Reliably Locking System V Shared Memory for User Level Communication in Linux*

Friedrich Seifert and Wolfgang Rehm
Chemnitz University of Technology
Department of Computer Science
Chair of Computer Architecture
Straße der Nationen 62, 09111 Chemnitz, Germany.
{sfr,rehm}@informatik.tu-chemnitz.de

Abstract

A major trend in recent cluster communication systems is to circumvent the operating system during the actual data transfers. That, on the one hand, reduces latency since there is no user–kernel transition needed and, on the other hand, increases bandwidth by avoiding additional intermediate copies. The data transfer is handled completely by the networking hardware and its DMA engine. For example, the Virtual Interface processes can access networking hardware directly in a protected manner. One of its characteristics is that it requires that all memory used for communication be locked down into physical memory. The same requirement holds true for the Infiniband ensures reliable locking of regular, i.e. private, virtual memory without altering the kernel. In this paper we present an extended solution that can handle System V shared memory as well.

Keywords—
Linux, User level networking, VIA, Memory locking, Shared Memory.

1 Introduction

The key point in user level communication is that the networking hardware has direct access to the user memory. In order to perform a DMA operation it must know the physical addresses of the particular virtual pages. There are two ways to achieve this in principle. The first requires a more sophisticated hardware since it must maintain a separate MMU or at least a Translation Lookaside Buffer (TLB), which must be kept in sync with the page tables of the host CPU by the operating system. U-Net/MM [10] is an academic implementation of this approach. However, there is also a commercial solution, QsNet by Quadrics [1], that applies the same principle. Whenever the NIC wants to access a virtual page that is not currently mapped by its MMU it requests the operating system to retrieve the physical address from the page tables. Eventually the page needs to be swapped in if it is not present. While this approach is quite flexible and puts no constraints on the communication buffers, performance can suffer significantly in case of frequent TLB misses and not-present pages. Incoming messages must be rejected and retransmitted or buffered by the hardware in such situations. Investigations on U-Net/MM have shown that the TLB miss rate can be as high as 35% in real applications.

The Virtual Interface Architecture (VIA), which was deeply influenced by the U-Net project, circumvents such awkwardnesses in that it requires that all communication memory be locked into physical memory. Besides the physical addresses of all those pages are permanently stored on the NIC in the so called *Translation and Protection Table (TPT)*. This is what VIA calls *memory registration*. Another aspect that has influenced our implementation of memory locking heavily is the fact that the VIA specification explicitly allows the same virtual memory region to be registered several times, see [5].

The problem of memory locking will also be an important issue in the upcoming Infiniband networking technology. Infiniband is basically an evolution of the Virtual Interface Architecture and employs a similar memory registration scheme where all pages must be locked [6, Ch. 10.6.4.2.1 Registered Memory Residency]. Moreover, any kind of user level communication facility that does not allow pages to be swapped must care about memory locking. Another example is our PCI–SCI bridge that provides Distributed Shared Memory (DSM) and VIA support [7, 8]. All exported pages of a process have to be locked down.

In this paper we address the issue of providing reliable memory locking for a VIA, IBA or similar implementation in a Linux environment. In [5] we have shown how regular, i.e. private, process memory can be locked reliably, allowing for multiple registration using a recent kernel mechanism called *kiobufs*. At that time the kiobuf functionality had to be patched into the kernel, but now with the 2.4 version it is an integral part of the standard kernel. Hence, no patches are required, which is important for the usability of a driver because if one wants to use several drivers, each of which requiring its own patch it is possible that those patches clash and one comes in trouble. Our solution also meets all conditions to be integrated into the main stream kernel.

One thing that remained open was the locking of System V shared memory (in the following referred to as shared memory). In this paper we show how the Locked Memory Manager (LMM) can be extended to also handle shared memory properly. This is desired in certain situations. Suppose, two VIA

*The work presented in this paper is carried out in strong interaction with the project GRANT SFB393/B6 of the DFG (German National Science Foundation).

processes A and B are running on the same node, say node 1, see figure 1. Both processes want to receive the same data from process C. This could be achieved by having separate VI connections between A and C resp. B and C. However, the data had to be transfered twice since VIA does not allow any kind of multicast. If a shared memory segment is used, a single transfer is sufficient to make the data available to both processes. Above that, process B might like to register the shared memory segment with one of its own VIs. Such setups can be applied to create efficient collective operations in MPI when there are several processes running on each node.

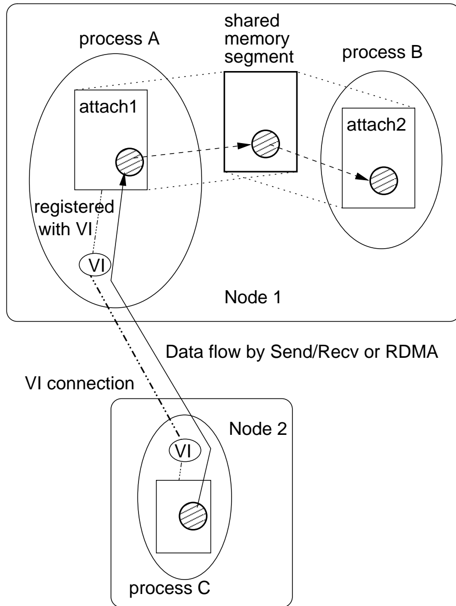

Figure 1. Using shared memory as VIA communication buffer

In the following we will discuss the problems that arise when shared memory comes into play and what are the ways to solve them. We will start with a short description of locking private memory. After that we will give an overview of the administration of shared memory in the Linux 2.4.x kernels. The fourth section shows the problems shared mappings cause and how they have been solved. Finally, we will make some conclusions.

2 The Locked Memory Manager so far

In this section we will explain briefly how we managed to lock private memory reliably. A detailed description can be found in [5].

The Locked Memory Manager was primarily targeted to a VIA implementation. So it was implemented inside the Kernel Agent, which is practically a device driver. In order to make the LMM generally usable it has been turned into a separate Linux kernel module that can be loaded at run time. The Kernel Agent invokes the LMM during its memory registration/deregistration functions. Figure 2 illustrates how the LMM and VIA go together.

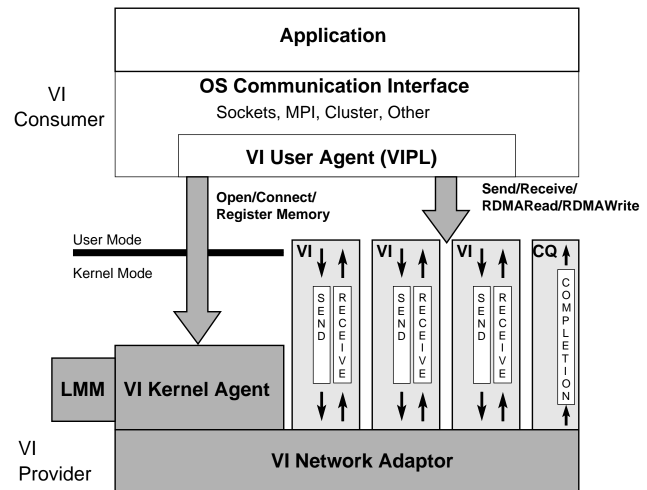

Figure 2. The VI Architecture using the LMM

2.1 The Interface

The LMM provides the following functions:

`lmm_lock_area(IN start, IN length, OUT la_handle)` Locks the given virtual address range and returns a handle to it. It can be called several times for the same addresses even with overlapping ranges, resulting in distinct handles.

`lmm_unlock_area(IN la_handle)` Releases the given locked area. Note, that the corresponding pages may still stay locked, if the memory area or parts of it have been locked several times.

`lmm_get_pages(IN la_handle, OUT pages)` Returns the physical page addresses of a locked area. There is also a function that returns a list consecutive page blocks for convenient creation of scatter/gather maps.

`lmm_cleanup()` Releases all locked areas of the calling process.

2.2 The Implementation

There are two principal ways to prevent user memory from being swapped out:

- page–based using physical addresses
- VMA[1]–based using virtual addresses.

The former is performed by setting the `PG_locked` flag in physical page's entry in the kernel's `mem_map` array. The swapping

[1] VMA stands for Virtual Memory Area, a kernel concept for describing the different parts of a process' virtual address space.

code leaves all pages with that bit set untouched. The latter method utilizes the VM_LOCKED flag in the VMA describing the virtual address range. When Linux is going to swap out something it first selects a process, and then goes through its list of VMAs to find one to move to secondary storage. While doing so Linux skips all locked VMAs. A user process can lock part of its memory by means of the mlock system call. Although this looks quite simple there are a number of obstacles in using this method in a device driver. A minor one is that the mlock function is not visible to driver modules in the standard kernel. Hence, a small kernel modification would be necessary. Also, some permission problems have to be solved, since only superuser processes are normally allowed to lock down memory. These problems are of minor importance and can be solved as will be shown in section 4.

A really severe problem is how to retrieve the physical addresses once a number of pages has been locked. Of course, the addresses are held in the process' page table and one could just walk through them and read the proper entries. However, a device driver should never apply this method. The reason is that Linus Torvalds, the Linux maintainer has decided to never accept a driver that walks the page tables, since there can be bad interferences with the memory management code. That means, in order to create a driver that could possibly be adopted to the main stream kernel, a different way to get the physical addresses must be used.

Fortunately, a new kernel mechanism, called *kiobufs* was introduced recently [9]. It is a integral part of the 2.4.x kernels, but there is also a patch for the 2.2.x series available. Kiobufs were created in conjunction with Raw I/O, which allows data transfers between disk and user memory without intermediate copies by the kernel. During the transfers the pages must be locked. A kiobuf can be mapped to a part of the user address space. After that the physical page addresses can be read directly from the kiobuf, and the page tables don't have to be touched. While all pages were locked immediately upon mapping a kiobuf in the 2.2.x kernels, 2.4.x introduced separate functions to lock and unlock an already mapped kiobuf.

The problematic aspect with kiobufs is that any physical page can be held in at most one locked kiobuf at any time. This requires a tracking mechanism for locked pages in order to allow for multiple registrations. Since kiobufs represent a page–based locking scheme one could assume that the lock count tracking could be based on physical page addresses. The problem is that the physical address had to be known prior to creating the kiobuf, and the only way to retrieve them is by walking the page tables. This, however, contradicts the requirements for a kernel compliant driver as explained above. That is why we have based the lock count tracking upon virtual addresses. This is possible because different virtual pages are always mapped to distinct physical pages for private memory, provided a COW (Copy–on–write) has been performed.

Locked Memory Areas: LMAs In [5] we introduced a structure, called *locked memory area*, or LMA. An LMA describes a contiguous interval $[start, end)$ of virtual pages with the same lock count. There is one locked kiobuf per LMA, that describes the physical pages. LMA's are kept in per process lists in ascending order of their start addresses.

Whenever a new area is to be locked the list of existing LMAs is scanned for intersections. There are five possible types of intersections that are illustrated in figure 3. Depending on the kind of intersection the proper actions are taken. An example situation is given in figure 4.

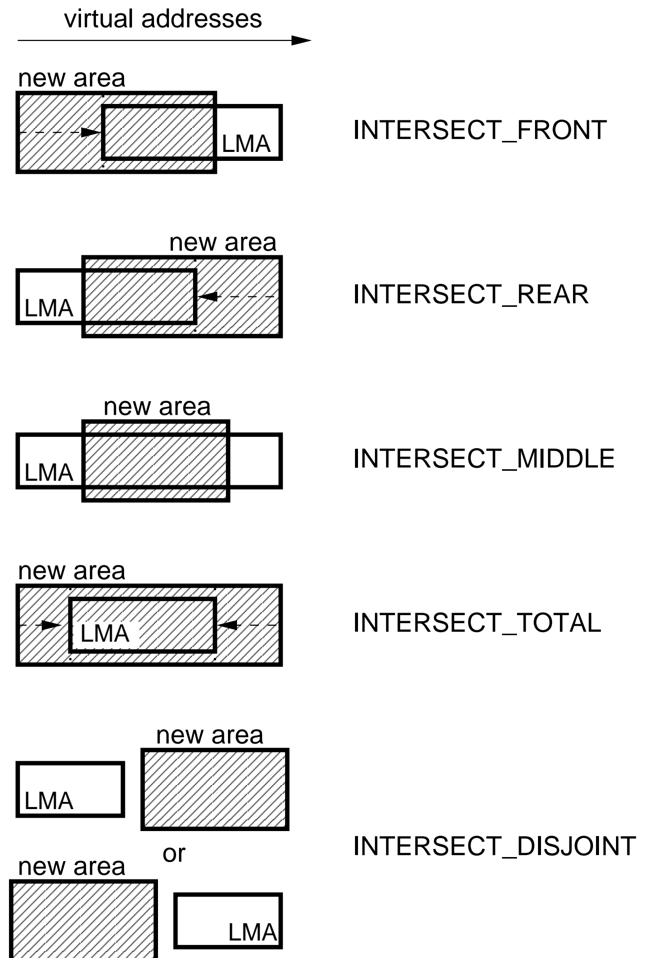

Figure 3. Possible types of intersection of new area and LMA

3 System V Shared Memory in Linux 2.4.x

Before we go on to explain how the Locked Memory Manager can be extended to handle shared memory, this section gives an insight in how shared memory is implemented in the 2.4.x kernels.

3.1 How to use SysV Shm

UNIX System V defines a number of system calls to control shared memory between processes. Generally, first a segment must be created, and then processes can attach to it to get it

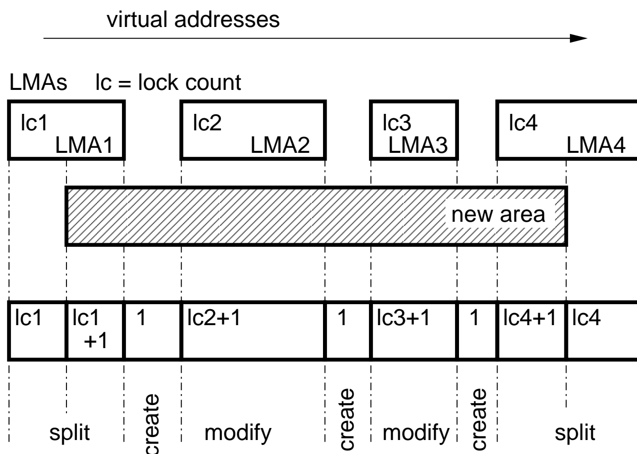

Figure 4. Locking a new area

mapped into their address space. The most important functions are:

`shmget(IN key, IN size, IN shmflg)` Opens an existing segment or creates a new one. It returns a so called *shmid*.

`shmat (IN shmid, IN shmaddr, IN shmflg)` Maps the segment corresponding to *shmid* into the process' user address space. The virtual start address of this area is returned.

`shmdt (IN shmaddr)` Removes the shared segment from the calling process' address space.

There are other functions to destroy a segment and to change its attributes.

3.2 The Implementation of SysV Shm

Following we will have a look at the internals. The implementation of shared memory in Linux 2.4.x differs significantly from former versions in that it is now based on the so called *shmem* filesystem.

A shared segment is treated as a file basically with an associated `shmid_kernel` data structure describing the segment size, its access permissions and times, and the number of attaches. Moreover, a segment may have the SHM_LOCKED flag set, that will play an important role later.

In order to create a new segment a new `shmid_kernel` structure is allocated and filled properly and a new file structure is allocated, which is assigned a special `mmap` method. Further, an inode structure is created, which again points to an `address_space` structure [2, Ch. 4 Linux Page Cache]. The latter describes the actual pages that make up the shared segment. Finally, a new *shmid* is created and the segment is added to the kernels pool of shared segments.

Once a segment has been established, any process that knows the *shmid* and has got the proper permissions can attach to it. The kernel's attach function retrieves the `shmid_kernel` structure by means of the identifier and simply calls `do_mmap` with the segment's size and the file structure assigned during creation. This results in reserving a free virtual address range, creating a new VMA and invoking the special shared memory `mmap` method. This one assigns special open, close and nopage methods to the VMA. Finally, the number of attaches of the segment is increased and the attach operation in complete.

Although the shared area can be accessed now, there are no pages allocated to it yet. This is done page by page by the nopage method upon the first access to the attached segment. If a page is accessed for the very first time, a new page is allocated and added to the segment's `address_space` structure. The page address is also returned and written to the page table, so that future accesses produce no more page faults.

If, now, a second process attaches to the segment and accesses one of its pages, an exception is raised, since that process has not got a valid page table entry for that page. Though, no new page has to be allocated in this case, since this was already done when the first process accessed the page. Instead, the page address is simply retrieved from the `address_space` structure and written to the page table.

To summarize this section figure 5 shows the interaction of the various data structure that are used to established shared memory at the example of two processes sharing a three page long segment.

4 Locking Shared Memory

As explained above the LMM was based on the assumption that never two virtual pages are mapped to the same physical page in order to maintain the requirement that any physical page is locked at most once by an kiobuf. Obviously this does not hold true for shared mappings. If one process has locked a shared segment by means of the LMM and another one tries to lock its own mapping of that segment, the procedure will fail at the moment when the kiobuf is to be locked. The kiobuf code will recognize that the physical page's `PG_locked` flag is already set, and return an error.

One could suppose a solution is to have only one process actually lock a shared segment, while the others just rely on that. Processes using a shared segment need to cooperate anyways. While this sounds quite easy to achieve there are two subtle difficulties.

1. In a VIA context memory is locked implicitly by VipRegisterMem, and there is no possibility to tell the Kernel Agent whether or not to lock the memory, even when all processes on that node have agreed upon which one should do the locking. Hence, the Kernel Agent must control which process actually locks the memory.

2. Suppose, we have found a strategy for the Kernel Agent to determine the locking process, and that process has created a kiobuf for some part of the shared segment. Now we come to another, ultimate problem. Any process that wants to access its own mapping of the locked part, and theses pages have not been accessed before, will block as long as the physical pages are locked. The reason for this unintentional behavior can be found in the *nopage* method. As ex-

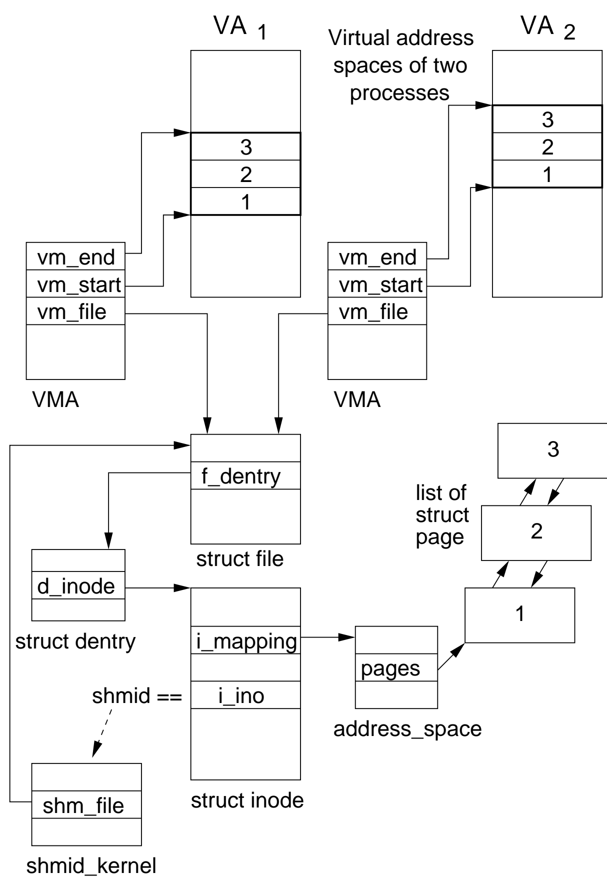

Figure 5. Implementation of shared memory in Linux 2.4.x

plained above the nopage routine of shared mappings tries to get a page from the corresponding `address_space` structure. This should be successful in our case, since the physical pages were allocated when the first process locked the area. However, the function retrieving the page needs to lock it, which is done by setting that `PG_locked` flag in the struct page. If the page is already locked, the process is put to sleep and reawakened only when that page is unlocked. This means, when the first process unlocks the kiobuf in our case.

Thus, we must state that, first, the device driver (i.e. the Kernel Agent in a VIA context) must control memory locking and, secondly, the pure page–based approach is unsuitable for shared mappings.

What is left is the opportunity to use some kind of VMA–based locking. However, a virtual–address only approach is not enough, since we need a way to determine the physical addresses without touching the page tables. The solution is to combine VMA–based locking and kiobufs.

4.1 Combining the VMA–based approach and kiobufs

As mentioned before there is a SHM_LOCKED flags for shared segments. It can be altered by the `shm_ctl` system call. Its purpose is to prevent the segment as a whole from paging. Furthermore there is the `mlock` system call that can be used on any virtual address range, including shared areas.

In the following we will present the general idea of combining kiobufs and virtual address based locking as well as discuss the pros and cons of `mlock` and `shm_ctl`.

4.1.1 Introducing Unlocked LMAs

We apply basically the same data structures for shared areas that are used for private locked memory, the central one of which is LMAs. Though, we introduce a new kind of LMAs, *unlocked* LMAs. This terminology might sound contradictory but *unlocked* refers to the kiobuf and, hence, the physical pages. So the correct naming should be *physically unlocked* LMA. For convenience, we simply say *unlocked* knowing that it means the kiobuf. The area itself *is* locked, though by other means. So the L in LMA retains its legitimacy.

When an LMA is created it can be specified whether or not the appendant kiobuf should be locked physically. This is possible since the kiobuf implementation in the 2.4.x kernel separates mapping and locking. Due to this each process attached to a shared segment can have its own LMAs and kiobufs.

Now we have got a clean way to get the physical page addresses, that are needed for the DMA engines of the I/O hardware. What is missing is a way to ensure locking.

4.1.2 `shm_ctl` vs. `mlock`

Both functions have two common flaws. First, only superuser processes are normally allowed to use them, and secondly the functions are not exported to kernel modules by default. The first problem can easily be solved by having the driver grant the proper capability to the process. The second one can be solved in differently elegant ways. The less elegant way is to modify the kernel slightly to export the symbols in question. Although it should be really easy to apply such a patch, even manually, it makes the usage of such a driver less convenient.

If we use `shm_ctl` to lock the segment, we only need a single counter per segment. Its value equals the number of LMAs within the segment across all attached processes. This means it must be incremented whenever an LMA for this segment is created, and decremented whenever an LMA for this segment is destroyed. Upon creation of the first LMA, the segment is locked via the shared memory control function, while it is unlocked as soon as the lock counter reaches zero. This version has been implemented and works reliably. However, during our investigations a kernel bug was discovered, namely the swapping code did not honor the SHM_LOCKED flags of shared segments at all.

In order to circumvent the kernel modification, `shm_ctl` should be called from user level through the normal system call interface. This requires that the user process has got enough

permissions. This can be accomplished by granting the appropriate capability, strictly speaking CAP_IPC_LOCK to the process when the device is opened. The crucial problem in moving the lock operation into the VIPL[2] is that the *shmid* is not known there anymore, only the virtual address. `shm_ctl`, however, needs this parameter.

This problem does not occur with mlock, since it takes the virtual address as parameter. According to the Linux manual page for mlock a shared mapping stays in RAM as long as at least one process has mlocked it, however, the current kernel code (up to 2.4.6) apparently does not honour that.

At the example of VIA we can do the following. The VipRegisterMem function of VIPL invokes the corresponding function of the Kernel Agent (by means of the ioctl system call). The Kernel Agent creates resp. modifies the LMAs appropriately and returns an indication whether or not the shared segment must be locked along with the segment boundaries. Upon return of the ioctl, the VipRegisterMem performs an mlock operation on the given area if necessary. Some care must be taken here to prevent race conditions with the swapper. It is possible and even probable that the process is descheduled when it returns from the I/O control. Since the area has not been locked yet by that time it is possible that parts of it get paged out before the mlock call is performed. This would lead to inconsistencies as different physical pages will be allocated when the area is paged in again, but the original physical addresses have been passed to the I/O hardware already. In order to prevent that race condition, VipRegisterMem must call mlock on the area to be registered *before* invoking the Kernel Agent. Now the Kernel Agent can safely create the LMAs for the area. It must also return the virtual start address of the whole shared segment, its length and an indication for the mlock operation to be performed, which is one of those:

MLOCK_NOP Do nothing.

MLOCK_LOCK Lock the whole segment. This is used when the process registers some part of the segment for the first time. According to the specification of the mlock system call it would be sufficient if only one process, i.e. the first one, locks the area. A problem arises if that processes deregisters the area before the other processes. The memory would be unlocked in this case, or another process had to lock it subsequently, which is practically impossible. For this reason all processes have to lock the shared mapping. Hence, each process needs its own lock counter for every segment. Note that it is not a problem that a part of the segment (namely the area to be registered) has been locked before, since mlock simply sets a bit in the VMAs. Hence, nothing is changed for the previously locked parts.

MLOCK_UNLOCK Unlock the area. This value is returned by the Kernel Agent when the process' first registration within the segment has failed for some reason. In this case the initial mlock operation must be made undone. Although it is not erroneous to unlock the whole segment, since just a bit in the VMAs is cleared, it is more efficient to only unlock the originally given address range.

[2] Virtual Interface Provider Library, Intel's name of the VIA User Agent, introduced in the Developer's Guide [3]

Unlocking memory is done in a similar way, execpt that nothing needs to be done prior to calling the Kernel Agent. The driver modifies and/or destroys the LMAs appropriately, and if the process' last LMA for the shared segment has been removed, MLOCK_UNLOCK is returned, whereupon VipDeregisterMem should call munlock for the whole segment, the boundaries of which are returned by the Kernel Agent function as well. Otherwise MLOCK_NOP is returned and VipDeregisterMem doesn't need to perform any additional action.

Figure 6 illustrates this method.

5 Performance Evaluation and Impact on Locking Private Memory

Although the intention of all user level communication is to exclude operating system calls from the communication paths, in cases such as VIA, where communication buffers must be registered, zero–copy protocols need on-the-fly registration. However, it is highly recommended to avoid those expensive operations by applying cache like strategies on registered memory [4].

We have conducted several measurements on a Pentium III machine at 450 MHz running Linux kernel 2.4.0. We examined the memory registration and deregistration of our VIA implementation, in the course of which the Locked Memory Manager has been developed.

First of all we measured the registration and deregistration times for shared memory using the `shm_ctl` function from inside the kernel. The results are shown in table 1. In this test the shared segment had been registered by another process before. The times for the very first registration of a shared area are slightly higer for small buffer sizes, see table 2. The additional times can be attributed to the locking of the shared segment that has to be carried out upon the first registration.

As explained in section 4 the method tested so far needs a small kernel modification. The solution shown above, that does without kernel changes, adds some more overhead due to the additional mlock/munlock system calls. Figure 7 compares the registration times for a shared memory segment using `shm_ctl` inside the kernel on the one hand and calling mlock from the user level on the other. The difference is about $3.5 \mu sec$ for small blocks. For large blocks there is no significant difference to be seen.

The last question to be answered is how much does the user mlock affect registration of private memory. Note, that mlock has to be performed in any case, since the VIPL cannot distinguish between private and shared memory. Figure 8 shows the results. It can clearly be seen that there is a significant overhead for all block sizes. The bottom most line of the graph shows the relation of the times with and without user mlock, which varies from 1.7 for a single page to 1.3 for 16 megabytes.

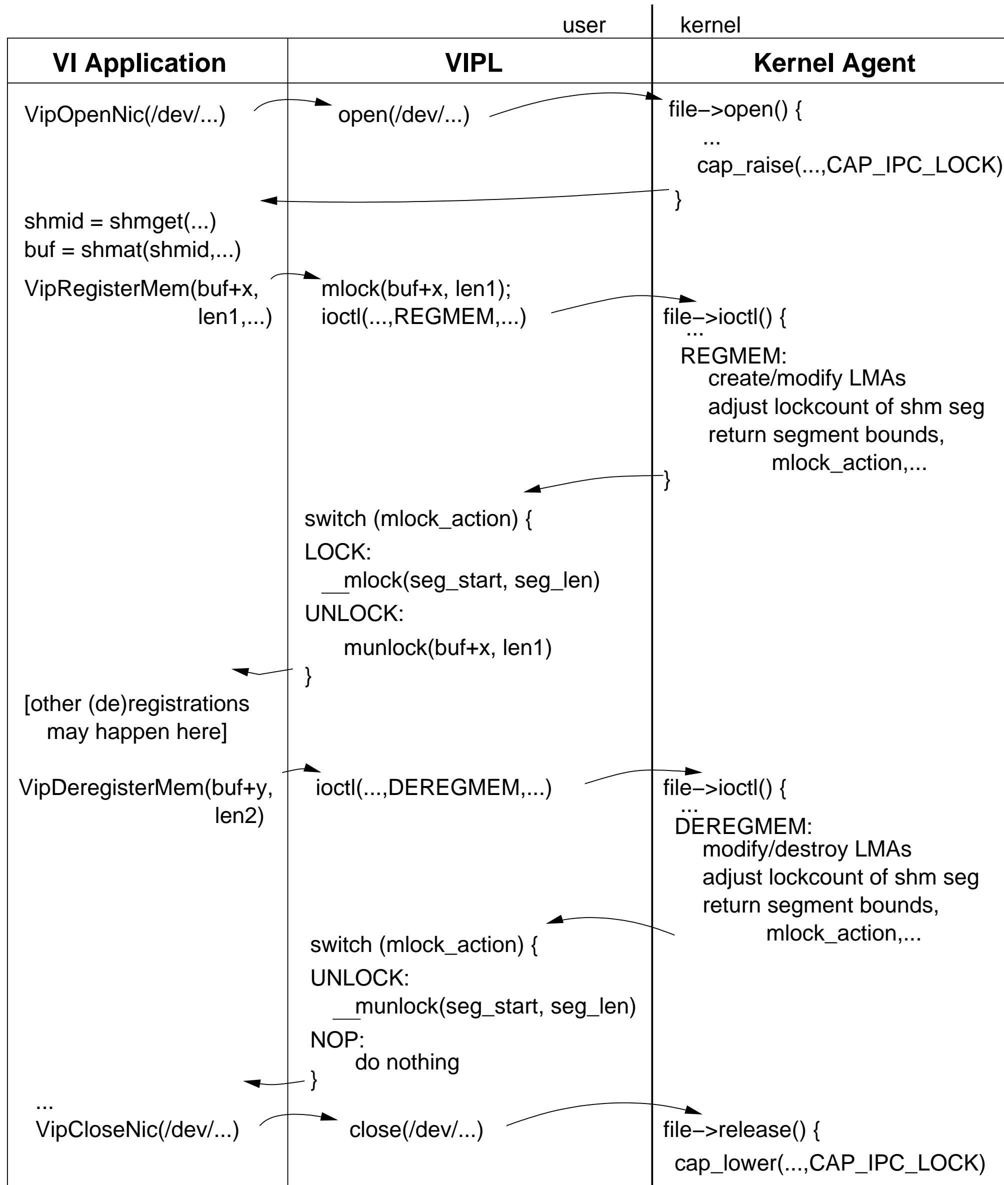

Figure 6. Using mlock from user level

6 Conclusions

In this paper we have proposed an extension to the Locked Memory Manager [5] that enables it to handle System V shared memory properly. Each process that has attached to a shared segment can lock parts of it regardless of other processes attached. We have shown that a purely page-based approach is not suitable due to the internals of the Linux memory management. Instead we have developed a solution combining virtual address based locking and the kiobuf mechanism. Further, we have shown that it is possible to find a solution that needs no kernel changes. However, it poses some extra overhead on locking shared as well as private memory. Hence, one has to trade off between performance and convenience of installing the driver. The authors are planning to make a proposal to integrate the LMM with the main stream kernel, since they believe it is helpful for all kinds of user level communication like VIA, Infiniband, SCI, direct sockets and so on.

An extension to support all kinds of shared memory mappings is expected to be easily derived from the current solution, since they uses the same underlying mechanism. The difference is that there is no `shmid_kernel` structure for such mappings and, hence the locking must be based on the file mapped and be done by means of `do_mlock`. Further, there are options for optimizations left. The LMAs of a process are still stored in linear lists. A performance improvement could be achieved by using AVL trees. Besides, it should be figured out if it is useful to merge adjacent LMAs if they have the same lock count.

Table 1. Times for registration/deregistration shared memory when is was already registered by another process (in μs)

Size (KB)	register	deregister
4	6.9	4.4
8	8.4	4.7
16	10.8	5.3
32	15.8	6.7
64	25.5	9.3
128	47.2	14.8
256	87.8	25.9
512	177	49.5
1024	340	98.8
2048	697	205
4096	1371	409
8192	2739	814
16384	5414	1646

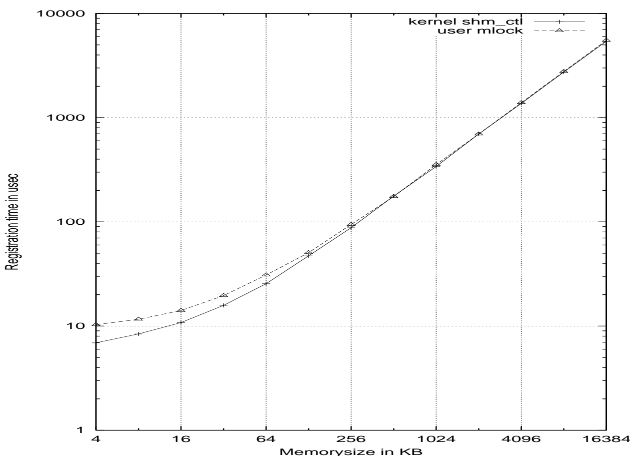

Figure 7. Registering shared memory with/without user level mlock

Table 2. Times for registration/deregistration shared memory for the first time (in μs)

Size (KB)	register	deregister
4	7.4	4.5
8	8.8	4.9
16	11.4	7.5
32	16.4	7.2
64	26.1	9.8
128	49.5	15.6
256	86.8	26.5
512	167	49.2
1024	334	101
2048	668	206
4096	1373	409
8192	2841	815
16384	5507	1650

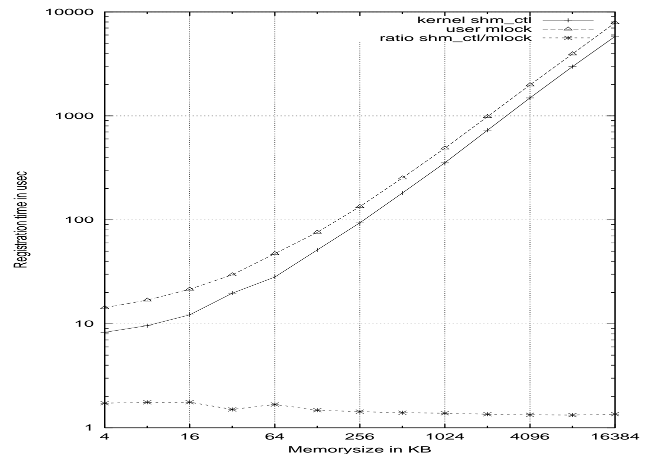

Figure 8. Registering private memory with/without user level mlock

References

[1] Quadrics QsNet High Performance Interconnect. http://www.quadrics.com/web/public/fliers/qsnet.html.

[2] T. Aivazian. Linux kernel internals. http://www.linuxdoc.org/LDP/lki.

[3] Intel Corporations. *Intel Virtual Interface (VI) Architecture Implementation Guide, Draft Revision 0.95*, May, 15 1998.

[4] L. Jordan. Entwicklung eines effizienten speichermanagementes fuer das chempi via/sci device. Study thesis (german), Chair of Computer Architecture, Chemnitz University of Technology, 2000.

[5] F. Seifert and W. Rehm. Proposing a mechanism for reliably locking via communication memory in linux. In *In proceedings of IEEE International Conference on Cluster Computing CLUSTER2000*, Chemnitz, Germany, Nov 28 - Dec 1 2000.

[6] I. Specification and V. Release. Infiniband trade association, 2000.

[7] M. Trams. Design of a system-friendly PCI-SCI bridge with an optimized user-interface. Diploma thesis, Chair of Computer Architecture, Chemnitz University of Technology, 1998.

[8] M. Trams, W. Rehm, D. Balkanski, and S. Simeonov. Memory management in a combined VIA/SCI hardware. In *Proceedings to PC-NOW 2000, International Workshop on Personal Computer based Networks of Workstations held in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS 2000)*, Cancun, Mexico, May 1-5 2000.

[9] S. C. Tweedie et al. Raw I/O enhancements. http://oss.sgi.com/projects/rawio.

[10] M. Welsh, A. Basu, and T. v. Eicken. Incorporating memory management into user-level network interfaces. Technical report, Department of Computer Science, Cornell University, Ithaca, 1997.