Why Aren't You Using mmap() Yet?

Kevin Sheehan
Uniq Professional Services
PO Box 70
Paddington, NSW, 2021 Aust.
kevin@uniq.com.au

1.0 Abstract

Since the inclusion of the Virtual Memory (VM) facilities in SunOS 4.0, the mmap() system call has provided a powerful and very efficient alternative to read() and write() to access regular files. Unfortunately, due to somewhat abstract and obscure documentation, developers have not been making great use of the interface.

With in inclusion of the VM facilities in SVR4, it is time for developers to start using mmap() more often as the preferred interface. It is no longer a Sun-specific interface, and the benefits demonstrated on the Sun platform will almost certainly apply to applications used on other platforms.

Mmap() provides many benefits. Ease of use is certainly one, but the major benefit of using mmap() is increased performance. This is partly due to the more efficient use of system resources, partly due to the fact the mmap() allows the application developer more control over use of resources, and also that the system can be given more information in how to deal with those resources.

This paper is also intended to become an application note supplementing the obscure documentation that exists. In that spirit, this paper and the examples will be made freely available. It is the author's hope that others will supply examples and comments to ease the transition for those not yet fortunate enough to be using mmap().

2.0 A Brief Look at VM

2.1 Paging

In order to facilitate the concept of virtual memory and demand paging, the address space of each process is divided up into a number of fixed size pages. Each process is assigned a particular memory management unit (MMU) context in which to run. The job of the MMU is to translate the references of virtual addresses to physical addresses of one kind or another.

In order to have a process address space larger than physical memory, the concept of virtual memory was implemented. In this scheme, not all of the pages a process is using need be mapped at one time. In order to get access to a page that is not mapped, the process references the unmapped page, and takes a “page fault”. This is a trap that tells the kernel that an unmapped page has been referenced. The kernel responds by checking to see that it is a valid reference, and arranging for the proper contents to be mapped. The process is then transparently restarted, seeing nothing from the fault to the restart, hence the term virtual memory.

In short, the process address space is treated as an array of pages, each of which is associated with some physical page. In the next two sections, we will discuss how this was done in pre-VM systems, and how it is now done under VM.

A topic that is well discussed in *4.3 BSD Unix [1]* is fetch, placement, and replacement policies. Fetch policy is what determines when a page is loaded, placement policy determines where you place a page, and replacement policy determines when you select a page to be replaced by another (when do I throw it out?).

Pure demand paged systems typically fetch on demand only (when a page is referenced), and replacement is achieved by a Least Recently Used caching scheme that uses the working set as an approximation to the optimal set of pages for each process. These policies arise because the system generally has no a priori information about page usage. We’ll discuss the facilities that VM provides to alleviate this shortcoming, but the author highly recommends the above book for further reading by those seriously interested in performance issues.

2.2 Classic Unix Memory Usage

In pre-VM UNIX, physical memory was divided into two pools, one for the text and data pages, and another (called the buffer pool or buffer cache) for file I/O. The reasoning behind the buffer cache was that disks are much slower than memory, and keeping recent file blocks in core meant much less disk I/O. This meant that processes doing read() and write() calls to regular files had to deal with a (necessary) memory copy, but for blocks used by more than one process, or used more than once by a process, no I/O was done. The only type of filesystem that UNIX had to deal with was UFS, and the buffer cache was used for “block” I/O devices, which typically meant disks.

For a demand paged process (the only kind we consider here) running the process caused the systems to 1) allocate space on the swap/paging device (in a virtual memory system, you have to put the page someplace when it is not in memory) 2) copy the appropriate pages to the paging device (in the case of “pure” text programs, the text pages may have already been on the device and another copy is not required) and 3) loading some subset of those pages and starting execution of the process. As time goes on, the process will fault in pages it needs, and pages that are not referenced will age and be replaced (written out to swap if dirty and made available) if the system needs physical memory.

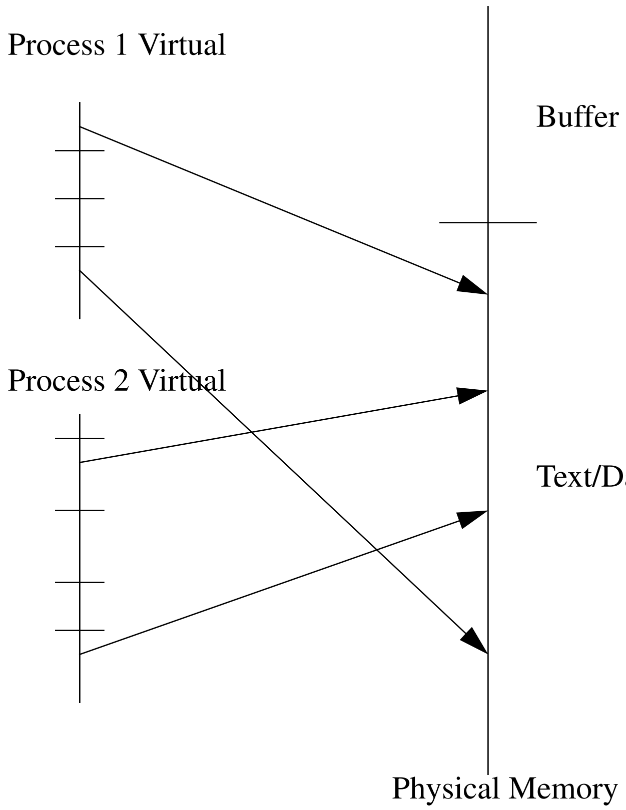


FIGURE 1. Physical Memory Usage in Pre-VM UNIX

Great, we have virtual memory, a demand paging system that does a pretty good job of figuring out which pages (on average) are needed, we’re caching disk blocks to minimize I/O done to the disks. Can we do better?

As UNIX grew from a time sharing system to meet the needs of a diverse number of platforms, the pre-VM facilities for mapping and management of the physical memory pool no longer proved sufficient.

Perhaps the biggest problem from a performance point of view was the static boundary between the buffer cache and the text/data pages. For a system doing large amounts of file I/O, the buffer cache is too small, and the text/data pages go to waste. For a system that needs many pages for text/data, the memory allocated to the buffer cache is not available to ease the load. More importantly, memory was being managed by two separate and non-cooperating sub-systems.

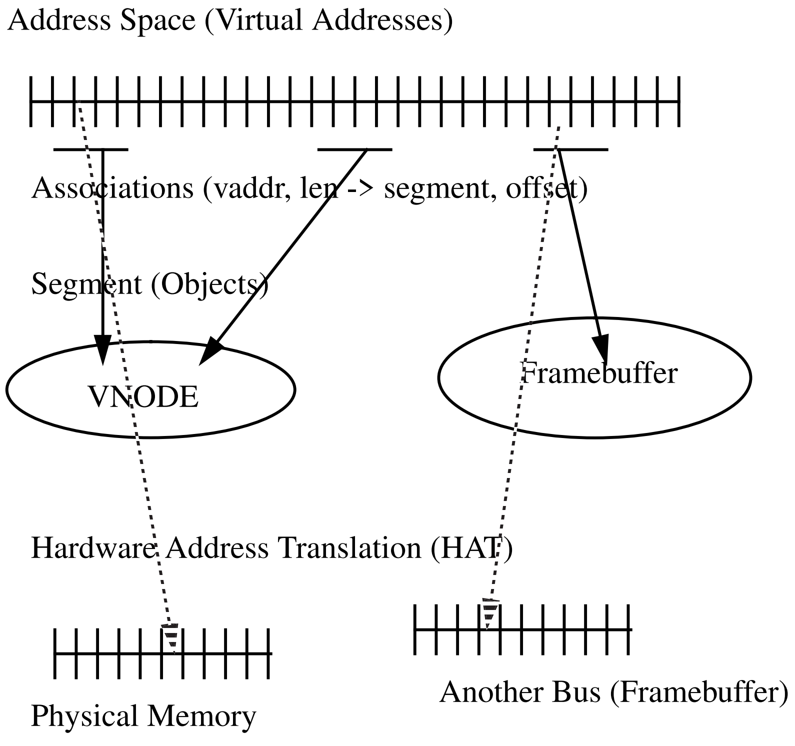
The requirements of mapping devices such as frame buffers were not well provided for - in order to map that memory, an allocation of virtual address space (backed by the swap device) was required, followed by the replacement of that mapping by the physical mapping to the device.

No method of interposing some action on the part of various subsystems in response to a page fault was provided, which meant that little or no management of the objects being mapped was really possible.

But perhaps the most telling argument for the VM rewrite was that the management of the mappings was heavily machine dependent. When the only model you have to worry about is the VAX or Sun2/3, that is one problem. When you have to start worrying about multiple architectures, caches, I/O caches and the like as well as the problem of many consumers of both virtual and physical memory, unifying and abstracting your management of virtual and physical memory resources becomes imperative.

2.3 VM Model

Essentially, VM is an abstraction of the three levels needed to manage the mapping from a process' virtual address space to a physical page. The three layers are the Address Space (AS) Layer, the Segment (SEG) Layer, and the Hardware Address Translation (HAT) Layer.



The Address Space layer is responsible for managing the allocation of the process virtual space, and keeping track of the associations between the virtual addresses and the objects underlying them. Each mapping contains a virtual address, a length, the segment involved in the mapping, and an offset within that segment.

The segment layer is where the work of managing the objects is done. Examples of segment types are vnodes (the abstraction of files used in the Virtual File System), generic device drivers (such as frame buffers), and devices which require management of the context associated with each mapping, such as graphics accelerators.

The Hardware Address Translation layer is where the machine dependent work of loading and unloading mappings is done. Allocation of MMU resources, cache consistency and the like happens here.

What does this buy us? Instead of having a page fault routine that has to know everything about the possible ways to fill in a page, the problem is now factored into nice neat chunks. The address space operations allocate and free associations with various types of segment, and call the proper segment ops when a page fault is encountered. The segment ops make sure the proper page is prepared and calls the HAT layer to load (or unload) a translation when appropriate. The HAT layer handles all the grotty details of getting the page mapped and consistent.

So instead of having to rewrite the page fault handler to extend mapping facilities, all you have to do is provide a new segment driver. Instead of encapsulating knowledge of the mapping hardware throughout the OS, the HAT layer handles it. With the abstraction of the Virtual File System, only one driver was needed for “regular” files, the segment driver for vnodes

Simply put, mmap() is the interface that creates the associations between the process address space and various objects in the system. With the factoring of functionality provided by VFS and VM, the range of objects available for mapping is considerably extended. In fact, mmap() is now used extensively by the system for management of resources in the lifetime of a process.

In addition to the familiar read, write and execute permissions that mmap() allows you to associate with a page, there is the concept of shared and private mappings. In a shared mapping, *all* mappings of a particular object are shared. In other words, if two different processes both map the same page of an object, they will have the same physical page mapped in for both address spaces at some point. In a private mapping, if the page is marked writable, the shared copy is mapped in read-only. On the first write reference to the page, an anonymous copy is made for that address space to use privately. This is also known as a Copy On Write (COW) mapping.

Since you are now able to explicitly manage your address space with mmap(), it is now possible to give the system more information about your usage of a particular mapping with the madvise() system call. In a classic demand paged system, the kernel knows nothing about your pattern of access, and has no a priori information about which pages you will or wont need at any given time. Madvise provides for three patterns of access to modify the fetch and replacement policies, explicit notification of whether a range of pages is needed or not, the ability to flush blocks back to secondary storage synchronously or asynchronously, and the facility to invalidate a mapping (cause the pages to be re-read on next fault).

The three access patterns are normal, sequential and random. The normal access pattern is to do read ahead, i.e. to read page N+1 when a page fault for page N is received. This presumes that page N+1 will be needed soon, and that page N-1 (if it is valid) may be needed and the resources should not be freed. Sequential access does read ahead as well, but to also do free behind. This assumes that page N+1 will be needed, but that page N-1 will not, and that the resources from page N-1 will be needed soon. Random access presumes nothing, i.e. it only fetches the page required by the fault, and allows the LRU algorithm to age pages as appropriate.

For cases where more explicit control is desired, madvise provides WILLNEED and WONTNEED advice. You can tell the system that a certain range of pages will be needed, and it will attempt to make sure they are available. When you no longer need them, and want the system to start making them re-usable, you simply tell it.

Msync() also allows you to explicitly flush pages either asynchronously or synchronously if you need to guarantee consistency without affecting the mapping. It also allows you to invalidate the mapping without undoing it if you with to reload the pages on the next reference.

Combined, these facilities offer the application developer a set of powerful tools to control the use of system resources. Since everything is handled by a page fault (or the simulation of a page fault) the various segment operations that need physical memory compete on an equal basis, and system resources can in theory be optimally allocated.

3.0 Examples and Benefits

Let's look at some examples of how the system and various applications use mmap(), and the benefits over the older techniques used.

3.1 Executables and Shared Libraries/Objects

As we saw in pre-VM UNIX, executables were loaded onto the paging device and paged in from there. There existed the possibility of sharing pure text, but all else was copied over. In particular, all executables had to be statically linked so they could be copied over, as no provision for mapping for dynamic linking existed.

In post VM UNIX, the vnode associated with the executable is simply mmap()ed Copy on Write, and the uninitialized data is mapped COW to /dev/zero. Let's see what this means for the lifetime of an executable called foo.

When you exec() foo, the system does a lot of what it used to - find foo. The first difference is that it will be dealing with a vnode, and not a collection of blocks in a particular flavor of filesystem. A small run time loader does two mmap() calls. The first creates a private mapping for the text and initialized data in the executable. The second creates a private mapping for the bss (uninitialized data) area to /dev/zero (/dev/zero is a device that returns zeros when read or mapped, as opposed to /dev/null, which returns nothing).

At this point no I/O (other than the page needed to get the exec header) has been done, so we have our first advantage - we don't need to copy the text or data to the paging area on start-up. All we have is an association and enough of the process actually in core to start it running.

Let's assume that it doesn't write to the text pages, but just faults them in as a result of fetches. In other words, when we touch a new page in text, we take a page fault, which calls the segment ops for vnodes, which allocates a page, schedules I/O to fill the page from the vnode, calls the HAT layer to load a translation, and then restarts our process.

The advantage here is that the text need never appear on the paging device, the pages can always be fetched from the vnode.

What happens when we reference the data or bss pages? If it is a read reference, the same thing happens as for text. In other words, if we have a lot of read-only data, we can read that out of the vnode as well, and don't need to have it on the paging device either. If it is a write reference, then a private copy is made. We take a page fault (either because it is unmapped, or because it is mapped read-only at this point) and the system notices that it is a write reference. The segment ops allocate another page, and copy the contents of the original page to it. This page is then "anonymous" i.e. only we can ask for it, and it is now backed by the paging device, not the vnode.

Well, not a huge advantage for written pages, but at least the I/O is done when we need it, and the previous savings apply.

What happens when we run another copy of foo? The actions are the same, but the effects differ because some of the pages are already in core, read-only and sharable. When the vnode segment ops resolve the page fault for the text pages (or read-only data pages) they will find the pages already mapped in for foo1.

Now we can see one of the major advantages of the VM approach - data will be shared by different processes whenever possible. We don't have to keep a sticky bit and clog up the paging device, we just re-map a page when we can. The segment ops look up a page in physical memory by vnode and offset (where appropriate) and simply map it into another process address space.

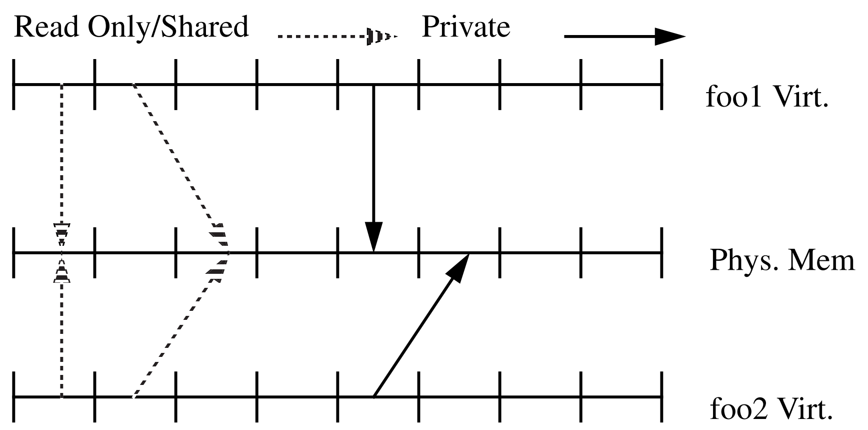


FIGURE 2. Two processes sharing pages.

If we can share pages in an executable, can we find a way to share things that every process in the system is likely to use? Yes, if we can map one executable into a process space and share portions, then it is a straightforward step to map another executable into the same address space. This is exactly the mechanism that SunOS 4.x and SVR4 use to implement shared libraries and shared objects.

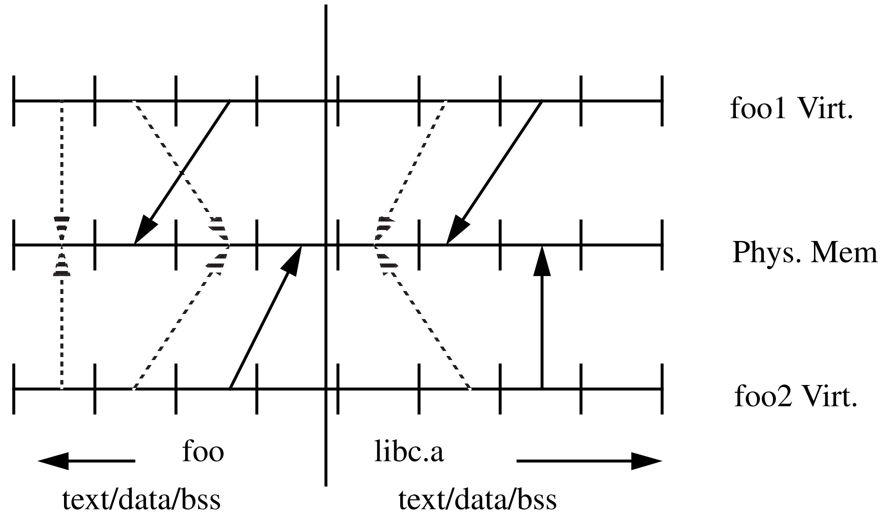


FIGURE 3. Two processes sharing pages and multiple text/data/bss mappings.

In this figure you can see that a process address space can be made up of multiple text, data, and bss segments. Instead of one each all lumped together, dynamic linking allows libraries and objects to be mapped and resolved as needed. More importantly, it allows common pages like library text to be shared among all processes, significantly reducing the amount of required memory resources.

Another huge advantage to dynamic linking and shared objects is that since the object is not linked statically into the executable, you can change it without relinking. In the case of libraries, it means developers can release and distribute portions of the product instead of doing full releases.

As an example, picture a C++ class implemented as a shared object. If you update the implementation of the class, but do not change the class definition, you only need to release the new class implementation, not your whole application.

3.2 Reading Files

Now that we have a rough idea of how VM works, and how it benefits the system, let's take a look at how the facilities can be used by the application developer. The simplest example is reading a file. We'll use a small read by one process to see how read() differs from mapping a file, then look at the benefits for large reads, and more than one process using a file.

While the system does not have a pre-allocated portion of physical memory in which to store filesystem I/O, it uses a similar technique as before to perform reads to user buffers. Assume that we are reading the first 10 bytes from a file. The system first simulates a page fault, which allocates a page and does the disk I/O in order to get the required disk blocks into memory., then makes sure that the portion of your process space that contains the buffer is in memory, then it copies the requested bytes into that memory. The resources consumed are two pages of memory (one of which is now dirty and may need to be written out), the space on the paging device to back the private page for the process, an I/O operation to get the filesystem block into memory., and a copy to move the info from one page to the other. We haven't even looked at the data yet...

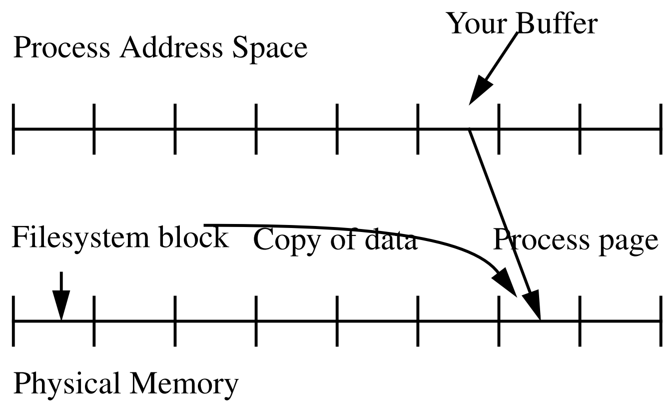


FIGURE 4. read()

Now let's use mmap() to get to the same 10 bytes. When we call mmap(), no I/O or allocation is done, all that happens is a record is made at the Address Space level of the association between a virtual address (which the system typically picks) and the vnode and offset (which in this case is 0). When we reference those 10 bytes for the first time, we generate a page fault, the same one that was simulated in the read() example. The system allocates a page, does the I/O, but instead of copying the data to our buffer, it maps the block used directly to our process address space. The resources consumed are one page of memory (and not dirty, so no further I/O is necessary) and one I/O operation. Half as much memory, half as much disk, and no copying.

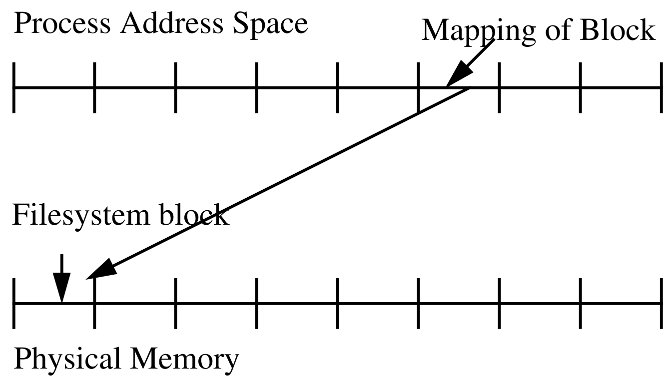


FIGURE 5. mmap()

Now let's take a look at what happens for a larger file. Let's assume that we are looking at a 2 MB file. When the read() is issued, the system has to allocate the pages for the file systems blocks, and do the I/O, then allocate pages for the private copy of the file, and copy the pages over. 2 MB of (in this example) useless pages, 2 MB of dirty pages, a big chunk of I/O, and we haven't even started yet. In this case the dirty pages in the process address space are likely to be written back to disk, creating further I/O.

For mmap(), the association is made, and no I/O is done. As each page is referenced, a page fault is taken, a page is allocated, and the required I/O is done when needed (actually, in the normal case, read-ahead would halve the fault rate). Instead of a glut of I/O, and twice as much memory and disk consumed, we have used exactly what was needed when it was needed.

As a further example, image the case of a file half the size of physical memory or larger. For read(), by the time we are done we have used twice as much memory as needed, so the pages we probably want at the beginning are guaranteed to be out of memory, and will have to be read in again. For mmap(), references to the page we want cause it fault into memory as we need it. Consider a huge file - mmap() doesn't need the backing of swap for the private copy, so as long as the file exists, and we have enough address space, the file can be mapped. For read(), we need as much swap as there is file, and this only to read the file!

Now add multiple processes all using the file. For read(), we have each process needing a copy of the private pages, all the I/O done for the pages up front (to be fair, if the page is already in memory from another page fault, just the copy is done) memory to memory copies done for each of the processes, and swap allocated to cover all those pages. In other words, for N processes, we increase the system resource usage 2*N times.

For mmap(), since the mapping looks directly for the page allocated by the segment layer, the pages are shared once the I/O is done. If every process is using the same pages at the same time, there is only one copy in memory, and no swap backing it. In other words, the worst case for mmap() is 1 copy on disk (the original) and 1 copy in memory.

The cases for programs that read a file, modify it, and then write it back are even more striking. In addition to the work done to read the file, the whole file has to be doubly paged again to write it back as well! For mmap(), the original pages can be modified, resulting in exactly the amount of work that needs to be done on the pages.

The write only case is closer to the read only case - one copy of the file is in core, one copy of the file takes up disk space.

Last but not least, the user has no control over what the system does with the pages in their address space. With an mmap()ed file, you can use the facilities of madvise() and msync() to manage the resources, and give the system all the information it needs to manage optimally on your behalf.

Less filling, tastes great - why aren't you using mmap() yet?

3.3 Cp and Cat

A good example of mmap() used to improve a simple program is the method cat and cp now use for input. If a file can be mmap()ed, both these programs will do so. What this means is that no user pages need be dirtied in order to do the I/O. When the mmap() is done, only the association is made for the process. When the write() call is issued to the output file, the kernel allocates the necessary pages to read the file in, and then does the write directly from there, halving the memory usage, eliminating an unecessary copy of the data to user space and the use of swap backed private memory.

In the trivial case of "cp X /dev/null" or "cat foo >/dev/null", no I/O is done at all. The association is made (no I/O is performed) and then the write to /dev/null does nothing (in particular, it does not request the pages to be read in). While some consider this cheating, I'd submit that it is simply doing what it has been told to do - nothing.

4.0 Examples

4.1 map_file() and make_file()

The steps needed to map a file are 1) open the file 2) find the length of the file 3) mmap() the file and 4) close the file descriptor. There is no need to keep an open file descriptor for a mapping, which is very convenient for applications using many files.

In these two examples, we assume that the file is smaller than the address space, and map the whole file for use. For most applications, this is a reasonable assumption. For applications such as cp and cat, a more general method of mapping a chunk at a time is necessary.

```
struct mapping {
        void *addr;
        unsigned long len;
};
void *
map_file(mapping, name, permit)
        struct mapping *mapping;
        char *name;
        int permit;
{
        struct stat s;
        void *p = 0;
        int fd, prot;
        u_long len;

        if ((fd = open(name, permit)) < 0) {
                fprintf(stderr, “open of %s failed (%s)\n”, name, sys_errlist[errno]);
                return (p);
        } else if (fstat(fd, &s)) {
                fprintf(stderr, “stat of %s failed (%s)\n”, name, sys_errlist[errno]);
        } else {
                if (permit == O_RDWR) prot = PROT_READ | PROT_WRITE;
                else if (permit == O_WRONLY) prot = PROT_WRITE;
                else prot = PROT_READ;
                if ((p =
                 (void *) mmap(0, len, prot, MAP_SHARED, fd, 0)) == (void *) -1) {
                        fprintf(stderr,
                                “mmap of %s failed (%s)\n”, name, sys_errlist[errno]);
                        p = 0;
                }
        }
        close(fd);
        if (mapping) {
                mapping->addr = p;
                mapping->len = s.st_size;
```

```
        }
        return (p);
}
```

The mapping structure can be used to record the length of the file. If a null pointer is passed in for the structure, the routine just hands back the address of the mapping. If no mapping is performed, then a null pointer is returned. Map_file() uses the file permissions from the open() system call declarations - this was an aid to the original conversion for which this routine was written, it should probably use the flags that mmap() uses for specifying the protection desired. You will also note that this routine uses void * as a generic pointer, where mmap() is declared as returning char * (caddr_t). Another routine caught in the ANSI C transition.

It is not difficult to create a file using mmap(), as the following (very similar) routine shows. It is important to note that no allocation of disk space is done until a given page is written, so there is no penalty for creating a huge file to map, and then making it smaller with truncate() once the final size is known.

```
void *
make_file(name, len)
        char *name;
        int len;
{
        void *bits = 0;
        int fd;

        if ((fd = open(name, O_RDWR | O_CREAT, 0666)) < 0) {
                fprintf(stderr, “make_file failed for %s (%s)\n”,
                        name, sys_errlist[errno]);
        } else if (ftruncate(fd, len)) {
                fprintf(stderr, “sizing failed for %s (%s)\n”,
                        name, sys_errlist[errno]);
        } else {
                len = (len + getpagesize() - 1) & ~(getpagesize() - 1);
                if ((bits =
                 (void *) mmap(0, len, PROT_READ | PROT_WRITE,
                                MAP_SHARED, fd, 0)) == (void *) -1) {
                        fprintf(stderr, “mmap failed for %s (%s)\n”,
                                name, sys_errlist[errno]);
                        bits = 0;
                }
        }
        close(fd);
        return (bits);
}
```

The size of the mapping is rounded up to the machine page size, but this doe *not* change the size of the file. If a page fault occurs at the end, and the offset is not beyond the real size of the file, the page fault succeeds, and a whole page with the end portion is mapped. If a page fault sees an access beyond the real end, a segmentation violation is signalled.

4.2 Simple cp

A simple copy program using the above routines would then be something like this:

```
main(int argc, char **argv)
{
        int fd;
        struct mapping in;

        /* check arguments */

        if ((fd = open(argv[2], O_WRONLY, 0666)) < 0) {
                /* error processing for write side */
        } else if (!map_file(&in, argv[2], O_RDONLY)) {
                /* error processing for read side */
        } else if (write(fd, m.addr, m.len) != m.len) {
                /* error processing for copy */
        }
}
```

In the above example, the pages will not be read in until the write() call asks that the pages be read in as part of I/O. More importantly, no user pages are dirtied. Also, all of the I/O will be done in the kernel, allowing it to choose the optimal size chunk, instead of making the application decide on a buffer size.

4.3 Accounting Summary

As another example of an application reading a file more efficiently, let's look at a quick accounting summary. The accounting file (typically /var/adm/pacct) is organized as a series of 32 byte fixed length records as defined in /usr/include/sys/acct.h. For our example, the sum of all characters transferred will be computed. ctoi() is a routine that converts between the comp_t format used by accounting to an integer.

```
main(int argc, char **argv)
{
        struct mapping af;
        struct acct *ap;
        u_long          nap, nchars = 0;
        char *filename = (argc > 1) ? argv[1] : "/var/adm/pacct";

        if (!!map_file(&af, filename, O_RDONLY)) {
                /* error processing */
                exit(1);
        }
        ap = (struct acct *)af.addr;
        nap = af.len/sizeof(*ap);
        while(nap--) {
                nchars += ctoi(ap->ac_io);
                ap++;
        }
        printf("%d\n", nchars);
        exit(0);
}
```

The equivalent example using read() and fstat() would read the records one at a time into a private copy or read the entire file into an allocated array of acct structs. The advantage of mapping the file in the first case is that no read() calls are done - on a machine with a 4K pagesize, this eliminates 128 system calls and copies per page. In the second case, as we have seen before, we eliminate reading the entire file at once and the swap backed private pages used to hold our copy.

4.4 Read Modify Write

We've seen the advantages for simple reader programs, let's look at an application that does an in-place modification of data. In this example, lets assume we have an 8-bit pseudo-color Sun rasterfile with a ramp in gray, and we want to make it a bit brighter We will assume that the rasterfile header correctly describes the size of the file. In practice, an application should check this assumption, otherwise accesses to invalid parts of the mapping could create segmentation violations.

```
main(int argc, char **argv)
{
        u_char *p;                      /* pointer to pixels */
        u_char pixel;                   /* pixel value */
        struct rasterfile *rp;          /* pointer to raster header */
        u_long                  size;

        /* check arguments */

        if (!(p = (u_char *) map_file( (struct mapping *)0, argv[1], O_RDWR))) {
                /* error processing */
        }
        rp = (struct rasterfile *)p;    /* pointer to raster header */
        /* check raster header */
        p += ((sizeof(*rp) + rp->ras_maplength);            /* skip header and cmap */
        size = rp->ras_width * rp->ras_height;
        while(size--) {
                pixel = *p;
                ii (pixel != 255) *p++ = ++pixel;
                else p++;
        }
        exit(0);
}
```

Fortunately, this is not a paper on C programming or image enhancment :-) Let's look at the benefits of an in-place mapping for applications that do read-modify-write. Using the normal malloc()/read()/process/write() model, we would have to read the whole file into a private copy. This would dirty our swap backed private pages and cause all of the file I/O to be done at once. We then have to manipulate that copy, and write it back, causing all of our private pages and all of the file pages to be loaded one more time. Two memory copies, use of a swap copy, and two complete runs thru the file.

For the mapped version, we access the pages actually used by the file, so no private copy is required, the system is far more likely to be able to keep the pages in memory, and when we are done, the actual pages that comprise the file have already been modified, so no write is necessary. The pages will age as appropriate and be written when necessary.

4.5 Large and Huge Files

Presuming that the previous example was a typical 1152x900 rasterfile, we were only dealing with approximately 1MB worth of data. What if the size of the file approaches the size of physical memory? What if it exceeds it? What if it exceeds the size of available swap space?

As already noted, using read() and write() presumes that the file pages will be read in and copied to or from a private copy. When the size of the desired manipulation starts approaching half of physical memory, read()/write() applications will start to "turn the bend” in performance, because the limiting factor will be disk I/O as the system will be bound by the paging rate - and this happens before you access the data. For a mapped application, the bend is at worst about the size of physical memory, and depending on how fast an application moves thru the pages, it may never become paging limited.

As the size of the chunk goes beyond physical memory, the read()/write() application still has to thrash thru the whole file before accessing the first byte, where the mapped application is still in the same situation as before - pages are drawn in as needed, and with use of madvise(), the application can help the system do exactly the right thing.

As the size of the chunk approaches the size of the swap device, the read()/write() application has to break up the problem into smaller chunks, and it is using a somewhat precious system resource (and presumably I/O bandwidth) to do so. The mapped application needs no backing from swap, so the first limit it runs into is the size of the virtual address space.

It was in fact the manipulation of large image files that first led the author to use mmap(). The benefits in performance for 1MB files is noticeable. When you are dealing with 225MB files, the path quickly becomes obvious.

4.6 XImages

A brief note here - most raster display formats can be described by XImage with the right settings. It is a big performance win (especially in systems like Sun that have a Direct X interface) to mmap() the image data, and use a pointer to the mapping in the XImage structure instead of reading a private copy. A recent animation example showed a 6X performance enhancment using mmap() to access all the data, and just switching the pointer instead of reading different files all the time...

4.7 Shared Data as Memory

The benefits of sharing pages being used by many cooperating processes has been discussed earlier in this paper. An interesting application for mmap() is that of using a commonly mapped file as shared memory.In addition to the simpler interface and the benefits of being able to rendezvous using the file system namespace, the model has some less obvious benefits as well when file locking is used.

As an example, consider a print spooler that has to process files in an input queue, and then move the processed files to an output queue to be processed by an output manager. RPC calls could be used to move the files about, but in a resilient system, the information must be kept around in order to restart the jobs.

The input queue would have a header describing the number of entries, the next free slot in the table, and the output queue would have the same. Manipulation of the queue entries can be accomplished by a common set of routines. Before updating an entry, the routines would acquire a write lock using the flock() system call (this allows a portion of the file to be locked) in order to make the access atomic. When information from the file is read, a read lock is acquired. If another process is in the midst of writing or reading that portion of the file, then competing process will block. If non-exclusive access (such as two reads) is attempted, both processes will succeed.

The processes share the data (which can be extensive), there is effectively no limit of the size of the shared memory that can be used, and the files can use advisory locking to guarantee atomic access to the file contents. The programming model is that of shared memory, instead of having to read() and write() entries as well as performing locking.

In the author's experience, mapping files as shared data is much easier, much more extensible in size, and more efficient (when you consider the control that flock() and madvise() give you over the file) than using shared memory and/or semaphores.

4.8 Shared Objects

We've discussed the implementation of shared libraries and objects before, but it is a small step from using shared data with mmap() to using shared objects as well. The point to be made here is that with the benefits of access methods and data hiding provided by languages such as C++, classes can use mmap() in their implementation for performance, and the application programmer need never know.

5.0 Caveats

While VM and the facilities it provides have a number of advantages, there are some caveats to consider.

The first is that mmap() won't work on stream based files such as sockets and pipes, so applications that have to deal with these will have to have a separate access method. It is worth considering re-writing the application to use regular files. As an example, image processing systems that use a series of filters to effect transformations might better be implemented as routines that mmap() (possibly temporary) files in order to eliminate the private pages and better control use of memory. In some cases, it just isn't possible.

The second is the use of NFS files for mapping. NFS files will mmap(), but only if file locking (using lockf() or fcntl()) is *not* being used. The typical way to get around this is to use a secondary file for the locking, and mmap() the original. Use of msync() to guarantee that you have a coherent copy in conjuction with ancillary locking works, but is obviously not as easy as mapping UFS files locally. It is also worth noting that network locks are slower than flock(), and have been known to have the odd problem.

The last caveat is that you are using a new method of access for files. You will make mistakes, and there will be some differences in the way the system behaves. You'll notice update more often (consider turning it off for large numbers of written pages running a long time), you'll get core dumps with addresses in strange places, you'll forget the offset parameter, you'll forget to use both PROT_READ and PROT_WRITE, you'll forget to open the file with the right permissions. Be patient, it's worth it.

6.0 Summary

VM and the mmap() and madvise() facilities are generally much more efficient and allow you to give the paging system explicit information on how to deal with the use of memory for applications. It is present in SVR4, which means it can no longer be considered a SunOS specific interface.

Examples of all the applications mentioned above are available, as well as electronic copies of this paper. The documentation for mmap() has never been extensive, but it is the hope of the author that this paper will be the start of a communal application note, and that the examples made available will come back to form a good toolkit for those seeking to move quickly to mmap().

With all the advantages, I can only ask: *Why aren't you using mmap() yet?*

7.0 References

Leffler, McKusick, Karels, Quarterman, The Design and Implementation of the 4.3BSD Operating System, Addison Wesley, ISBN 0-201-06196-1 (1989)

Gingell, Moran, Shannon, Virtual Memory Architecture in SunOS, USENIX Association Conference Proceedings, pp 81-94, (1987)