Thread Segment Stacks

Markus Pizka
Technische Universität München, Informatik XIII
Germany, 80290 Munich

Abstract *This paper presents enhanced memory management concepts and their implementation providing better support for multi threading. The virtual address space of the multi threaded process is dynamically partitioned by a dynamic set of cooperating managers. Special thoughts are given to detect and solve possible thread stack and heap overflows and collisions. Both stacks and heaps associated with threads are organized non-contiguously with linear segments to fully exploit possibly large virtual address spaces. Crucial for the efficiency of this approach are modifications of the compiler and parts of the runtime system. The proposed solutions are implemented an evaluated on the SUN Sparc V9 architecture.*

Keywords: operating systems, multi threading, memory management

1 New Features and Flaws

Multi-tasking operating systems (OS) usually provide private 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 a single virtual address space (VA) for all processes this and other problems can be evaded. Each memory object is uniformly identified with a unique memory address instead of separately maintained identifiers. Usually, the VA of a process is partitioned as sketched on the left hand side in figure 1. Besides code segments, one stack and one heap grow and shrink in opposite directions. A collision of stack and heap implies that a servere 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. Hence, this situation is usually accepted although it decreases reliability.

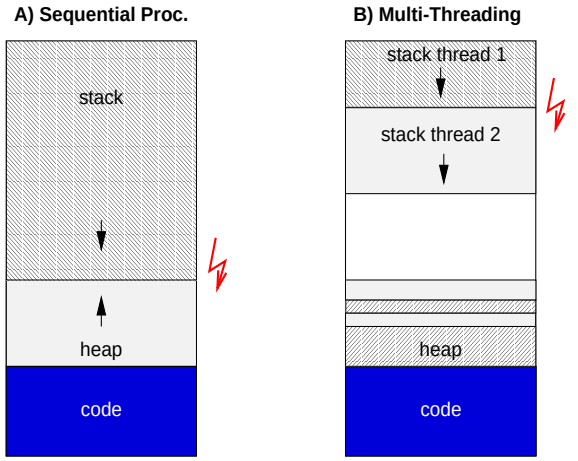

Figure 1: VA partitioning

In multi-threaded systems [1], each thread needs its dedicated stack somewhere within the shared VA. Unfortunately, multi-threading is hardly supported by the memory management system. As shown on the right hand side of figure 1 thread stacks eventually collide, although the VA is far from being exhausted. Malfunc-

[0]This project is sponsored by the DFG (German Research Council) as part of the SFB #342 – *Tools and methods for the utilization of parallel architectures.*

tions of this kind might affect many independent computations making such approaches insufficient reliable. As a matter of fact, this problem stays unsolved in all implementations known to 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 appropriate. This shifts the problem to the programmer contradicting the goal of simplicity and even worse, is no solution to the problem. In general, neither stack size demands nor the number of threads can statically be predicted.

2 Related Work

Hardware supported paged *segments* as used in former OS like MULTICS on Honeywell 6000 machines [2] would solve the problem. Thread stacks, heaps and extensible code fragments could be placed in separate segments but after years of predominant sequential processes these features are missing.

Concurrent Oberon [3] 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. Though consumption of physical memory is adaptive, unweakened *linearity* of stack spaces disallows the exploitation of the whole VA for stacks larger then the predefined limit. Hence, 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 in Solaris [4], is fast and compatible. While overflows are handled sufficiently, correction of collisions is nearly impossible. Collisions stay undetected till objects on the restricted page are touched, although other objects within the same frame and their addresses might already be used. At the time of detection, registers and objects would have to be examined globally along with pointer swizzling. Compiler-based approaches like dynamic stack probing in `gcc` [5], also suffer from *late detection*.

In [6] problems of maintaining multiple stacks are described. The proposed solution is to implement the conceptual *cactus stack* as a per processor *meshed stack*. This technique requires expensive garbage collection of activation records within the meshed stack and obstacles hardware protection. The technique presented in this paper provides similar space but superior time efficiency.

3 Thread Context Managers

To enforce transparent, scalable and adaptable resource management in parallel and distributed environments, we developed a reflective management architecture [7, 8, 9, 10]. The key idea is to associate a dedicated *manager* with each flow of control. One thread and all its termination dependent [11] passive objects are clustered to *thread-contexts* (TC). Each TC is guided by exactly one manager, which has to satisfy all demands for resources of its TC. Besides standard tasks such as allocating memory for the stack, heap and code, a manager might also have to enforce access restrictions. Conflicts, such as overflows or concurrent heap allocations are conceptually solved by inter manager cooperation. Crucial for the efficiency of this approach is a systematic realization of the conceptual managers. Any software instance involved in resource management is regarded as implementing parts of managers. At runtime, each thread is guided by a small data structure — *thread control block* (TCB) — representing the anchor of the manager implementation. Fields within the TCB provide access to thread specific runtime data structures, such as pointers to heap and stack space of the thread.

4 VA Management

For the dynamic distribution of virtual addresses to TCs, the VA is structured into disjoint memory *regions*. A virtual memory region is a complete interval of addresses starting and ending on page boundaries. According to

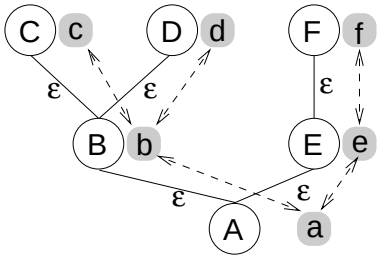

Figure 2: Spreading the VA across threads

the management model introduced in section 3 the task of VA distribution as distributed among the managers (see fig. 2). At first, the complete range of addresses is assigned to manager a of the root TC A. Subsequently created TCs are provided with regions for autonomous use by their creators. If the initial provision proofs to be insufficient additional regions are dynamically requested either from the creator or reclaimed from children. At the time of termination, each TC returns its regions back to its creator.
The implementation of this concept is based on maintaining all currently unallocated regions in a *region pool* encapsulated in the *region allocator*.

4.1 Segments

A *virtual memory segment* is a complete interval of virtual addresses consisting of at least one virtual memory region. A *segment stack* contains individual segments which are dynamically pushed and popped. Additionally, the top most segment may grow and shrink. Notice, virtual addresses within a segment stack need neither be monotonous nor linear.

With its regions, each manager autonomously maintains two segment stacks to provide stack and heap memory to its TC. Every segment has a header specifying its size and a link. The header 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

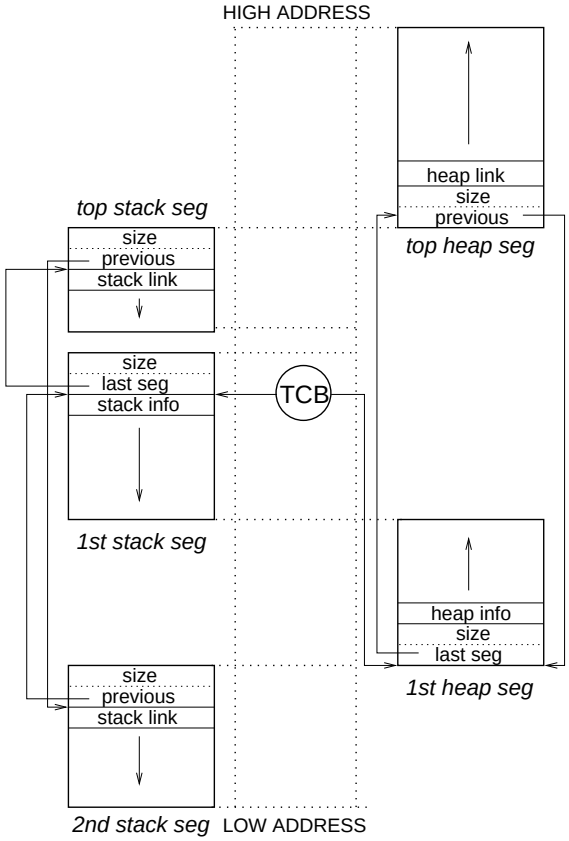

Figure 3: Per thread segment stacks

segment, it is first tried to *linearly extend* the top segment by requesting a connecting region from the region allocator. Otherwise, a *non-linear extension* is performed by pushing the region received as the new top segment onto the segment stack. An underflow occurs, if the stack pointer or the heap limit drop below (above for heap) the start of the top segment. *Reductions* triggered by underflows can as well be linear (shrinking top) or non-linear (top is popped).

Figure 3 illustrates stack and heap space based on segment stacks. The link field of bottom elements references for performance reasons the top segment. Management data usually kept in a static data part, e.g. heap library variables, are placed in the information part of the bottom segments. The figure also depicts a non-monotonous stack space for this TC. The

top stack segment starts and ends above its preceding segment.

All kind of memory in this system is `mmap`'ed. Fast access to the TCB is crucial. We modified GNU `gcc` to use a hardware register to hold the current TCB [12].

4.2 Unlimited Thread 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.

Only three possibilities for segment crossings must be considered. First, when *a call level is entered*, the stack pointer (SP) is decremented (downward growing stacks) to allocate the activation frame. Second, *dynamic stack objects*, such as fields with statically unknown ranges, are allocated by decrementing SP. While these two operations may cause overflows, *leaving a call level* is the source for underflows. 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.

4.2.1 Compiler Modification

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 with inlining code around around call instructions or into the prologue and epilogue of subprograms. Latter was chosen because it reduces code size and is eligible of supporting extensibility where the caller might have no knowledge about the callee.

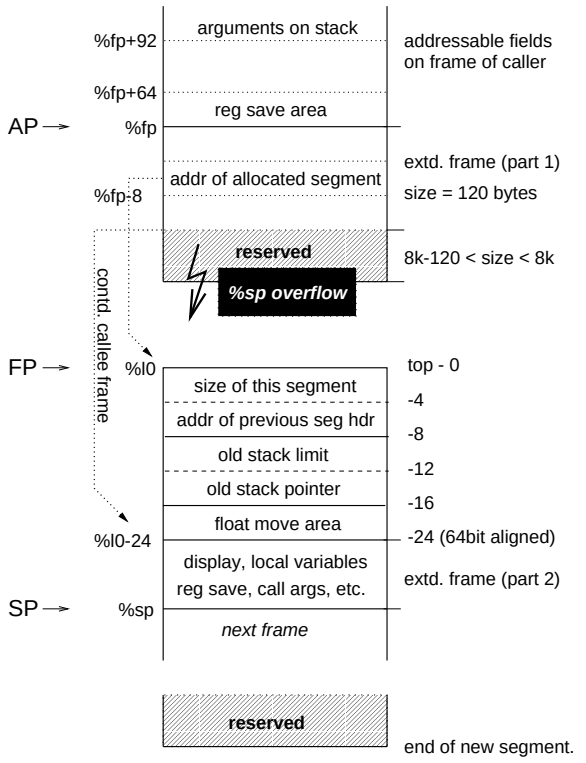

Figure 4: Non-linear stack extension

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, the size of the possible gap between arguments and locals is statically unknown. This 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 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 stack limit recorded in the TCB underflows are detected via the extension flag. Due to alignment reasons 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. 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.

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.

4.2.2 Performance Considerations

The computational costs for dynamic stack checking are comparably low. 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. I. e. checks are only needed at points of recursion.

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.

4.3 Segmented 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 [13] and its both, short and understandable source code, D. Lea's freely available memory allocator `G++ malloc` [14] 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`.

Several modifications were made to support positive or negative *holes* caused by non-linear extensions (see figure 5). 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

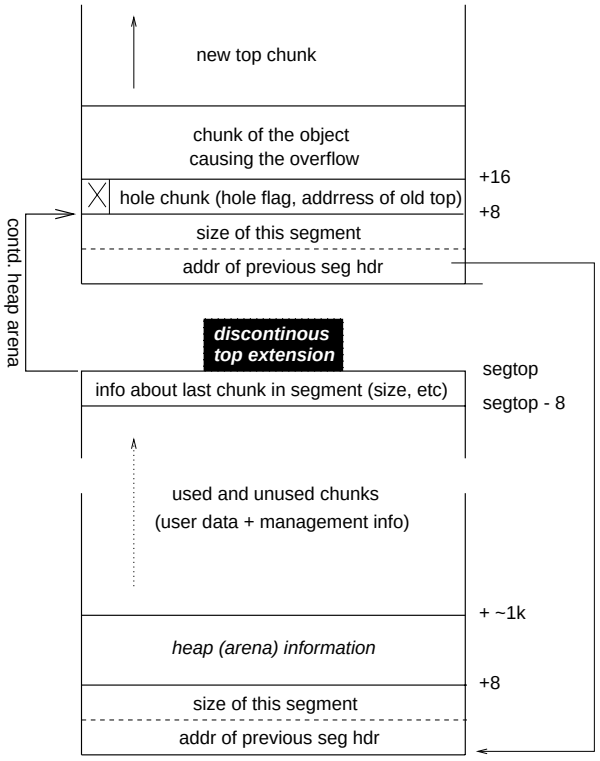

Figure 5: Heap extension

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 4.2.2.

5 Conclusion

The memory management techniques presented, support parallelism as an integral part of the OS architecture. The motivation is to free the application level from repetitive and error prone management tasks. The programmer is not burdened with 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 corresponds to application-level requirements and these features do not induce significant overhead. Instead of constructing layers and to preserve compatibility to a certain degree, existing tools are modified according to the changed requirements. Existing binaries can be integrated into the system but to fully profit from these new features, applications have to be recompiled. Implementation and evaluation of segmented stacks as well as modifications of the malloc library, are mostly finished with the exception of linear stack extensions.

References

[1] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language).* IEEE CS Press, 1995.

[2] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, New Jersey, 1992.

[3] P. Reali A. R. Disteli. Combining Oberon with active objects. In *Proc. of JMLC*, Linz, Austria, March 1997. Springer.

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

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

[6] G. Hogen and R. Loogen. A new stack technique for the management of runtime

structures in distributed environments. Technical Report 93-03, RWTH Aachen, 1993.

[7] C. Eckert and H.-M. Windisch. A top-down driven, object-based approach to application-specific operating system design. In *Proc. of IWOOOS*, Sweden, August 1995.

[8] C. Eckert and H.-M. Windisch. A new approach to match operating systems to application needs. In *Proc. of the ISMM*, Washington, DC, October 1995.

[9] S. Groh. Designing an efficient resource management for parallel distributed systems by the use of a graph replacement system. In *Proc. PDPTA*, pages 215–225, August 1996.

[10] S. Groh and M. Pizka. A different approach to resource management for distributed systems. In *Proc. of PDPTA*, July 1997.

[11] M. Pizka and C. Eckert. A language-based approach to construct structured and efficient object-based distributed systems. In *Proc. of HICSS-30*, volume 1, pages 130–139, Maui, Hawai, January 1997. IEEE CS Press.

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

[13] D. Detlefs, A. Dosser, and DB. G. Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 24(6):527–542, June 1994.

[14] D. Lea. A memory allocator, December 1996. http://g.oswego.edu/dl/html/malloc.html.