Kernel Support For Network Protocol Servers

Franklin Reynolds
fdr@osf.org

Jeffrey Heller
jeffreyh@osf.org

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142

Abstract

The current Mach 3.0 network device interface provides support for user space protocol servers. This support includes mechanisms for user space device drivers and a message based device interface including a programmable packet filter for demultiplexing network packets. The flexibility and other benefits of these features are not without cost. The goal of our work is to improve network performance, cpu utilization, OSF/1 network driver compatibility, and support for protocol server compression while avoiding any significant perturbation of the Mach 3.0 interfaces. The design and implementation of a shared memory communication channel between the kernel interrupt handler and the user space network protocol server is described. A variety of performance measurements are reported and compared to measurements made on similar systems. Some of the implications of protocol servers on hardware and software architectures are discussed.

Introduction

Traditional operating systems kernels such as BSD 4.3, Mach 2.5 and OSF/1 provide all OS functionality, including networking, as part of a single integrated kernel. When network packets arrive the networking protocol code is able to process this data directly. In some cases no data copies are necessary. For instance, for packets with network specific information such as ARP and routing information, and only one copy is needed from the kernel's address space to the users buffer for many other types of incoming Packets.

With the advent of Mach 3.0 and the BSD single server [1] the interactions between the network device driver and the network protocols changed. Most BSD functionality, including network protocols, was evicted from the kernel and into a user space server. With the separation of BSD from Mach, device drivers could no longer assume the presence of BSD services. The ability to share information and state between the network device driver and the network protocols was eliminated by the separation by of the address spaces. A new device interface was created to allowing the kernel to export device services to Mach tasks. Most if not all device drivers and the clients of the device driver services written for these traditional operating systems must be modified to work in the new Mach 3.0 environment.

The network device interface for Mach 3.0 is message based. Network packets arrive, are copied into a message and sent to servers listening for packets via IPC. There is a thread that runs as part of the BSD single server which receives these messages. An mbuf is pointed at or wrapped around each message then fed to the network protocol code and processed in the same manner it would have been in a traditional operating system kernel. The kernel exports a programmable packet filter as part of the device interface [2]. The packet filter provides packet shedding, demultiplexing and duplication. This filter interprets a simple, stack oriented language. Filter scripts can be created and installed by user tasks.

To use this new interface and kernel, a network device driver of a BSD or OSF/1 origin requires many changes. For instance, the original network drivers assumed the existence of mbufs and had explicit knowledge of the supported protocol families. The Mach 3.0 drivers do not have BSD or protocol family dependencies. Functionality such as the first level of network packet demultiplexing and filtering is implemented as a packet filter script. Instead of putting the data into some space in the kernel that the driver and the protocol code know about, a message must be put together, data copied into it and sent off to the network

code. This method adds buffer copies and context switches to the cost of receiving a network packet. In the case of small packets, a message based interface adds significant overhead relative to monolithic systems. The 3.0 network device architecture is powerful and flexible and complex. This power and flexibility is not without cost.

Mach 3.0 has experimental support for user space device drivers. There is a reference implementation of an ethernet driver for the PMAX in the 3.0 distribution. The user space ethernet device driver work was motivated by, among other things, the need for better performance. One limitation of this approach is that a single network server is favored over all others. Only one task has direct access to the network driver. Only that task gets the performance advantage. Other tasks wanting information off the wire must arrange with the favored task for it to pass packets they are interested in on to them. Perhaps more important is that some hardware platforms lack the necessary features to make user space device drivers practical or efficient.

The goal of our work is to improve the network performance and efficiency of the OSF/1 single server. We do this by attempting to reduce the number of context switches, data copies and scheduling interactions during message processing. We hope to improve OSF/1 monolithic kernel and single server network driver compatibility by increasing the reuse of OSF/1 driver and low level protocol code. This would reduce the cost to vendors of migrating from monolithic kernel technology to micro-kernel based technology. In addition we attempt to avoid significant perturbation of the Mach 3.0 kernel interfaces.

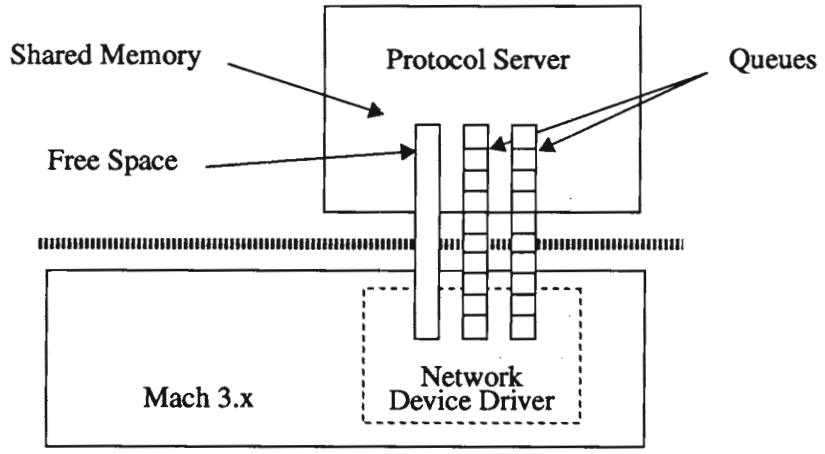
A Shared Memory Device Architecture:

We have developed a experimental network device driver framework that attempts to address our performance, portability and compatibility requirements. Our framework incorporates the use of a shared memory communication channel between the device interrupt handler and the network protocol servers. Our design relies on two kernel extensions.

1) The ability to create shared memory between an in-kernel device driver and a user space task

2) The ability of a kernel interrupt handler to perform a counting unblock of a user space thread.

In this approach the network device driver shares a region of memory with the protocol server. Both the driver and the server have read and write access to the shared memory. Mach 3.0 already defines a dev_map() service that can be trivially extended to provide the desired semantics. The shared memory consists of queues, free space for network buffers and a few shared variables. Queues are associated with devices, protocol families and other services. These queues provide the primary channel for communication between devices and protocols.



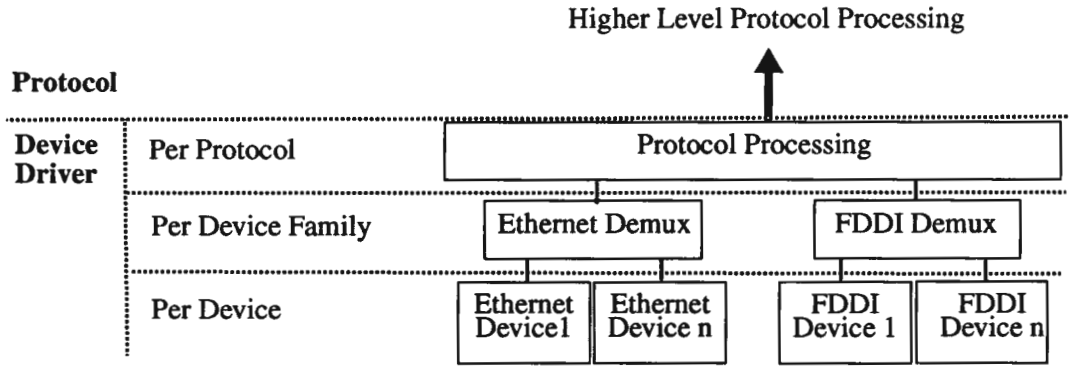
The design is derived from OSF/1 which, though a monolithic kernel, has a similar design. The network interrupt handlers communicate with the protocol threads by placing network buffers on appropriate protocol

queues and then, if necessary, unblocking a network thread. Our approach, of enhancing and transparently extending the functionality of extant abstractions increases OSF/1 device drive compatibility and the reuse of OSF/1 code within the server.

Device Driver and Protocol Server Interdependency:

The OSF/1 device drivers execute in the same context as the protocols. The Drivers exploit apriori knowledge of the framework and the configured protocol families to improve performance. Regardless of the truth and beauty of this issue, the practical consequence in this case is that the network device driver understands the network buffer structures and have some apriori knowledge of the configured protocol families. In the interest of OSF/1 compatibility we BSD 4.3 Reno mbufs, clusters and Stream mblks. BSD 4.3 mbufs are fairly general memory descriptors and have been used extensively. In fact, OSF/1 is able to transparently and inexpensively layer mblks on top of mbufs. Protocol frameworks that use alternative data structures, such as the x-Kernel, which do not wish to adopt mbufs can always make the relatively simple changes to the device drivers necessary to use their own buffer descriptors.

Knowledge of the configured protocol families allows early packet filtering. This is currently done in the OSF/1 device interrupt routine. To take advantage of this will require new protocols to add specific code in the device interrupt handler. In the worst case this will mean a new version of all network device drivers for each new family of protocols. Fortunately, new protocols are relatively uncommon and in most cases the necessary driver modification s are trivial. The first level of packet filtering could be evicted from the kernel but it would, without a doubt, slow some things down. In particular, protocol selection by the driver allows us to support multiple protocol servers without paying the cost of indirection through a demultiplexing server. Another advantage of doing the first level of demux inside the interrupt handler is that corrupt packets can be shed without delay. This improves the robustness of the system by preventing corrupt packets from tying up resources (buffers and CPU cycles) that could be used to handle real data.



The figure above illustrates the hierarchy of packet processing within the device driver. A the lowest level there is per device, usually per network device controller, processing. Device errors would typically be handled at this level. At the next level, the Family level, processing that is specific to ethernet but independent of the individual manufactures versions of ethernet, takes place. For example, the ethernet header is examined for supported protocol types. Packets with bad types, such as test protocols, are dropped. The top layer is entirely hardware independent.

Shared Memory & Synchronization:

Mach 3.0 exports a dev_map() interface. This interface did not change. The original device pager that is established via dev_map() calls dealt with physical pages. A small change to the implementation of the device pager allowed it to handle virtual address. The dev_map() service in conjunction with vm_map() are sufficient to build the shared memory window.

While it is possible to share memory between the kernel and a user task, it is very difficult on some hardware platforms to make the virtual addresses of the shared region align. This alignment, or rather the lack of alignment of the virtual addresses of the shared memory in the kernel and the protocol server pose some problems. Pointers and data references are usually absolute virtual addressees rather than offsets from a base register or segment. As a consequence shared pointers only work without some sort of conversion when the shared memory is mapped to the same location in each virtual address space. Our strategy is to assume that the shared memory structures and pointers should correct for the protocol server and appropriate conversions will be made in the kernel. Translations are done in the driver to localize and minimize the impact on the code base.

Consider the example of a queue of buffers in shared memory that does not align. For the purposes of this example the kernel maps the window at 0xc000000 and the server maps the window at 0x6000. Assume that a shared memory queue begins at the beginning of the window. Both the kernel and the protocol server have non-shared memory pointers that refer to the shared memory queue control block. The value of the server's queue pointer is 0x6000 and the value of the kernel's pointer is 0xc000000. The queue points to a buffer in shared memory. The buffer is offset 0x100 bytes from the beginning of the shared memory window. The value of the queue's pointer to the buffer is 0x6100. Server code can use the queue pointer in a natural fashion. The kernel must convert 0x6100 (which in the kernel's context may refer to almost anything) to 0xc00100 before it can be used as a pointer.

Protocol to interrupt (PTOI) and interrupt to protocol conversion macros have been defined.

```
#define PTOI(addr) (((addr)) ? (unsigned int)(addr) +
                        (unsigned int)plus_shm_offset - (unsigned int)minus_shm_offset : 0 )
#define ITOP(addr) (((addr)) ? (unsigned int)(addr) -
                        (unsigned int)plus_shm_offset - (unsigned int)minus_shm_offset : 0 )
```

Note the test for 0. This is necessary to avoid converting NULL pointers to non-NULL pointers. While the cost of these conversions are not dramatic they could be avoided completely on friendly hardware.

We need to synchronize access to the shared memory between threads on multiple processors and interrupt handlers. Synchronizing access by the user space protocol servers and the kernel space device driver to data in the shared memory present some difficulties. Typically, unprivileged programs use a system call or use either atomic operations such as compare-and-swap instructions to synchronize with the kernel. Privileged tasks, such as the kernel, frequently disable interrupts to synchronize with interrupt handlers. This is because the naive use of locks without disabling interrupts, even spin-locks, to synchronize with interrupt handlers has deadlock problems [3]. The performance characteristics of system calls make them unsuitable for our purposes. In the absence of hardware support across the majority of interesting platforms, a portable software synchronization method must be devised. In the absence of the ability to disable interrupts another method must be employed to prevent the interrupt handler from waiting forever for a lock.

It has been asserted that wait-free synchronization primitives can be engineered with an atomic compare-and-swap instruction [4]. Presumably these primitives would be suitable for synchronizing with interrupt handlers. However, many hardware platforms do not provide hardware support for atomic operations such as compare-and-swap or fetch-and-theta. In fact, there are popular hardware architectures that do not provide even a simple atomic test-and-set instruction. Hardware that allows unprivileged tasks to disable interrupts, a frequently used technique for synchronizing with interrupt routines, are even less common. Because of the general lack of adequate hardware support we chose to provide a software solution. Our hope is that the abstractions will prove amenable to optimization on friendly hardware platforms.

Our queue and the buffer allocation abstractions provide wait-free operations to the device driver's interrupt routines. The queue and buffer management routines use a low level software mutex to implement critical sections. This mutex can be replaced or optimized on friendly hardware platforms. Critical sections are

used to coordinate access to the new data structures necessary to implement the queue the various wait-free operations. There is a per queue conditional variable used to indicate that the agent that removes buffers from the queue has blocked as well as a new mechanism for unblocking threads.

Software Mutex:

We chose to implement a software mutex algorithm to serve as the basis for the higher synchronization protocols. A software approach has the twin advantages of being very portable and amenable to hardware dependent optimizations. We were also interested in assessing the cost of software synchronization algorithms.Software algorithms for mutual exclusion [5] are designed for coordinating multiple processes - not processes and interrupt processing routines. Interrupt handlers have different execution characteristics from processes. Generally speaking, device drivers do not like long busy - wait or spin loops When synchronizing with device drivers, monolithic kernels usually take advantage of privileged instructions to disable and re-enable interrupts. Unprivileged tasks, such as the protocol server must find another way to synchronize with interrupt processing routines.

We have designed an asymmetric variation of Peterson's [6] mutual exclusion algorithm. Descriptions and analysis of the original algorithm can be found in [5],[6].The only significant hardware assumptions we make are integer (32 bit) sized atomic loads and stores. This maximizes the hardware independence of the code. The important characteristic of this variant of the algorithm is that the interrupt side does not spin indefinitely, instead it gives up after of couple of attempts to get the lock. This is required in order to remove the possibility of deadlock. Otherwise deadlock could occur if the device driver interrupted and attempted to acquire the mutex immediately after the protocol server acquired the mutex.

Protocol Side:

```
protocol = TRUE;                        /* announce intent to acquire mutex*/
turn = INTERRUPT;                       /* give the interrupt routine a chance */
while ( interrupt == TRUE && turn == INTERRUPT ) {
        spin() ;                        /* wait if the interrupt routine got there first */
}
< critical section >
protocol = FALSE;                       /* renounce intent to enter and exit*/
```

Interrupt Side:

```
interrupt =TRUE;                        /* announce intent to acquire mutex*/
turn = PROTOCOL;                        /* give the protocol task a chance */
if ( protocol == TRUE && turn == PROTOCOL) {
        interrupt = FALSE;              /* if the protocol routine got there first */
        return(FALSE);                  /* renounce intent to enter and exit */
}
< critical section >
interrupt = FALSE;                      /* renounce intent to enter and exit*/
```

Note the asymmetry of the algorithm. The Protocol side of the algorithm is identical to Peterson's original. The protocol or single server is permitted to wait (by spinning) for the interrupt processing to complete. On the other hand, the interrupt handler gives up almost immediately rather than spinning. This removes the risk of deadlocking on the mutex. After the interrupt handler relinquishes its attempt to acquire the first mutex it is free to attempt to acquire another mutex or take other action.

Queues

OSF/1 1.0 already uses queues as the communication channel between the network device driver and the protocol threads. Queue initialization, lock, enqueue, dequeue and flush operations are already defined. We have attempted to preserve these abstractions. While we were largely successful some trivial side effects of our different implementation strategy are visible.

Access to the original OSF/1 queues is synchronized by simple locks. These are spin locks that depend on the ability of the protocol code to disable interrupts in order to synchronize with device drivers. The protocol server, as a unprivileged task, does not have the ability to disable interrupts. Consequently the locks for the shared memory queues are built from the software mutex algorithm described earlier.

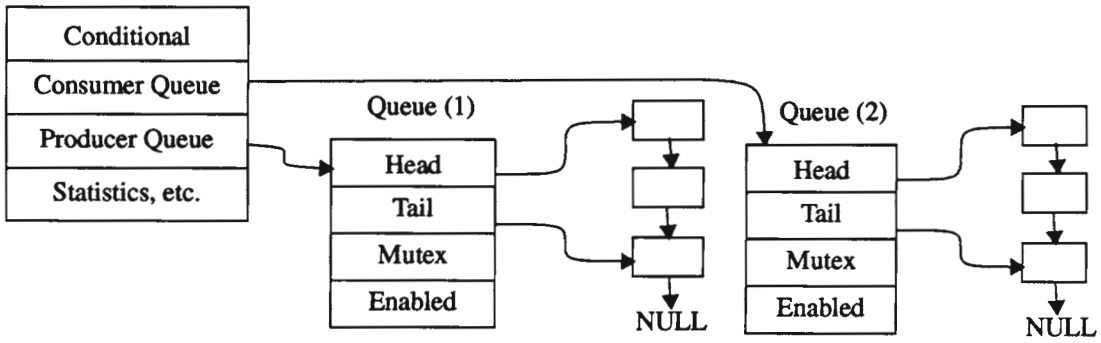
Another consequence of the protocol code executing in a user task is that its execution can be pre-empted for long periods of time. The naive implementation of locks and queues using the software mutex can be problematic if the protocol server blocks or is pre-empted. It is important for good performance that the queues used for sending buffers to protocol stacks be available to the driver. If the queue is unavailable the driver can drop the packet while trusting to the higher level protocols to generate a retry. There are a variety of reasons why this can happen in the OSF/1 monolithic kernel today. However, dropping packets almost always has a bad effect on performance and should not be done lightly.

In some situations the driver **must** be able to successfully enqueue a buffer to avoid leaking resources. It is possible that the device driver is finished with an outbound buffer but cannot successfully return it to free space. This can occur if the free space is locked by the garbage collector or if the buffer is complex such as a chain of buffers or a buffer containing a pointer to a cluster. When the driver decides that a buffer is too complex to free it enqueues the buffer on a deferred-free queue. The protocol pulls buffers off of this queue and frees them at a more leisurely pace. The driver must be able to enqueue the complex buffers onto the deferred-free queue in order to guarantee that the buffer will be eventually freed and not forgotten.

The problem is: design a multi-processor safe queue that is always that is essentially wait-free to the device driver without the hardware support to disable interrupts or perform complex atomic operations.

To accomplish this we define a shared memory queue. There are two types of shared memory queues, IN queues and OUT queues. The Consumer (whatever performs dequeue operations on a particular queue) of an IN queue is the protocol server. The Consumer of an OUT queue is the device driver. The asymmetric nature and needs of the different Consumers dictates an asymmetric implementation of the enqueue and dequeue operations. Each queue is actually composed of two separate queues. There is pointer for the Producer and the Consumer indicating which sub-queue is being used. An enabled flag associated with each sub-queue. is used by the enqueue and dequeue operations to preserve queue order. Each IN or OUT queue has a conditional variable used to indicate that the Consumer for that queue is idle and must be restarted in order to resume dequeue operations.

IN/OUT Shared Memory Queue



Coordinating the use of the sub-queues is a bit complicated. In addition to the software mutex protocol there is a condition variable protocol, a queue ordering protocol as well as the enqueue and dequeue protocols. A conditional variable is associated with each shared memory queue. The Producer reads the condition variable and the Consumer writes it. The Producer tests the condition and if the condition is TRUE then a device event is generated. The Consumer sets the condition to TRUE if there is nothing to do. Then it checks again, in case new work arrived in the interval between the time it last checked for work and when the condition was set. If there is still nothing to do then the Consumer becomes idle. Otherwise the condition is set to FALSE and the Consumer does the appropriate work.

It is possible for the Producer to detect that the condition is TRUE before the Consumer has actually become idle. In order to avoid losing device events the device event service counts events. In our prototype device events are implemented using a version of thread_resume() that counts the resumes as well as the suspends. If the Producer generates a device event before the Consumer attempts to block then when Consumer eventually does attempt to block it will immediately return instead. The condition variable protocol, in conjunction with device events, provides a mechanism for Producers to activate idle Consumers.

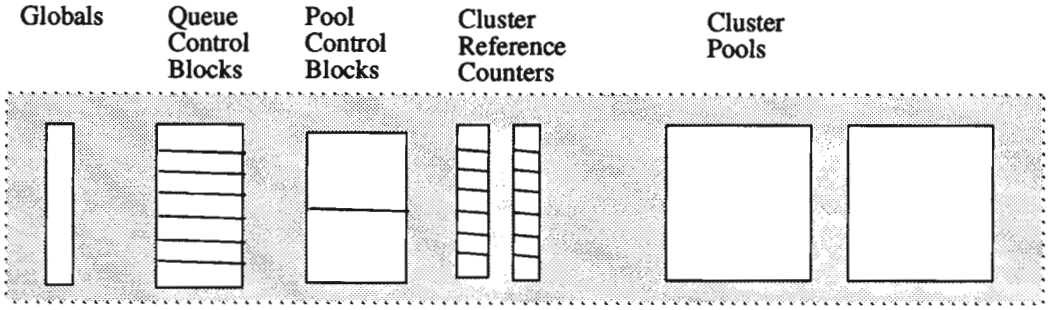
The Enable protocol is used to guarantee that buffers are dequeued from the set of sub-queues in the order that they were enqueued. The Producer only uses the Enabled flag when it discovers that its current sub-queue is locked by the Consumer. In that case it locks the other sub-queue and enables it. The Producer never changes its current sub-queues unless it was blocked. When that occurs it sets the Enabled flag on the new sub-queue to indicate that it successfully changed sub-queue and enqueued a buffer. The Consumer only references the Enabled Flag in the event of a sub-queue becoming empty. When the Consumer's current sub-queue becomes empty the Consumer checks the Enabled Flag of the other sub-queue. If the Producer has successfully changed sub-queues and enqueued at least one buffer the Enabled bit will be TRUE. In that case the Consumer changes its sub-queue, sets the Enabled but to FALSE and dequeues the buffer.

Free Space:

The design of the shared memory queues is complex. Synchronizing access by interrupt handlers and unprivileged tasks to shared memory is difficult. The same difficulties arise in the design of shared buffer space management (allocate, free, garbage collection). An additional source of complexity is both Device drivers and Protocol servers need to allocate as well as free buffers.

A solution to the problem is to divide the buffer free space into multiple pools. All of the control structures are duplicated for each pool. This includes statistics, reference counter arrays, and other control information. Most of the control information is located within a pool control block. Access to each pool is mediated by a Mutex variable. When the device driver needs a buffer it can query each pool. If it is unable to lock the pool or if the pool is empty it can go on to the next pool. If there are no buffers available then the device will discard the packet. This strategy allows us to reuse the bulk of the OSF/1 mbuf and cluster code.

Shared Memory Window



One side effect of multiple free pools is that the buffers need to be returned to the pools from which they were allocated in order to be coalesced into clusters. This is not difficult since each pool is composed of a

contiguous region of memory. The correct pool can be deduced from the address of the buffer. Note the fact that the cluster pools are adjacent in shared memory. This simplifies and improves the performance of the mbuf to pool (mtop) calculations before the original mbuf to cluster number (mtocl) calculation can be performed.

When the server is finished with a buffer or cluster the resource is returned to the appropriate free pool and the reference counters are decremented. When the device driver is finished with a buffer it attempts to return it to the appropriate free pool. If the attempt fails, either because the pool was locked or the buffer was complex (a buffer chain, special free requirements, or a buffer with associated external memory) the interrupt routine places the buffer on the deferred-free queue and the protocol server eventually returns the buffers to the correct free pools.

Sometimes a thread in the OSF/1 monolithic kernel goes to sleep because there are no free buffers available. When a buffer is freed any threads blocked waiting for buffers are receive a wakeup(). OSF/1 uses the global variable "m_want" as a conditional to indicate that there are sleeping threads in need of buffers. Since our design needs to be able to free buffers from protocol or interrupt level processing m_want was moved to shared memory. The server sets m_want to TRUE if any networking threads block in need of buffers. The kernel never changes the state of m_want. This permits m_want to be referenced without first acquiring a lock. After each successful release of a buffer by the kernel m_want is tested. If it is TRUE then the network threads are unblocked. The first of these threads resets m_want to FALSE.

There is an interesting trade-off to be made when choosing the number of pools. Assuming the total amount of free space is constant, a large number of pools decreases the amount of free space controlled by any single mutex. This means that during the time the Protocol holds a particular mutex it is restricting access to a smaller portion of the free space. This gives the Device a better chance of finding a pool with some free space in it. This is a Good Thing. On the other hand, the fragmentation of the free space into many pools will have a tendency to limit garbage collection. It would be problematic to join data into contiguous chunks spanning pool boundaries. Smaller pools also increase the chances that multiple pools will have to be checked before any free space can be found since any individual pool will be more quickly exhausted.

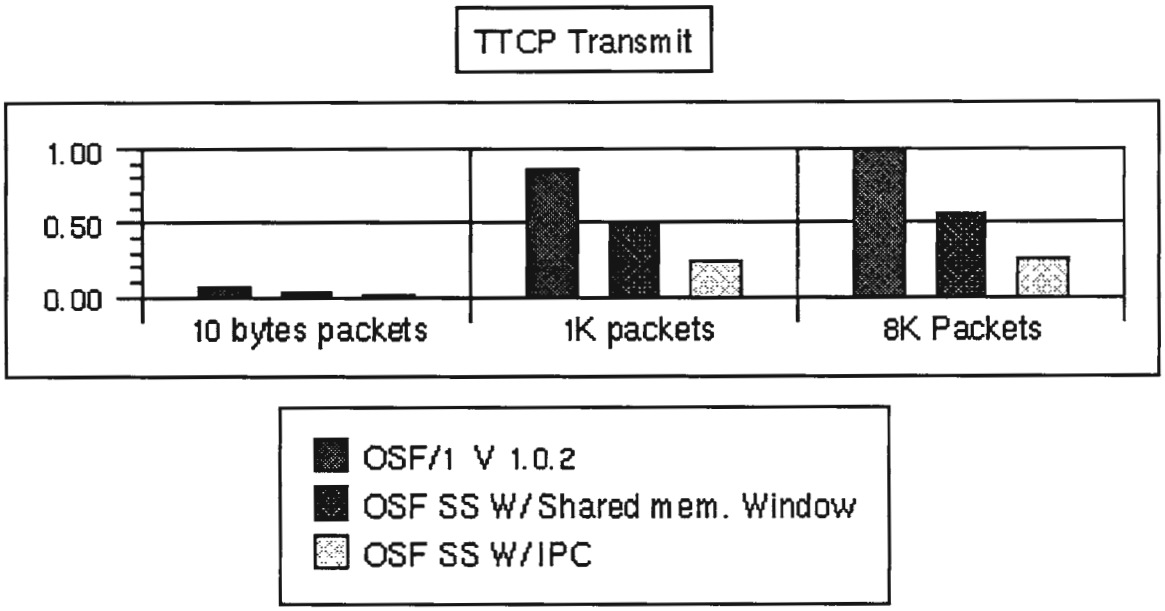
Network Performance:

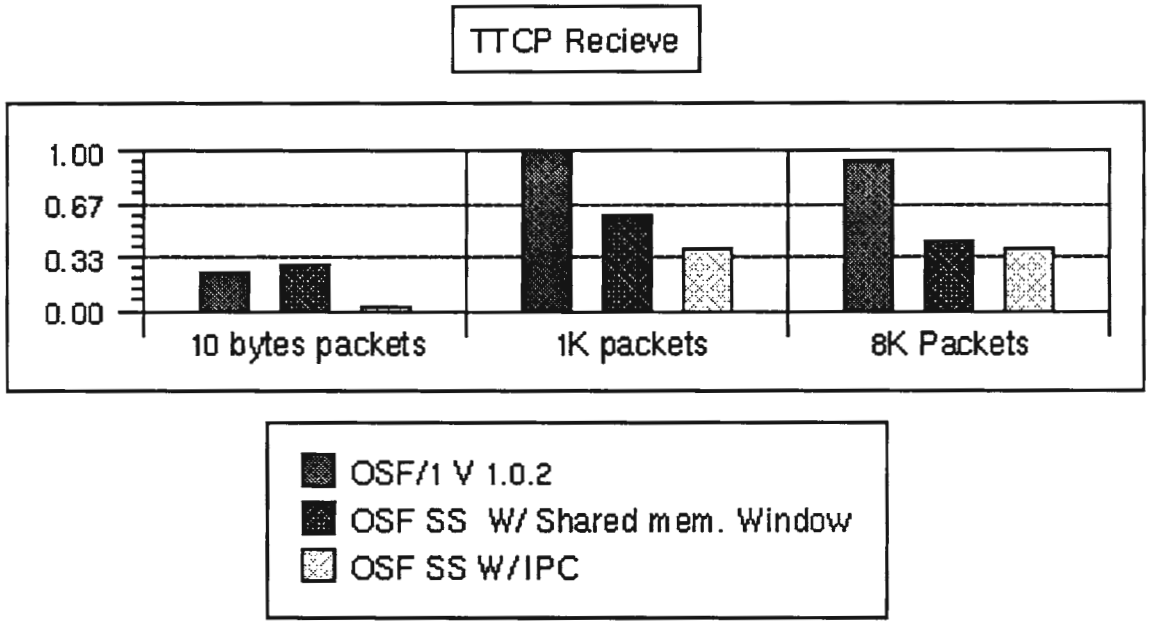
The goal of our performance testing plan is to understand the performance and the cost of this architecture. All of our measurements were done on the same Hp Vectra 386 based computer with 16 megabytes of memory, a western digital wd80013EBT networking card, and an OSF/1 1.0.2 binary set. The only difference between the tests was in the kernel (and single server for the two Mach 3.0 based system) chosen at boot time. The systems that were measured were OSF/1 V1.0.2, the second pre-alpha release of the OSF/1 single server, and that same single server with only the changes described in this paper.

All of the tests were conducted between the 386 machine and a 68040 based NeXT Station. The NeXT has sufficiently greater ethernet performance than the 386 platform that we have confidence that it was never a limiting factor in our tests. As the implementation of all three versions of networking code were at least 90% the same, all measured differences between systems were due to the changes at the device interface layer. The limited number of variables in the performance tests give us give us confidence in our analysis of the results. All of our performance numbers are presented on the bar graphs in a form normalized to the fastest of the tests in a series.

We have chosen to run three benchmarks. The ttcp program - version 1.10 , ftp and the dot benchmark provided in version 1.2 of the X11perf package.

Our first test, ttcp is a streaming network test. No file system interaction happens during the tests, and the size of the packets can be chosen. Six repetitions of transmit and receive tests were done for three different packet sizes. We have chosen a small 10 byte packet size, an average 1K packet size and a larger 8K packet size.

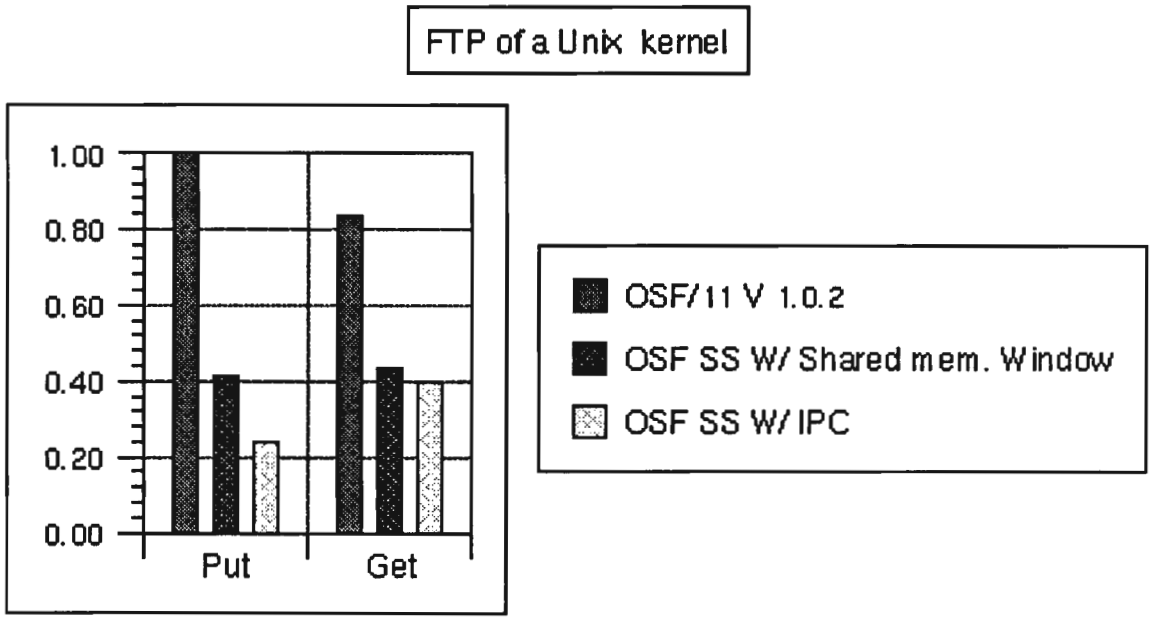




TTCP: Ttcp numbers are much more effective in judging network streaming performance then ftp. This program is purely a network benchmark. As used by us it has no file system interactions. For transmits our tests showed that the OSF single server's performance was less then one third of the integrated system across all three packet sizes. As can be seen in the "TTCP Transmit" bar graph, our modified single server has recovered about half of the difference for all three packet sizes. All three system perform poorly on very small packets, yet still the OSF/1 system is more then twice as fast as the other two systems. Our modified single server was uniformly in the middle of the two systems. We are currently winning back about half of the performance lost by the current interface in transmits. Ttcp receives present a slightly different

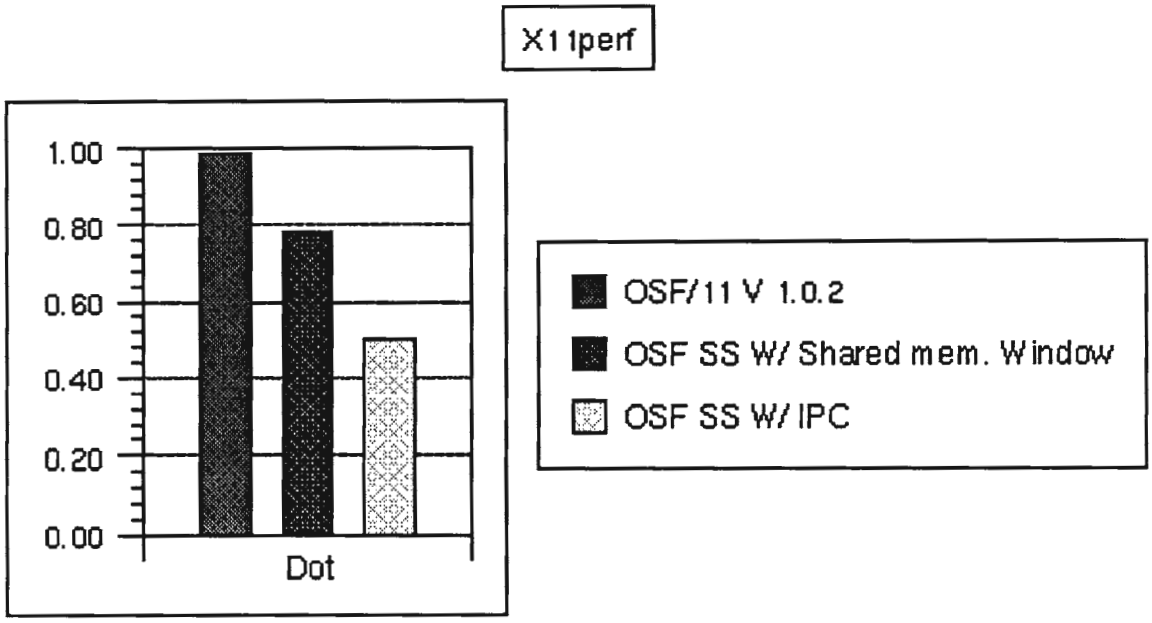
story. On ten byte packets the OSF single server is less then a fifth the performance of the OSF/1 system. Our modified server outperforms OSF/1 by about ten percent. This is the only place where the benchmarks show our server outperforming the OSF/1 system. On one kilobyte packages the OSF single server was about one third of the performance of the OSF/1 system, and we are again in the middle of the two with performance just over half the performance of the OSF/1 system. On 8 kilobyte packages the OSF single server was a little under half as fast as the OSF/1 system, and our modified system was only a little faster then the standard single server.

Ftp was chosen due to the property that it is universally used to check network performance. Although it is not a great network benchmark due to interactions it has with the file system, it is common, easy to use and easy to report. No network performance discussion would be complete without some ftp numbers. This ftp test was the ftp a unix kernel from disk on one machine to /dev/null on the other. The test was run six times in each direction.



FTP: In ftp OSF/1 1.0.2 was the best to both transmitting and receiving files. The OSF single server using the standard IPC based network interface from CMU is about twenty five percent the speed of the OSF/1 system on transmits and just over forty percent the speed of OSF/1 in receives. The closer numbers for receives over transmits are not too hard to believe as the OSF single server's file system, and disk interfaces are not yet final and complete. Work on file system is currently underway in another project. Our enhanced OSF/1 single server suffers from this same problems, and we expect performance of both systems to improve over time. Ftp's reliance on file system performance is one of the reasons why it is not a reliable network benchmark. Our modified OSF single server did provide some performance improvement in the ftp case. In transmitting a file we saw an improvement of almost eighty percent over the standard interface, however we still are only about 45 percent the performance of the OSF/1 integrated kernel. On receives the story is not quite as good. While we are just over 50 percent the speed of the integrated kernel, we are just over ten percent faster then the OSF single server.

Our third and final benchmark is X11perf's dot test. This was run as an X client application from the 386 to an X server running on the NeXT. The benchmark reports back the number of dots per second that it was able to cause the server to plot. This test has the most varied workload of all three tests, not only is the test sending to the network, but we have the intervention of Xlib packaging the server requests up in what it considers to be reasonably sized request.



X11perf: The numbers from the X11perf suite's dot test are valuable because that are not just network streaming number of a single packet size. This test was the closest we came to testing a real network application. It numbers should give us a feel for how the performance of network applications such as X clients would be affected by the network interface. Again here we see that the OSF/1 system performed the best. The OSF/1 single server performed with about half the performance, and our modified server was about seventy five percent the performance of the OSF/1 system.

The results contain few surprises. Most of the benchmarks show that we have made progress towards our goal of closing the gap in performance between traditional integrated kernels and protocol servers. In our benchmarks we see no areas where we are outperformed by the message passing network interface, although there are areas where we have made less progress to date then we had hoped.

In most cases we have reduced the cost of out of kernel networking. In the majority of the streaming tests we have split the difference between the OSF/1 system's performance and that of the standard OSF single server with it's Mach IPC based network interface. There were two surprises for us. In the ttcp receive case of eight kilobyte packages we had a larger drop in performance over the one kilobyte package size then we anticipated. Our working hypothesis is that the buffer pools are being exhausted but we not yet verified this.

The other surprise was in the ttcp receive of ten byte packets we outperformed even the OSF/1 1.0.2 system. Again it is still early in our analysis of the results, however we speculate that this advantage in performance over the integrated kernel stems from reducing the number of scheduling events. Where OSF/1 schedules a thread for each incoming packet, we only need to schedule a thread if the Consumer thread for that queue is idle queue. With lots of small packets we believe that the producer is able to put packets on the queue faster then the consumer can get them off. The consumer just works its way down the incoming queue, not needing to be scheduled for each packet, because it checks for more work before it goes to sleep. The producer being able to tell that the consumer is still working does not need to go to the work of scheduling some consumer to process the packet, as it does in the current OSF/1 system.

While our numbers are not neutral to the OSF/1 system, these numbers from our early prototype give us a belief that we are on the right track. We still are using mechanisms that are very expensive, such as using the device_setstat mechanism to wake the device up to dequeue outbound packets, yet we are still seeing an across the board performance improvement. We are optimistic that our network interface performance will continue to improve.

Discussion:

The disparity between the performance figures of the monolithic kernel and the standard single server give some indication of the costs associated with user space protocol servers. It should be noted that both single servers tested were early prototypes. Their overall performance, as well as their network performance will improve in due course. All of the performance improvement shown by the experimental single server was due to the network enhancements described in the paper. No other modifications to the code base were made. This suggests that as the performance of the standard single server improves so will the experimental server.

The performance numbers suggest that the biggest improvement comes reducing the number of times a buffer is copied. Moving data to or from the device via Mach IPC is expensive. Even the use of IPC to initiate device writes when there is little or no data to be copied is expensive if it has to be done for every network message. Using the shared memory queues reduces the number of copies. But on the system tested, which has a single CPU and a dumb network device, the number of IPC messages is not significantly reduced.

On the test system, writes are almost always synchronous even though the mechanisms are in place for asynchronous, parallel operation. There is only one CPU. After on out-bound packet is queued on an OUT queue the device is started via a trap to the kernel. The kernel dequeues the packet, writes it to the device and looks for something to do. There is never anything to do because the CPU has been busy doing the device write. Since there is nothing to do the routine becomes idle. Eventually the server gets the CPU back and puts another buffer on the queue and checks to see if the device is idle. It almost always is and the cycle repeats. A multiprocessor system or a system with a smart device controller would permit the protocol to enqueue buffers and the device to write them out at the same time.

IN queues are structured in a way that permits the asynchronous nature of receives to be exploited. However, single threaded applications that exhibit request reply behavior work against asynchronous receives. If the protocol server must ACK before the next message is sent then the asynchronous nature of the receive will never be exercised.

Future Work:

In the immediate future we plan to port the system to coherent shared memory and message passing multi-CPU computers. Our system has enough characteristics in common with UPRC [7] that we expect similar performance on a shared memory machine. URPC may suggest to our design. We also plan to port to a platform with a different CPU and I/O architecture. This will give us the experience we need to judge how successful we were at designing abstractions capable of exploiting friendly hardware. Eventually we plan to run other network applications and benchmarks in an effort to refine our understanding of the behavior characteristics of the system.

We plan to continue work to improve the performance of user space protocol server. An investigation into ways to further reduce context switching and scheduling interactions. One approach is to improve the asynchronous operation of device write without the need for the server to explicitly interact with the kernel. For example, some devices can generate an interrupt upon completion of a write or after a timer elapses. This would be a good cheap way to get into the kernel without an IPC message or system call. Another opportunity is when a message is received. At that time an interrupt is generated. Once the received message in enqueued on the correct IN queue the driver could look for outgoing buffers on the OUT queue. Unfortunately the window between the time when the protocol has enqueued something and when decides to call device_start() is very small.

There may be a way for the application C-thread technology to make the window larger by delaying the write. It may also be possible to implement a delayed or gang write policy. Delaying the device_write after the enqueue operation not only allows other enqueue operations to piggy back but it widens the window exploitable by the receive interrupt handler described above.

C-Threads may have an application in the receive path as well. In the protocol server we associate a single Mach thread with each IN queue. This limits that amount of parallelism available to any particular protocol family. If the thread for a particular family blocks the device driver must unblock that particular thread. This approach simplified the design and initial implementation of the prototype. However the judicious use of C-Threads may improve the potential for parallelism and reduce the number of IPC messages. Network threads would not be attached to particular protocol queues. Instead, Device events would wakeup one of several network threads which would look in shared memory for an indication of which IN queue it should start with. When there was no more work to be done with the first IN queue the thread would check all the other queues before blocking.

Summary:

The primary goals of this project were to improve the performance and efficiency of protocol serves using portable technology. A secondary goal was to maximize the reuse of the OSF/1 monolithic kernel code base. The preliminary performance results from the first prototype are encouraging. The current prototype provides enough improvement in performance to continue the project. The perturbation of the code base and the additional complexity needed to manage the shared memory data structures was greater than originally expected.

As has been observed by other researchers [8] current practice in hardware design is not in step with the evolving practice in OS design. The lack of compare-and-swap instructions to build synchronizers, the lack of good segmented architectures or equivalently flexible VM architectures, the costs incurred by PIC code, the costs of context switching or system calls, dumb device controllers, and the lack of support for safe access to devices by unprivileged processes contributes to the difficulty of building safe and efficient servers. To avoid exacerbating the situation operating system designers and CPU designers need to play a greater role in one another's work in the years to come.

Acknowledgments:

The authors would like to thank David Black of OSF and Alessandro Forin of CMU for their contributions to the design and implementation of the system. We would also like to thank Keith Loepere and Paul Neves for their help in designing the various synchronization algorithms.

References:

[1] "Unix as an Application Program", David Golub, Randall Dean, Alessandro Forin Richard Rashid, Usenix Conference Proceedings, Summer 1990

[2] "The Packet Filter: An Efficient Mechanism for User-level Network Code", Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta, Proceedings of the 11th Symposium on Operating Systems Principles, ACM SIGOPS, November 1987

[3] "Locking and Reference Counting in the Mach Kernel", David L. Black, Avadis Tevanian, Jr. , David B. Golub, Michael W. Young, 1991 International Conference on Parallel Processing

[4] "Impossibility and Universality Results for Wait-Free Synchronization" Maurice P. Herlihy, Proceedings of the Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, August 1988

[5] Algorithms for Mutual Exclusion, M. Raynal, Scientific Computaion Series, MIT Press, 1986

[6] Operating System Concepts (chapter5 section2), A. Silberschatz, J.Peterson, P. Galvin, Addison-Wesley, December 1990

[7] "User-Level Interprocess Communication for Shared Memory Multiprocessors", Brian Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, ACM Transactions on Computer Systems, May 1991, Volume 9 Number 2

[8] "The Interaction of Architecture and Operating System Design", Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska, Proceeding of the ACM Fourth International Conference on Archetectural Support for Programming Languages and Operating Systems, April 1991