Distributed Virtual Address Space Management in the MoDiS-OS

Markus Pizka
Technische Universität München
Department of Computer Science
80290 Munich (Germany)
pizka@in.tum.de

Keywords: operating systems, distributed systems, parallel systems, memory management

Abstract

This reports motivates and explains concepts developed in the project MoDiS to organize large virtual address spaces comprising fine-grain concurrent computations in parallel and distributed environments. The single distributed address space is adaptively partitioned by a dynamic set of cooperating managers. The partitioning scheme is decentralized and scales with growing system configurations. Deficiencies as known from centralized or static organizations are prevented. In contrast to common operating systems, thoughts have also been given to possible thread stack and heap overflows and collisions. Both stacks and heaps associated with a thread are realized non-contiguously with linear stack and heap segments to enable the desired exploitation of the possibly large virtual address space. Distribution of data is coupled with garbage collection and based on objects instead of pages while still making use of the hardware faulting mechanism. Implementation is based on "off the shelf" hardware components. Crucial for the efficiency of this approach is a thorough top-down oriented construction of all operating system entities comprising the compiler and libraries as well as the kernel.

Contents

Observations

1.1 Unsatisfying Operating System Technology

The acceptance of distributed and parallel[1] processing techniques in practice lacks far behind the expectations associated with the tremendous computing power provided by ubiquitous high-speed interconnected workstations. This is mostly due to a comparable rate of complexity coming along with it. Such platforms tend to either burden the programmer with additional concepts and their effects, or demand load and memory management tasks from the resource management system that are hard to fulfill. Hence, to correct this situation, development of distributed systems has to be simplified with amongst others adequate programming concepts. Additionally, new methods for automated yet efficient application transparent resource management have to be emerged.

Definition: 1 (Purpose of an Operating System)
The purpose of an OS is to release the application level from difficult, repetitive, or – due to rights – impossible tasks which can be performed without significant losses transparently by the system.

History of operating systems (OS) shows that management tasks are handled at the application level only as long, as powerful OS solutions are missing. For example, early overlay techniques [Flo89] for computers with small main memories have been replaced with OS and hardware support for large virtual address spaces (VA) combined with paging. Similar OS shortcomings can nowadays be observed in distributed and parallel environments reflected in application-integrated resource management decisions. In fact, on parallel or even distributed platforms OS technology drastically fails to comply with its objective target as stated in definition 1. In such environments, applications still have to handle many resources by themselves e.g. perform load balancing or special handling of sharable memory regions in case of distributed shared memory (DSM).

Undoubted, memory management as a fundamental task of an OS should be performed completely transparent to the application level. This obvious statement is often violated because of the cost to integrate functionality supporting parallelism and distribution into all management instances including the compiler, runtime system, and the kernel. Overcoming this deficit is a milestone of major importance for the transition form centralized and sequential to distributed and parallel processing.

Outline Section 1.2 sketches the impact of several hot topics in OS technology on memory management. In 1.3 the deficiencies revealed are briefly compared with methods used in existing systems followed by a description of the system model underlying the work presented in this paper in 2.1. Fundamentals of the adaptive distributed OS architecture and considerations concerning its implementation are found in 2.1.2 & 2.1.3. Sections 2.3 and 3 detail the techniques developed for efficient single address space management while focusing on distributed partitioning and changes in stack and heap organization. Information about the approach taken to incorporate DSM and garbage collection functionality is given in 4. This paper concludes with information on the current state of the project and summarizing results in section 5.

[1] Throughout this paper *concurrency* and *parallelism* are used as synonyms

1.2 New Features and New Flaws

Multi-tasking OS usually provide separate address spaces for processes. In order to share data amongst processes, IPC interfaces such as *shared mappings*, *signals*, or *sockets* along with error prone techniques like pointer swizzling have to be used. Of course, tight coupling of processes needed for cooperative parallel algorithms can not be achieved this way without considerable overhead.

By employing one large address space for all processes as supported by modern 64bit architectures this and other problems can be evaded. Each memory object is identified with its unique memory address instead of separately maintained object identifiers. Therefore, object accesses are uniform and can be performed efficiently.

Using virtual addresses as globally unique identifiers seems to be extremely helpful especially in distributed environments because it simplifies naming, sharing, and migration, as well as it eases the enforcement of persistence for distributed objects.

1.2.1 Multi-Threading and Overflows

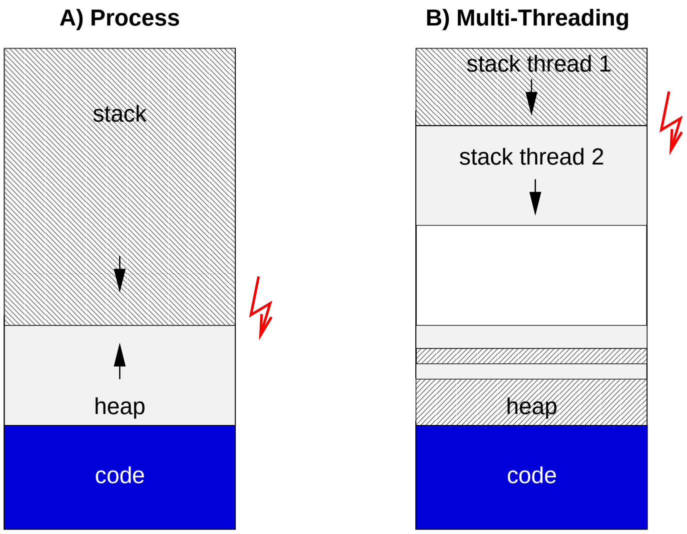

Figure 1.1: Single and multi-threaded VA partitioning

Usually the VA is divided into partitions as sketched on the left hand side in figure 1.1. Besides static code segments, one stack and one heap grow and shrink in opposite directions. A collision of stack and heap implies that no more free virtual addresses are available and an irreparable error state has been reached. In reality, exhausted physical memory or shared libraries mapped somewhere in between stack and heap will cause faults in advance. This situation is usually accepted although it decreases reliability, because only one computation is directly affected.

In parallel systems with multiple threads executing in one VA, each thread receives a dedicated stack. Unfortunately, multi-threading is a typical example for a bottom-up constructed and weakly integrated concept. It is provided to the application level with hardly any further support by the memory management system. The right hand side of figure 1.1 illustrates a new severe problem. Thread stacks eventually collide, although the VA is not close to be exhausted. In single address space systems, malfunctions of this kind might affect many independent applications making such approaches insufficient reliable. As a matter of fact, this problem stays unsolved in all thread implementations known by the author. Some libraries allow the definition of custom stack sizes if the default size (varying from 16k to 1M depending on the implementation) does not seem to be sufficient. Of course, this shifts the problem to the programmer contradicting the goal of simplicity and even worse, is no solution in case of recursion or incremental extensibility. In general, neither stack size demands nor the number of threads is statically predictable.

1.2.2 Extensible Systems

Currently, a broad spectrum of research activities e.g. [vDHT95] investigates methods to dynamically construct complex systems aiming at enhanced adaptability with higher quality, less effort and better performance. Architectural changes in this direction decrease the possibilities of static analyzes and therefore impose further restrictions — also on memory management techniques. For example, as discussed in newsgroups, static prediction of stack sizes becomes nearly impossible.

```
> Date: 4 Mar 1997 19:25:43 GMT
> Nice idea ... but ...
>
> What about using function pointers where you don't
> know where your function is (in this case I guess the
> max stack requirement for any function will do) or
> run-time linking where you can't know the stack
> requirements for the code because it might not have
> even been written yet ...
```

Executable code must also be placed in dynamically growing and shrinking partitions preferably without programmer intervention, creating further sources for overflows and collisions.

1.2.3 Distributing the Virtual Address Space

Distributed systems developer tend to statically bind node information to virtual address ranges, by using some high or low order bits as workstation identifiers. Besides the simplification for locating objects this approach has several disadvantages. For example, object migration requires costly pointer swizzling and the maximum size for allocatable objects becomes unnecessarily restricted. Hence, binding location information on this level of abstraction opposes the intentions of the single address space concept.

1.2.4 Mapping Virtual Memory to Nodes

Mapping to nodes refers to the question how virtual addresses are assigned to workstations in the cluster if not already determined by hardware-related static partitioning. Providing the abstraction of a distributed shared memory (DSM) [Li86] seems to be a promising approach to exploit distributed storage facilities with existing programming paradigms. In reality, DSM systems suffer from two drawbacks.

First, most DSM implementations do not provide the desired level of transparency. Programmers have to cope with new concepts such as allocating and freeing sharable segments. In addition to this, handling dynamic data structures within shared segments is most times up to the programmer due to a lack of combination with dynamic storage allocation techniques. As a consequence, transparency of access is missing and applicability is limited.

Second, DSM systems often only provide poor performance. Bottom-up constructed DSM systems are oriented on hardware properties usually employing a number of hardware as management unit. Pages are huge compared to the granularity of application-level objects, such as integers. This inadequacy leads to the effect of *false sharing*, resulting in a strong performance degradation for a wide range of applications. In contrast to this, object-based DSM systems allow individual handling of application objects but are often realized "all in software" producing high constant costs for runtime management. As DSM is still a hot topic in distributed processing several important issues have not yet been investigated such as advanced replication control [PT98] in DSM environments, or selective creation and elimination of replicates to support long term running systems.

1.2.5 Requirements and Goals

The difficulties revealed and additional experiences such as with dangling pointers are summarized in following incomplete[2] list of requirements:

1. Simple and safe application programming interface:
   (a) Automatic collection of unused objects (garbage).
   (b) Support for concurrent light-weight activities within one VA.
   (c) Uniform and location transparent creation and access to objects.
   (d) No distinct limitations on the amount and size of allocatable objects besides the size of the VA and existing physical resources.
2. Time and space efficient automatic management:
   (a) No distinct constant performance deterioration.
   (b) Scalability with growing configurations.
   (c) Adaptive management of heap, stack and code.
   (d) Exploitation of existing hardware features.
   (e) Fast remote accesses to objects of any granularity.

Each item of this list has numerous consequences. For example, 2a drives optimization of local processing to avoid overhead relative to sequential systems. Because of 2b, defacto improvements should be noticed if additional resources are consumed. Furthermore, 2b necessitates decentralization of shared data structures and elimination of synchronization as far as possible which in turn requires sophisticated protocols, partitioning algorithms, et cetera. Item 2c addresses transparent solutions for overflows and collisions, support for extensibility as well as allowing for thread migration including data and code. This explosion of limitations and requirements points out, that respecting all of these items is probably only possible in a top-down oriented approach.

The goal of the work presented in this paper, is to develop memory management methods as part of a distributed OS guided by definition 1 and the requirements listed above. This distinguishes it from equally important work where details such as different coherence protocols [TF95] are investigated.

1.3 Related Work

In fact, hardware supported paged *segments* as used in former OS like MULTICS on Honeywell 6000 machines [Tan92] would nowadays be helpful to efficiently solve some of the problems mentioned. Thread stacks, heaps and extensible code fragments could be placed in separate segments without the danger of collisions. After

[2] Of course, items such as protection would have to be added.

years of predominant sequential processes with private VA these features are missing.

Stacks Concurrent Oberon [ARD97] for example substitutes segments with compiler inlined stack checking code and a predefined limit of 128k for the stack of each "Active Object". Overflows below the limit are detected and corrected with additional allocations. Linearity is preserved and consumption of physical memory is adaptive. Unweakened *linearity* of stack spaces on the other hand, disables the exploitation of the whole VA for larger stacks. In other words, OS supported stack adaption is limited and demands may only vary within narrow boundaries.

Using restricted pages at the end of the stack for the detection of overflows combined with deferred mapping as for example in Solaris [Sun95] is fast, compatible, and mostly independent from the compiler. While detection is cheap, correction may be extremely difficult. Overflows stay undetected as long as objects located on the restricted page are untouched, although other objects of the same frame or even their addresses are used. At the time of detection, registers and objects may have to be examined globally along with pointer swizzling in order to correct the overflow. Hence, avoidance or early detection should be preferred instead of late correction. Compiler-based approaches as for example dynamic stack probing implemented in `gcc` [Sta95], also suffer from *late detection*.

In [HL93] problems of maintaining multiple stacks are described. The proposed solution is to implement the conceptual *cactus stack* as a per processor *meshed stack*. Although this technique is an improvement it also requires expensive garbage collection of activation records within the meshed stack and obstacles hardware enforced protection.
The technique presented in this paper is based on dynamically extending and splitting stacks which provides similar space but superior time efficiency.

Memory Allocators W. Gloger's `ptmalloc` [Glo97] implements a parallel memory allocator based on POSIX threads [IEE95]. Lock contention is reduced by employing multiple heaps with separate locks. Performance improvements of nearly factor 3 on Solaris/Sparc are the benefit. Unfortunately, application-specific properties are ignored. Objects are placed on the first currently unlocked heap. Hence, consecutively allocated objects become scattered through the VA which has negative effects on locality of reference and fragmentation.

The memory allocator `Mmalloc` [Hae] supports multiple dedicated heaps within one VA. Each heap grows and shrinks separately using the system call `mmap` but has to be linear. Similar to stacks, linearity restricts dynamic adaption and full exploitation of the VA as only overflows can be solved. Collisions are only detected.

Garbage Collection Extensive work has been performed in the context of memory allocation strategies and garbage collection (GC) in uniprocessor environments [Wil94, ea95]. Furthermore, a comprehensive comparison of distributed GC methods based on extensions of centralized algorithms such as weighted references [Cor91] as well as new distributed shared stores allowing for fault tolerance and replication is given in [PS95]. It leads to the conclusion, that integrated solutions are superior to layering, hierarchical methods providing locality are mandatory, and most of all, distributed GC is still unsatisfactory. For example the language-based software DSM LEMMA [ML95] for ML [HMT89] uses global and local two-space GC. Although it provides "useful speed-ups" it is also recognized, that "there is considerable work to be done in a number of areas". With a tight coupling of programming model, GC, and object distribution, we expect the ability to reduce the cumulative overhead for distributed memory management.

DSM Li's Ivy system [Li86] was the first implementation of a page-based DSM. Since then, variations of this idea with weakened forms of consistency and other improvements were developed in projects such as Quarks [SSC98] or TreadMarks [ea96b]. Although most of these projects provided technological progress, they all suffer from being based on page sizes and using one uniform coherence protocol at once for all managed objects. The consequences are false sharing and inefficient protocols for a large number of objects. These problems are partially circumvented in software-based DSM systems such as Midway [BZS93], CRL [JKW95] and Munin [Car95]. But especially the latter fails to provide simplicity and transparency. In [Car98] the situation of DSM systems after almost 15 years of research is characterized as "very little real world impact". It is stated, that the reasons are either "pretty lousy" performance or inapplicability because of significant user input. Future DSM research will focus on support for distributed services and wide area applications in less specific contexts. We argue, that this in turn prerequisites seamless integration of DSM features into distributed OS architectures.

Single Address Spaces and Protection The question of how to define and enforce protection in a single address space has been investigated in numerous projects such as Mungi and Opal [Elp93, CLBHL93]. An overview and comparison of these approaches amongst others can be found in [ea96a]. Commercial processor designs slowly start to incorporate support for advanced protection in a large address space. For example, SUN provides TLB[3] supported clustering of pages to *page contexts* with its V9 architecture. Unfortunately, there are no means to hierarchically structure page contexts, yet.

[3] Table Lookaside Buffer

Address Space Structuring

2.1 Basics of the Project MoDiS

In MoDiS (Model oriented Distributed Systems) [EW95b, EW95a] a top-down driven and language-based approach is followed to systematically develop efficient yet simple to use concepts. Homogeneous and distribution transparent language concepts allow the development of parallel algorithms with varying degrees of parallelism, granularity, and cooperation. Objects representing new functionality (especially applications) dynamically extend the running system, forming a globally structured system encompassing applications and OS functionality.

2.1.1 Programming Model

INSEL [Win96] provides the grammar to the more formal MoDiS concepts. It is a high-level, type-safe, imperative and object-based programming language with explicit tasking parallelism. Encapsulated objects are dynamically created as instances of class describing objects, called *generators*[1]. Generators can be nested within other generators or instances and vice versa. Objects may either be active (*actors*) or passive determined by the generator. Each actor defines a separate flow of control and performs its computation concurrently to its creator. Actors may interact directly in a synchronous rendezvous (message passing) or mediately via shared passive objects (shared memory).

Named objects are identified by exactly one reference within a function or block while *anonymous* objects are identified by references which can be passed, duplicated and deleted. No further pointer arithmetics are supported. All objects are automatically deleted according to their conceptually defined lifetime [PE97]. The lifetime of an anonymous object depends on the lifetime of the generator for references to this object, whereas named objects depend on the enclosing object or method.

2.1.2 Scalable Operating System Architecture

To enforce transparent, scalable and adaptable distributed resource management, we developed a reflective management architecture [Gro96, GP97]. Though originating in MoDiS, this architecture is also highly applicable in other parallel or distributed systems. The key idea is to associate a *manager* with each flow of control on the conceptual level. In the context of INSEL, one actor and all its termination dependent [PE97] passive objects are clustered to *actor-contexts* (AC). Each AC is guided by exactly one manager, which has to satisfy all demands for resources of its AC. Besides standard tasks such as allocating memory for the stack, heap and code, a manager might also have to enforce coherence of replicates, initiate migration, or enforce access restrictions. Conflicts, such as overflows, concurrent heap allocations, or processor allocation are solved by inter manager cooperation.

This management scheme is top-down oriented as it is constructed independently from the physical hardware configuration. Furthermore, it is scalable, because it does not have a potential bottleneck and the number of managers corresponds to

[1] similar to type or class in common languages

the number of actors. Adaption is assisted due to the close relation of management with dynamically changing requirements of application-level objects.

2.1.3 Implementation Philosophy

Crucial for the efficiency of this approach is a systematical realization of the conceptual managers. Prototypes on top of Mach [Win96] and HP-UX [Rad95] have shown, that limiting the implementation to an adaption layer in an otherwise adopted environment does cause unacceptable disadvantages for the long term goal.

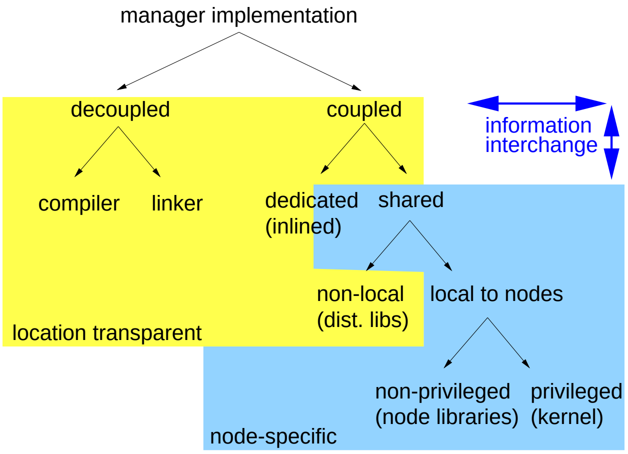

Figure 2.1: Instances used to implement managers

Based on these experiences, any software instance involved in resource management is now regarded as implementing parts of managers. Figure 2.1 illustrates typical different possibilities. Efficient and flexible managers are constructed by tightly integrating the capabilities of this framework by means of bidirectional information interchange and coordination of actions. The distributed manager architecture with this implementation concept leads to following redefinition of the term “operating system” in the context of MoDiS:

Definition: 2 (Distributed Operating System)
The OS is the complete management of the distributed computing system. It consists of cooperating process managers implemented by an integrated tool set.

The assignment of functionality to a certain instance e.g. *dedicated* (inlined) or *privileged* (kernel) must be based on sound criteria. For example, realizing manager functionality in the shared (functionality or data implementing more then one manager) portion constructs flexible interpreting services while the utilization of decoupled techniques leads to more static production characteristics. Transition from interpretation to compilation is soft without a strict separation between statics and dynamics and management is regarded as continuous regulation.

2.2 Memory Management Subsystem

2.2.1 Architecture

Figure 2.2 provides an overview of the memory management subsystem. Note, that the abstractions shown, represent conceptual *levels* in contrast to *layers* which

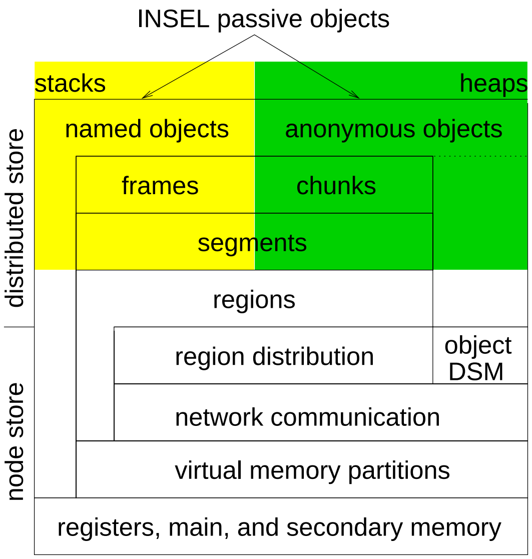

Figure 2.2: Memory management levels

would already imply a certain style of implementation. A horizontal marker separates the distributed – location transparent – portion from the storage subsystem on each workstation. The former splits up vertically into stack and heap down to the level of segments. As sketched in this figure, two orthogonal levels of distribution. First is the distribution coarse grain memory regions while the second is an object-based DSM, migrating and replicating individual heap objects. This separation of distribution functionality instead of a uniform low level page transportation layer is a prerequisite to develop mostly independent and in turn powerful strategies for object sharing and VA partitioning. Each object placed on heap is a sharable object per definition. If named objects are to be shared among distributed entities they are transparently transformed into anonymous objects by the compiler. While anonymous objects are mapped onto chunks and the object DSM, named objects are mapped onto activation frames or registers. Similarly, memory regions are either bound to node virtual memory or become dynamically distributed. These shortcuts represent flexibility which is exploited by the OS to improve performance.

2.2.2 Node and Shared Partitions

Although the goal is to provide a single distributed address space it proofs to be helpful to preserve some addresses for node-specific purposes. Objects only locally referenced or low level data structures reflecting the local state of a node, such as kernel code, communication buffers, etc. are placed in the *node partition*. Interpretation of addresses in this range is node dependent. Among the advantages are:

+ No need for coordination, migration, or replication

+ Fast address translation and object location

+ Simplified enforcement of protection

+ Exploitation of hardware features (TLB lock, etc.)

The major part of the address space is *shared* amongst all nodes with addresses uniquely identifying objects. Figure 2.3 indicates that the internal organization of

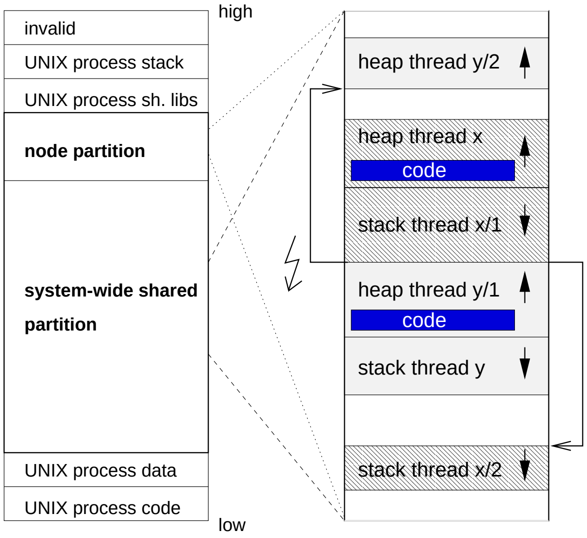

Figure 2.3: Memory layout

the system-wide shared and the local partition is identical. Both consist of ranges used as stack or heap segments for actors performing concurrent computations. Collisions or overflows are transparently solved. In fact, each actor may allocate memory in both partitions. The manager attribute `node_allocation`, inherited from the creator, determines the kind of allocation and can be changed with a privileged system call. Usually, this feature is only used to satisfy management requirements and is transparent to the application level.

Current state of the project still requires a UNIX host system. The dynamic loader of the chosen host system Sparc/Solaris does not support initialization before shared library initialization. Therefore, some partitions of the address space are preserved for the UNIX process environment. Start addresses and sizes of all partitions are fix.

Based on this partitioning, the memory subsystem bootstraps as follows: First, runtime data structures of boot AC managers are created and initialized within the UNIX data section on all nodes. Afterwards, the boot ACs themselves and other node-specific actors such as network communication handlers are created within the node partition. Finally one boot AC becomes elected as the boot master, switches to global allocation, and starts with the creation of distributed ACs.

2.3 Memory Regions

Virtual addresses are dynamically spread to ACs. Fur this purpose, both, the node and the shared partition are internally structured into disjunct memory *regions*.

Definition: 3 (virtual memory region)
A virtual memory region is a complete interval of virtual addresses starting and ending on page boundaries.

Because the region concept mainly aims at overcoming the physical distribution of workstations, this section will concentrate on the shared partition. Most of the explanations also hold for the node partition, with the difference that network communication has no impact.

```
region_t region_get  (pref_addr,min_size,direction)
void     region_put  (addr     ,size )
bool     region_split(addr     ,size )
bool     region_merge(addr1    ,addr2)
```

Figure 2.4: Region interface

The dedicated runtime portion of an AC manager calls `get` and `put` of its shared portion to dynamically allocate and free regions. The arguments of `get` specify a preferred starting address, the minimum required size, and positive or negative orientation to pass information about the intended usage of regions as heap or stack space for a certain AC. Split and coalescence (`merge`) of regions are prerequisites to keep fragmentation under control. Internal versus external fragmentation is dynamically tunable. First, the preferred address and the minimum size are only guidelines instead of accurate values. And second, preferred continuous allocations as a consequence of stack and heap growth are anticipated.

2.3.1 Distribution Concept

Distribution of the VA has to be scalable to support growing hardware configurations as well as dynamic software systems consisting of parallel computations with varying quantity and granularity. Scalability in general, is based on decentralization to circumvent bottlenecks and the reduction of synchronization. An eligible method should also meet diverging requirements of applications by exploiting application-level knowledge as far as possible. Furthermore, ancillary conditions resulting from the requirements listed in 1.2.5 must be respected. For example, addresses of shared objects should not be used to code e.g. workstation identifiers.

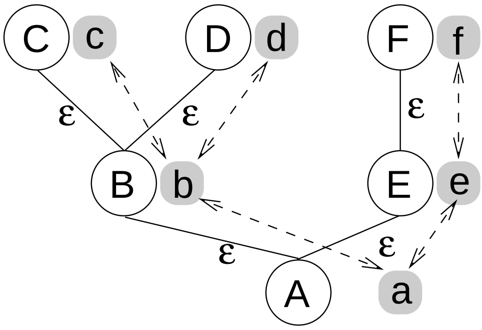

Figure 2.5: Distribution Strategy

According to the general top-down orientation, the management model introduced in section 2.1.2 with its structure of termination dependent actors is used to split the task of VA distribution among the AC managers as shown in figure 2.5. At first, the complete range of addresses is assigned to manager a of the root AC A. In the path of computation, new ACs are created. Each AC is provided with regions for autonomous use by its creator. If this initial provision proofs to be insufficient at a certain point of execution, additional regions are dynamically requested by either asking the father within the termination dependency (ϵ) i.e the creator, or reclaiming regions formerly delegated to children. At the time of termination, each AC returns its regions back to its creator.

Obviously, this high-level strategy provides scalability and adaptability. It also allows to exploit the complete VA with little external fragmentation because the recursive style of cooperation allows to retrieve available regions globally. For example, requests of c are satisfied with regions retrieved from f if necessary.

2.3.2 Implementation Based on Resource Pools

The main characteristics of this high-level scheme are intense and cascading cooperation among managers whereas their number is large compared to workstations. Straight-forward implementations with chains of signals or even network messages would deliver unacceptable performance. The strategy to forward regions to sons has to cope with large numbers of small regions, if many light-weight actors are forked, as well as just a few but extremely large regions in case of recursion. But in general, neither source analyzes nor runtime monitoring could provide the information needed to steer a suitable policy with little tolerance considering limited local resources. Though, the resource competed for – unallocated virtual address intervals, is available in abundance (considering 64 bits) — somewhere in the system.

Analogical to strategy and mechanism, these problems are solved by thoroughly separating levels of abstraction and connecting methods on different levels via sound mappings.

The characteristics depicted indicate, that region distribution belongs to a typical management task class where reducing low-level communication by means of group communication is crucial. Because dynamic grouping based on the node of execution provides a natural way to reduce network messages, manager tasks of this class are mapped onto node *resource pools*. Notice, cooperation is in no way limited to exchanging messages. E.g. shared data is a technique to implement high bandwidth cooperation.

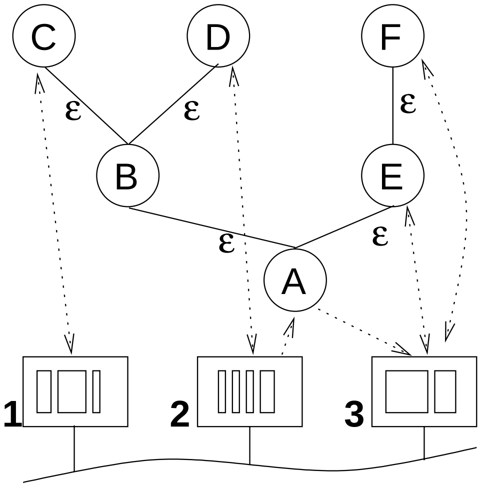

Figure 2.6: Regions implemented with node pools

As shown in figure 2.6, each node maintains an own dynamic pool of regions encapsulated in the *region allocator*. Each pool is provided at system startup by the boot master. The region allocator is tailored to the specific properties of regions such as page aligned, just a few different sizes, and double-ended stack alike handling. Each AC gets/returns regions directly from/to the region allocator where it is executing. To further avoid communication, regions may be allocated and returned on different nodes in case of migration (e.g. A allocated on node 2 and returns on 3). These may lead to a certain degree of additional external fragmentation. Idle cycles or region shortages trigger a *region pool reorganization* which is hierarchically coordinated by *cluster masters* and a designated *system master* node. This *lazy* or *optimistic* strategy is eligible, because it can be supposed that region shortages occur infrequently.

□ This subsection also demonstrated the importance of the ability to systematically map abstract concepts to generalized management methods. Unfortunately, it seems as if there was hardly any support for systematic top-down derivations of this kind in the context of operating systems. The reasons are mainly missing abstraction and categorization of existing successful techniques.

Segmented Stacks and Heaps

Each manager has to provide heap and stack space for its AC. Obviously, due to multiple ACs within one address space, heap and stack growth either has to be limited or classical management has to be rethought. We decided for the latter.

Definition: 4 (virtual memory segment)
A virtual memory segment is a complete interval of virtual addresses consisting of at least one virtual memory region.

Definition: 5 (segment stack)
A segment stack contains individual segments which are dynamically pushed and popped. Additionally, the top most segment may dynamically grow and shrink.

Notice, virtual addresses within a segment stack are in general neither monotonous nor linear.
With its regions each manager autonomously maintains two segment stacks (see def. 4,5) to implement stack and heap of its AC. Every segment has a header specifying its size and a link. For performance reasons, segments of a segment stack are chained in a circular list through the link field. The header itself is placed at the highest address in case of stack, respectively the lowest address in case of heap to enable linear segment extensions for downward growing stacks and upward growing heaps.

In case of an overflow of the top segment, it is first tried to extend the top segment by requesting a connecting region from the region allocator. If the region returned complies to this preference it is simply added to the top segment as a *linear extension*. Otherwise, a *non-linear extension* is performed by pushing the region received as the new top segment onto the corresponding segment stack. An underflow occurs, if the stack pointer or the heap limit drop below the start address of the stack respectively heap top segment. Analogously to extensions, *reductions* triggered by underflows can as well be linear (shrinking the top segment) or non-linear (top segment is popped). In either case, regions formerly contained in segments are returned to the node region pool.

Figure 3.1 illustrates stack and heap space based on segment stacks. Each thread, implementing the flow of control of an AC, is guided by a *thread control block* (TCB) representing the dedicated data portion of the manager. Fields within the TCB provide access to the bottom elements of both segment stacks. Unlike all other segments, the link field of bottom elements references the top segment. Management objects usually kept in a static data part, e.g. global heap library variables, are placed in the information part of the bottom heap or stack segment. The figure also shows an overall non-monotonous stack space for this AC. The current (top) stack segment starts and ends above its preceding segment.

Notice, that all kind of memory in this system is `mmap`'ed. Abandoning `sbrk` and kernel stack handling has several consequences which are elaborated in the following paragraphs. It is also evident, that fast access to the TCB is crucial. For this purpose, we modified GNU `gcc` to amongst others use a fix hardware register to reference the TCB of the current AC [Piz97]. For example, on Sparc V9 `%g3` is used as the TCB designator.

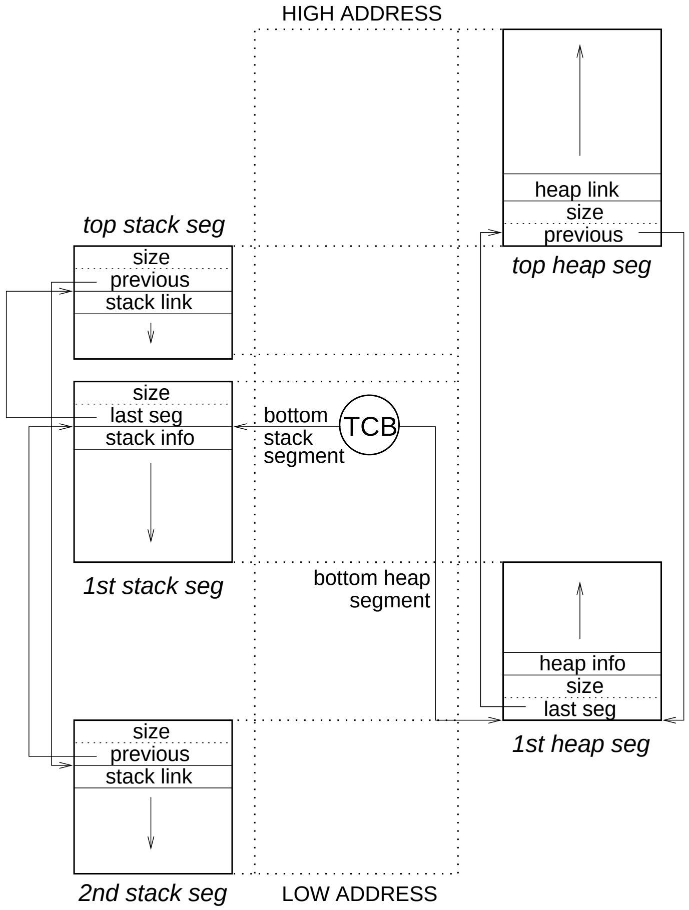

Figure 3.1: Per thread segment stacks for stack and heap space

3.1 Unlimited Stacks

Segment stacks allow to lazily adapt memory consumption without a rigid limit. Each thread is started with a single stack segment whose size is determined at compile time. At runtime, segment crossings are monitored and the usually linear stack space becomes eventually split to fit on separate segments.

Knowing the code generator, only three possibilities of segment crossings must be considered. First, when *a call level is entered* the stack pointer (SP) is decremented[1] to allocate the new activation frame. Second, *dynamic stack objects*, such as fields with statically unknown range, are allocated by decrementing SP. While these two operations may cause overflows, *leaving a call level* is the source for underflows.

Stack objects are bundled within activation frames for faster (de-)allocation. A sound possibility to split the stack is between activation frames. Dynamic stack objects could as well be separated with the effect of an awkward heap alike management within stack, causing strong internal fragmentation. As placing dynamic stack objects on stack is not essential, we decided to transparently place such objects in heap space. This, in turn has the advantageous effect that at most each call level entry and exit must be monitored.

[1] Assuming downward growing stacks.

3.1.1 Decoupled — Compiler Modification

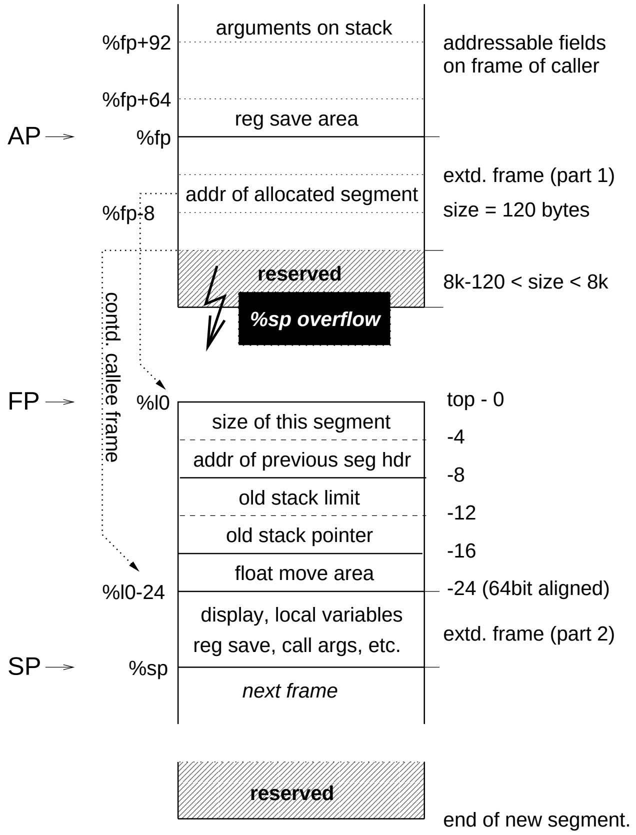

Figure 3.2: Non-linear stack extension

A hardware integrated compare logic checking SP against segment limits would be desirable but is not available. Hence, monitoring must be prepared by the compiler by generating inlined code. This code could be placed around calls or integrated into prologue and epilogue of subprograms. Latter was chosen because it reduces code size and most of all, is eligible to support extensible systems where a caller might have no knowledge about the callee.

Stack addressing had to be changed. Usually, a single frame pointer (FP) points in between two frames. Negative offsets reference local objects, while arguments are found via positive offsets. Now, frames are eventually separated as shown in figure 3.2. The size of the possible gap between arguments and locals is statically unknown. Besides the FP addressing locals, this also requires an explicit argument pointer (AP). On Sparc V9, we utilize register `%l0` as FP and changed the semantics of `%fp` to AP instead of solely using a new register for the AP. This approach provides compatibility (debugger, libraries, etc.) and better performance.

The activation frame layout was extended with a flag determining whether the frame has caused a non-linear extension. While overflows are checked against the current stack limit recorded in the TCB, underflows are detected with help of this extension flag. Due to alignment more than one bit must be allocated. This property is exploited for faster segment deallocation by storing the *address of the allocated segment* instead of just a boolean value with the extension flag.

All of these modifications were made to the low-level back-end of the GNU `gcc` compiler. Among the benefits are support for many languages (C, C++, INSEL, etc.) at once and compatibility with all compiler optimizations such as function inlining or leaf functions.

3.1.2 Coupled — Runtime Management

In addition to the linkage of segment stacks, stack segments are also doubly linked through the AP (backward) and extension flag (forward) of frames causing non-linear extensions (see figure 3.2). This eliminates searching within lists in order to correct underflows and speeds-up stack evolution across segment boundaries. Two more values must be remembered and reset in case of underflows: the SP and the stack limit at the time of overflow. Instead of wasting two words in every frame, inlined code writes these values directly underneath the header of stack segments.

```
    save %sp,-384,%sp
 1) clr [%fp-8]
 2) mov %fp,%l0
 3) ld [%g3+12],%l1
 4) cmp %sp,%l1
 5) bgeu .prolog_end
    nop
 6) clr [%g3+12]
 7) mov 384,%o0
 8) mov %sp,%l2
 9) call OVERFLOW
10) add %fp,-120,%sp
11) cmp %o0,%g0
12) bne .non_linear
    nop
13) b .prolog_end
14) mov %l2,%sp
.non_linear:
15) st  %l1,[%o0-12]
16) st  %sp,[%o0-16]
17) st  %o0,[%fp-8]
18) add %o0,-8,%l0
19) add %l0,-384,%sp
```

```
1) ld [%fp-8],%o0
2) cmp %g0,%o0
3) be .epilog_end
   nop
4) ld  [%o0-12],%l1
5) clr [%g3+12]
6) call UNDERFLOW
7) ld  [%o0-16],%sp
8) st  %l1,[%g3+12]
```

Figure 3.3: Sparc stack check prologue and epilogue

Correcting an overflow requires calls of subprograms consuming further stack space. This is accomplished by maintaining a *reserved area* at the end of the current stack segment. The technique implemented ensures, that at least the size of the reserved portion (currently 8k) minus the minimal frame (currently 120 bytes) is available for the overflow handler. It can easily be proofed, that overflows are always handled within this space. In case of non-linear extensions, the reserved area is temporarily lost. Linear extensions simply move the reserved area to the new end of the segment without losses.

□ Figure 3.3 lists the stack checking code used on Sparc V9 for the interested reader. In this example, the frame size is 384 bytes. Line (1) of the prologue clears the extension flag, FP is assigned the value of AP (2), and the effectual limit is fetched from the TCB (3). If the SP is below the limit, nothing is left to do (4,5). Otherwise, the stack limit is cleared (≡ maximum) to avoid recursion (6) and the overflow handler is called (9) after shrinking to the minimal frame (10). The handler returns zero in case of linear extensions which is checked in (11). If linear, then only the SP is reset to the value before the handler was called (14,8). If non-linear, the stack limit and SP are written to the new segment (15,16) and the segment address is written to the extension flag (18), before the frame space is moved to the new segment by setting FP and SP (18,19). Lines 1–3 of the epilogue check whether the current frame caused a non-linear extension by comparing the extension flag with zero. If yes, then the current limit is set ineffective (5), and SP is reset (7), before the underflow handler is called (6), and the stack limit becomes reset (8).

3.1.3 Distributed Display Handling

In turn of modifying stack addressing within the compiler, we also modified display [ASU86] handling to better support nested functions. The usually used static chain technique is unacceptable in a distributed environment, because tracing each link of the chain could cause network communication. Displays, on the other hand, are often implemented by copying data from the static predecessor. As this may still cause network communication although a local function is called, it is also unacceptable. The new technique integrated into the compiler copies the display either form the dynamic predecessor on the same node or prefetches it, if a potential remote function call is to be performed.

3.1.4 Performance Considerations

The computational costs for dynamic stack checking are comparably small. In the average case of no extension, $5 + 3$ additional instructions incur. The effect on real programs is debatable. Tests with a simple parallel prime generator indicate an insignificant overhead (40.3 versus 40.5 seconds). Widening the scope of checks could further reduce this overhead. E.g. checks are actually only needed at points of recursion. Other checks can be combined according to the statically predictable deepest call level.

Internal fragmentation only occurs in case of non-linear extensions. Let f be the average frame size, r the size of the reserved area, and s the average segment size. Following formula is an approximation of the internal stack fragmentation, if every extension was non-linear:

$$F_{avg} = \frac{r + ((s - r)\ mod\ f)}{s}; 8k - 120 < r < 8k$$

If $f = 256, r = 8192$, and $s = 32k$ internal fragmentation would be 25%. Non-linear extensions are problematical in two ways. First, they may cause noticeable fragmentation, which can be optimized by choosing adequate segment sizes. Second, in contrast to linear extensions, non-linearly extended segments become freed as soon as the call-level causing the extension is left and might already be reallocated with the next call leading to unfavorable *thrashing*. This situation is avoided by exploiting the region allocator to provide regions at preferred addresses.

3.2 Heaps

Throughout this paper, the term “heap” refers to a pool of memory available for allocation and deallocation in arbitrary order. To eliminate synchronization and communication as far as possible, each AC (de-)allocates objects on its own dedicated heap.

We investigated existing libraries concerning their eligibility to serve as a starting point for the implementation of the heap segment stack. Because of its excellent performance [DDZ94] and its both, short and understandable source code, D. Lea's

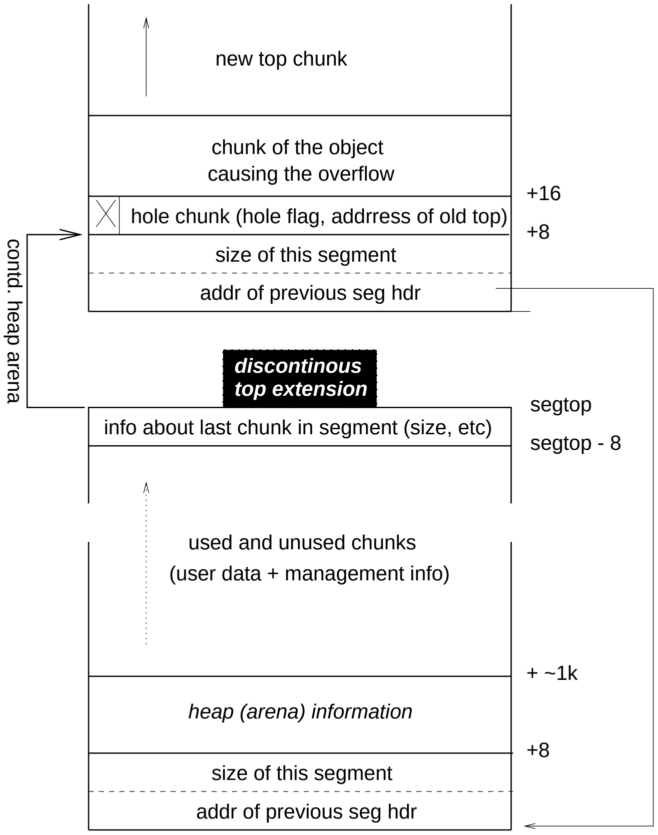
new top chunk
chunk of the object causing the overflow
+16
hole chunk (hole flag, addrress of old top)
+8
contd. heap arena
size of this segment
addr of previous seg hdr
discontinous top extension
segtop
info about last chunk in segment (size, etc)
segtop - 8
used and unused chunks (user data + management info)
+ ~1k
heap (arena) information
+8
size of this segment
addr of previous seg hdr

Figure 3.4: Heap extension

freely available memory allocator `G++ malloc` [Lea96] was selected. It structures heap space into free and allocated *chunks*. A special free chunk, called *top chunk* (TC), is used to grow and shrink the heap. It is split and coalesced as chunks are (de-)allocated at the top end of the heap while being increased and decreased at the upper end with the system call `sbrk`.

In contrast to stacks, the separate management of each application-level object in a chunk allows to easily spread a heap across segments, because splitting can be performed between arbitrary chunks. Obviously, linear extensions and reductions simply increase and decrease TC's upper limit, identically to `sbrk` without requiring changes to the library.

Several modifications were made to support positive or negative *holes* caused by non-linear extensions (see figure 3.4). If TC is non-linearly extended, the effectual TC is converted into an ordinary free chunk, which can be used to satisfy subsequent allocations. Its chunk information (size, etc.) is placed at the highest address of the old top segment. Above the segment header of the new segment, a special *hole chunk* is installed and the allocation causing the overflow is performed. The remainder of the segment is used as the new TC. The hole chunk serves two purposes. First, it stores the information about the old TC. Second, it has a flag set, that prevents this chunk from being coalesced with other chunks than the TC. Heap trimming operations, succeeding deallocations with coalescences, decrease TC's upper limit if its size exceeds a certain limit. Each time TC is trimmed, it is also checked, whether TC could be coalesced with the hole chunk, which would mean that no chunks are allocated within this segment. If this is the case, a non-linear reduction is performed instead of just linearly reducing the segment size. Before returning regions to the node region pool, the old TC is re-established based on information stored in the hole chunk and at the end of the previous segment.

The computational overhead introduced with the segmented heap organization is neglectable. Similarly to stack space, fragmentation increases with the amount of non-linear extensions which can be controlled with the region allocator. In contrast to stack space, there is no reserved area in heap space being wasted. Furthermore, lazy reduction can be employed by deferring heap trimming which nearly eliminates the thrashing effect explained in 3.1.4.

Object-Distribution and Garbage Collection

Current work is focusing on the transparent incorporation of garbage collection (GC) and DSM capabilities into heap management. In a long term running distributed single address space system, GC and DSM have strong interactions. A joint approach will be superior to individually optimized solutions. For example, indirections needed for hardware-supported distribution of individual objects [GPR97] can at the same time be exploited by the collector to move objects. The approach taken, is to widen the scope of GC to include management objects as well as application level objects in a *collection hierarchy.* References to objects and replicates of remote objects are locally monitored. Locally unreachable replicates become deleted. *Proxy pages* only mapped to hold replicates and migrated objects become further unmapped by the local collector if they do not contain any reachable representants of remote objects. “Original objects” are deleted if neither replicates nor local references exist.

A first prototype of the MoDiS DSM, providing distributed shared stack objects, is explained in detail in [GPR97]. The techniques developed, are currently adapted to provide efficient remote access to heap objects. The basic idea is to access objects mediately via indirection pointers in order to move shared objects between different memory regions. These regions represent per node *read-write*, *read-only* and *no* rights, which are checked in hardware because regions are page aligned. Accesses with insufficient rights trigger faults. Software handlers retrieve the requested object, enforce per object consistency with a dynamically chosen coherence protocol, and adjust the indirection. Pointer swizzling at fault time between different memory regions delivers the ability to exploit the page fault mechanism of “off the shelf hardware”. Thus, the DSM management only has to handle accesses to locally unavailable objects. Performance penalties as known from all-in software implementations are avoided while individual objects are still efficiently handled without false-sharing.

Conclusion

The reader might have noticed, that although this approach is introduced as being top-down oriented, concepts are explained rather in the opposite direction starting from coarse partitions and regions. In fact, concepts were elaborated top-down with the bottom in mind[1] Pure top-down construction seems to be at least as unsatisfactory as bottom-up driven methods. Where the latter fails to match application-level requirements, the former tends to miss real world possibilities.

The memory management techniques presented, aim to support parallelism and distribution as an integral part of a new distributed OS architecture. The motivation is to free the application level from repetitive and error prone management tasks. Although the context of this work is a language-based approach, most of the concepts elaborated are also applicable in other parallel or distributed environments.

Besides distinguishing stack and heap, memory management is invisible at the application level. The programmer is not burdened with object locations, network messages, special sharable regions, or stack size requirements. Instead, the OS performs adaptive segmentation to fully exploit the address space for concurrent computations dynamically varying in size and number. Memory consumption approximates application-level requirements. Furthermore, any application level object is shared across nodes with automatic migration or replication as necessary. It is also stated clearly, that these features do not induce significant constant overhead. This is a prerequisite to not solely provide speed-ups with the consumption of additional resources but also the possibility of defacto advantages compared to conventional systems.

Implementation is based on a tight coupling of tools and kernel into an integrated OS. Instead of constructing layers, all instances involved in management are considered as possibilities to implement management functionality. To reduce the effort needed to construct these instances from scratch and at the same time avoid reinventions of the wheel, existing software is modified to meet changed requirements. In turn, compatibility is limited. Existing binaries can be integrated into the system but to fully profit from these new features, applications at least have to be recompiled. Another important step is the introduction of new languages as briefly presented in this paper, supporting e.g. high level specification of concurrency.

The platform used for the implementation of these concepts consists of 14 SUN Ultra 1 workstations running Solaris 2.5.1 interconnected with a 100Mbit/s Fast Ethernet. Implementation and evaluation of segmented stacks as well as modifications of the malloc library is finished.

Partitioning into shared and node partitions, region distribution and the region allocator are realized to a great extend. Besides the object-based DSM for heap space, current implementation work concentrates on dynamic region redistribution and visualization tools. Conceptual work is focusing on the interaction between DSM and distributed garbage collection.

[1] Using this line of thought in this paper would probably not lead to a better understanding for the techniques.

Bibliography

[ARD97] Patrik Reali Andreas R. Disteli. Combining Oberon with active objects. In *Proc. of Joint Modular Languages Conf. (JMLC). LNCS 1024*, Linz, Austria, March 1997. Springer Verlag.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools.* Addison-Wesley, 1986.

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the IEEE CompCon Conf.*, 1993.

[Car95] J. B. Carter. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and Distributed Computing*, 29(2):219–227, September 1995.

[Car98] John B. Carter. Distributed shared memory: Past, present, and future. slides of tutorial; 3rd Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments, March 1998.

[CLBHL93] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Ed Lazowska. Opal: A single address space system for 64-bit architectures. In *Proc. of the Fourth Workshop on Workstation Operating Systems*, pages 80–85, 1993.

[Cor91] H. Corporall. Distributed heapmanagement using reference weights. In Arndt Bode, editor, *Distributed Memory Computing*, number 487 in LNCS, pages 325–336. 2nd European Conf., EDMCC2, Springer-Verlag, April 1991.

[DDZ94] David Detlefs, Al Dosser, and Benjamin G. Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 24(6):527–542, June 1994.

[ea95] Paul R. Wilson et al. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proc. of Int'l Workshop on Memory Management*, volume 986 of *LNCS*, Kinross, Scotland, September 1995. Springer-Verlag.

[ea96a] A. D. Skousen et al. The Sombrero operating system for a distributed single very large address space. Technical Report TR-96-005, Arizona State University, April 1996.

[ea96b] Cristiana Amza et al. TreadMarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, February 1996.

[Elp93] Kevin Elphinstone. Address space management issues in the Mungi operating system. Technical Report SCS&E Report 9312, University of New South Wales, Australia, November 1993.

[EW95a] C. Eckert and H.-M. Windisch. A new approach to match operating systems to application needs. In *Proc. of the 7th Int'l Conf. on Parallel and Distributed Computing and Systems (ISMM)*, Washington, DC, October 1995.

[EW95b] C. Eckert and H.-M. Windisch. A top-down driven, object-based approach to application-specific operating system design. In *Proc. of the Int'l Workshop on Object-orientation in Operating Systems (IWOOOS)*, pages 153–156, Lund, Sweden, August 1995.

[Flo89] Michael A. Floyd. Turbo Pascal with objects. *Dr. Dobb's Journal of Software Tools*, 14(7):56–63, 95–97, July 1989.

[Glo97] Wolfram Gloger. ptmalloc - a multi-threaded malloc implementation. FTP, April 1997. ftp://ftp.dent.med.uni-muenchen.de/pub/wmglo/ptmalloc.tar.gz.

[GP97] Sascha Groh and Markus Pizka. A different approach to resource management for distributed systems. In *Proc. of Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, July 1997.

[GPR97] S. Groh, M. Pizka, and J. Rudolph. Shadow stacks — a hardware-supported DSM for objects of any granularity. In *Proc. of the 3rd Int'l Conf. on Algorithms and Architectures for Parallel Processing (ICA3PP)*, December 1997.

[Gro96] Sascha Groh. Designing an efficient resource management for parallel distributed systems by the use of a graph replacement system. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 215–225, August 1996.

[Hae] Mike Haertel. Mmalloc. WWW. http://www.sdsu.edu/doc/texi/mmalloc_toc.html.

[HL93] Guido Hogen and Rita Loogen. A new stack technique for the management of runtime structures in distributed environments. Technical Report 93-03, RWTH Aachen, 1993.

[HMT89] Robert Harper, Robin Milner, and Mads Tofte. The Definition of Standard ML: Version 3. Technical Report ECS-LFCS-89-81, Laboratory for the Foundations of Computer Science, University of Edinburgh, May 1989.

[IEE95] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995.

[JKW95] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: high-performance all-software distributed shared memory. In *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, volume 29/5, 1995.

[Lea96] Doug Lea. A memory allocator. WWW, December 1996. http://g.oswego.edu/dl/html/malloc.html.

[Li86] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Department of Computer Science, Yale University, New Haven, CT, October 1986.

[ML95] D. C. J. Matthews and T. Le Sergent. LEMMA: A distributed shared memory with global and local garbage collection. In *Proc. of the Int'l Workshop on Memory Management (IWMM)*, pages 297–311, September 1995.

[PE97] M. Pizka and C. Eckert. A language-based approach to construct structured and efficient object-based distributed systems. In *Proc. of the 30th Hawaii Int. Conf. on System Sciences*, volume 1, pages 130–139, Maui, Hawai, January 1997. IEEE CS Press.

[Piz97] Markus Pizka. Design and implementation of the GNU INSEL-compiler gic. Technical Report TUM-I9713, Technische Universität München, Dept. of CS, 1997.

[PS95] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In Henry Baker, editor, *Proc. of Int'l Workshop on Memory Management*, volume 986 of *LNCS*, ILOG, Gentilly, France, and INRIA, Le Chesnay, France, September 1995. Springer-Verlag.

[PT98] H. Pagnia and O. Theel. Sacrificing true distribution for gaining access efficiency of replicated shared objects. In *Proc. of the 31st Hawaii Int'l Conf. on System Sciences (HICSS)*, volume VII, January 1998.

[Rad95] Ralph Radermacher. *EVA: A Runtime Environment with Integrated Load Balancing for Distributed and Parallel Systems.* PhD thesis, TU München, 1995. german only.

[SSC98] M. Swanson, L. Stroller, and J. B. Carter. Making distributed shared memory simple, yet efficient. In *Proc. of the 3rd Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 2–13, March 1998.

[Sta95] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, November 1995.

[Sun95] SunSoft, Mountain View, CA. *Solaris Multithreaded Programming Guide*, 1995.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, New Jersey, 1992.

[TF95] O. E. Theel and B. D. Fleisch. Design and analysis of highly available and scalable coherence protocols for distributed shared memory systems using stochastic modeling. In *Int'l Conf. on Parallel Processing, Vol.1: Architecture*, pages 126–130, Boca Raton, USA, August 1995. CRC Press.

[vDHT95] L. van Doorn, P. Homburg, and A. S. Tanenbaum. Paramecium: An extensible object-based kernel. In *Proc. of the 5th Workshop on Hot Topics on Operating Systems (HotOS)*, Orcas Island, WA, May 1995.

[Wil94] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[Win96] H.-M. Windisch. The Distributed Programming Language INSEL - Concepts and Implementation. In *High-Level Programming Models and Supportive Environments HIPS'96*, 1996.

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen :

Reihe A

Liste aller erschienenen Berichte von 1990-1994
auf besondere Anforderung

342/01/95 A Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids

342/02/95 A Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law

342/03/95 A Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space

342/04/95 A Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes

342/05/95 A Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems

342/06/95 A Maximilian Fuchs: Formal Design of a Model-N Counter

342/07/95 A Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology

342/08/95 A Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies

342/09/95 A Ketil Stølen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness

342/10/95 A Ketil Stølen, Max Fuchs: A Formal Method for Hardware/Software Co-Design

342/11/95 A Thomas Schnekenburger: The ALDY Load Distribution System

342/12/95 A Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm

342/13/95 A Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming

342/14/95 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Point-to-Point Dataflow Networks

342/15/95 A Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs

342/16/95 A Bernhard Schätz, Katharina Spies: Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik

342/17/95 A Georg Stellner: Using CoCheck on a Network of Workstations

342/18/95 A Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller: Workshop on PVM, MPI, Tools and Applications

342/19/95 A Thomas Schnekenburger: Integration of Load Distribution into ParMod-C

342/20/95 A Ketil Stølen: Refinement Principles Supporting the Transition from Asynchronous to Synchronous Communication

342/21/95 A Andreas Listl, Giannis Bozas: Performance Gains Using Subpages for Cache Coherency Control

342/22/95 A Volker Heun, Ernst W. Mayr: Embedding Graphs with Bounded Treewidth into Optimal Hypercubes

Reihe A

342/23/95 A Petr Jančar, Javier Esparza: Deciding Finiteness of Petri Nets up to Bisimulation

342/24/95 A M. Jung, U. Rüde: Implicit Extrapolation Methods for Variable Coefficient Problems

342/01/96 A Michael Griebel, Tilman Neunhoeffer, Hans Regler: Algebraic Multigrid Methods for the Solution of the Navier-Stokes Equations in Complicated Geometries

342/02/96 A Thomas Grauschopf, Michael Griebel, Hans Regler: Additive Multilevel-Preconditioners based on Bilinear Interpolation, Matrix Dependent Geometric Coarsening and Algebraic-Multigrid Coarsening for Second Order Elliptic PDEs

342/03/96 A Volker Heun, Ernst W. Mayr: Optimal Dynamic Edge-Disjoint Embeddings of Complete Binary Trees into Hypercubes

342/04/96 A Thomas Huckle: Efficient Computation of Sparse Approximate Inverses

342/05/96 A Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, Arndt Bode: OMIS — On-line Monitoring Interface Specification

342/06/96 A Ekkart Kindler: A Compositional Partial Order Semantics for Petri Net Components

342/07/96 A Richard Mayr: Some Results on Basic Parallel Processes

342/08/96 A Ralph Radermacher, Frank Weimer: INSEL Syntax-Bericht

342/09/96 A P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, H.-M. Windisch: Sprachkonzepte zur Konstruktion verteilter Systeme

342/10/96 A Stefan Lamberts, Thomas Ludwig, Christian Röder, Arndt Bode: PFSLib – A File System for Parallel Programming Environments

342/11/96 A Manfred Broy, Gheorghe Ştefănescu: The Algebra of Stream Processing Functions

342/12/96 A Javier Esparza: Reachability in Live and Safe Free-Choice Petri Nets is NP-complete

342/13/96 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Many-to-Many Data-flow Networks

342/14/96 A Giannis Bozas, Michael Jaedicke, Andreas Listl, Bernhard Mitschang, Angelika Reiser, Stephan Zimmermann: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project

342/15/96 A Richard Mayr: A Tableau System for Model Checking Petri Nets with a Fragment of the Linear Time μ -Calculus

342/16/96 A Ursula Hinkel, Katharina Spies: Anleitung zur Spezifikation von mobilen, dynamischen Focus-Netzen

342/17/96 A Richard Mayr: Model Checking PA-Processes

342/18/96 A Michaela Huhn, Peter Niebert, Frank Wallner: Put your Model Checker on Diet: Verification on Local States

342/01/97 A Tobias Müller, Stefan Lamberts, Ursula Maier, Georg Stellner: Evaluierung der Leistungsf”ahigkeit eines ATM-Netzes mit parallelen Programmierbibliotheken

342/02/97 A Hans-Joachim Bungartz and Thomas Dornseifer: Sparse Grids: Recent Developments for Elliptic Partial Differential Equations

342/03/97 A Bernhard Mitschang: Technologie f”ur Parallele Datenbanken - Bericht zum Workshop

342/04/97 A nicht erschienen

342/05/97 A Hans-Joachim Bungartz, Ralf Ebner, Stefan Schulte: Hierarchische Basen zur effizienten Kopplung substrukturierter Probleme der Strukturmechanik

Reihe A

342/06/97 A Hans-Joachim Bungartz, Anton Frank, Florian Meier, Tilman Neunhoeffer, Stefan Schulte: Fluid Structure Interaction: 3D Numerical Simulation and Visualization of a Micropump

342/07/97 A Javier Esparza, Stephan Melzer: Model Checking LTL using Constraint Programming

342/08/97 A Niels Reimer: Untersuchung von Strategien für verteiltes Last- und Ressourcenmanagement

342/09/97 A Markus Pizka: Design and Implementation of the GNU INSEL-Compiler gic

342/10/97 A Manfred Broy, Franz Regensburger, Bernhard Schätz, Katharina Spies: The Steamboiler Specification - A Case Study in Focus

342/11/97 A Christine Röckl: How to Make Substitution Preserve Strong Bisimilarity

342/12/97 A Christian B. Czech: Architektur und Konzept des Dycos-Kerns

342/13/97 A Jan Philipps, Alexander Schmidt: Traffic Flow by Data Flow

342/14/97 A Norbert Fröhlich, Rolf Schlagenhaft, Josef Fleischmann: Partitioning VLSI-Circuits for Parallel Simulation on Transistor Level

342/15/97 A Frank Weimer: DaViT: Ein System zur interaktiven Ausführung und zur Visualisierung von INSEL-Programmen

342/16/97 A Niels Reimer, Jürgen Rudolph, Katharina Spies: Von FOCUS nach INSEL - Eine Aufzugssteuerung

342/17/97 A Radu Grosu, Ketil Stølen, Manfred Broy: A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing

342/18/97 A Christian Röder, Georg Stellner: Design of Load Management for Parallel Applications in Networks of Heterogenous Workstations

342/19/97 A Frank Wallner: Model Checking LTL Using Net Unfoldings

342/20/97 A Andreas Wolf, Andreas Kmoch: Einsatz eines automatischen Theorembeweisers in einer taktikgesteuerten Beweisumgebung zur Lösung eines Beispiels aus der Hardware-Verifikation – Fallstudie –

342/21/97 A Andreas Wolf, Marc Fuchs: Cooperative Parallel Automated Theorem Proving

342/22/97 A T. Ludwig, R. Wismüller, V. Sunderam, A. Bode: OMIS - On-line Monitoring Interface Specification (Version 2.0)

342/23/97 A Stephan Merkel: Verification of Fault Tolerant Algorithms Using PEP

342/24/97 A Manfred Broy, Max Breitling, Bernhard Schätz, Katharina Spies: Summary of Case Studies in Focus - Part II

342/25/97 A Michael Jaedicke, Bernhard Mitschang: A Framework for Parallel Processing of Aggregat and Scalar Functions in Object-Relational DBMS

342/26/97 A Marc Fuchs: Similarity-Based Lemma Generation with Lemma-Delaying Tableau Enumeration

342/27/97 A Max Breitling: Formalizing and Verifying TimeWarp with FOCUS

342/28/97 A Peter Jakobi, Andreas Wolf: DBFW: A Simple DataBase FrameWork for the Evaluation and Maintenance of Automated Theorem Prover Data (incl. Documentation)

342/29/97 A Radu Grosu, Ketil Stølen: Compositional Specification of Mobile Systems

342/01/98 A A. Bode, A. Ganz, C. Gold, S. Petri, N. Reimer, B. Schiemann, T. Schnekenburger (Herausgeber): ”‘Anwendungsbezogene Lastverteilung”’, ALV’98

342/02/98 A Ursula Hinkel: Home Shopping - Die Spezifikation einer Kommunikationsanwendung in Focus

342/03/98 A Katharina Spies: Eine Methode zur formalen Modellierung von Betriebssystemkonzepten

Reihe A

342/04/98 A Stefan Bischof, Ernst-W. Mayr: On-Line Scheduling of Parallel Jobs with Runtime Restrictions

342/05/98 A St. Bischof, R. Ebner, Th. Erlebach: Load Balancing for Problems with Good Bisectors and Applications in Finite Element Simulations: Worst-case Analysis and Practical Results

342/06/98 A Giannis Bozas, Susanne Kober: Logging and Crash Recovery in Shared-Disk Database Systems

342/07/98 A Markus Pizka: Distributed Virtual Address Space Management in the MoDiS-OS

SFB 342 : Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B	Wolfgang Reisig: Petri Nets and Algebraic Specifications
342/2/90 B	Jörg Desel: On Abstraction of Nets
342/3/90 B	Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems
342/4/90 B	Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme
342/1/91 B	Barbara Paech1: Concurrency as a Modality
342/2/91 B	Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier- Toolbox - Anwenderbeschreibung
342/3/91 B	Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen
342/4/91 B	Werner Pohlmann: A Limitation of Distributed Simulation Methods
342/5/91 B	Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis
342/6/91 B	Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme
342/7/91 B	W. Reisig: Concurrent Temporal Logic
342/1/92 B	Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support
342/2/92 B	Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen
342/1/93 B	Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung
342/2/93 B	Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick
342/1/94 B	Andreas Listl; Thomas Schnekenburger; Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS