1988

Efficient Interprocess Communication Using Shared Memory

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Steven B. Munson

Report Number:
88-744

EFFICIENT INTERPROCESS COMMUNICATION USING SHARED MEMORY

Douglas E. Comer
Steven B. Munson

CSD-TR-744
February 1988

Efficient Interprocess Communication Using Shared Memory

Douglas E. Comer
Steven B. Munson

Computer Science Department
Purdue University
West Lafayette, IN 47907

CSD-TR-744
January 1988

Abstract

Multitasking operating systems allow multiple application programs to execute concurrently without interference. Each task generates addresses starting at zero and the operating system uses memory management hardware to map each task's addresses into a unique part of physical memory. The memory management unit maps all the addresses a task generates into an area of physical memory reserved for that task; one task cannot accidentally read or alter the memory assigned to another task. Only the operating system itself can access unmapped physical memory. Isolation of tasks makes it easy to write or debug a given application program but it makes sharing memory among multiple tasks impossible. Thus, when tasks need to communicate, one task must ask the operating system to copy the data from its address space to another task's address space.

This article reports the results of a research project that is exploring how to support memory sharing among multiple tasks. The scheme described here uses conventional memory management hardware in a new way. It has all the advantages of task isolation, but allows cooperating sets of tasks to share parts of their address spaces. The chief advantage of our scheme is that it permits tasks to share objects that contain pointers (addresses) like linked lists, even though each task usually interprets memory addresses privately. In our scheme, a task dynamically establishes an area of memory to be shared and grants other tasks access to that area. Access to objects in the shared area is no more expensive than access to objects in any other part of memory.

* This work was supported in part by grants from the National Science Foundation's Office of Advanced Scientific Computing (NSF-851-8369), with additional support from Digital Equipment Corporation, CSNET, ATT Foundation and Purdue University

1. Introduction

In any multitasking computer system, the part of the operating system known as the *task manager* or *process manager* switches the CPU among multiple tasks. It meticulously saves the registers and machine state for one task and loads the saved machine state for another. When a multitasking system supports virtual memory, each task, sometimes called a *process*, executes in its own private memory address space without knowledge of the address spaces of other tasks that execute concurrently. That is, each task uses memory addresses starting at zero and extending upward, and the operating system arranges for the memory management hardware to map each task's addresses into a different region of physical memory. The operating system must save and restore the mappings from a task's address space to physical memory, just as it saves other machine registers, when it switches the CPU from that task to another.

In a virtual memory system, each task imagines that it has all of memory to use, starting at location zero. As we have said, memory management hardware maps the addresses a task generates into an area of physical memory reserved for that task. We say that the task generates a *virtual* address to distinguish those addresses from the *physical* or *real* addresses that the underlying hardware uses. Later sections describe the mechanism most commonly used to perform the mapping. For now, it is only important to understand that each application program is written to use addresses starting at zero, and that conventional operating systems arrange to keep tasks completely isolated when they execute.

The key concepts of virtual memory that we will discuss in this paper are protection and communication. Clearly, isolating each task in a separate address space protects the task's memory, because other tasks cannot read or change data values. On some hardware, the operating system, or *kernel*, executes in its own virtual address space, called *kernel space*, which is distinct from that of any task. In such cases, the operating system must execute special instructions to address physical memory directly.

The consequence of using virtual memory for protection is that tasks cannot easily move data among themselves. It is impossible, for example, to have one task assign a value to an integer variable that belongs to another task. All communication between tasks must go through the operating system. However, placing a request with the operating system, or moving data between the task's address space and the operating system's address space can be expensive. On most processors, a task must execute a special machine instruction known as a *trap* or *system call* to make such transfers. The system call instruction transfers control to the operating system and makes it possible for the system to access data in the task's address space. The operating system copies the data into its address space and changes the memory mapping. Later, when some other task places a system call to extract the data, the operating system must copy the data into that task's address space.

To understand how inefficient communication can be, consider an example from UNIX. The command

```
who | wc -l
```

creates two tasks (processes), one to execute the "who" command and one to execute the "wc" command, and connects the output of one task to the input of the other. The "who" command produces a list of users who are logged into the computer, with one user per line. The "wc" command counts lines in its input. Thus, the effect is to count the number of users logged into the computer. Because the two tasks each execute in their own address space, communication between them must go through the operating system. As a result, the operating system must copy the data from the address space of the "who" command into the address space of the operating system, and from there into the address space of the "wc" command. In this example, the amount of data being transferred is so small that communication costs are unnoticeable compared to processing costs. However, if we consider tasks like print spoolers that transfer large volumes of data, the communication costs can dominate the cost of the computation.

Memory sharing is not new to operating systems. In the XINU operating system [Com84], the operating system and all tasks share the same address space, and they communicate by sending pointers to global data. V [Che83] and THOTH [Che79] allow tasks to be grouped into teams, and, within a team, all tasks share their whole address space. TENEX [Bob72] has system calls that change a task's address mapping so that different tasks can share memory. DEMOS [Bas77] uses links for communication which allow the communicating tasks to read or write sections of each other's memory. Accent [Fit86] has a system call called MoveWords, which moves data between tasks or between a task and the operating system by sharing the memory the data occupies until one of the tasks attempts to write in the shared memory, in which case the operating system traps the attempt and makes a copy first.

2. Hardware Page Tables

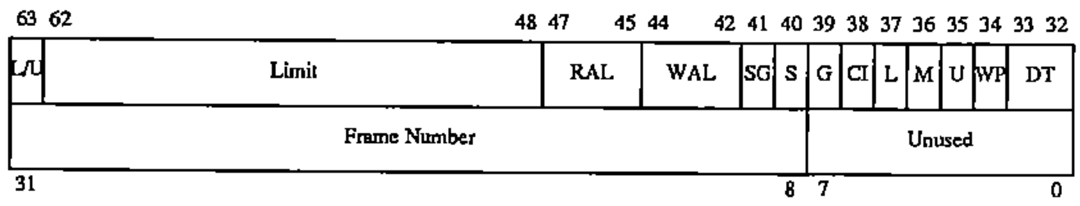
The hardware memory management unit (MMU) performs two functions that are essential to memory sharing; it maps addresses used by a program into addresses in physical memory, and it provides access protection. It performs both of these tasks using page tables. Each task has its own set of page tables, each of which describes a segment of its address space. A page table is simply an array of page table entries, each of which describes how to map addresses within a fixed range, called a *page*, onto a section of real memory, called a *frame*. Hardware designers determine the choices of page sizes (i.e., the range of addresses controlled by a single page table entry) when they build the MMU. Usually, they choose to make the page size a power of 2 bytes (e.g., 512, 1024 or 4196 bytes). A page table entry must specify the address of the frame in which the system has chosen to store the page. Since frames have to start at an address which is a multiple of the page size, only the multiple, called the *frame number*, has to be stored in the page table entry.

The page table entry must also specify what access is allowed for each page. The access control mechanism allows the operating system to specify, for example, that a given page is read-only, or that it can only be accessed by the operating system.

Figure 1 shows one of the page table entry formats for the Motorola MC68851 PMMU, used with the popular Motorola MC68020 processor. The page size for the MC68851 is software selectable, and must be a power of 2 from 256 bytes to 32K bytes. Hence, the frame number never occupies the low-order 8 bits of the page table entry, and might occupy less than 24 bits.

3. Implementation of Shared Memory

It might seem that a paging system makes memory sharing trivial. To share an object, the system arranges to have entries in two different tasks' page tables point to the same frame in real memory. However, if the shared memory contains structured objects like linked lists, the shared memory must appear at the same address in the address space of all tasks that share it. To achieve this goal, our sharing scheme requires the operating system to reserve a fixed segment of the address space of each task. We call the reserved space the *shared segment*. Tasks



Supervisor (S):	Access restricted to supervisor mode if S=1.
Modified (M):	Page has been written to if M=1.
Used (U):	Page has been referenced if U=1.
Write Protect (WP):	No writing to this page allowed in any mode if WP=1.
Frame Number:	Most significant 24 bits of page frame address.

Figure 1. Motorola MC68851 page table entry format. Each entry tells how to map one page into a frame in physical memory.

get access to the shared segment by asking the operating system to allocate pieces of it, called *shared objects*. The operating system keeps a table that describes the state of the shared segment and modifies the tasks' page tables to give them access to shared objects. Tasks acquire access rights to shared objects by allocating them, receiving them from another task, or inheriting them from their creating task. Tasks invoke system calls to allocate and delete shared objects, give access rights to other tasks, and alter the length and access rights of a shared object.

In order to prevent shared objects with different access rights from overlapping the same page, the operating system allocates memory for all shared objects in multiples of the page size used by the hardware. The operating system keeps all the information about the shared segment in the *shared segment frame table*, which has one entry for each page frame in the shared segment. An entry in the table contains a frame number, a length, and a reference count. For frames that are part of a shared object, the frame number points to the first frame of the object, the length is the total length of the object, and the reference count of the first frame in the shared object is equal to the number of tasks that have access to the object. There is also a free list of blocks of frames; the first entry in each block contains the frame number of the next free block, the length of the block, and a reference count of zero. Figure 2 gives an example of a frame table with a shared object allocated that takes up three frames. Notice that the reference count field of the first entry for the shared object contains the actual reference count, and those for its other frames are only non-zero to indicate that that frame is part of an allocated object, so that the reference count can be updated efficiently. Also, lengths are in bytes, so that a shared object might not be as large as the space allocated for it, which is rounded up to the nearest page size.

A shared object can be marked with the access rights *read* or *write*. The access rights each task has to an object are kept in its page table, and the operating system only implements the access rights supported by the hardware. Hence, access rights are checked directly by the hardware with every access, and the operating system only has to intervene when an access violation occurs.

A task allocates a shared object by executing the system call:

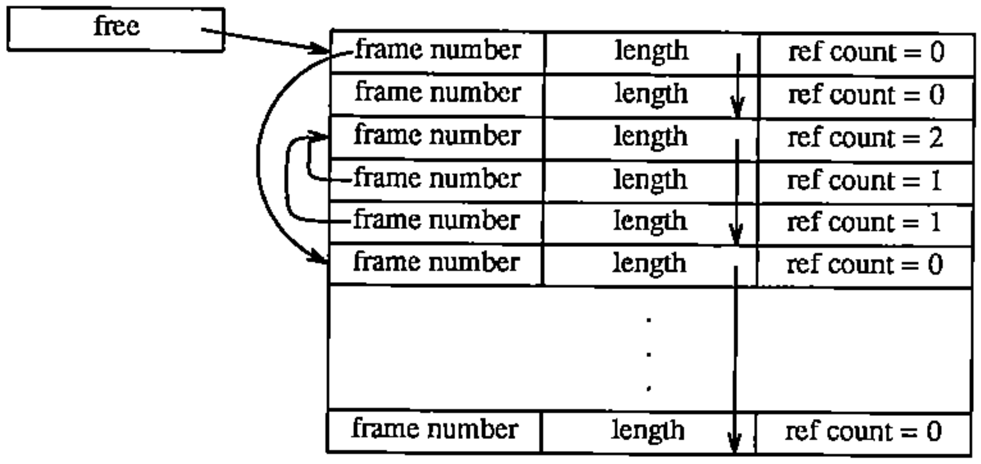


Figure 2. Shared segment frame table.

obj = shalloc(size, access),

where *size* is the requested size of the shared object in bytes and *access* specifies the requested access rights. Tasks communicate by passing access rights to shared objects. Access to a shared object can be given to another task with the system call

shsend(pid, obj, access).

This gives task *pid* access rights to the shared object *obj* specified by the intersection of *access* and the access rights of the calling task. When a task is ready to receive a shared object, it executes

obj = shreceive(pid),

which blocks until access to a shared object is sent to the calling task by task *pid*, or by any task if *pid* is the constant *ANYPROCESS*, and then returns the address of the shared object. An error value is returned if task *pid* does not exist or dies before it sends a shared object.

When a task is finished using a shared object, it calls

shdelete(obj),

which makes the object inaccessible to the calling task, decrements its reference count, and adds it to the free list if the count goes to zero. An additional system call,

obj2 = shexchange(pid, obj1, access),

is equivalent to

```
shsend(pid, obj1, access);
shdelete(obj1);
obj2 = shreceive(pid),
```

and is added for convenience. There are also functions that set and retrieve the length and access mode of a shared object:

```
shsetlen(obj, len);
len = shgetlen(obj);
shsetaccess(obj, access);
access = shgetaccess(obj).
```

A task can shorten a shared object, but cannot lengthen it, and it can restrict its own access rights, but cannot extend them.

4. More Efficient Communication

We now show some examples of communication using the the shared memory primitives, beginning with a task sending data to a print spooler. The client makes itself known to the server by sending its task id in a shared object. The server just waits for clients to send it shared objects and then spools data to print by exchanging shared objects with the client and writing the data in them to the spool area. Here is what the server and client might look like:

```
printspooler()
{
      char        *buf;
      int         clientpid;

      while (TRUE) {
            buf = shreceive(ANYPROCESS);
            clientpid = *(int *) buf;
            while ((buf = shexchange(clientpid, buf, RW)) != NULL
                  && buf != ERROR)
                   spool(buf, shgetlen(buf));
      }
}

client(fd, serverpid)
int   fd, serverpid;      /* file to print and server pid */
{
      char  *buf;

      buf = shalloc(BUFLEN, RW);
      *(int *)buf = getpid();
      shsend(serverpid, buf, RW);
      buf = shalloc(BUFLEN, RW);
      while ( (len=read(fd, buf, BUFLEN)) == BUFLEN)
            buf = shexchange(serverpid, buf, RW);
      if (len > 0) {
            shsetlen(buf, len);
            buf = shexchange(serverpid, buf, RW);
      }
      shdelete(buf);
      buf = shexchange(serverpid, NULL, 0);
      shdelete(buf);
}
```

Very little copying is done in this example. The server passes to its print spooling routine the very same buffer that the client used to read the data. Hence, the copies from the client address space into a buffer in kernel space, and from kernel space into the server address space are eliminated.

The shared segment can also make writing data to a disk file faster. One might consider this to be a slow operation anyway, because it involves disk I/O, but the write function does not actually wait for the data to be transferred to the disk. Instead, the operating system places the data to be written in a queue of disk I/O requests and returns control to the task, knowing that the disk driver will eventually write the data to the disk. Hence, copying the data from the task's address space to kernel space adds significantly to the cost of the operation. To make the write faster, the operating system can include a new system call,

shwrite(fd, buf, len),

which adds the shared buffer, *buf*, to kernel space and removes it from the task's address space, eliminating the copy to kernel space.

5. Conclusion

One of the hidden costs in an operating system is the time spent copying data. Communication in a multitasking, virtual memory system, whether between tasks or between a task and the operating system, usually requires copying to get data from one address space to another. One way to eliminate the need for copying between address spaces is to provide a mechanism by which sections of memory can be shared among tasks and the operating system. Using conventional memory management hardware, we have proposed simple system calls that make it possible to share memory among different address spaces while still providing protection from non-cooperating tasks. Examples have shown the elegance and efficiency of the shared segment primitives.

Some preliminary work has been done to implement a prototype system that uses the shared segment concept. At the Department of Computer Sciences at Purdue University, the 4.2BSD UNIX kernel and linker have been successfully modified to insert an inaccessible shared segment at the beginning of each task's address space, shifting the program text, data, and heap upward. A team of graduate students has written a version of the XINU operating system which uses the hardware memory management on a Digital Equipment Corporation MicroVAX to implement virtual memory and a shared segment through which tasks can share memory. The experiments show that the proposed mechanism is easy to implement and that it works well in practice.

6. REFERENCES

[Bas77] F. Baskett, J. H. Howard, and J. T. Montague, "Task Communication in DEMOS", In Proceedings of the 6th Symposium on Operating Systems Principles, (Purdue University, Nov. 16-18), ACM, New York, 1977, pp. 23-31.

[Bob72] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time-Sharing System for the PDP-10", CACM 15, 3 (March 1972), pp. 135-143.

[Che79] D. R. Cheriton, M. A. Malcolm, L. S. Melen and G. R. Sager, "Thoth, a Portable Real-Time Operating System", CACM 22, 2 (Feb., 1979), 105-115.

[Che83] D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations", In Proceedings of the 9th Symposium on Operating Systems Principles, (Bretton Woods, NH, Oct. 11-13) ACM, New York, 1983.

[Com84] D. Comer, Operating System Design, the Xinu Approach, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Fit86] R. Fitzgerald and R. F. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent", ACM Transactions on Computer Systems 4, 2 (May 1986), pp. 147-177.