DESIGN OF UNIVERSAL CONTINUOUS MEDIA I/O*

Charles D. Cranor and Gurudatta M. Parulkar

Washington University, St. Louis MO 63130, USA

Abstract. The problem this research 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.[2]

1 Introduction

The current Unix I/O application program interface (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 (due to data copying and system call overhead), and little support for continuous media.

Thus, our objective is to design and implement a universal continuous media I/O (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 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

* This work was supported in part by ARPA, the National Science Foundation, and an industrial consortium of ascom Timeplex, Bellcore, BNR, Goldstar, NEC, NTT, Southwestern Bell, SynOptics, and Tektronix

[2] This paper is an extended abstract of [1].

– *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 the I/O subsystem as a whole. The first three ideas are briefly described in the subsequent sections.

2 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. The two main I/O system calls are called `import` and `export`. The `import` function takes data from the I/O system and imports it to the application. The `export` function sends data from the application to the I/O system. These calls support vector based I/O, and both connectionless and connection-oriented I/O with the same interface. Other functions include functions which start and stop continuous media I/O, buffer allocation functions, and functions that open and close UCM I/O descriptors. Also, note that 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 the application or device driver sends data into UCM I/O subsystem there are three options:

1. The application can force the UCM I/O layer in the kernel to copy data between application and kernel memory.
2. The application can allow the kernel to remap the data (in effect the application is giving away its buffer).
3. The application can allow the kernel to choose whether the data buffers should be remapped or copied.

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 within the OS using a soft real-time 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 effort but is being explored in a related project [2].

3 SuperCall

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.

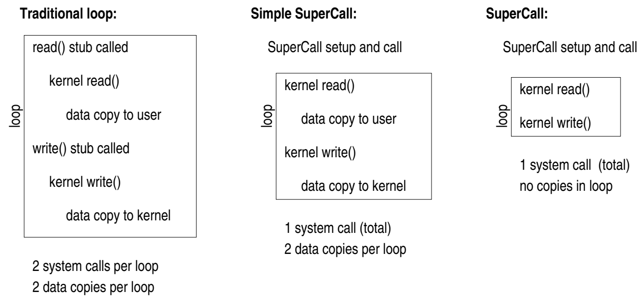

Fig. 1. SuperCall

Figure 1 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 1. 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.

4 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 uses a circular pool of data buffers, as shown in Figure 2. 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 the polling interval the kernel may send an invalid data buffer. By folding the polling of the tag into an already existing interrupt, cost is minimized.

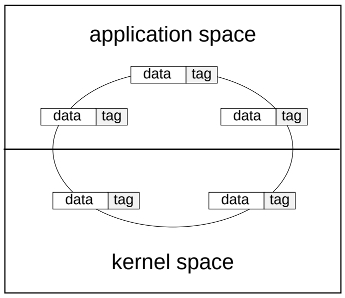

Fig. 2. A pool of buffers

References

1. Cranor, C., Parulkar, G., "Universal Continuous Media I/O: Design and Implementation," Washington University Department of Computer Science, Technical Report WUCS-94-34, 1994.
2. Gopalakrishnan, R., and Parulkar, G.M., "Application level Protocol Implementations to provide Quality-of-service Guarantees at Endsystems," Ninth IEEE Workshop on Computer Communications, Duck Key, Florida, Oct 1994.