μ-kernel Memory Management

Sebastian Schönberg   Volkmar Uhlig

Dresden University of Technology
Faculty of Computer Science
D-01062 Dresden, Germany
email: {schoenbg,volkmar}@os.inf.tu-dresden.de

Abstract

In this paper we present a concept to manage structures required by μ-kernels to support a hierarchical user level memory management, based on FlexPages. FlexPages are variable sized memory objects. We describe a structure and algorithm which allow high concurrency, fast access and also time guaranties for real-time systems. It is also designed for use in sparsely occupied address spaces and has been tested on the 64 bit version of the L4-μ-kernel on Alpha processor. It performs the map operation in the worst-case time of $8.2\mu s$. The time for the flush operation depends only on the number of affected mappings and has a worst-case completion time of $3.5\mu s$.

1 Introduction

μ-kernels provides mainly inter process communication (IPC) and a basic memory management scheme. For security reasons, both issues need support by the kernel. IPC relies on trusted sender and receiver IDs, memory management on protection schemes to enforce access control to physical frames; for that, systems with virtual memory uses the translation from virtual into physical memory. In [HWL96, HC92] techniques are presented that enable to manage address spaces on user level. A mapping originally belongs to exactly one address space, but can be transferred to another space with less or same access rights. By use of this, address spaces can be created hierarchically. Figure 1 illustrates that.
Moreover, all mappings issued directly or indirectly by an address space can be revoked by the issuing address space.
These two services enable to build a memory management completely on user level. Every thread has its appropriate user level pager. On a page fault, the pge fault exception handler in the kernel initiates an IPC to the pager and waits for a reply that establishes a mapping in the caller's address space.
Secondly, μ-kernels are often used in real-time and embedded system environments, which strongly rely on predictability and worst-case times of kernel services. The map operation can be used to cheaply transfer huge amounts of data accross address spaces without copying the data.

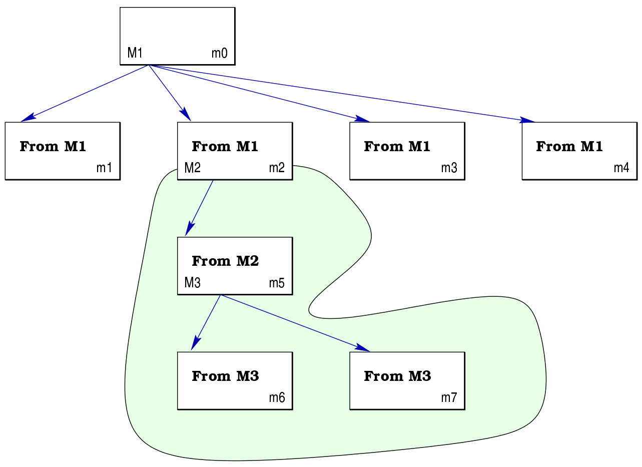

Figure 1: Hierarchical Memory Mappings

In opposite to current buffer management algorithms like [DP93] where the mappings are established in advance and the file is mapped in a round-robin scheme, due to the predictability a mapping can be established at a guaranteed time.
Especially multi-media data streams can now be mapped contiguously in the application's address space because the mapping will be established only for a certain time. This time is calculated upon program start by negotiation of Quality of Service parameters and depends on the data stream and the available buffer space.
To establish such a "timed-mapping" at a precise time, at least the map operation needs to be predictable. Hence, it is necessary to give worst-case times for the map operation. For a flush operation, the kernel needs information which mappings have been established by an address space into which address spaces. This data is stored in a structure called memory mapping database. Since more than one activity can operate over this structure, it must support concurrent operations. This can either be implemented by short, atomic instructions disabling interrupts or non-blocking fully preemptable synchronization as described in [Val95].
In the following, we present a structure for a 64-bit version of the L4 μ-kernel on Alpha [Sch96] that allows to use the technique of hierarchical address space management under

real-time constraints. Finally, we give some measurements to prove the ability of our technique.

2 Objects and Operations

L4 supports multiple address spaces. As described in [Lie95], address spaces can be generated hierarchically. Based on an initial space which is almost identical to the physical memory space, new address spaces can be generated. Figure 2 illustrates: an address space A_0 maps two regions, $R1$ completely and $R2$ partially into A_1. Furthermore, it maps $R2$ completely into A_2. A_1 maps $R1$ entirely into A_3 and partially into A_4. Additionally, A_4 gets $R2$ partially mapped by A_3.

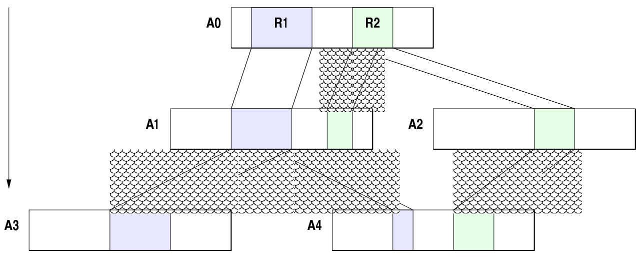

Figure 2: Recursive Construction of Address Spaces

To enforce security, a mapping must not be established without agreement of the receiver's space. Since spaces are no active entities, the map operation is bound to the kernel's IPC operation. By use of a flag, the kernel distinguishes to interpret the IPC arguments as FlexPage and not as message data. A FlexPage describes a memory region of 2^n contiguous pages.
The μ-kernel provides the following three operations:

map is the operation to send a FlexPage with the same or less privileges to a thread. This establishes a mapping to the source frames in the receiver's address space. The mapping in the source address space leaves unchanged. An already existing mapping for a page in the region described by the FlexPage is not overridden; if the destination refers to the same page, but with less access rights, the access rights are extended.

grant is similar to map, but the mapping is removed in the source's address space.

flush revokes either all mappings referring to frames described by the given FlexPage in caller's address space issued by threads of this address space or changes the access to read-only. This operation is an own system call and not bound to IPC. This is neccesary to perform a flush operation independently of an activity in the receiver's space.

When for example in Figure 1 m2 revokes its mappings by the flush operation, m5, m6 and m7 are also affected and flushed.

3 Memory Mapping Database

The mapping database was designed to meet the following requierements:

- the kernel reserves only a few pages of main memort for its management, hence the database must minimize its memory usage
- database operations run in the context of the calling thread; that means they use the kernel stack of that thread. Due to the restricted size of the stack, no recursive algorithms can be used
- database operations must either guarantee timing requirements or must be non-blocking with synchronization
- operations that return must guarentee, that the address spaces are in a well defined state and no pending operation is in progress
- the execution order caused by the hard priorities of the kernel must be strictly adhered.

3.1 Database operations

FlexPage operations modifying the address spaces have their counterparts for the mapping database. The operation to send a FlexPage inserts a new entry or modifies an existing entry in the database. The operation to flush a Flexpage removes entries from the database.

add entry inserts an entry into the database. The operation must be executed atomic to ensure database consistency as discussed in the following section.

change entry modifies an existing entry. This is the fastest operation, since the entry only needs to be modified and no memory allocation and database insertion is necessary.

flush entry removes all sub-mapping entries and if chosen, the top entry. This operation is not time-critical but must be interruptible. The freed entries are added to the free pool.

The map and grant operations are implemented as blocking and time-deterministic functions with disabled interrupts. When the total amount of affected pages for a flush operation is unknown, it can not be implemented as time-deterministic and a non-blocking design is chosen.
The maximum number of mappings in a system cannot be estimated on startup. Hence, a pre-allocation of memory for a fixed number of entries is not possible. The database is designed as an array in virtual memory. On touching an address in this array, the kernel maps a page from its kernel page pool to that address and restarts the faulting instruction. Freed entries in the database are linked in a list. Next time, an entry needs to be allocated, the entry is taken from that list.

3.2 Structure

Since a frame can be mapped in many other address spaces, the structure of the database is a tree representing the relation of the mapping to the issuing address spaces. To reduce memory space, the structure can be realized by linking all destination nodes together in form of a list. The parent entry only holds a reference to the first entry in the destination list.
Each mapped frame is represented by one entry in the mapping database. An entry contains the page's virtual address and the address space ID.

The overall pointer structure is designed to support a fast lookup of the nodes.
A running flush operation must either leave the database in a consistent state or lock inconsistent database areas. As mentioned above, despite a running flush operation, the map and grant operations must be successfully completed without interruption. Paying attention to that, the required pointer links are displayed for an example in Figure 3.

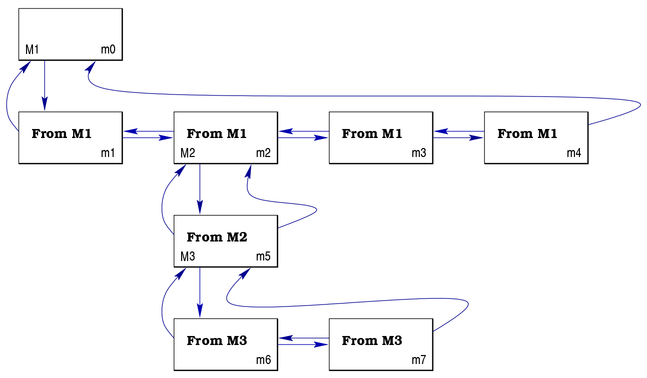

Figure 3: Mapping Database Memory Representation

3.3 Referencing a Mapping Entry

To find a reference to the database entry corresponding to a virtual address in address space, the kernel needs an appropriate lookup function. This function can either be implemented as a hash function, or a shadow page table. Since the Alpha has software managed Translation Look-aside Buffers (TLB), we extended the leaf entry of a page table by a pointer into the database. This allows a fast lookup of the database entry and secondly, page tables and mapping database always represent the same state.

4 Concurrency

Concurrency always occurs when more than one thread performs map or flush operations over the same physical frame. As described, the flush operation needs to be interruptible. When a thread reaches an entry locked by another thread it immediately switches to the thread that keeps the lock.
The following shows the flush implementation in pseudo code.

```
void flush(mapping_ptr)
{
  t_map_ptr = mapping_ptr;

flush_loop:
  while (!(aquire_lock(t_map_ptr))) {
    switch_to_locking_thread();
  }

  if (has_submappings (t_map_ptr)) {
    unlink_submapping (t_map_ptr);
    t_map_ptr = t_map_ptr->submapping;
    goto flush_loop;
  }

flush_no_lock_loop:
  flush_page_table_entry (mapping_ptr);
  if (has_nextmapping (t_map_ptr)) {
    t_map_ptr = t_map_ptr->next;
    goto flush_loop;
  }

  if (parent_of(t_map_ptr) != mapping_ptr) {
    t_map_ptr = parent_of (t_map_ptr);
    goto flush_no_lock_loop;
  }
  release_lock (t_map_ptr);
}
```

The critical operation is to unlink the entry. Inconsistency of the database would occur, when many threads wait to acquire the lock and the thread currently holding the lock releases the entry and a map operation of a high priorized thread reuses exactly this entry. A field that counts the pending operations over that entry was added to remove the entry after all operations have been completed. For the operation that changes the attributes to read-only the described operation is uncritical since the mapping database is only used to lookup the corresponding entries in the page tables.
In opposite to a global lock for the frame, locking every single map entry allows to simply implement dependencies between the threads acquiring locks. This implementation avoids possibly occuring problems due to dead-lock situations or priority inversion. This can be done without changing the priority of a thread while running a database operation.
The lock as mentioned above, also affects the map and grant operation that way that a new mapping cannot be added with read-write permissions while a read-only flush operation is still in progress on the same frame.
The resulting entry structure is as follows:

- the child list, this is a doubly linked list; the first and the last entry point back to the parent entry
- a pointer to the list of destination entries
- the virtual address and the
- address space ID,
- the thread currently holding the lock and
- the counter for the number of pending threads

To implement a flush of an entire address space, all mappings of the same address space are linked together.

5 Performance

The performance measurements were done on an Alpha 21164, clocked 433 MHz with 64 MB RAM and 1 MB second level cache.
The test program structure for the map operation looks like:

```
server thread:
  from = wait (ANY, &pagefault);
  while (true) {
    fpage = flexpage_of (pagefault);
    from = reply_and_wait (from, fpage,
                    ANY, &pagefault);
  }
```

```
client thread:
  start = get_time ();
  for (page = 0; page < NR_TESTS; page++) {
    /* generate request to server thread */
    read_memory (address);

    address = next_page(address);
  }
  time = get_time () - start;
```

The benchmark test accesses every page in an address range. The first run establishes a mapping only in that address space. This case is standard for handling a page fault. Here, the responsible pager already has all pages mapped in its address space.
In the second case, even this pager has to request the page from its pager.
We measured the time from generating the request until the reply from the pager. We get costs from 3.5μs to 8.2μs for a one-level page fault and costs from 8.2μs to 18μs for a two-level page fault. For the second case, scheduling in the kernel generates additional costs. The peek at every 256^{th} run occurs due to the allocation of a blank page for the mapping database.

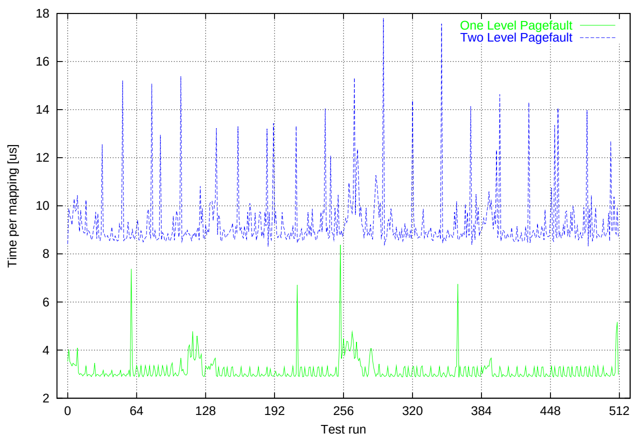

Figure 4: Memory Mapping Costs

For flush we measured two cases. A flat structure in which all mappings are initiated from the same address space and a hierarchical structure where a mapping is passed from one address space to the next. Figure 5 illustrates the costs for these both cases. Obviously the costs are independent on the mapping structure. Consequently, the costs for flushing an entry only depend on the number of entries in the tree and not from its structure. The higher costs for flushing a FlexPage with less mappings are caused by cache and TLB misses in the kernel.

6 Conclusion & Future Work

We presented an algorithm and a structure for a memory mapping database that allows μ-kernel based user level memory management providing the **map** and **flush** operation even under real-time constraints. The current state only provides a predictable map, but further work will be done to get a predictable flush operation, too.

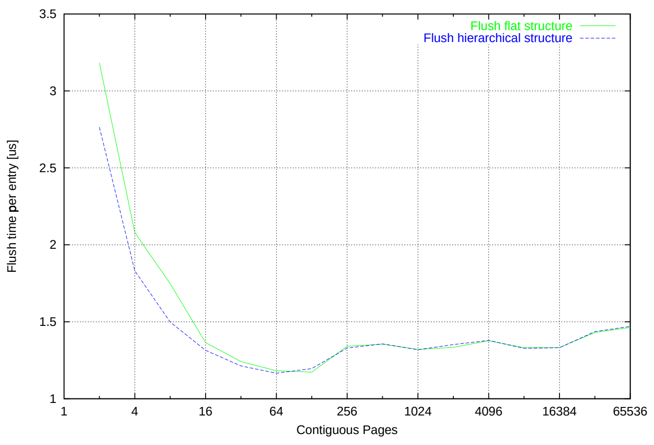

Figure 5: FlexPage Flush Costs

In the DROPS [HBB+98] project that deals with coexistence of real-time and time-sharing systems, a predictable flush operation allows to give memory pages originally reserved for the real-time part as read-only pages to the time-sharing environment. Since the page is only mapped read-only, it can simply be flushed and passed on to the real-time part within a deterministic time.

Acknowledgements

Hermann Härtig, Uwe Dannowski and Jean Wolter provided helpful feedback and commentary on earlier versions of this paper.

References

[DP93] Peter Druschel and Larry L. Peterson. Fbufs: A high bandwidth cross-domain transfer facility. In *14th ACM Symposium on Operating System Principles (SOSP)*, December 1993.

[HBB+98] Hermann Härtig, Robert Baumgartl, Martin Borris, Claude Hamann, Michael Hohmuth, Frank Mehnert, Lars Reuther, Sebastian Schönberg, and Jean Wolter. DROPS – OS support for distributed multimedia applications. In *Euro Sigops '98*, 1998.

[HC92] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 187–197, Boston, MA, October 1992.

[HWL96] H. Härtig, J. Wolter, and J. Liedtke. Flexible-sized page-objects. In *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 102–106, Seattle, WA, October 1996.

[Lie95] J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[Sch96] S. Schönberg. L4 on Alpha, design and implementation. Technical Report CS-TR-407, University of Cambridge, 1996.

[Val95] John .D Valois. *Lock-Free Data Structures.* PhD thesis, Rensselaer Polytechnic Institute, Troy New York, 5 1995.