A New Virtual Memory Implementation for Berkeley UNIX®

Marshall Kirk McKusick
Michael J. Karels

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

With the cost per byte of memory approaching that of the cost per byte for disks, and with file systems increasingly distant from the host machines, a new approach to the implementation of virtual memory is necessary. Rather than preallocating swap space which limits the maximum virtual memory that can be supported to the size of the swap area, the system should support virtual memory up to the sum of the sizes of physical memory plus swap space. For systems with a local swap disk, but remote file systems, it may be useful to use some of the memory to keep track of the contents of the swap space to avoid multiple fetches of the same data from the file system.

The new implementation should also add new functionality. Processes should be allowed to have large sparse address spaces, to map files into their address spaces, to map device memory into their address spaces, and to share memory with other processes. The shared address space may either be obtained by mapping a file into (possibly different) parts of their address space, or by arranging to share ‘‘anonymous memory’’ (that is, memory that is zero fill on demand, and whose contents are lost when the last process unmaps the memory) with another process as is done in System V.

One use of shared memory is to provide a high-speed Inter-Process Communication (IPC) mechanism between two or more cooperating processes. To insure the integrity of data structures in a shared region, processes must be able to use semaphores to coordinate their access to these shared structures. In System V, these semaphores are provided as a set of system calls. Unfortunately, the use of system calls reduces the throughput of the shared memory IPC to that of existing IPC mechanisms. We are proposing a scheme that places the semaphores in the shared memory segment, so that machines that have a test-and-set instruction can handle the usual uncontested lock and unlock without doing a system call. Only in the unusual case of trying to lock an already-locked lock or in releasing a wanted lock will a system call be required. The interface will allow a user-level implementation of the System V semaphore interface on most machines with a much lower runtime cost.

1. Motivations for a New Virtual Memory System

The virtual memory system distributed with Berkeley UNIX has served its design goals admirably well over the ten years of its existence. However the relentless advance of technology has begun to render it obsolete. This section of the paper describes the current design, points out the current technological trends, and attempts to define the new design considerations that should be taken into account in a new virtual memory design.

1.1. Implementation of 4.3BSD virtual memory

All Berkeley Software Distributions through 4.3BSD have used the same virtual memory design. All processes, whether active or sleeping, have some amount of virtual address space associated with them. This virtual address space is the combination of the amount of address space with which they initially started plus any stack or heap expansions that they have made. All requests for address space are allocated from available swap space at the time that they are first made; if there is insufficient swap space left to honor the allocation, the system call requesting the address space fails synchronously. Thus, the limit to available virtual memory is established by the amount of swap space allocated to the system.

Memory pages are used in a sort of shell game to contain the contents of recently accessed locations. As a process first references a location a new page is allocated and filled either with initialized data or zeros (for new stack and break pages). As the supply of free pages begins to run out, dirty pages are pushed to the previously allocated swap space so that they can be reused to contain newly faulted pages. If a previously accessed page that has been pushed to swap is once again used, a free page is reallocated and filled from the swap area [Babaoglu79], [Someren84].

1.2. Design assumptions for 4.3BSD virtual memory

The design criteria for the current virtual memory implementation were made in 1979. At that time the cost of memory was about a thousand times greater per byte than magnetic disks. Most machines were used as centralized time sharing machines. These machines had far more disk storage than they had memory and given the cost tradeoff between memory and disk storage, wanted to make maximal use of the memory even at the cost of wasting some of the disk space or generating extra disk I/O.

The primary motivation for virtual memory was to allow the system to run individual programs whose address space exceeded the memory capacity of the machine. Thus the virtual memory capability allowed programs to be run that could not have been run on a swap based system. Equally important in the large central timesharing environment was the ability to allow the sum of the memory requirements of all active processes to exceed the amount of physical memory on the machine. The expected mode of operation for which the system was tuned was to have the sum of active virtual memory be one and a half to two times the physical memory on the machine.

At the time that the virtual memory system was designed, most machines ran with little or no networking. All the file systems were contained on disks that were directly connected to the machine. Similarly all the disk space devoted to swap space was also directly connected. Thus the speed and latency with which file systems could be accessed were roughly equivalent to the speed and latency with which swap space could be accessed. Given the high cost of memory there was little incentive to have the kernel keep track of the contents of the swap area once a process exited since it could almost as easily and quickly be reread from the file system.

1.3. New influences

In the ten years since the current virtual memory system was designed, many technological advances have occurred. One effect of the technological revolution is that the micro-processor has become powerful enough to allow users to have their own personal workstations. Thus the computing environment is moving away from a purely centralized time sharing model to an environment in which users have a computer on their desk. This workstation is linked through a network to a centralized pool of machines that provide filing, computing, and spooling services. The workstations tend to have a large quantity of memory, but little or no disk space. Because users do not want to be bothered with backing up their disks, and because of the difficulty of having a centralized administration backing up hundreds of small disks, these local disks are typically used only for temporary storage and as swap space. Long term storage is managed by the central file server.

Another major technical advance has been in all levels of storage capacity. In the last ten years we have experienced a factor of four decrease in the cost per byte of disk storage. In this same period of time the cost per byte of memory has dropped by a factor of a hundred! Thus the cost per byte of memory compared to the cost per byte of disk is approaching a difference of only about a factor of ten. The effect of this change is that the way in which a machine is used is beginning to change dramatically. As the amount of physical memory on machines increases and the number of users per machine decreases, the expected mode of operation is changing from that of supporting more active virtual memory than physical memory to that of having a surplus of memory that can be used for other purposes.

Because many machines will have more physical memory than they do swap space (with diskless workstations as an extreme example!), it is no longer reasonable to limit the maximum virtual memory to the amount of swap space as is done in the current design. Consequently, the new design will allow the maximum virtual memory to be the sum of physical memory plus swap space. For machines with no swap space, the maximum virtual memory will be governed by the amount of physical memory.

Another effect of the current technology is that the latency and overhead associated with accessing the file system is considerably higher since the access must be over the network rather than to a locally-attached disk. One use of the surplus memory would be to maintain a cache of recently used files; repeated uses of these files would require at most a verification from the file server that the data was up to date. Under the current design, file caching is done by the buffer pool, while the free memory is maintained in a separate pool. The new design should have only a single memory pool so that any free memory can be used to cache recently accessed files.

Another portion of the memory will be used to keep track of the contents of the blocks on any locally-attached swap space analogously to the way that memory pages are handled. Thus inactive swap blocks can also be used to cache less-recently-used file data. Since the swap disk is locally attached, it can be much more quickly accessed than a remotely located file system. This design allows the user to simply allocate their entire local disk to swap space, thus allowing the system to decide what files should be cached to maximize its usefulness. This design has two major benefits. It relieves the user of deciding what files should be kept in a small local file system. It also insures that all modified files are migrated back to the file server in a timely fashion, thus eliminating the need to dump the local disk or push the files manually.

2. User Interface

This section outlines our new virtual memory interface as it is currently envisioned. The details of the system call interface are contained in Appendix A.

2.1. Regions

The virtual memory interface is designed to support both large, sparse address spaces as well as small, densely-used address spaces. In this context, ‘‘small’’ is an address space roughly the size of the physical memory on the machine, while ‘‘large’’ may extend up to the maximum addressability of the machine. A process may divide its address space up into a number of regions. Initially a process begins with four regions; a shared read-only fill-on-demand region with its text, a private fill-on-demand region for its initialized data, a private zero-fill-on-demand region for its uninitialized data and heap, and a private zero-fill-on-demand region for its stack. In addition to these regions, a process may allocate new ones. The regions may not overlap and the system may impose an alignment constraint, but the size of the region should not be limited beyond the constraints of the size of the virtual address space.

Each new region may be mapped either as private or shared. When it is privately mapped, changes to the contents of the region are not reflected to any other process that map the same region. Regions may be mapped read-only or read-write. As an example, a shared library would be implemented as two regions; a shared read-only region for the text, and a private read-write region for the global variables associated with the library.

A region may be allocated with one of several allocation strategies. It may map some memory hardware on the machine such as a frame buffer. Since the hardware is responsible for storing the data, such regions must be exclusive use if they are privately mapped.

A region can map all or part of a file. As the pages are first accessed, the region is filled in with the appropriate part of the file. If the region is mapped read-write and shared, changes to the contents of the region are reflected back into the contents of the file. If the region is read-write but private, changes to the region are copied to a private page that is not visible to other processes mapping the file, and these modified pages are not reflected back to the file.

The final type of region is ‘‘anonymous memory’’. Uninitialed data uses such a region, privately mapped; it is zero-fill-on-demand and its contents are abandoned when the last reference is dropped. Unlike a region that is mapped from a file, the contents of an anonymous region will never be read from or written to a disk unless memory is short and part of the region must be paged to a swap area. If one of these regions is mapped shared, then all processes see the changes in the region. This difference has important performance considerations; the overhead of reading, flushing, and possibly allocating a file is much higher than simply zeroing memory.

If several processes wish to share a region, then they must have some way of rendezvousing. For a mapped file this is easy; the name of the file is used as the rendezvous point. However, processes may not need the semantics of mapped files nor be willing to pay the overhead associated with them. For anonymous memory they must use some other rendezvous point. Our current interface allows processes to associate a descriptor with a region, which it may then pass to other processes that wish to attach to the region. Such a descriptor may be bound into the UNIX file system name space so that other processes can find it just as they would with a mapped file.

2.2. Shared memory as high speed interprocess communication

The primary use envisioned for shared memory is to provide a high speed interprocess communication (IPC) mechanism between cooperating processes. Existing IPC mechanisms (*i.e.* pipes, sockets, or streams) require a system call to hand off a set of data destined for another process, and another system call by the recipient process to receive the data. Even if the data can be transferred by remapping the data pages to avoid a memory to memory copy, the overhead of doing the system calls limits the throughput of all but the largest transfers. Shared memory, by contrast, allows processes to share data at any level of granularity without system intervention.

However, to maintain all but the simplest of data structures, the processes must serialize their modifications to shared data structures if they are to avoid corrupting them. This serialization is typically done with semaphores. Unfortunately, most implementations of semaphores are done with system calls. Thus processes are once again limited by the need to do two system calls per transaction, one to lock the semaphore, the second to release it. The net effect is that the shared memory model provides little if any improvement in interprocess bandwidth.

To achieve a significant improvement in interprocess bandwidth requires a large decrease in the number of system calls needed to achieve the interaction. In profiling applications that use serialization locks such as the UNIX kernel, one typically finds that most locks are not contested. Thus if one can find a way to avoid doing a system call in the case in which a lock is not contested, one would expect to be able to dramatically reduce the number of system calls needed to achieve serialization.

In our design, cooperating processes manage their semaphores in their own address space. In the typical case, a process executes an atomic test-and-set instruction to acquire a lock, finds it free, and thus is able to get it. Only in the (rare) case where the lock is already set does the process need to do a system call to wait for the lock to clear. When a process is finished with a lock, it can clear the lock itself. Only if the ‘‘WANT’’ flag for the lock has been set is it necessary for the process to do a system call to cause the other process(es) to be awakened.

Another issue that must be considered is portability. Some computers require access to special hardware to implement atomic interprocessor test-and-set. For such machines the setting and clearing of locks would all have to be done with system calls; applications could still use the same interface without change, though they would tend to run slowly.

The other issue of compatibility is with System V’s semaphore implementation. Since the System V interface has been in existence for several years, and applications have been built that depend on this interface, it is important that this interface also be available. Although the interface is based on system calls for both setting and clearing locks, the same interface can be obtained using our interface without system calls in most cases.

This implementation can be achieved as follows. System V allows entire sets of semaphores to be set concurrently. If any of the locks are unavailable, the process is put to sleep until they all become available. Under our paradigm, a single additional semaphore is defined that serializes access to the set of semaphores being simulated. Once obtained in the usual way, the set of semaphores can be inspected to see if the desired ones are available. If they are available, they are set, the guardian semaphore is released and the process proceeds. If one or more of the requested set is not available, the guardian semaphore is released and the process selects an unavailable semaphores for which to wait. On being reawakened, the whole selection process must be repeated.

In all the above examples, there appears to be a race condition. Between the time that the process finds that a semaphore is locked, and the time that it manages to call the system to sleep on the semaphore another process may unlock the semaphore and issue a wakeup call. Luckily the race can be avoided. The insight that is critical is that the process and the kernel agree on the physical byte of memory that is being used for the semaphore. The system call to put a process to sleep takes a pointer to the desired semaphore as its argument so that once inside the kernel, the kernel can repeat the test-and-set. If the lock has cleared (and possibly the wakeup issued) between the time that the process did the test-and-set and eventually got into the sleep request system call, then the kernel immediately resumes the process rather than putting it to sleep. Thus the only problem to solve is how the kernel interlocks between testing a semaphore and going to sleep; this problem has already been solved on existing systems.

3. References

[Babaoglu79] Babaoglu, O., and Joy, W., ‘‘Data Structures Added in the Berkeley Virtual Memory Extensions to the UNIX Operating System’’ Computer Systems Research Group, Dept of EECS, University of California, Berkeley, CA 94720, USA, November 1979.

[Someren84] Someren, J. van, ‘‘Paging in Berkeley UNIX’’, Laboratorium voor schakeltechniek en techneik v.d. informatieverwerkende machines, Codenummer 051560-44(1984)01, February 1984.

4. Appendix A − Virtual Memory Interface

4.1. Mapping pages

The system supports sharing of data between processes by allowing pages to be mapped into memory. These mapped pages may be *shared* with other processes or *private* to the process. Protection and sharing options are defined in *<sys/mman.h>* as:

```
/* protections are chosen from these bits, or-ed together */
#define  PROT_READ           0x04    /* pages can be read */
#define  PROT_WRITE          0x02    /* pages can be written */
#define  PROT_EXEC           0x01    /* pages can be executed */

/* flags contain mapping type, sharing type and options */
/* mapping type; choose one */
#define MAP_FILE             0x0001  /* mapped from a file or device */
#define MAP_ANON             0x0002  /* allocated from memory, swap space */
#define MAP_TYPE             0x000f  /* mask for type field */
```

```
/* sharing types; choose one */
#define  MAP_SHARED            0x0010  /* share changes */
#define  MAP_PRIVATE           0x0000  /* changes are private */

/* other flags */
#define MAP_FIXED              0x0020  /* map addr must be exactly as requested */
#define MAP_INHERIT            0x0040  /* region is retained after exec */
#define MAP_HASSEMAPHORE       0x0080  /* region may contain semaphores */
```

The cpu-dependent size of a page is returned by the *getpagesize* system call:

```
pagesize = getpagesize();
result int pagesize;
```

The call:

```
maddr = mmap(addr, len, prot, flags, fd, pos);
result caddr_t maddr; caddr_t addr; int *len, prot, flags, fd; off_t pos;
```

causes the pages starting at *addr* and continuing for at most *len* bytes to be mapped from the object represented by descriptor *fd*, starting at byte offset *pos*. The starting address of the region is returned; for the convenience of the system, it may differ from that supplied unless the MAP_FIXED flag is given, in which case the exact address will be used or the call will fail. The actual amount mapped is returned in *len*. The *addr*, *len*, and *pos* parameters must all be multiples of the pagesize. A successful *mmap* will delete any previous mapping in the allocated address range. The parameter *prot* specifies the accessibility of the mapped pages. The parameter *flags* specifies the type of object to be mapped, mapping options, and whether modifications made to this mapped copy of the page are to be kept *private*, or are to be *shared* with other references. Possible types include MAP_FILE, mapping a regular file or character-special device memory, and MAP_ANON, which maps memory not associated with any specific file. The file descriptor used for creating MAP_ANON regions is used only for naming, and may be given as −1 if no name is associated with the region.† The MAP_INHERIT flag allows a region to be inherited after an *exec*. The MAP_HASSEMAPHORE flag allows special handling for regions that may contain semaphores.

A facility is provided to synchronize a mapped region with the file it maps; the call

```
msync(addr, len);
caddr_t addr; int len;
```

writes any modified pages back to the filesystem and updates the file modification time. If *len* is 0, all modified pages within the region containing *addr* will be flushed; if *len* is non-zero, only the pages containing *addr* and *len* succeeding locations will be examined. Any required synchronization of memory caches will also take place at this time. Filesystem operations on a file that is mapped for shared modifications are unpredictable except after an *msync*.

A mapping can be removed by the call

```
munmap(addr, len);
caddr_t addr; int len;
```

This call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references.

4.2. Page protection control

A process can control the protection of pages using the call

---

† The current design does not allow a process to specify the location of swap space. In the future we may define an additional mapping type, MAP_SWAP, in which the file descriptor argument specifies a file or device to which swapping should be done.

```
mprotect(addr, len, prot);
caddr_t addr; int len, prot;
```

This call changes the specified pages to have protection *prot*. Not all implementations will guarantee protection on a page basis; the granularity of protection changes may be as large as an entire region.

4.3. Giving and getting advice

A process that has knowledge of its memory behavior may use the *madvise* call:

```
madvise(addr, len, behav);
caddr_t addr; int len, behav;
```

Behav describes expected behavior, as given in <*sys/mman.h*>:

```
#define MADV_NORMAL      0   /* no further special treatment */
#define MADV_RANDOM      1   /* expect random page references */
#define MADV_SEQUENTIAL  2   /* expect sequential references */
#define MADV_WILLNEED    3   /* will need these pages */
#define MADV_DONTNEED    4   /* don't need these pages */
#define MADV_SPACEAVAIL  5   /* insure that resources are reserved */
```

Finally, a process may obtain information about whether pages are core resident by using the call

```
mincore(addr, len, vec)
caddr_t addr; int len; result char *vec;
```

Here the current core residency of the pages is returned in the character array *vec*, with a value of 1 meaning that the page is in-core.

4.4. Synchronization primitives

Primitives are provided for synchronization using semaphores in shared memory. Semaphores must lie within a MAP_SHARED region with at least modes PROT_READ and PROT_WRITE. The MAP_HASSEMAPHORE flag must have been specified when the region was created. To acquire a lock a process calls:

```
value = mset(sem, wait)
result int value; semaphore *sem; int wait;
```

Mset indivisibly tests and sets the semaphore *sem*. If the previous value is zero, the process has acquired the lock and *mset* returns true immediately. Otherwise, if the *wait* flag is zero, failure is returned. If *wait* is true and the previous value is non-zero, *mset* relinquishes the processor until notified that it should retry.

To release a lock a process calls:

```
mclear(sem)
semaphore *sem;
```

Mclear indivisibly tests and clears the semaphore *sem*. If the ‘‘WANT’’ flag is zero in the previous value, *mclear* returns immediately. If the ‘‘WANT’’ flag is non-zero in the previous value, *mclear* arranges for waiting processes to retry before returning.

Two routines provide services analogous to the kernel *sleep* and *wakeup* functions interpreted in the domain of shared memory. A process may relinquish the processor by calling *msleep* with a set semaphore:

```
msleep(sem)
semaphore *sem;
```

If the semaphore is still set when it is checked by the kernel, the process will be put in a sleeping state until some other process issues an *mwakeup* for the same semaphore within the region using the call:

```
mwakeup(sem)
semaphore *sem;
```

An *mwakeup* may awaken all sleepers on the semaphore, or may awaken only the next sleeper on a queue.