in the cluster, hence Plurix uses a global address space shared by all nodes and organized as DHS containing both data and code. Sharing the programs in the DHS reduces redundancy concerning code segments and simplifies the administration of the system.

Plurix works in a fully object oriented fashion and is entirely written in Java. The development of an operating system and its drivers requires access to device registers which is not possible in standard Java. To support operating system and hardware level programming we have developed our own Plurix Java Compiler (PJC) with appropriate language extensions. The compiler directly generates Intel machine instructions and initializes runtime structures and code segments in the heap. Traditional object-, symbol-, library- and exe-files are

avoided. Each new program is compiled directly into the DHS and is thereby immediately available at each node.

Plurix is a lean and high speed OS and therefore able to start quickly. The start-up time of the primary node, which creates a new heap (installation of Plurix) or restarts an preexisting heap from the PageServer, is less than one second (excluding BIOS). Additional nodes which join the existing heap starts in approximately 250 ms. This quick-boot operation of Plurix is also crucial for achieving fast node and cluster recovery in case of (unlikely) critical errors.

2.1. Consistent Distributed Shared Memory

The transfer of the DHS-objects between cluster nodes is managed within the page-based DSM and takes advantage of the Memory Management Unit (MMU) hardware. The MMU detects page faults, which are raised if a node requests an object on a page which is not locally present. Each page fault results in a separate network packet which contains the address of the missing page (PageRequest). This packet is broadcast to all nodes in the cluster (Fast Ethernet LAN) and the current owner of the page sends it to the requesting node.

An important topic in distributed systems is the consistency of shared and replicated objects. In Plurix this is synonymous to the consistency of the entire DSM. Memory consistency is achieved using a new consistency model, denoted *transactional consistency*. Appropriately all actions are encapsulated in transactions. At the start of a transaction, write access to all pages is prohibited. If a page is written, the system creates a shadow image of it and then enables write access. All modified pages are logged and at the end of a transaction (commit phase) these addresses are broadcasted. All partner nodes in the cluster will then invalidate these pages. If an active transaction in a partner node detects a collision with the write set of the committing transaction it aborts and restarts later. In this case all modified pages are discarded and the previous state of the node is reconstructed, using the saved shadow images. A token mechanism makes sure that only one node at a time enters the commit phase.

2.2. False Sharing and Backchain

All page-based DSM systems suffer from the false-sharing syndrome, which occurs if two or more nodes access separate objects which nevertheless reside on the same page. If a node writes to such an object, other accessing nodes are forced to abort their current transaction. Such an abort is semantically unjustified if only the page but not the object is shared. To cure this problem some of the false-shared objects must be relocated to another page. This also implies that all pointers to the relocated object must be adjusted. Due to the substantial network latency in the cluster environment, it is not possible to inspect each object whether it contains a pointer to the relocated object. To adjust the affected references, Plurix uses the Backchain [5] which links together all references to an object, by recording the addresses of these pointers (see fig. 1). To reduce invalidations of remote objects when a new Backchain entry is inserted references on the stack are not tracked in the Backchain.

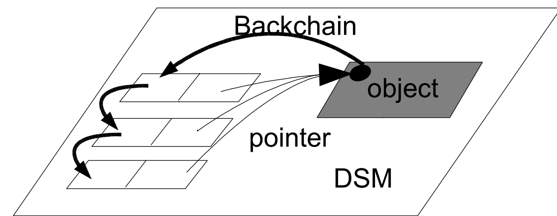

Figure 1 The Backchain Concept

3. A Type-Safe Interface for a DSM-Kernel

The SSI concept requires, that all nodes in the cluster have the same programs installed. In a distributed environment the easiest why to achieve this is to share both data and code of the programs. Protection of code segments from unwanted modification either by corrupted pointers or by malicious attacks can be achieved by using a type-safe language like Java. The requirement for type safety in the DSM affects also the interface to the OS. As data in the DSM is represented by objects and these data must be passed to the kernel, either the objects must be serialized before they are used as parameters or the kernel must be able to handle objects. Language-based OS development has been successfully demonstrated by the Oberon system [6].

3.1. Traditional Kernel interfaces

Traditionally, distributed systems are implemented as a middleware layer on top of a local OS which is often written in C and does not provide objects. The communication between the distributed system and the local OS takes place using primitive data types or structures. If the kernel cannot handle objects proper, they are serialized before being passed. This kind of raw communication does not provide type checks for parameters and signatures.

To increase system reliability and to limit programming complexity it is preferable to pass typed objects to the OS-kernel. Plurix was therefor created as microkernel OS not as a middleware layer. Since the kernel is written in Java a type-safe communication between the DSM applications and the OS is natural. All Java types and objects can be handed to the kernel methods. The programmer has no need to pay attention to the type of the passed object because this is checked by

the compiler and in some cases by the runtime environment. The kernel method obtains a reference and accesses the object directly, therefor data included in these objects need not be copied.

3.2. Inter Address Space Pointers

In systems without inherent type safety it is mandatory to provide separate address spaces for the OS kernel and for user applications. Some kernel classes and their methods must be continuously available on each node the straight-forward approach of implementing the system would be to place the kernel in the local address space. Local addresses are not shared and each node in the cluster will use local memory in its own way.

In a distributed memory system the separation between kernel and user address space would lead to a distinction between local- or NonDSM-space one the one hand and the DSM-address-space on the other hand. If in such an environment objects are used as parameters, references will point from the local- into the DSM address space. This kind of references reduces the cluster performances, as they inhibit the relocation of objects so that attenuation of memory fragmentation and false-sharing is no longer possible. The cause of this is, that the Backchain entries are not longer unambiguous if an locally referenced object migrates to another node and is afterward relocated. The Backchain will lead into the original nodes local memory and as addresses in local memory are not unique it is not possible to detect which node was originally specified. The correct forward reference can not be found and an adjustment of the local memory location, specified by the Backchain will lead to invalid pointers or even destroyed code segments (see figure 2).

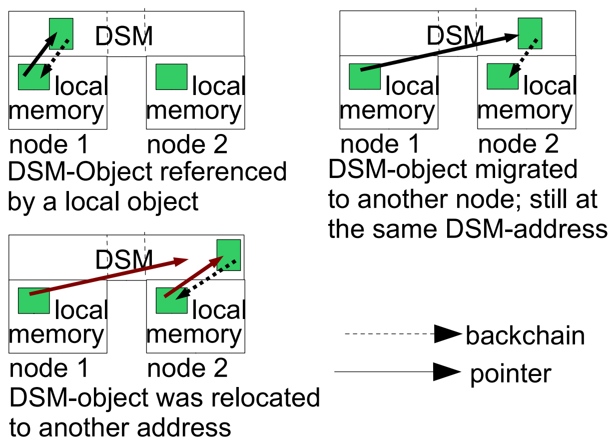

Figure 2 Migration and subsequently relocation of a DSM-object

As a consequence all objects which are currently referenced by a kernel instance must be marked as non-relocatable. As it is not possible to specify, which objects are used as parameters for kernel methods this could lead to unsolvable false-sharing if objects which are used by kernel instances on different nodes reside on the same page. Such a situation will derogate cluster performance dramatically, hence direct pointers from the Non-DSM into the DSM-address-space must be avoided.

Another design consideration is how kernel methods are called from (DSM) applications. Two alternative methods are conceivable:

1. Software Interrupt: most traditional systems invoke the kernel methods using OS- or system-traps. Here the switching to another address space is automatic. But since the software interrupt itself is intrinsically untyped the question arises how to pass along typed parameters. One possible solution is to pass data to the kernel through a fixed address. If an object is passed as a parameter, this location would contain a reference to the object. Since each kernel method requires different parameter sets, this object must be of a generic type and any object may be passed. OS methods need to explicitly check the parameter type because this could not be done by the runtime environment. Explicit checking requires additional programming effort, reduces the system performance and creates opportunities for programming errors at the system level.
2. Object oriented invocation: OS methods are invoked in an object oriented fashion via direct pointers. This implies that on all nodes the kernel classes and their methods have to reside at the same respective address because application classes can only have one pointer to a given kernel class. If the local address of some OS class varies from node to node then application references might point to invalid addresses and the corresponding kernel methods could not be called on some nodes (see figure 3). Direct pointers require, that each node in the cluster is running the same kernel, which may never change during runtime. Otherwise all pointers in the applications, which reference kernel methods require adjustment. Adjustment requires that all kernel methods must contain a Backchain which points from Non-DSM into the DSM creating the problems described above.

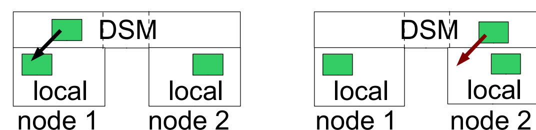

Figure 3 Invalid reference to kernel methods

Both techniques give rise to an additional problem. The compiler is running in the DSM and any new program is automatically created in the DSM. If the new program is a device driver (which typically resides in kernel space) the code segments must be transfered from the DSM into the Non-DSM address space and this must occur simultaneously on each node.

Our implemented solution which solves the challenges above by removing the kernel from the local memory address space and placing it into the DSM. Further benefits of this approach are described in the following section.

4. Extending the SSI Concept

We extend and refine the SSI concept by moving the OS and the Kernel into the DSM. Local memory is only used for a few state variables for the network device drivers and -protocol and for the so called Smart-Buffers, which help to bridge the gap between not restartable interrupts and transactions [7].

4.1. Benefits of a kernel running in the DSM

If the OS kernel runs in the DSM space, parameter passing between applications and kernel is elegant and all objects can be used as parameters. Kernel methods are called directly as described in section 3.3. There are no references between different address spaces. Since all device drivers now reside in the DSM the problem of moving recompiled drivers from the DSM into the kernel space vanishes. Because the code segments of the kernel methods are in the DSM code duplication is avoided.

Some interesting questions surfaced when moving the kernel and OS into the DSM, but before we describe these topics and our corresponding solutions, we describe the memory management of Plurix and the object allocation mechanism underlying our solution.

4.2. Distributed Heap Management

A basic design topic of the Plurix system is the page-based DSM – giving raise to the false sharing problem. The allocation strategy of the memory management must try to avoid false sharing wherever possible. Collisions between concurrent transactions during the allocation of objects in the DHS must be minimized in order to avoid serializing all allocations in the cluster. To achieve those goals, Plurix uses a two stage allocation concept consisting of *allocator*-objects and a cluster memory manager. The partitioning of the memory space must not be static as this would limit the maximum size of the objects.

To avoid fragmentation of the logical address space the allocators are comparatively small and will only accommodate small objects. The cluster memory manager is used to create large objects and allocators. Using allocators for large objects would lead to large allocators and thereby to a static fragmentation of the heap. The alternative for this is to limit the size of the allocators and thereby the maximum size of the DHS-objects which is unacceptable.

Allocator-objects represent a portion of empty memory. The size of an allocator is reduced for each allocated object. If it is exhausted a new allocator is allocated by the cluster memory manager.

When a new object is requested the size of the new object is considered. If it is greater than 3 KB it is directly allocated by the cluster memory manager. To avoid false sharing on these objects, their container is increased to a multiple of 4 KB (page granularity of the 32-bit Intel architecture). Each such object starts at a page border and consumes entire pages, hence these objects do not co-reside with other objects on the same page. Most objects in Plurix are less than 3 KB and are created by an allocator. As each node has its own allocator, collisions during allocation can only occur if a large object is allocated or if an allocator is exhausted and a new one must be created. This technique substantially reduces the frequency of collisions.

The benefit of the two level allocation of objects is that small objects from one node are clustered in the memory. As a consequence collisions do not occur during the allocation of small objects and are rare for large objects. As large objects are not allocated within the allocator, its size can be limited, without limiting the maximum size of the objects. No static division of the memory is needed and therefore no static fragmentation is created.

Generally speaking objects can be divided into two categories: Read-Only (RO) and Read-Write (RW) objects. False-sharing on RW-objects is reduced by the mechanism described above. To further reduce false-sharing it is reasonable to make sure, that RO-objects like code segments and class descriptors without static variables do not co-reside with RW-objects on the same page. These objects are only written by the compiler, which uses separate allocators for RO-objects.

If the entire system is running in the DSM, some code segments and instances of classes must be protected against invalidation, because these objects are vital for the system (SysObjects). These are all classes and instances implementing the Page-Fault-handler, the DSM protocol and the network device drivers. As SysObjects reside in the DSM they might be affected by the transaction mechanism. In case of a collision the corresponding page risks to be discarded and o system will hang, as the node is no longer able to request missing pages.

The protection of SysObjects against invalidation is easily achieved by providing two additional allocators. SysObjects are either code segments or instances of SysClasses. As described above, code segments are only

written during compilation, later they are read only. The compiler will create the new kernel classes in a different memory area, using a special allocator for each compilation and after that the remaining part of the last used page is consumed by a Dummy-SysObject.

RW-SysObjects are instances of SysClasses which are meaningless for remote nodes and pertain only to the node which has created the instance. For this reason RW-SysObjects are not published through the global name service and no other nodes can access a RW-SysObject. The only case where a RW-SysObject could be invalidated is as a result of false-sharing. To prevent this, each node acquires a SysRW-Allocator during the boot phase. All instances of SysClasses are allocated in this private allocator, so that there are only SysObjects from one node on the same page.

These two additional allocators and the described techniques to use them are sufficient to protect all SysObjects against invalidation during run time.

4.3. Restart of device drivers

In case of an abort the state of the entire node is reset to the state just before the current transaction was started. Hardware devices, however, can not be automatically reset and the device driver programmer must implement an *Undo*-method, which is called by the system in case of an abort. This method has to ensure that both the state of the hardware and content of the state variables in the device driver object are reset. To make this possible the state of all devices before the transaction must be conserved.

An example for such an Undo-method is shown for graphics controller devices. In this case the current On- and Off-screen memory-area in the display card must be reset. Since between two transactions the On-and Off-screen contains the same data, it is sufficient to reset the Off-screen memory and afterward copy this value to the On-screen area. This is easy to implement as most graphics controllers contain substantial amounts of memory for textures and vertices. A small part of this memory can be used to save the committed state of the graphic controller. After the commit phase the current On-Screen area is copied into this separate memory area and can be restored if necessary.

The serial-line controller though is more difficult to handle. In case of an abort it is not possible to “undo” the data sent. In fact there are two solutions for this problem: either the serial line protocol is able to handle duplicated and corrupted data or the driver makes use of smartbuffers. This special buffer type contains uncommitted date which will only become visible to the device after a successful commit phase, so the device can only access committed and valid data.

5. Experiences

Moving the kernel into the Distributed Heap Storage and building on the SSI concept leads to a type-safe kernel interface and clears the problem of pointers between multiple address spaces. OS methods can be invoked in a perfectly natural and object oriented fashion and it is now simple to take a snapshot of the system.

The current version of Plurix is very stable and runs in the cluster environment without excessive collisions. Our two level allocation strategy minimize false-sharing and collisions within the allocation management. When objects are created by an application and published to other nodes, the allocation mechanism can not prevent false-sharing but we are working on a monitoring tool to detect false-sharing spots. Relocation of objects to disentangle false-sharing is already available.

Plurix uses an incremental garbage collection algorithm which collects garbage (including cyclic garbage) without stopping the cluster. The detection algorithm for cyclic garbage checks for all objects whether they are part of a garbage cycle. Advanced heuristics for detecting false-sharing and garbage cycles are under study.

6. References

[1] 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.

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

[3] Amza C., Cox A.L., Drwarkadas S. and Keleher P., „TreadMarks: Shared Memory Computing on Networks of Workstations“, Proceedings of the Winter 94 Usenix Conference, pp. 115-131, January 1994.

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

[5] S. Traub, “Speicherverwaltung und Kollisionsbehandlung in transaktionsbasierten verteilten Betriebssystemen”, PhD thesis, University of Ulm, 1996.

[6] N. Wirt and J. Gutknecht, „Project Oberon“, Addison-Wesley, 1992.

[7] T. Bindhammer, R. Göckelmann, O. Marquardt, M. Schöttner, M. Wende, and P. Schulthess, “Device Programming in a Transactional DSM Operating System”, in: Proceedings of the Asia-Pacific Computer Systems Architecture Conference, Melbourne, Australia, 2002.