Report Number: WUCS-94-34

1994-01-01

Universal Continuous Media I/O: Design and Implementation

Charles D. Cranor and Gurudatta M. Parulkar

The problem this paper addresses is how to modify an existing operating system's I/O subsystem to support new high-speed networks and high-bandwidth multimedia applications that will play an important role in future computing environments. The proposed I/O subsystem is called universal continuous media I/O (UCM I/O). This paper will cover the preliminary design of UCM I/O, some of the trade-offs and issues that need to be addressed in order to implement UCM I/O, and a summary of work in progress.

UNIVERSAL CONTINUOUS MEDIA I/O: DESIGN AND IMPLEMENTATION

Charles D. Cranor and Gurudatta M. Parulkar

WUCS-94-34

December 19, 1994

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

The problem this paper addresses is how to modify an existing operating system's I/O subsystem to support new high-speed networks and high-bandwidth multimedia applications that will play an important role in future computing environments. The proposed I/O subsystem is called universal continuous media I/O (UCM I/O). This paper will cover the preliminary design of UCM I/O, some of the trade-offs and issues that need to be addressed in order to implement UCM I/O, and a a summary of work in progress.

This work supported by ARPA, NSF, Ascom Timeplex, BNR, Goldstar, NEC America, NTT, Southwestern Bell, SynOptics, and Tektronix.

UNIVERSAL CONTINUOUS MEDIA I/O: DESIGN AND IMPLEMENTATION

Charles D. Cranor
chuck@ccrc.wustl.edu
+1 314 935 4203

Gurudatta M. Parulkar
guru@flora.wustl.edu
+1 314 935 4621

1. Introduction

As computers become more common and powerful, new applications are being proposed. Applications in areas such as distributed computing, imaging, and multimedia are becoming more sophisticated. In order to run properly, many of these new applications require very high levels of performance from the entire computer system. While CPUs and networks have been getting faster at a remarkable rate and can meet the needs of new applications, the I/O system is lagging behind and is starting to become a major bottleneck for these applications.

There are three major problems with the current I/O system:

- The I/O system has poor performance. Solutions to this problem have been devised for special cases (e.g. network I/O), but they have not been generalized for all forms of I/O.
- The I/O subsystem lacks abstractions to support new classes of applications such as those that make use of continuous media.
- The I/O applications programmer interface (API) has become too unwieldy. There are too many I/O calls and they are often non-uniform.

To address these I/O subsystem issues, our high level objective is to modify the traditional Unix operating system's I/O subsystem to support new high-speed networks and high-bandwidth multimedia applications and devices. The proposed I/O subsystem is called universal continuous media I/O (UCM I/O). UCM I/O is called "universal" because it combines all types of I/O into a single abstraction. This abstraction consists of two parts: the API and the buffer management system. In order to achieve this objective, we will introduce four novel ideas: a new I/O API with improved semantics, the use of polling interrupts to improve the handling of continuous media I/O streams, a SuperCall mechanism that allows an application to download I/O critical code into the kernel, and a buffer management system that allows efficient page remapping and shared memory mechanisms to be used between user programs and the kernel (and its devices).

This paper contains the preliminary design of UCM I/O. It details some of the trade-offs and issues that must be considered in order to implement UCM I/O, and it also contains a summary of work in progress.

This paper is organized as follows: Section 2 motivates the need for UCM I/O. Section 3 describes the problems UCM I/O is expected to solve. Section 4 contains an overview of the proposed solutions to the problems described in Section 3. Section 5 describes the functions and data structures that make up UCM I/O's application program interface (API). Section 6 presents the design of UCM I/O's buffer management and its interaction with the virtual memory system. Section 7 describes research related to UCM I/O. Finally, Section 8 contains the current status of the work in progress.

2. Motivation and Environment

This section defines the hardware and software environment in which we plan to work. Assumptions made in this section will hold throughout the paper. In a majority of cases, these assumptions will merely reflect either the state of the art or future trends widely accepted by the research community.

Network Layer. At the network layer, we assume the existence of high speed (150 Mbps and more) packet switched networks such as ATM networks. Prototype testbeds of such networks already exist, and commercial products have already appeared during the past year. We shall also assume that these fast packet switched networks will provide support for multipoint connections, and provide a specified quality of service (QOS).

Internet and Transport Layer. At the internet and transport layer we shall assume the presence of protocols that can provide support for multipoint connections with statistical resource reservations and application oriented flow and error control. Because many research efforts (including our research group's work) are aimed at either creating new internet and transport protocols or extending existing ones to provide such services [5, 25, 9, 19, 23, 24, 18, 22, 11, 4], our assumptions about the underlying protocols are realistic.

Application Layer. The specific class of applications targeted in this research include point-to-point as well as multipoint multimedia applications. Multimedia applications are characterized by their requirement for transport of multiple synchronized media streams, possibly between a number of simultaneously participating and widely distributed hosts. Some of these streams (typically video streams) have high bandwidth and stringent real-time QOS requirements.

It is important to note that these target multimedia applications involve a varied amount of user interaction and may require not only high bandwidth but significant computing power as well [14, 15, 2]. Unlike first generation multimedia system that used special hardware to bypass the operating system to directly display multimedia data, future multimedia workstations and servers must have operating systems that can handle multimedia streams directly so that the application can manipulate and transform the data.

Figure 1a shows several types of end-points that I/O subsystems need to support. These endpoints include traditional I/O endpoints such as disks and tapes and traditional IPC endpoints such as another process. They also include new multimedia devices such cameras and graphic displays. These devices require periodic transfer of data with stringent performance guarantees.

All these types of I/O endpoints can be connected in many different ways. Figure 1b shows some example configurations. These examples include traditional I/O configurations such as process-to-process or process-to-network. Multimedia applications require new types of configurations, such as local-device-to-local-device and local-device-to-network, in which the CPU is not involved in the data transfer. When the CPU is not involved in data transfer, then the I/O API needs to provide a way to set up or "join" the two endpoints so that they can do I/O on their own. The UCM I/O system includes this type of I/O abstraction in its API.

The I/O subsystem in the operating system is complex and interacts with many other OS components such as memory management, network protocols, scheduling, and device drivers, as shown in Figure 2. Thus, modifications to the I/O subsystem involves upgrading other parts of the OS as well. The current Unix I/O environment is not equipped to support all types of I/O efficiently, as explained in the next section.

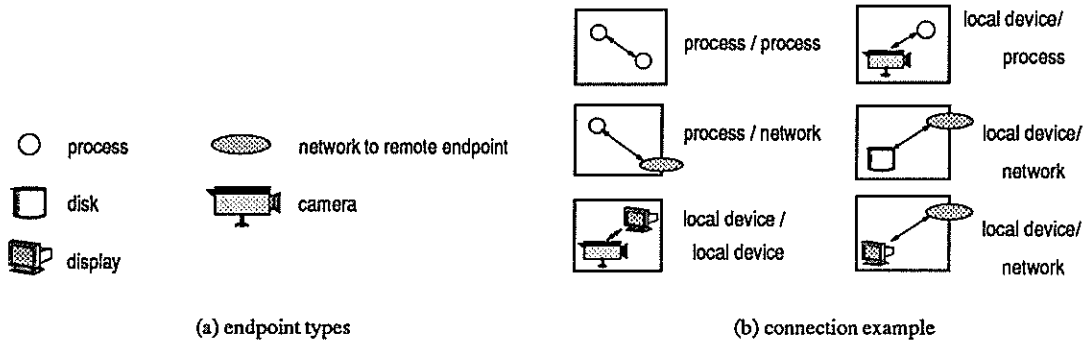

Figure 1: UCM I/O connections

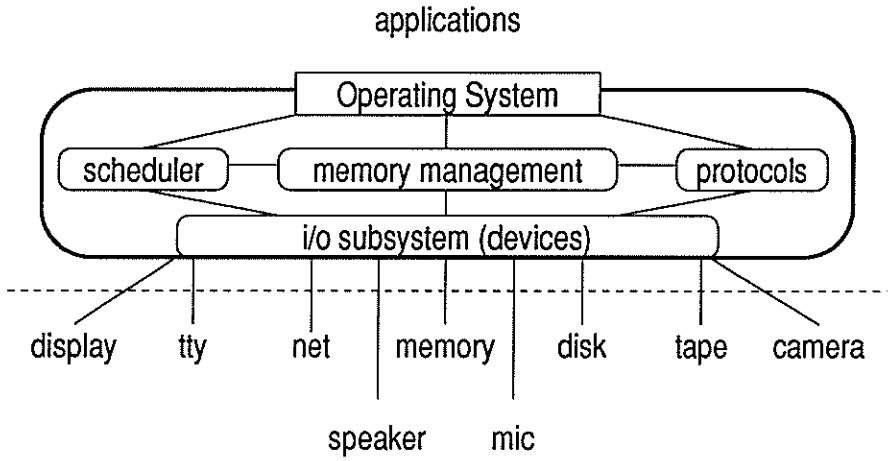

Figure 2: Types of I/O devices

3. Problem Statement

The current Unix I/O API is a cross between file I/O and socket IPC I/O. While this API is flexible and compatible with older versions of Unix, it has a number of weaknesses that need to be addressed for future applications. These weaknesses include the API's unwieldiness, performance problems in the I/O subsystem, and little support for continuous media. The rest of this section describes these weaknesses in more details.

3.1. Non-uniform API with too many functions

An API is uniform if it has the same interface for all types of I/O. The current API is non-uniform. Figure 3 shows an example of this non-uniformity. The shaded box on the left contains six functions that are part of the socket interface for sending and receiving data. The shaded box on the right contains four functions that are part of the file (and device) I/O API. For network I/O any of the system calls in the boxes can be used to

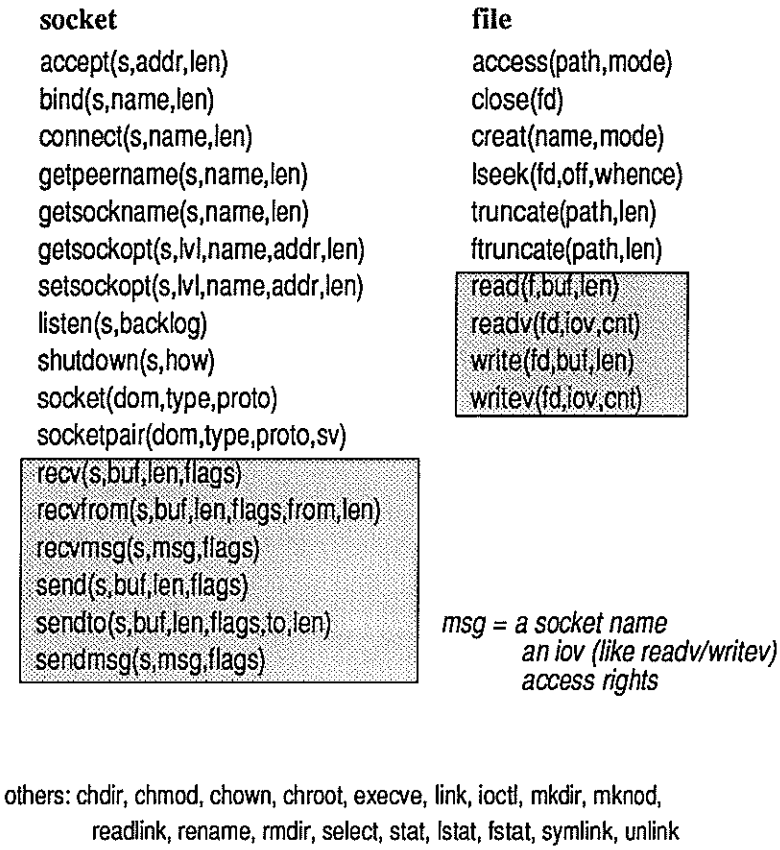

Figure 3: The current API

send and receive data. However, for file I/O only the the four functions on the right can be used to transfer data. Thus, the current I/O API is non-uniform.

Figure 3 lists the I/O system calls in the current API. These system calls can be divided into socket system calls, file system calls, and other system call that change the shape and attributes of the file system tree. The system calls in the shaded boxes are calls that input or output data.

A non-uniform interface leads to extra system calls that perform the same function, but can only be used by certain types of I/O. This requires that there be a lot of extra code in the kernel to support all the interfaces and to check to make sure the right interface is being used. The more extra code in the kernel, the more effort is required to support it. Also, non-uniform interfaces are sometimes harder to learn, and users will learn the minimal amount required and may miss out on features that can be used to improve I/O speed. For example, a `write` system call passes one buffer to operating system, while the `writev` system call can pass multiple buffers in one call. If an application has many small buffers, it can reduce its system call overhead by using `writev` to pass all the buffer in at once.

While the API needs to present a uniform interface to the application, it must be able to handle non-uniformity in the layers below it. For example, it must be able to cope with both in-band and out-of-band control information, it must be able to move data in different device dependent ways, it must handle I/O to devices that have caching and devices that do not need caching, and it should be able to handle each type of I/O stream's state information as well.

3.2. Performance Problem

Adding layers of software to a system reduces the bandwidth available to applications. Usually, at the hardware level there is sufficient bandwidth, however, the effective bandwidth at higher layers is much smaller. For example, in the case of file I/O, after data passes through the device driver layer, the disk file system code, the virtual file system code (VNODE), and finally the system call layer the bandwidth seen by the application is significantly reduced because of the extra overhead imposed by these layers. For network I/O, as the data passes through the network driver layer, the protocol layer, the socket/IPC layer, and the system call layer the same problem occurs. For a network filesystem, the problem is even worse since the data has to pass through both filesystem and network code.

3.2.1. The API and Data Copying. The current API and buffer management scheme make it difficult to avoid data copying for both the reading and writing of data. Consider writing data to an I/O descriptor. The application does a `write` system call with the buffer it wants to write out as an argument. The kernel processes the `write` system call, starts I/O on the buffer, and resumes the application. If the kernel does not copy the buffer and the application reuses the buffer, it is possible that the application will overwrite part of the buffer being sent, thus corrupting the data. To solve this problem, the kernel must either copy the data, or protect the pages than contain the data while the data is in transit. Because this involves virtual memory operations, it can be expensive.

The same sorts of problems arise when reading data. For example, consider an application that is reading data from a network connection. The application passes the address of the buffer in its address space that is to receive data to the kernel with a `read` system call. If the data has already arrived or is cached it must be copied into the application's buffer. If the data has not arrived yet then the application must sleep waiting for data. When the network interface receives the data it has no way of knowing which process the data is for (unless you have an advanced ATM interface that does multiplexing on VCI). So, the network interface puts the inbound data into an arbitrary kernel buffer. To store the data into the application's buffer the kernel must copy it.

Past work to solve these problems in the I/O system has been specialized and not integrated into the operating system. For example, in some implementations of the network file system (NFS) the overhead of extra software layers is bypassed by having a special non-layered protocol function that does all of UDP and IP in one pass. There are also systems that attempt to use virtual memory to reduce copying. Some of these systems, such as the device driver for the SGI FDDI card, rely on adding extra semantics to standard `read/write` system calls just for networking connections. Other systems such as FBUFS focus on data movement without specifically addressing the API issue[8].

3.2.2. Interrupt and context switching overheads. An important part of the cost of I/O is the expense of interrupts and context switching. Often I/O bound applications enter a loop reading data from one I/O descriptor only to immediately write it back out to another I/O descriptor. These applications are characterized by doing little processing and making a large number of I/O system calls. Performing repeated system calls is expensive due to the overhead of switching back and forth between kernel and user mode and the copying of data. Efforts to address this problem in the networking field have focused on protocol processing overhead. This effort has been successful. Currently protocol processing costs on the order of one hundred instructions for the common case[16]. But it is important to note that the operating system overhead of context switching and interrupt handling is on the order of thousands of instructions. So, there is still room for improvement.

3.3. No support for continuous media

Another problem with the current API is that it was not designed to support continuous media. Continuous media has two main requirements: QOS support and efficient periodic data transfer.

QOS support consists of QOS specification, reservation, and enforcement. QOS specification consists of providing hooks in the API for setting QOS requirements and then translating them into a format the underlying hardware can understand. Once the QOS has been specified, then the resources can be reserved. Then, while data is being transfered, the network and operating system must make sure the process does not use more resources than it is entitled to. New networks like ATM are being designed to provide QOS support for various applications. However, there is no standard way in the API for the application to provide its QOS specification to the operating system. Without this information the operating system cannot do resource reservation on the local machine and request resources from the network.

Continuous media also needs efficient periodic data transfer. The current I/O subsystem requires the application to explicitly make a system call to transfer data. This is expensive and wasteful because with continuous media the application knows in advance the period at which data will be sent, but it does not take advantage of this knowledge.

4. Proposed Solutions

Based on the problem statements in Section 3, our objective is to design and implement a UCM I/O subsystem with the following features:

- a clean and uniform I/O API
- a high performance I/O subsystem with minimal data copying and system calls
- support for continuous media including QOS specification and periodic data transfer support

The UCM I/O structure is shown in Figure 4. The application uses the UCM I/O API interface to perform I/O. To accompany this new API, a new data movement facility is needed in the kernel. This facility needs to have a clean interface with the new API. The UCM I/O system will be an integral part of the virtual memory system and software caches and will support as many types of devices as possible.

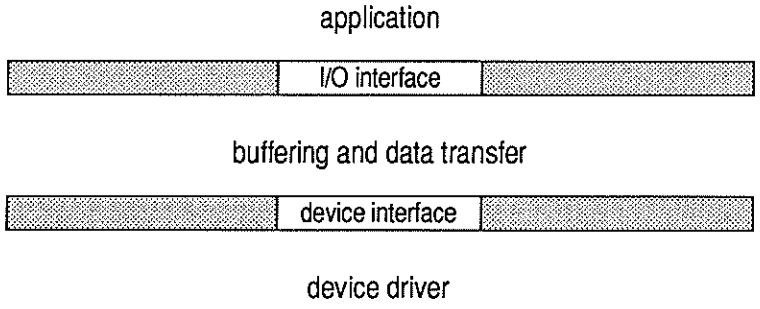

Figure 4: UCM I/O layers

The four most significant ideas behind UCM I/O that will contribute to UCM I/O's meeting of its objectives are:

- a new I/O API that has a small number of functions, can support a variety of I/O devices (traditional as well as continuous media), can work with different underlying buffer management systems, and can support all the I/O semantics of application programs

- the use of clock interrupts for polling I/O devices and user programs in order to support periodic data transfer and to reduce the number of asynchronous interrupts and system calls
- a supercall mechanism that allows the kernel to run an I/O program on behalf of an application, thus reducing the number of system calls and data copying operations performed
- a buffer management system that works with different I/O devices and allows efficient page remapping and shared memory between user programs and the kernel (and the devices controlled by the kernel)

The last three ideas lead to efficient support for continuous media devices and significant performance improvements for I/O subsystem as a whole. These ideas are briefly described in this section and then elaborated on in subsequent sections.

4.1. New I/O API

The UCM I/O applications program interface consists of a small number of functions that take general data structures as arguments. For example, the function that sends data from the application to the kernel takes buffer descriptors, and addresses as part of its arguments. Simpler backward compatible functions can be implemented as library functions rather than systems calls to keep the kernel interface focused.

UCM I/O provides flexibility to applications and devices. This is done by absorbing some complexity in the buffer management of UCM I/O. When outside layers send data into UCM I/O they have three options:

1. The application can force the UCM I/O layer in the kernel to copy data between application and kernel memory. In this case the application can provide any buffer it wants. The buffer is flagged "must copy" to the API interface. The kernel then allocates a kernel buffer and copies the data into it. After this is done the application retains control of the buffer. This is how the current API works.
2. The application can allow the kernel to remap the data (in effect the application is giving away its buffer). The application must specify a page (or pages) as a buffer. The kernel then uses the virtual memory system to remove the buffer from the application's address space and the application no longer has access to the buffer. In this case the data flows into and out of the application's address space. (It is also possible to mark it "copy-on-write"). The UCM I/O API also provides the application the option of having the kernel allocate a new buffer to replace the one that was sent.
3. The application can allow the kernel to choose whether the data buffers should be remapped or copied. The kernel can base its decision on buffer length, memory availability, and the alignment of the buffer. Because this decision is machine dependent it is important that it be made by the kernel, rather than the application. The application must be able to handle both remaps and copies (the first two cases).

When data is in the UCM I/O system, the I/O system owns the data buffers. This allows it to let the outside layers either copy or remap the buffers depending on the size of the data (UCM I/O can provide a hint as to which is more efficient). Note that to use the new features of UCM I/O the application's semantics may change.

UCM I/O also supports continuous media by providing for QOS specifications in the API. These specifications are given at I/O descriptor creation time. This allows for multimedia applications and for application oriented flow and error control in protocols. To meet the requirements, resources must be allocated on the network and on the host, and enforced on the OS using a real-time OS scheduling mechanism. There also needs to be an interface for changing the attributes of a descriptor. This can be combined with normal file system attributes. It should be noted that QOS enforcement is beyond the scope of this research.

4.2. SuperCall

One problem that many applications have is that they have to do system calls repeatedly to get something done. Consider a file transfer program. The core of the program simply reads a block of data from one I/O descriptor and writes it out to another I/O descriptor. Each read or write of data costs one system call, and two data copies. This is very expensive given that the application does not even need to look at the data. One solution to this problem is to use a SuperCall. A SuperCall is a way of passing a small routine that is interpreted in the kernel. Figure 5 shows an example file transfer SuperCall routine. The kernel can execute a SuperCall in one system call and avoid mapping the data into the user's address space by using the UCM I/O API described in Section 5.

```
1: READ from file
2. if RET_VAL == 0 return 0
3. if  RET_VAL < 0 return -1
4. WRITE data to socket
5. goto 1
```

Figure 5: A SuperCall routine

There are several issues involved in the implementation of SuperCalls. First, an instruction set must be determined for the interpreter. The design of this instruction set must take into account how buffers will be exchanged and protected between the application and the kernel. Since the instructions will be interpreted at run time in the kernel, it is important that the kernel does not access an invalid area of memory due to an application error.

4.3. Clock Interrupt for Polling

UCM I/O adds support for continuous media to the operating system. In traditional I/O, the only way to trigger a data transfer is with a system call. This is not efficient for continuous media because it does not take advantage of the periodic nature of the data stream. UCM I/O provides a way to take advantage of the periodic nature of continuous media to transfer data. It use a circular pool of data buffers as shown in Figure 6. The buffer pool can be used to transfer data between the application and kernel without the need of a system call. It is accessed by both the application process and the kernel at the same time. The application arranges for the kernel to check the circular buffer on a periodic basis. The application can ask the kernel to operate in one of two modes. In one mode of access, the kernel polls a tag in the shared buffer area to see if data needs to be sent. In the other mode, there is no tag and the kernel always sends the data at the polling interval. The main difference in cost between the two modes is that the first mode requires one access to the tag area per polling interval, and the second mode does not. However, in the second mode, if the application does not meet polling interval the kernel may send an invalid data buffer.

Periodic interrupts from the clock chip facilitate polling by the kernel. These interrupts are used for keeping the time of day, process scheduling, and timeouts for periodic events. One of the new periodic events can be to poll these shared buffer pools to see if data needs to be sent. By folding the polling of the tag into an already existing interrupt, cost is minimized.

4.4. Shared Memory and Efficient Page Remapping

UCM I/O improves the I/O performance of the operating system by reducing overhead. One way to reduce overhead is with the new buffer management system that allows data buffers to be aligned on page boundaries.

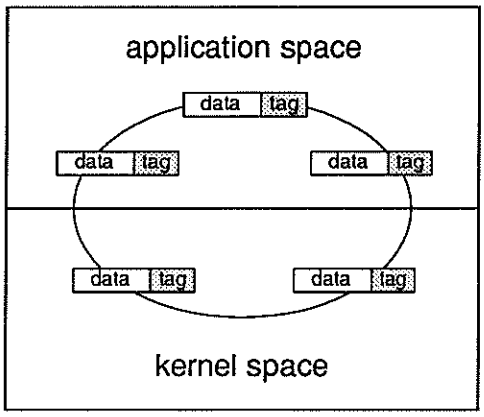

Figure 6: A pool of buffers

By aligning data buffers this way, the virtual memory hardware of the computer can be used to move data without copying it. In effect, the application and the kernel are sharing the data buffer rather than having their own separate copies of it. However, this arrangement does not make efficient use of memory due to page alignment constraints and it requires the virtual memory system to support fast and efficient remapping of pages.

Normally, an application's buffer resides in virtual memory that can be paged out to disk. If a buffer is going to be used for I/O frequently, it should be "wired down" to prevent page faults and mapping changes. To help with this, UCM I/O allows a process to allocate a pool of buffers that can be wired down so that there is always a buffer available for data. In this case, the pool of memory is shared statically between the application and the kernel (rather than remapped on a per I/O call basis). One drawback of this feature is that the user is given more of a chance to abuse the virtual memory system by tying down lots of memory.

5. UCM I/O API

This section presents the data structures and functions that make up the UCM I/O API.

5.1. Data Structures

The main UCM I/O data structure, `bufio`, holds pointers to buffers in an application's address space. The structure consists of an array of buffer vector structures and an integer representing the number of buffer vectors structures stored. In the example shown in Figure 7 there are three buffer vectors stored in the `bufio` structure. Having more than one buffer vector allows an application to do scatter-gather I/O.

A buffer vector is a data structure that describes one single buffer. Shown in Figure 8, a buffer vector structure has four fields. The first field holds a pointer to the buffer area. The second field holds the length of the buffer area. The third field holds flags that indicate whether the data is copied or remapped on data transfer operations. The final field is the replacement or new buffer field (described in Section 4.1).

Generic data structures such as the buffer vector structure are useful because they can describe almost any part of an address space without requiring the programmer to collect the data in one place. They also allow multiple buffers to be specified at once. On the other hand, they are not as easy to incorporate into application programs as single buffers. Functions that use a simpler buffering data structure can be added to the C library.

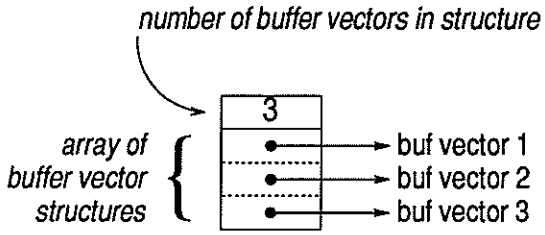

Figure 7: An example `bufio` structure

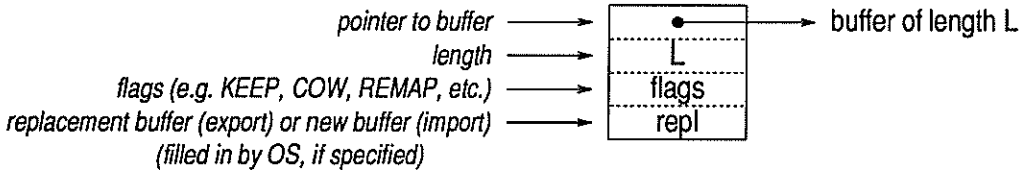

Figure 8: An example buffer vector structure

5.2. Functions

In this section the various API functions that are part of the UCM I/O are described. The UCM I/O API is more powerful than the Unix API. By increasing the uniformity of system calls the total number of system calls has been decreased in the UCM I/O API. The UCM I/O system calls are somewhat more complex in terms of the number and structure of arguments than the traditional Unix calls. However, this complexity can be hidden by a user level library, if needed.

5.2.1. Buffer allocation functions. Although UCM I/O can work with any buffer, optimal page-aligned wired-down buffers are necessary to fully exploit UCM I/O's strengths. Buffer allocation functions are provided for creating these buffers. The `balloc` function can be used to allocate a buffer in units of pages. There is an option that will allow the buffer to be wired down. The `bfree` function can be used to free one of these buffers when they are no longer needed. Because the `balloc` function can use up all physical memory, a mechanism that limits the number of buffers a process can have will be included.

5.2.2. Export function. After studying the current API calls, the `export` function was created. It has all the features included in the current I/O API, and additional features designed to meet it was also was designed to meet the needs of future I/O applications. The format of the `export` function is shown in Figure 9. The `export` function takes an I/O descriptor, a `bufio`, flags, and an optional destination address buffer and exports the data buffers described by the `bufio` structure from the application to the kernel. The `export` function allows three options when exporting data buffers as explained in Section 4.1.

If an `export` function remaps a buffer out of an application's address space, then the application will need to allocate a new buffer later. The `export` function has a feature that allows this new buffer to be allocated as the old one is shipped out. If the correct flag is set in the buffer vector structure, the "replacement buffer" field of the structure is set to a pointer to a new buffer of the same length as the exported buffer.

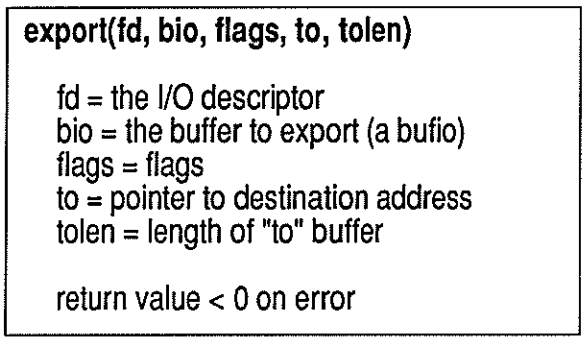

Figure 9: Export function prototype

As can be seen from the above, the `export` function is highly flexible. It allows for simple I/O, or complex scatter-gather I/O. It is also easy to build a compatibility library on top of it to implement the old API.

5.2.3. Import Function. The `import` function is the opposite of the `export` function. The `import` function reads data using two alternative methods. In the first method, the application specifies where to read the data from. For example, it could specify a specific block of a disk file. In the second method, the application passively waits for data such as a network packet to arrive. The format of the `import` function is shown in Figure 10. The `import` function takes an I/O descriptor, a `bufio` structure, and some flags. It may also contain an optional buffer that stores the source of the data. If the first method is used, the source is a memory address. If the second method is used, the source is the address of the host or endpoint that sent the data. Like the `export` function, the `import` function allows the application to specify how it wants the buffers handled.

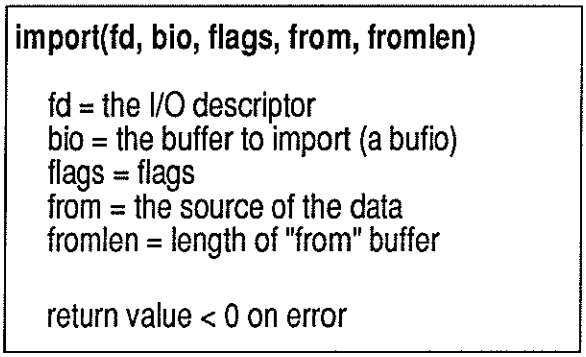

Figure 10: Import function prototype

5.2.4. Opening a I/O Descriptor. The current API has two main paths for opening an I/O descriptor. One is the traditional file I/O path, and the other is the more generic socket I/O path. These two paths are shown in Figure 11. Traditional file I/O uses the `open` call with a file name, flags, and the protection mode to create the file with. Note that the open call can also be used to open a device in `/dev`, rather than a file. Socket I/O requires two or three system calls. First, the `socket` call creates an unconnected socket that can

be optionally bound to a local address using the `bind` call. The `connect` call is then used to connect the socket to a remote endpoint. The `connect` call takes the I/O descriptor, a pointer to a buffer, and the buffer's length. This buffer is a `sockaddr` structure that contains the address of the remote endpoint. The format of the `sockaddr` structure depends on what type of socket is being used. For internet sockets, the `sockaddr` is a `sockaddr_in` structure where it contains an IP address and a port number. For Unix sockets, the address is a `sockaddr_unix` and it contains the file name of a Unix domain socket.

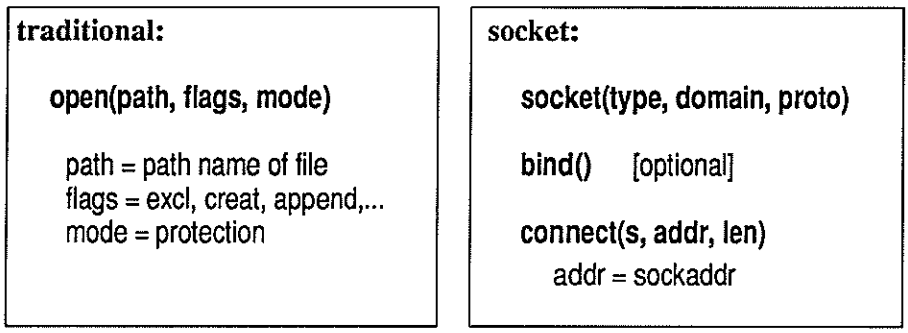

Figure 11: Opens

The socket based open is more general than the traditional that file one because it can allow for "connections" to more than just files and devices. UCM I/O combines socket I/O with traditional `i/o` to create a new `open` call for the UCM I/O API. The format of this call is shown in Figure 12. The new `open` call takes a type, domain, and protocol, just like the `socket` call. These parameters are used to select what type of I/O endpoint will be used. The `addr` and `len` fields are used to specify the `sockaddr` structure. In order to support normal file I/O, a new type of `sockaddr` will have to be created. The `sockaddr_file` structure contains a file name and an optional offset.

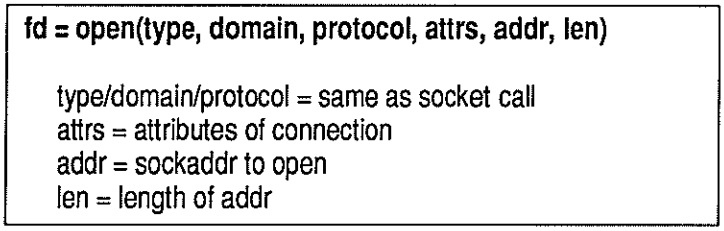

Figure 12: New open

The attribute field (`attrs`) in the new `open` call is used to point to a structure that contains the attributes of the I/O descriptor. This is a generalization of the `flags` and `mode` arguments in the traditional `open` call. Generalizing these parameters creates a natural place to specify attributes for an I/O descriptor. For example, for normal file I/O, the attribute structure can be a pointer to a structure that contains the flags and protection mode of the file. For network I/O the attribute structure could be null, or for connections that require performance guarantees the attribute pointer could point to a structure that contains them.

This simplifies state management since the application's interface to descriptor dependent state information can be centralized into one place in the API rather than spread out over several calls. For example there are several system calls to change a file's attributes (e.g. change protection, ownership, lock file). For sockets

there is a "set socket options" call. All these interfaces to I/O attributes can be bundled into one universal API call.

5.3. Continuous Media I/O

This section describes the support UCM I/O has for continuous media I/O. In continuous media I/O data is being sent on a periodic basis. UCM I/O takes advantage of the periodic nature of continuous media I/O to improve performance.

5.3.1. Continuous media data structures. In UCM I/O, continuous media I/O is built using the buffer allocation system and `import/export` system call interface. Logically there is a ring of pre-allocated buffers (a circular array) on which all I/O is performed. This ring is described by a control buffer data structure. The format of the control buffer data structure is shown in Figure 13. For efficiency a control buffer must be stored in a wired down area of virtual memory, so it is best to store it in a buffer allocated with `balloc`. The control buffer contains a fixed length header and an array of arguments for either `import` or `export`. Each element of the array describes one buffer in the circular array. The fixed header part of the control buffer contains several items. It contains two pointers into the circular array. These pointers indicate which buffer the kernel and application are working on. Because there is only one writer[1] to a pointer there is no need for a semaphore to protect the pointers. When sending data, the application's pointer is ahead of the kernel's pointer. When receiving data, the application's pointer is behind the kernel's pointer. The length field in the fixed head tells the kernel how many buffers are in the circular list. The "time to poll" field tells the kernel how often it needs to check the control buffer to see if there is something ready to send. There is also a place to put `import`, `export`, and other flags.

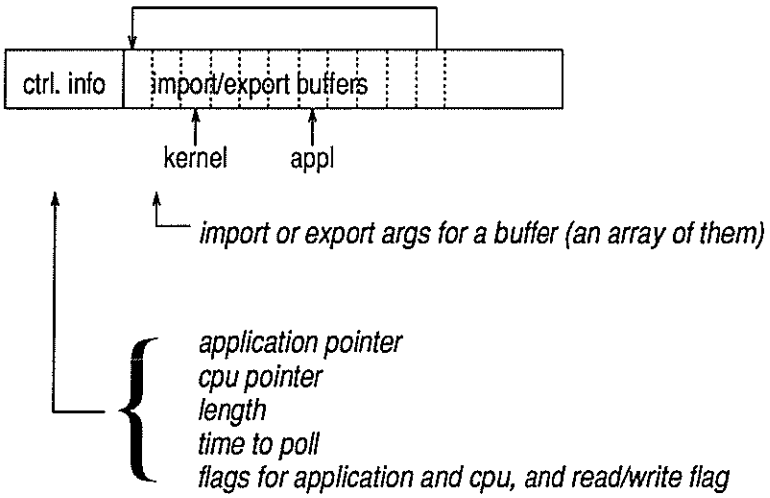

Figure 13: Control buffer format

5.3.2. Continuous media functions. To start UCM I/O's continuous media I/O a control buffer must be allocated and initialized properly. Then the application can call `auto_io` on the control buffer to start the I/O

[1] there can be any number of readers

process. To stop the I/O process the application can either exit, close the I/O descriptor, or call `stop_io` on the control buffer.

Once the `auto_io` is in progress the application can send data on a periodic basis without performing any more system calls. If an application wants to generate a continuous media stream it just has to fill one of the buffers in the ring and update the control information. The kernel will poll the control information in the clock interrupt to see if data needs to be sent. Since the control buffer is wired down the polling overhead should be minimal because control buffer memory accesses will not cause page faults. UCM I/O does not provide any quality of service guarantees in the kernel, however, these guarantees may be provided by a scheduler[13].

5.4. Super Calls

UCM I/O also improves application performance by reducing the number of system calls by allowing them to be aggregated into "super" system calls or SuperCalls. This is useful for applications such as data transfer programs and daemons whose execution time is spent mostly in system calls. A SuperCall is a short program passed into the kernel for interpretation. This program can include multiple system calls.

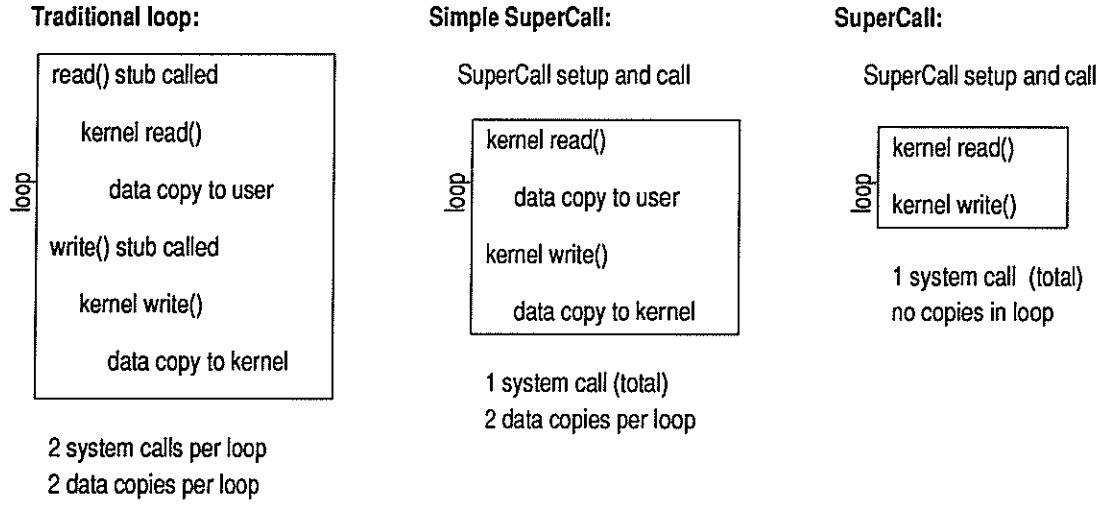

Figure 14: SuperCall

Figure 14 shows how a SuperCall can reduce the cost of a file transfer loop. The left-hand side of the figure shows the cost of the file transfer loop if a SuperCall is not used. The cost consists of two system calls and two data copies per loop. The number of loops executed depends on the size of the buffers being used and the size of the data being sent. A system call consists of a stub that is called by the user program and a function in the kernel (usually with the same name as the stub) that is called on behalf of the user to perform the system call. A naive implementation of a SuperCall would simply take the SuperCall program and call the kernel routine directly, as shown in the middle of Figure 14. While this is not difficult to implement, it is still expensive because the kernel routines will still be copying (or mapping) the data into and out of the user space. A more efficient scheme would be for the SuperCall to only copy the data between the user and kernel at the ends of the SuperCall, as shown in the right-hand side of the figure. While this is more efficient, it is also much harder to implement because the kernel usually defers copying until the data is actually needed. One of the main design challenges of the SuperCall system is that it needs to be designed in such a way that useless copying is eliminated. This issue must be taken into account when deciding where and when the `import` and `export` system calls move data between user and kernel space.

The SuperCall instruction set must also be designed with ease of use in mind. Application programmers will not use SuperCalls if they require too much effort to understand and use. One important issue that will influence the design of the SuperCall instruction set is when to copy data between kernel and user space. Current system calls defer this copy until they are deep within a system call. Figure 15 shows the copy depth for two system calls. For the `open` call the copy in of the file name happens three levels down into the code. For a `write` call the copy in happens four levels down. When a system call is being executed as part of a SuperCall the copy may or may not need to be done. The layering of the kernel code will have to be re-thought in order to implement SuperCalls efficiently.

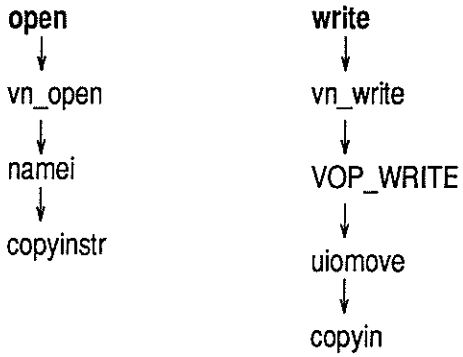

Figure 15: Copy depth

Another issue that must be considered is the single-threaded nature of many operating systems. Because a process running a system call cannot be preempted by another process, the SuperCall driver must be careful to check to see if the process needs to be put to sleep for a while. Combining SuperCalls with UCM I/O will hopefully lead to very efficient I/O performance.

6. Buffer Management and Virtual Memory

Buffer management is an important component of the UCM I/O system. The buffer management system:

- supports various application data transfer semantics such as the three options described in Section 5 (copy, remap, and copy/remap).
- provides efficient page remapping to avoid data copying.
- provides shared memory between applications and the kernel to reduce copying and virtual memory management overhead.
- supports I/O endpoints that use polling, a variety of DMA schemes, caching, and in-band or out-of-band control information.

This section contains the initial design of the UCM I/O buffer management system. Section 6.1 describes how the buffering system will support API semantics. Sections 6.2 and 6.3 describe how the UCM I/O buffering systems interact with the file cache and control information, respectively.

6.1. Supporting API Semantics

The UCM I/O API offers a wider range of options than the traditional API. This section describes the buffer management options that support the new API semantics. The new `import` and `export` operations require several system and virtual memory related operations. These operations are pictured in Figure 16. For an `export` operation, the I/O descriptor and buffer description are available from the system call parameters. These parameters must be checked by the kernel for valid data. The I/O descriptor must be converted to a file pointer that points to a data structure which contains all the state information about the open I/O descriptor.

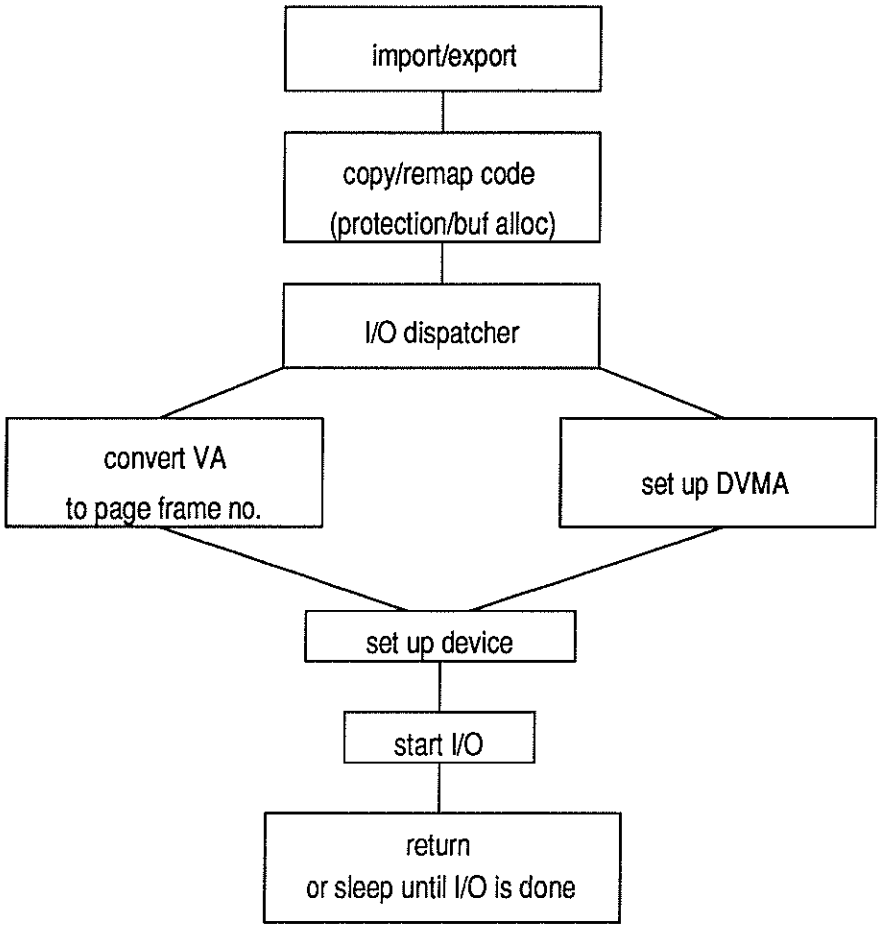

Figure 16: VM operations

For connectionless I/O operations, a "route" to the destination must be determined through a filesystem file lookup, or a network address translation. If the I/O descriptor is already connected then the output destination information was determined at I/O descriptor connection time and is stored in the structure pointed to by the file pointer.

Once the destination is determined the data buffers must be processed. Each buffer can be either copied or remapped. For buffers that must be copied the kernel must allocate a kernel level buffer to copy in the data. This involves allocating a buffer of kernel virtual memory and some physical memory to back that kernel virtual memory. If the data is to be remapped, a destination kernel virtual address must be determined.

Then the processes address space description must be locked and the page table entries must be adjusted. The translation lookaside buffer (TLB) and data cache may have to be flushed. After the copy or remap operation is done, the UCM I/O system will own the data buffers.

Once the data is in the UCM I/O system, it may need to be processed by one or more parts of the I/O dispatcher. The internal structure of the I/O dispatcher is shown in Figure 17. When the I/O dispatcher receives a data request it performs an operation on it and dispatches the I/O request. There are several ways an I/O request could be dispatched. For example, for protocols, the I/O dispatcher may prepend in-band header information on the buffer and then dispatch it to a network device driver. Or, for disk files, the I/O dispatcher may first try and satisfy the I/O request from a cache. If the data is in the cache the I/O dispatcher can return the data immediately. However, if the data is not in the cache, the I/O dispatcher can place an I/O request with a device driver (possibly caching the result). By using the I/O dispatcher concept it is possible to integrate different I/O systems together into one abstraction and still provide all the flexibility that is available in today's systems. Note that the full semantics of the I/O dispatcher have not been fully defined yet.

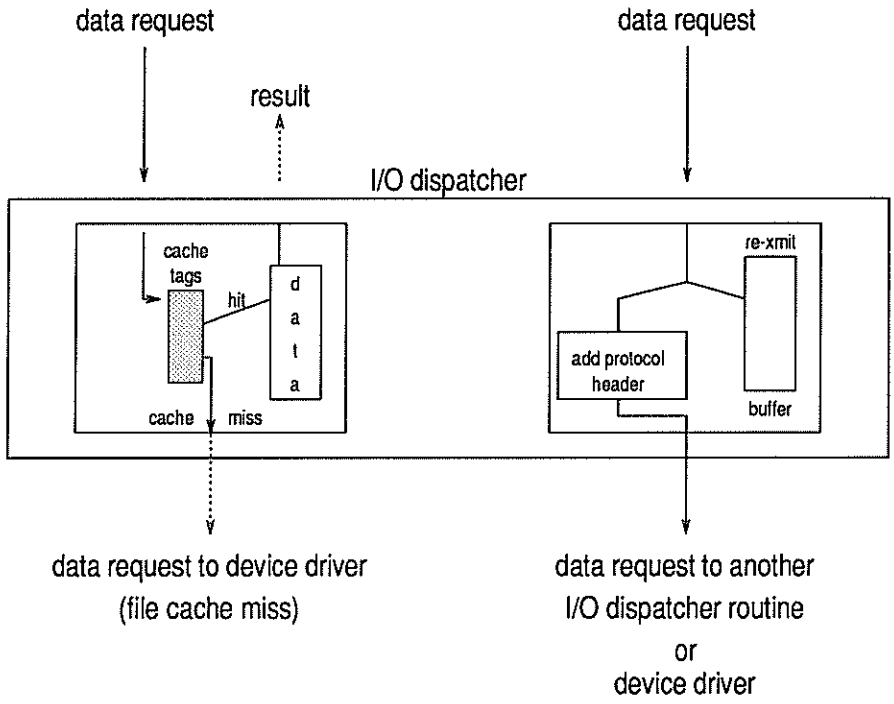

Figure 17: Two example paths through the I/O Dispatcher

Once the data is filtered and is safely wired down in kernel virtual memory the I/O device must be programmed. For DMA transfers this involves setting up the buffers and initiating the transfer. For programmed I/O there is no setup; the kernel must explicitly load the data. For I/O to another process some of the virtual memory operations described above must be performed.

If the sending process requests new buffers, then UCM I/O allocates and maps them. If so, it must do the allocation and map in these buffers. The process itself may have the option of going to sleep to wait for the I/O to finish or it may continue without waiting for the I/O.

6.1.1. Shared memory. The UCM I/O buffering system supports memory being shared between the kernel and the application. To use shared memory, an application must allocate a ring of buffers. The application then accesses these buffers one at a time using the remapping interface of the `import/export` API. The shared memory buffers appear to the application to be remapped following the `import` or `export` system call. However, the buffers actually remain mapped into both the application and kernel address spaces. As a result, a remap and its associated overhead are avoided. For this system to work properly the application must be careful not to modify buffers that it has passed to the kernel using the `import` or `export` system calls. Fortunately, if the application is careless it does not affect the rest of the system.

6.1.2. SuperCalls. In an ideal world, a SuperCall should be able to call any system call and return the results of the SuperCall to any memory location in the application's address space. This is a difficult process to implement given that copies of data between the application and the kernel are spread throughout the kernel. So, the implementation constraints of the UCM I/O buffering system may force extra constraints on the design of the SuperCall mechanism.

6.1.3. Performance. To have efficient I/O, the buffering system described above must be efficient. UCM I/O's buffering system has been designed to achieve this efficiency requirement by:

- avoiding expensive data copying where ever possible. The `import/export` API of UCM I/O allows the buffering system to choose when to copy small chunks of data and when to remap large chunks of data.
- minimizing the software cost of page remapping as much as possible. For example, to do DMA the kernel needs to be able to convert from a virtual address to a physical address quickly. An inefficient software implementation could perform a software-based memory management unit (MMU) tree walk. A more efficient software implementation might be able to map the page table entries (PTEs) in such a way that the conversion can be done with some simple address arithmetic, thus reducing the software overhead of page faults from the order of thousands of instructions to hundreds of instructions.
- using shared memory between the application and kernel to eliminate the cost of copying and remapping data between the user and kernel.
- allowing the I/O dispatcher to provide caching and to perform standard algorithms such as protocol header prediction.

6.2. UCM I/O and the File Cache

For efficiency, parts of data files are cached in RAM. UCM I/O buffer management must interact properly with the file cache. When data is read or written the kernel first checks to see if the pages requested are in the cache. The data is updated after the buffering system flushes the page from the cache. This process is shown in Figure 18. When remapping pages UCM I/O must be careful not to map out a page from the file cache leaving the cache empty for the next request. It should also be noted that for files containing continuous media it may not be useful to cache data block since these files are often traversed sequentially. This can be controlled with a flag as part of the I/O descriptors attributes. (The caching code in the I/O dispatcher will interpret this flag.)

UCM I/O buffer management is designed to use the same mechanism that the `mmap` system call uses to handle the file cache. For read-only access the page can be mapped read-only by as many processes as necessary. For write access if changes should be private, then a copy-on-write scheme should be used. If changes should be shared by the whole system, then the same page can be mapped read/write into as many processes as necessary.

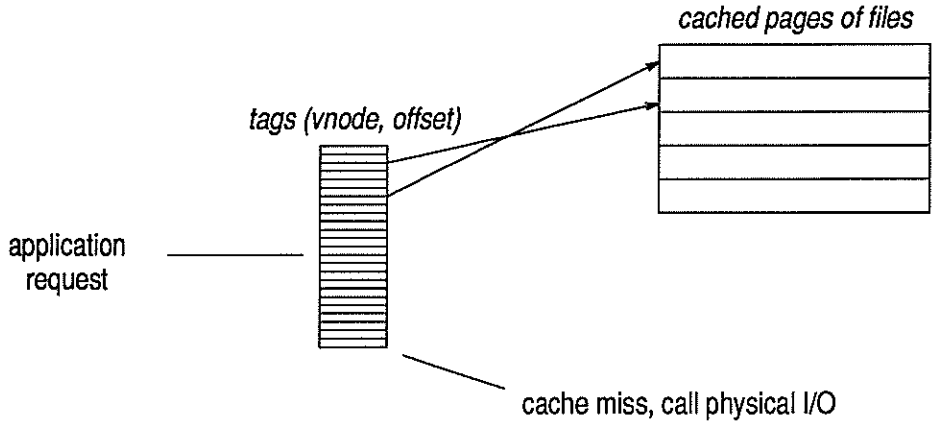

Figure 18: File cache

6.3. Out-of-band vs In-band Control Info

There are many ways that control information is passed between endpoints in an I/O operation. Devices such as disk controllers pass control information separately from the data, where as devices such as a network interface pass control information in the data stream. For example, ethernet interfaces pass the ethernet, IP, and transport protocol headers in the data area. However, as long as the data is not larger than the maximum transport unit this is not a major problem. a problem may occur if the data is so large that gets fragmented into multiple packets. When these packets are received it is not possible to reassemble the data without having headers embedded in the middle of the data area.

6.3.1. Protocol Processing. UCM I/O has three buffering options: copy, remap, and copy/remap. UCM I/O's handling of these cases in the context of protocol processing is described below.

Copying: This case can be handled in a traditional way. UCM I/O will copy packet data if the application requests it, or if the application leaves the copy/remap decision up to UCM I/O. On output, UCM I/O can save a copy of the data buffers for retransmission by incrementing an "in use" reference count on the buffer pages. On input the data can be copied as needed. Headers can be stored in a separate buffer, or copied into the main buffer. Buffers saved for retransmission are released when an acknowledgment is received by the I/O dispatcher.

Remapping (transmit): To transmit a remapped buffer, header data must be prepended to the front of the buffer. There are several ways to do this: the application can leave room in the front of the buffer pages for the header, if scatter/gather I/O is available then a separate buffer can be used for the header, or if DVMA is available, then the header can be mapped in front of the data pages in the kernel virtual address space. Buffers can be held for retransmission with a reference count. For shared memory, these buffers should not be touched by the application (at the risk of corrupting the connection). If applications obey the semantics of the API this will not be a problem. Note that if the application misbehaves, it causes problems for itself but not the rest of the system.

Remapping (receiving): When data comes in from the network the headers are usually in the front of the buffer. When the buffers get mapped into the application's address space the headers get mapped as well. Because the application is not supposed to access header information the application's pointer

is set to point directly to the data. In this case the application could access header data if it looked in the front part of the buffer. However, this would be a violation of the semantics of the API, and the application should not expect valid data in that area. Another way to deal with this problem is to have the network interface hardware separate the headers from the data as data arrives (as proposed in the APIC interface).

Copy/remap: This case is simply a superset of the previous cases.

Note that the copying and remapping phase of data I/O is done before the data reaches the I/O dispatcher. The I/O dispatcher has control of the data it receives from the copy and remapping layers. Also, note that for the remapping case, the data may reside in a shared memory buffer. In that case the data may not actually be remapped (but to the application the semantics of the I/O operation are the same as a remap).

7. Related work

This section provides pointers to work related to this research.

Historic Unix I/O and IPC API. The Unix I/O API[17] started off with just a few functions and the concept of "everything is a file." These functions included `open`, `close`, `read`, `write`, and `ioctl`. For interprocess communications the `pipe` system call was provided.

The BSD socket IPC system was added to Unix to support networking protocols such as TCP/IP. The socket API is very flexible and supports both connection-oriented and connectionless network I/O. It uses its own buffering system that consists of a data structure called an `mbuf` which allows data to easily be added to the ends of a buffer. All the API weaknesses presented in Section 3 apply to the BSD API. For example, the socket interface does not use advanced buffer management techniques such as page remapping, instead it always copies data. It also has no support for continuous media or file I/O.

New operating systems such as Plan9 from Bell Labs attempt to get back to the "everything is a file" I/O model and the `read`/`write` API interface. However, Plan9 does not support a page remapping based I/O API or continuous media data streams.

Uniform I/O Interfaces. One often cited work in the area of uniform I/O interfaces is the UIO interface of the V operating system[3]. The UIO work creates a framework into which all I/O is supposed to fit. However, the UIO effort focuses on providing support for a distributed environment based on remote procedure calls or message passing. It provides uniform interfaces for locking, data replication, and atomic transactions, but it does not address current issues such as multimedia data streams or the reduction of data copying.

VM Systems. Older Unix VM systems emulated the VAX hardware system so that they can use VAX memory management software. As VAX hardware became less popular the VM system became dated and inefficient. Two VM systems that have replaced the old VAX based systems are the SunOS VM system[10] and the Mach VM system[21].

The SunOS VM system steps away from the old VAX hardware and supports a machine-dependent/machine-independent layer system in a more portable way. It merges file and buffer cache memory management into one system and supports the BSD memory map (`mmap`) interface. SunOS style shared libraries were first built on top of this VM system. The SunOS VM system does not directly support page remapping, and uses the traditional I/O API.

The Mach virtual memory system is from the mach operating system. It has been imported into the 4.4BSD operating system. It is a portable VM system that requires only one machine dependent module (the "pmap" module). We expect to modify and replace parts of this VM system as UCM I/O is implemented. Specifically, it needs to be modified to provide efficient remapping and sharing of memory between the kernel

and application processes. While the full details of these changes have not been worked out, it is clear that they will be non-trivial.

FBufs. The FBUF system[8] is a data transport service which is part of the *x*-kernel. The *x*-kernel is built into the Mach 3.0 operating system and provides support for protocol composition in a microkernel based operating system. The FBUF system consists of only a data transport service which uses page remapping, and doesn't address API issues. It also does not provide system call free I/O for continuous media or a way to perform multiple I/O operations in one system call (e.g. SuperCalls). Also, the FBUF system is not very portable and has not yet been generalized to work with non-network based I/O.

Network Interfaces. Previous work at Washington University in network interfaces focused on shared memory. The Axon project[22] defined an I/O model based on shared memory segments. These memory segments are "streamed" between machines using an application-oriented lightweight transport protocol. The idea of "segment streaming" is a precursor to the idea of continuous media streams[12].

The current trend with new host network interfaces is to make them more intelligent. For example, the Silicon Graphics FDDI interface is a newer host network interface which takes advantage of virtual memory techniques without changing the API. On packet sending, the buffer is marked "copy-on-write" and passed to the network interface. On packet reception, if the buffer provided by the application is page aligned and its length is a multiple of the page size then that buffer is replaced by page remapping with the data.

Newer ATM interfaces such as the APIC[6, 7] have been designed with advanced features such as on-board segmentation and re-assembly, and automatic separation of headers from data on packet delivery. This allows the OS to use the VM system to deliver data to the application.

Packet Filter. The BSD packet filter (BPF)[20] is a tool which allows for user-level network packet capturing. It allows the application to load a "filter" into the kernel, which is used to discard uninteresting packets without having to copy them into the application's address space. The "filter" is, in fact, a small interpreted program. In the first version of the packet filter the program was "executed" on a stack-machine. The current version of the BPF the program is "executed" on a virtual RISC machine. The RISC machine has access to the packet, some registers, and a small area of scratch memory. This is similar in spirit to the SuperCall. However, while the BPF is narrowly focused on one question which has a simple yes or no answer (should the packet be passed to the application?), the SuperCall is aimed at allowing more general forms of I/O from the kernel.

8. Current Status and Future Research

The topic of this paper is the design and implementation of the UCM I/O system. This research can be divided into four stages.

- First, the problem to be solved must be fully identified and clear objectives for the problem's solution must be set. This has been accomplished.
- Second, a design for the solution must be developed that meets all the proposed objectives. The proposed solution, UCM I/O, consists of two main parts: the API and the buffering system. At this time, the UCM I/O API has been almost fully defined and appears to support all the I/O operations presented in this proposal. Based on the requirements of the API, a high level design of the UCM I/O buffering system has been developed.

  The final step in finishing the design is to adjust it as needed to fit in with NetBSD specific details.

- The third stage of the proposed research is the implementation phase. The first step of this phase is to support the traditional Unix API on top of the new buffering system. Once the new buffering system is working with the old API, the implementation of the new API can begin (the old API will still be supported). After the new API is implemented, the final step is to implement the SuperCall system call driver.
- Finally, once the implementation is complete, it needs to be evaluated. For novel ideas such as shared buffer ring based I/O and the SuperCall, simply showing that they work is a first step to proof of concept. After that, the next thing to do is to start inserting probes in the kernel to measure the overhead of UCM I/O. For the API, the cost of remapping vs copying needs to be measured (in terms of delay and throughput). Also the impact of the new API on existing applications needs to be evaluated. For SuperCalls, the benefits of moving part of an application into the kernel needs to be measured. Existing applications such as `ftp` and `cp` can be ported and their performance should be compared to the original versions. For continuous media, a raw video or audio application which sends data periodically can be written. This application can be evaluated by using probes to measure the polling cost, and by monitoring the network. For the buffer management and VM system the remapping time and page fault time need to measured.

Additionally, we will investigate any unusual results that the above timings uncover. Finally, after implementing and measuring UCM I/O we will try to evaluate where the I/O costs lie and if there are any changes that could be made to the hardware or software that would further improve the performance of UCM I/O.

References

[1] Accetta, B., Golub, D., Rashid, R., Tevanian, A., and Young, M., "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of Summer 1986 USENIX Conf.*, pp. 93-112, 1986.

[2] Bolduc, L., Culbert, J., Harada, T., Harward, J., and Schlusselberg, E. "The AthenaMuse 2 Functional Specification," *AthenaMuse Software Consortium E40-300*, Massachusetts Institute of Technology.

[3] Cheriton, D., "UIO: A Uniform I/O System Interface for Distributed Systems," *ACM Transactions on Computer Systems*, Vol. 5, No. 1, pp. 12-46, 1987.

[4] Chesson, G., et. al., "XTP Protocol Definition," Revision 3.1, Protocol Engines, Inc., PEI 88–13, Santa Barbara, California, 1988.

[5] Clark, D., Schenker, S., and Zhang, L., " Supporting Real-Time Applications in an Integrated Packet Network: Architecture and Mechanism," *ACM SIGCOMM'92 Conference: Communications Architectures and Protocols*, Vol. 22, No. 4, pp. 14-26, October 1992.

[6] Dittia, Z., Cox, J., and Parulkar, G., "Washington University's Gigabit ATM Desk Area Network," *Proceedings of the Ninth Annual IEEE Workshop on Computer Communications*, pp. 176-187, October, 1994.

[7] Dittia, Z., Cox, J., and Parulkar, G., "A Gigabit ATM Desk Area Network: System Architecture Document for the ATM Port Interconnect Controller (APIC) Draft Version 1.0," August 1994.

[8] Druschel, P., "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," University of Arizona Department of Computer Science, Technical Report #TR93-05, 1993.

[9] Floyd, S., "Issues in Flexible Resource Management for Datagram Networks," *Preliminary Proceedings of Third Annual Workshop on Very High Speed Networks*, Baltimore, pp. 238-250, March 1992.

[10] Gingell, R., Moran, J., and Shannon, W., "Virtual Memory Architecture in SunOS," *Proceedings of 1987 Summer Usenix*, pp. 81-94.

[11] Gong, F. "A Transport Solution for Pipelined Network Computing," Doctoral Dissertation, Department of Computer Science, Sever Institute of Technology, Washington University, St. Louis, Missouri, December 1992.

[12] Gong, F., and Parulkar, G., "Segment Streaming for Efficient Pipelined Televisualization," *Proceedings of IEEE Military Communications Conference*, MILCOMM'92.

[13] Gopalakrishnan, R., Parulkar, G.M., "Efficient Quality-of-Service Support in Multimedia Computer Operating Systems," Technical Report WUCS-94-26, Dept. of Computer Science, Washington University in St.Louis, November 1994.

[14] Gove, R., Lee, W., Kim, Y., and Alexander, T., "Image computing requirements for the 1990s: from multimedia to medicine," *Medical Imaging V - image capture, formatting, and display*,, SPIE, Vol. 1444, pp. 318-333, February 1991.

[15] Guttag, K., Gove, R., and Aken, V., "A Single-Chip Multiprocessor for Multimedia: The MVP," *IEEE Computer Graphics and Applications*, pp. 53-64, November 1992.

[16] Jacobson, V., "query about TCP header on tcp-ip," Message-ID <9309080530.AA05271rx7.ee.lbl.gov>, E-Mail to Craig Partridge from TCP-IP mailing list.

[17] Leffler, Samuel J., McKusick, Marshall K., Karels, Michael J., and Quarterman, John S., *The Design and Implementation of the 4.3* BSD *Unix Operating System*, Addison-Wesley Publishing Company, Inc., Redding, Massachusetts, 1989.

[18] Netravali, A., Roome, W., and Sabnani, K., "Design and Implementation of a High-Speed Transport Protocol," *IEEE Trans. on Communications*, Vol. 38, No. 11, pp. 2010-2024, November 1990.

[19] Mazraani, T., and Parulkar, G., "Specification of a Multipoint Congram-Oriented High Performance Internet Protocol," INFOCOM'90, *IEEE Computer Society*, Washington D.C., June 1990.

[20] McCanne, S., and Jacobson, V., "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *Proceedings of 1993 Winter Usenix*, pp. 259-269.

[21] Rashid, R., Tavanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectuers," Carnegie Mellon Department of Computer Science Technical Report # CMU-CS-87-140, July, 1987.

[22] Sterbenz, J., and Parulkar, G., "Axon: A Host-Network Interface Architecture for Gigabit Communication," Doctoral dissertation, Department of Computer Science, Sever Institute of Technology, Washington University, St. Louis, Missouri, December 1991.

[23] Topolcic, C., "Experimental Internet Stream Protocol: Version 2 (ST-II)," RFC-1190, October 1990.

[24] Zhang, L., "A New Architecture for Packet Switching Network Protocols," Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, July 1989.

[25] Zhang, L., Deering, S., Estrin, D., Shenker, S., and Zappala, D., "A New Resource ReSerVation Protocol," (unpublished document).