Providing Asynchronous File I/O

for the Plan 9 Operating System

by

Jason Hickey

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 21, 2004 [June 2004]

Author _____

Jason M. Hickey
Department of Electrical Engineering and Computer Science
May 21, 2004

Certified by _____

Russell S. Cox
Graduate Student, Parallel and Distributed Operating Systems Group
Thesis Supervisor

Certified by _____

M. Frans Kaashoek
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by _____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Providing Asynchronous File I/O
for the Plan 9 Operating System
by
Jason M. Hickey

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 2004

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis proposes a *mux* abstraction that multiplexes messages of a network file protocol to provide asynchronous access to all system resources on the Plan 9 operating system. The mux provides an easy-to-program asynchronous interface alleviating the need to manage multiple connections with different servers. A modified version of the Plan 9 Web server demonstrates that the mux can be used to implement a high-performance server with user-level threads without having to use a kernel thread for each user-level thread.

Scalability tests demonstrate that the mux implementation scales well with hundreds of clients and hundreds of servers. Furthermore, the user-threaded version of the web server performs comparably with the kernel-threaded implementation on disk bound workloads and exhibits an 18% decrease in performance on in memory workloads. These results suggest that the mux could provide performance benefits for more intricate applications that can exploit the fine-grained control of user-level scheduling.

Thesis Supervisor: Russell S. Cox
Title: Graduate Student, Parallel and Distributed Operating Systems Group

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Computer Science and Engineering

Acknowledgements

I would like to thank Russ Cox, my primary thesis advisor for this thesis. Without his insight, feedback, and assistance throughout the entire process, this thesis would never have been possible. Russ provided the Plan 9 kernel modifications used to export the network device and ran the web server benchmarking tests presented in the evaluation. I am particularly indebted to him for the nearly full-time assistance he provided during the last few weeks of the thesis. His involvement went above and beyond the call of duty for a thesis advisor, and the thesis would never have been completed on time without it.

Secondly, I would like to thank Frans Kaashoek for overseeing this project and providing extensive feedback on the writing. I would also like to thank him for allowing me to undertake this research in the first place and for ensuring that I had the necessary resources to complete the work.

I also owe a word of thanks to the members of the PDOS group for discussions and feedback at various times throughout the process. I'd specifically like to thank Frank Dabek for routine discussions as the work progressed and for providing the hardware support needed to carry out the work.

Finally, I would like to thank my parents. Without their continuing support and encouragement, I would never have made it this far.

1 Introduction

High performance systems require asynchronous I/O to exploit the parallelism provided by the hardware. Existing asynchronous interfaces lack completeness or consistency across different devices. These deficiencies make implementations of systems that require non-blocking access to multiple devices both difficult and error-prone.

In this thesis, we address the need for an interface for asynchronous I/O on the Plan 9 operating system [10]. The next section defines several terms used in the thesis and details the general need for asynchronous I/O. Section 1.2 presents an example that illustrates the problems that stem from the lack of completeness and consistency in existing asynchronous interfaces. Section 1.3 details the specific problems with existing methods of performing asynchronous I/O on Plan 9. Finally, section 1.4 details the contributions of this thesis and outlines the remainder of the thesis.

1.1 Problem Statement

In this thesis, we use the following definitions for several terms:

Process – A flow of control defined by a single register set operating within a single address space. Processes are scheduled by the kernel.

Kernel Thread – A flow of control defined by a single register set operating in a shared address space with zero or more other kernel threads. Kernel threads are scheduled by the kernel.

User Thread – A flow of control defined by a single register set operating in a shared address space with zero or more other user threads. User threads are scheduled by a user-level thread manager.

For each of these abstractions, we use the definition to refer to the abstraction running in user mode.

Processes provide an abstraction of the physical processor and isolation from other concurrently executing processes. Context switches between processes have a high overhead from entering and leaving the kernel and from the need to switch address spaces. Kernel threads alleviate some of this overhead by eliminating the need to switch address spaces. Both of these abstractions must obey the kernel scheduling policy. This can be detrimental to performance because the kernel scheduling decisions may not match the needs of the application.

User threads allow scheduling at the application level. Eliminating the need to transfer control to the kernel also makes context switching less expensive. However, user threads must be implemented on top of existing operating system abstractions, such as kernel threads. This leads to restrictions on the ability of user threads to provide a complete abstraction of the physical processor.

In particular, if a user thread performs an I/O operation that blocks in the kernel, none of the other user threads running on top of the same kernel thread can run until the operation completes and the kernel returns control to the blocked kernel thread. The next section provides a concrete example of this general problem in the context of a web server.

1.2 Example: Asynchronous I/O in Web Servers

In this section, we illustrate the benefits from overlapping processing and device operations by reviewing work on the Flash web server [7]. This work also demonstrates the difficulty in using the available operating system interfaces to achieve concurrency.

To provide high throughput, web servers serve requests from multiple clients concurrently. A web server running on a system with a single processor overlaps processing with network operations. As an example, a web server might locate the content for one request while the network interface receives a new incoming request. Additionally, for web servers with disk-bound workloads, obtaining concurrency requires that disk operations overlap with both request processing and network I/O. Web server designs achieve this overlap by employing one of several different approaches.

One approach relies on the operating system to manage concurrency. Web servers designed with the multi-process (MP) architecture allocate one process to handle each request. The operating system overlaps processing with disk I/O by switching to a runnable process whenever the active process performs a blocking operation. The MP approach suffers from per-request memory overhead and expensive context switching. Additionally, the MP architecture makes it difficult to incorporate centralized caching and information gathering used to improve performance. These penalties cut into the performance benefits provided by overlapping disk operations with processing.

An improvement on this architecture, called multi-threaded (MT), uses kernel threads instead of processes. This reduces memory and context switching overheads, and allows more convenient use of centralized information gathering. Despite these improvements, performance tests presented in the Flash paper demonstrate that MT implementations perform only marginally better than MP implementations. This lack of improvement indicates that the reliance on application-unaware kernel mechanisms to manage concurrency prohibits extremely high-performance web server implementations.

To avoid reliance on the kernel for scheduling, another approach to web server design explicitly manages concurrency at the application level by interleaving the processing stages of multiple outstanding requests. The single-process event-driven architecture (SPED) consists of one process that makes non-blocking system calls to perform I/O asynchronously. In principle, this approach should perform better than MP/MT implementations because it does not suffer from substantial per request overhead or context switching.

Many existing operating system interfaces impede high-performance implementations of SPED. For example, the BSD UNIX `select` system call allows applications to register a set of file descriptors to determine which can be read from or written to without blocking [4]. The `select` system call works correctly on network sockets and pipes, but cannot handle disk files. Since SPED uses a single process, all request processing ceases when an operation blocks. Other interfaces exist to circumvent blocking on disk I/O, but no *uniform* interface exists that covers all devices. The authors of the Flash paper summarize this problem:

> Many UNIX systems provide alternate APIs that implement true asynchronous disk I/O, but these APIs are generally not integrated with the `select` operation. This makes it difficult or impossible to simultaneously check for completion of network and disk I/O events in an efficient manner. Moreover, operations such as open and stat on file descriptors may still be blocking [7].

The Flash web server implements a hybrid design to attain some of the performance gains of the SPED architecture without altering existing operating system interfaces. The asymmetric multi-process event-driven (AMPED) architecture starts with a single server process that dispatches on ready I/O operations. When an event requires a potentially blocking I/O operation, the server process delegates the operation to a helper

process. The server and helper processes communicate through a pipe, which the server registers with `select`. When the I/O operation completes, the helper process notifies the server process using the pipe. Flash shows marked performance gains over MP/MT and blocking SPED implementations. However, the use of helper processes in Flash and the overheads incurred therein imply that a non-blocking SPED implementation would outperform an AMPED implementation.

In addition to AMPED's overheads, ensuring that the main process never blocks requires a careful implementation. Analysis of the Flash web server performed using DeBox, a system call performance profiling tool, revealed that the main process still blocked on seven different system calls [9]. Since the main process opens the same file that the helper process retrieves, some of the system calls blocked because of contention on file system locks. To prevent the main and helper processes from both trying to open a file, the designers changed the helper processes to pass open file descriptors to the main process with `sendmsg`.

Deeper analysis of the other reasons for blocking discovered a subtle performance bug in the Flash implementation. The Flash web server employs two different caches: one for URL-to-path name translations and a second for data of memory mapped files. In the process of translating a URL to a path name, Flash retrieves the file metadata and stores it in the data cache. If a name lookup hits its cache, Flash assumes that it just performed the name translation and that it has the file metadata in memory. However, the metadata could have been replaced in the data cache even though the URL-to-path name entry still remains. When the main process calls `open`, it may block. This

demonstrates the difficulty in guaranteeing non-blocking behavior without a complete asynchronous interface.

In addition to the blocking behavior discovered with DeBox, the analysis pinpointed the major overheads in the Flash design. The main process originally used the `mincore` call to determine memory residency of files before calling a potentially blocking operation. However, the system call profiling demonstrated that that the operations required for memory residency checking constituted a large portion of the total CPU time. The Flash designers eliminated the memory residency checks and chose to rely instead on heuristics based on bookkeeping for the mapped-file cache. This method caused occasional blocking in calls to `sendfile`, a zero-copy I/O operation [8]. To solve this problem, the authors added a flag to `sendfile` so that it returns an error if the operation will blocking on disk.

The preceding discussion demonstrates the need for a complete and intuitive interface for asynchronous I/O. In this thesis, we provide such an interface for the Plan 9 operating system. The next section motivates the use of Plan 9 and outlines the specific shortcomings of current methods for performing asynchronous I/O in Plan 9.

1.3 Limitations of Asynchronous I/O in Plan 9

Plan 9 presents access to all resources through a standard file interface. A single asynchronous interface for file I/O therefore provides asynchronous access to all resources in the system. Additionally, Plan 9 is a distributed operating system that constructs a computing environment from a network of independently operating components. The inherently asynchronous behavior of this environment motivates the need for a usable asynchronous interface.

For remote resources, Plan 9 already provides an asynchronous interface through 9P, the network-level file protocol. (See the Appendix for a description of the 9P protocol.) This does not immediately provide a user-level interface for asynchronous file I/O, however, because the kernel presents a 9P connection to an application as a file descriptor. A client reads and writes 9P messages on the file descriptor to correspond with a 9P server. Since these reads and writes could block, the client application achieves asynchronous behavior by creating additional processes to handle the I/O. As the number of servers that a client wishes to communicate with increases, the task of managing multiple connections and the corresponding I/O processes becomes burdensome.

As a further complication to the task of performing asynchronous I/O, the kernel presents local devices through a blocking procedural interface. This thesis addresses these issues to provide more convenient mechanisms for performing asynchronous file I/O in Plan 9.

1.4 Contributions and Outline

The primary contribution of this thesis is the construction of a user-level library for a 9P multiplexer (mux). The mux presents a new abstraction of a 9P connection that allows clients to asynchronously carry out conversations with multiple servers over a single connection. Clients using the mux call non-blocking procedures to send and receive 9P messages. To implement these procedures, the mux manages the I/O processes needed for asynchronous communication with servers.

We also demonstrate the feasibility of providing user-level 9P access to local devices by exporting a 9P connection to the network device. We combine the network 9P

connection and a 9P connection to a disk file server through the mux in an implementation of a user-threaded web server. In constructing the web server, we modify a user-level library to provide implementations of the standard file I/O system calls that block at the user-level.

The remainder of this thesis is structured as follows. Section 2 surveys work related to asynchronous I/O and the more general topic of exposing control to the user level. We present the design of the mux in section 3 and describe the mux implementation in section 4. Section 5 describes the design for providing user-level 9P access to local devices. In section 6, we evaluate the design and present the user-threaded web server used in the benchmarks. We conclude in section 7 and offer suggestions for future work.

2 Related Work

In this section we describe and evaluate previous work on providing asynchronous interfaces. Additionally, we present work detailing the performance benefits of exposing fine-grained access to hardware to the user-level.

2.1 Asynchronous Interfaces

Existing asynchronous interfaces only achieve non-blocking behavior for a restricted class of operations. A number of interfaces exist that allow applications to poll sets of file descriptors to determine if an I/O operation on that file descriptor will block. These interfaces only support a subset of I/O operations such as `read` and `write`.

The BSD UNIX `select` system call allows applications to inquire about the blocking status of three sets of file descriptors [4]. An application passes a set of file descriptors for reading, a set of file descriptors for writing, and a set of file descriptors to check for exceptions as arguments to `select` and `select` returns subsets containing the descriptors ready for reading, ready for writing, and those with exceptions respectively.

The System V `poll` system call performs similarly to `select`, except that an application only passes one set of file descriptors as an argument [4]. The structure associated with each file descriptor contains a bit mask that allows the application to register the type of events that `poll` should check for.

The Solaris 7 Operating Environment provides the `/dev/poll` (`devpoll`) pseudo-device [13]. This device provides the same functionality as the `select` and

`poll` calls, while allowing an application to register kernel-level polling information that persists between system call invocations. An application acquires access to a new instance of the `devpoll` device by opening `/dev/poll`. To register file descriptors for `devpoll` to monitor, the application writes to on the file descriptor returned by `open`. The application can use the `ioctl` system call to retrieve the set of file descriptors ready for I/O. With `devpoll`, an application that repeatedly polls a set of file descriptors need only pass the set to the kernel once. For this reason, `devpoll` provides better performance than `select` or `poll` in applications monitoring large number of file descriptors.

The `select`, `poll`, and `devpoll` (`select`-style) interfaces suffer from the drawback that they allow an application to achieve non-blocking I/O only with socket and pipe file descriptors. In particular, `select`-style interfaces do not allow an application to register an offset in a file or specify the number of bytes to read or write. For example, if an application registers a file descriptor for a file stored on a local disk, these interfaces will return a ready status for the file descriptor when the application polls for non-blocking read events. If the application calls `read` on the file descriptor for data not currently in cached in memory, the operation will block while the disk retrieves the data.

The POSIX Asynchronous I/O (AIO) interface addresses this problem by providing asynchronous read (`aio_read`) and write (`aio_write`) operations [4]. To asynchronously read or write a file, an application calls the corresponding AIO function. The application passes a structure to the call containing the relevant information such as the offset and number of bytes and a pointer to a buffer for the data. The application then

repeatedly calls `aio_error` to determine whether the operation has completed. On completion, a call to `aio_return` gives the return status of the file. This interface solves the specific problem of asynchronous reading and writing files, but leaves out other potentially blocking I/O operations, such as `open`. `Open` calls have a high chance of blocking because the kernel must resolve each element of the path, which may require multiple disk operations. Additionally, the AIO interface does not integrate well with the `select`-style interfaces. As noted in the Flash paper, this makes it difficult to design applications that need to both poll sockets and pipes and perform asynchronous reads and writes.

The `kqueue` API provides a more general polling interface for asynchronous I/O [4]. With `kqueue`, an application registers a set of events with the kernel by calling `kevent`. The application calls `kevent` again to check for action on a specified list of previously registered events. The kernel calls event-specific filter procedures when it detects the specified action on the registered event. `Kqueue` supports several different types of events including both file descriptor read and write monitoring of the `select`-style interfaces and support for the AIO interface. An application can register a call to `aio_read` or `aio_write` with `kqeueue` and the kernel will modify the associate event structure when the operation completes. On a subsequent call to `kevent`, the application will receive the notification of completed I/O and call `aio_return`. Although `kqueue` provides a common polling interface for some of the existing specialized asynchronous I/O operations, it does not support other asynchronous calls such as `open`. Adding support for new interfaces in `kqueue` requires modifying the kernel to implement filters for the new event type. The complexity of providing access to

all resources with the `kqueue` interface stems from the underlying problem of having different asynchronous interfaces to different resources. A uniform asynchronous interface to all resources eliminates the need to modify the kernel to provide asynchronous support for each new resource.

The libasync library solves many of these problems by providing a generic interface for non-blocking functions [6]. Libasync uses C++ templates that allow an application to wrap a callback function and arguments into an asynchronous call. Libasync uses the `select` system call to poll for ready network events and performs NFS transactions to handle file operations. When the operation completes, libasync calls the callback function with the result and any passed in arguments.

2.2 Exposing Control to the User Level

In their paper on scheduler activations, Anderson *et al.* argue that user threads achieve inherently better performance than kernel threads [1]. The authors note that previous implementations of user threads suffer from a lack of control over scheduling. Processes, on top of which user threads are implemented, do not provide a complete representation of the underlying physical processor. In particular, when a user thread performs an I/O operation that causes it to block, the entire user thread package blocks even though another user thread could run on the now idle processor. Scheduler activations work around this problem by communicating scheduling information between the kernel and the user thread manager. A scheduler activation plays a similar role as a kernel thread. A single user thread runs on top of a scheduler activation. When a user thread performs a blocking operation, the kernel creates a new scheduler activation and performs an upcall to inform the user thread manager of the blocking user thread. The

user thread manager removes the state of the blocking user thread from its scheduler activation and notifies the kernel that it may reuse the old scheduler activation. The user thread manager then selects a ready user thread and runs it on the new scheduler activation. This scheme provides a way around blocking I/O, but requires that the user thread scheduler interact with the kernel in order to manage threads correctly. Ideally, the kernel would provide a user thread package with a time-slice, and within that time-slice, the user thread manager could detect blocking threads and switch to a runnable thread without interacting with the kernel.

In a generalization of the concept of increasing user-level control, the exokernel design concentrates on separating the protection of physical resources from the management of those resources [3]. The exokernel architecture securely exports low-level access to the underlying hardware to the user-level, leaving the development of abstractions for resource management to operating systems libraries. This design provides flexibility and efficiency by allowing applications to take advantage of the full capabilities of the hardware. Experiments on an exokernel operating system prototype bear the design logic out in practice. Most primitive kernel operations execute ten to 100 times faster on the prototype than on Ultrix, a standard UNIX operating system. These results illustrate the benefits of providing fine-grained access to hardware at the user level.

3 Mux Design

The mux provides clients a manageable, non-blocking interface to interact with servers by multiplexing 9P conversations between one or more clients and one or more servers. The mux maintains one 9P connection with every client and one 9P connection with every server. Using these 9P connections, the mux provides three functions:

1) The mux multiplexes the requests from multiple clients to access files on a server over the single 9P connection with that server.
2) The mux multiplexes the responses from multiple servers to requests from a client over the single 9P connection with that client.
3) The mux hides the multiplexing from both the clients and the servers. Each server believes it is carrying out a 9P conversation with a single client. Likewise, each client believes it is carrying out a 9P conversation with a single server.

3.1 Interface

```
Mux* muxalloc(void)
int muxmount(Mux *mux, int fd, char *old, int flag)
int muxbind(Mux *mux, char *new, char *old, int flag)
Muxcon* muxattach(Mux *mux)
void muxsend(Muxcon *mc, Fcall *msg)
int muxrecv(Muxcon *mc, Fcall *msg)
```

Figure 1: Mux interface. This figure shows the external interface to the 9P mux.

Figure 1 shows the procedures provided by the mux interface. The `muxalloc` procedure creates a new 9P multiplexer. It takes no arguments and returns a pointer to a `Mux` structure.

The muxmount procedure allows an application to attach the root of a file hierarchy served by a 9P server at an arbitrary place in the name space of a Mux. In the arguments to muxmount, fd corresponds to an open connection to a 9P server. After a successful call to muxmount, the root of the file tree served by fd appears in the name space of mux with file name old. The flag argument specifies how the root of the file tree served by fd (new) will be mounted on old. A flag value of MREPL causes new to replace the existing directory old. A value of MBEFORE or MAFTER makes old into a union directory with new at the beginning or end respectively. Muxmount returns a positive integer on success and –1 on failure.

A call to muxbind rearranges the existing name space. A successful call to muxbind causes the file new in the name space of mux to appear in the directory old in the same name space. The flag argument has the same effect as in muxmount. Muxbind returns a positive integer on success and –1 on failure.

The muxattach procedure establishes a connection with the mux. It returns a pointer to a Muxcon. A Muxcon uniquely identifies a client of the mux.

The muxsend procedure communicates 9P messages to a mux. Muxsend sends the 9P message msg to the input queue of the mux connected to mc.

The muxrecv procedure reads from mc and fills out the Fcall structure msg with the first 9P message received. Muxrecv returns a positive integer if the client must free buffers associated with msg and 0 otherwise. *A* call to muxrecv blocks until a message arrives.

3.2 Using the Mux Interface to Perform Asynchronous I/O

A client uses the mux to perform asynchronous I/O by sending and receiving 9P messages to servers. Currently, a client performs 9P transactions with servers by reading and writing the file descriptor representing the 9P connection. The reads and writes to the file descriptor could cause the calling thread to block.

To provide the ability to send and receive 9P messages without blocking, the mux replaces the file descriptor interface to a 9P connection with a `Muxcon`. A call to `muxsend` or `muxrecv` that cannot execute immediately causes the calling thread to wait on a user-level queue. A user-level thread manager can then schedule a ready user thread until the send or receive completes.

3.3 Overview of Internal Structure

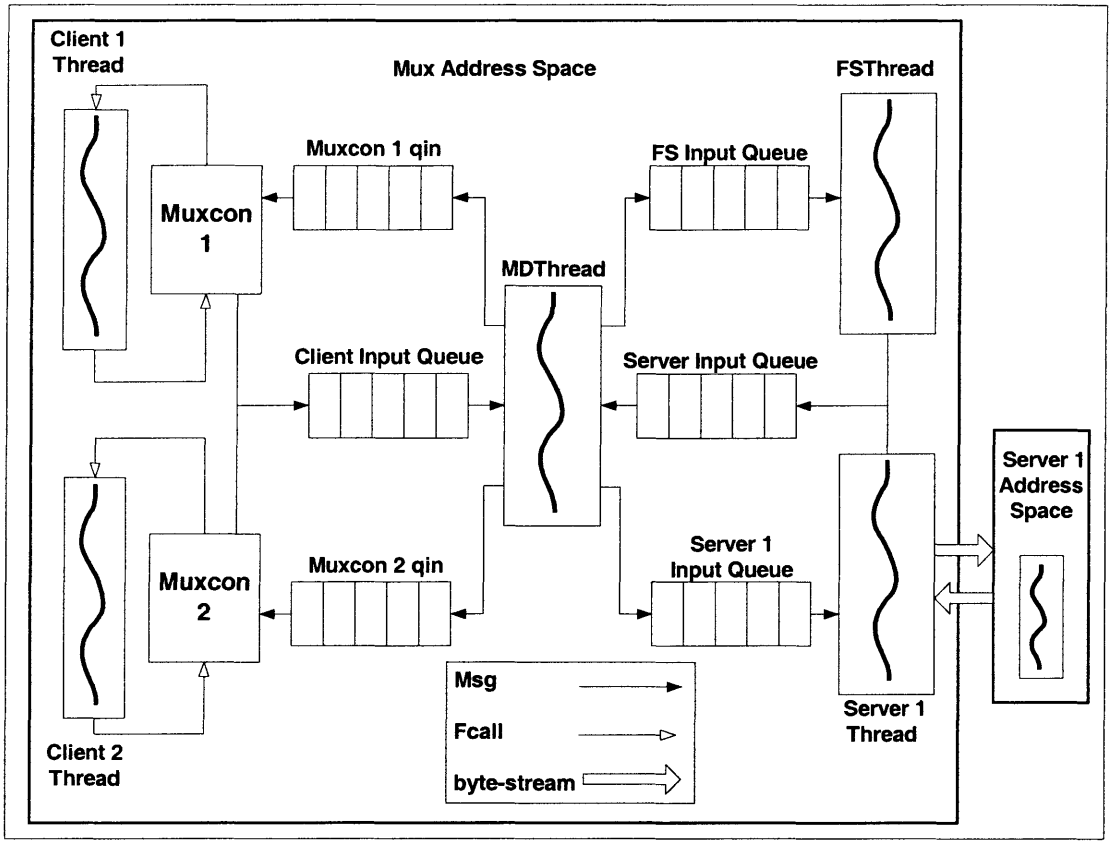

Figure 2: Internal mux structure. This diagram shows the internal structure for a mux connecting two clients and one external server together. Threads use shared queues to pass 9P messages between one another. Different arrow types represent the three different 9P message formats used by the mux. The byte-stream arrow indicates the presence of an I/O process.

```
struct Msg
{
   Queue *qret;  //return queue for the response
   Msg *mwrap; //chain of saved messages
   uchar type; //9P or muxcall
   union {
      Fcall f; //9P message structure
      Muxcall m; //muxcall message structure
   };
};

struct Muxcon
{
   Muxcon *next; //next Muxcon in list
   Queue *qin; //queue of incoming messages
   Queue *qout;  //queue of outgoing messages
   CFid *cfids;  //list of client fids
};
```

Figure 3: Msg and Muxcon structures.

Figure 2 depicts the internal structure of the mux. The mux consists of multiple user threads that send and receive 9P messages by manipulating fixed-sized message queues. We use a queue size of 512 in the implementation of the mux. In the process of serving 9P transactions, the mux converts messages between two external 9P message formats and an internal format used by the mux. Clients use the `Fcall` representation of a 9P message. Calls to `muxsend` and `muxrecv` use the client's `Muxcon` to convert between the `Fcall` representation and the internal `Msg` structure which contains additional information that the mux uses in servicing requests (Figure 3 details the `Muxcon` and `Msg` structures). Server threads convert messages between the `Msg` structure and the machine independent byte-stream representation. An I/O process reads and writes the byte-stream representation to the file descriptor for the 9P connection to the server.

A call to `muxalloc` creates a new `Mux` and creates two threads in the address space of the calling thread: the message dispatching thread (`mdthread`) and the file server thread (`fsthread`). The `mdthread` processes and routes 9P messages and is described in detail in section 4.1. The `fsthread` serves the internal mux file tree. This file tree provides the backbone onto which an application mounts file trees for external 9P servers. Section 4.2 describes the implementation of the `fsthread`.

A client thread attaches to the mux by calling `muxattach`. `Muxattach` allocates a new `Muxcon` (see Figure 3). The `Muxcon` contains a pointer to an input queue (`qin`) and an output queue (`qout`). Each `Muxcon` has a unique `qin`, but the `qout` for all `Muxcons` points to the Client Input Queue of the `mdthread`. A call to

muxsend wraps the Fcall structure representation of the 9P request into a Msg structure and sets the return queue (qret) of the Msg structure to point to the Muxcon's qin. Muxsend then sends the request to Client Input Queue of the mdthread. After the mdthread processes the request, it sends the response to queue specified in the qret field of the Msg structure of the request. Muxrecv retrieves responses from the qin of the Muxcon and returns the Fcall in the Msg structure to the client.

A call to muxmount attaches the file tree of an external server into the mux namespace. Muxmount sets up a server thread and two I/O processes to manage communication with the external server being mounted. Each server thread has a unique input queue from which it retrieves requests send by the mdthread. The server thread converts requests from the Msg structure into the byte-stream representation and uses the first I/O process to write the converted requests to the file descriptor representing the connection with the server. To retrieve responses from the external server, the server thread uses the second I/O process to read from the server file descriptor. After retrieving a response, the server thread converts the message to the Msg structure format and sends it to the Server Input Queue of the mdthread.

The previous discussion described the functions of the constituent components of the mux and how messages flow between them. The next section explains the problems encountered in message multiplexing and the data structures employed to solve those problems.

3.4 Data Structures

The mux must provide three primary functions. Each of these functions raises specific problems that the mux design solves. In order to multiplex requests from multiple clients over a single server connection, the mux solves the problem of collisions in the tag and fid spaces. To multiplex responses from multiple servers over a single client connection, the mux solves the problem of collisions in the qid space. To provide the single server/single client abstraction, the mux manages a name space that integrates file trees from multiple servers into a single file hierarchy presented to the clients and hidden from the servers.

3.4.1 Avoiding Tag Collisions

Tag collisions arise when multiplexing multiple clients over a single 9P connection. Clients choose tags to uniquely identify 9P requests. However, two (or more) clients may each send a request with the same tag. The mux uses the `qret` field of the incoming `Msg` structure to associate a client with the tag of incoming requests. When the mux needs to send a message to a server in order to satisfy a request, the `mdthread` saves the original request in an array and uses the index into this array as the tag for the request that it sends to the server. The saved message remains in the array until a response arrives from the server. At this point, the `mdthread` removes the saved message and the tag can be reused. This convention ensures that the every outstanding message on a server has a unique tag.

3.4.2 Avoiding Fid Collisions

```
struct CFid
{
   uint num;        //client fid number
   CFid *next;      //next CFid in list
   Fid *fid;        //fid used on servers
   Fid *fidcur;       //current fid in unionread
   vlong offset;      //offset in file, for unionread
   MheadHistory *mhHist;  //stack of mount points crossed, for ".."
};

struct Fid
{
   Queue *qsrv;       //input queue that specifies the server
   uint num;        //fid number
   Qid qid;         //qid of the file
};
```

Figure 4: CFid and Fid structures.

Multiplexing multiple clients over a single 9P connection to a server also raises the possibility of fid collisions. Since clients choose the fids to use in 9P sessions, two (or more) clients may choose the same fid to refer to two different files on the same server. To address this problem, the mux uses a single set of fids (maintained in `Fid` structures) in all of its 9P sessions with servers. The mux translates the fids chosen by clients (maintained in `CFid` structures) to these internal fids. This method ensures that each fid sent to a server unambiguously refers to a file handle managed by the server.

Figure 4 outlines the `Fid` and `CFid` structures. Each `Fid` contains the fid number and a pointer to the input queue of the server on which the fid is valid. Each `Muxcon` contains a list of pointers to `CFid` structures. `CFids` contain the number of the fid chosen by the client and a pointer to the `Fid` corresponding to the `CFid`.

The `mdthread` manages the mapping of `CFids` to `Fids`. The `mdthread` looks up the `Muxcon` for a client from the `qret` in the `Msg` structure of a client request. The `Muxcon` provides the mdthread access to the `CFids` corresponding to a client. When a

client sends a message that assigns a new fid to a file, the `mdthread` creates a new `CFid` to `Fid` mapping and adds it to the list stored in the `Muxcon` of the client. When a client sends a request containing an existing fid, the `mdthread` looks up the corresponding `CFid` and follows the pointer to retrieve the `Fid`. The `Fid` tells the `mdthread` which server and fid to use to satisfy the client's request. The `mdthread` destroys the `CFid` and corresponding `Fid` when it receives a clunk request.

3.4.3 Avoiding Qid Collisions

```
struct Queue
{
   Channel *c; //underlying queue implementation
   union {
      Muxcon *mc;   //Muxcon, if this is a Muxcon input queue
      Qidmap *qm;   //qid mappings, if this is a server input
queue
   };
};

struct Qidmap
{
   Qidmap *next; //next mapping
   uint from;  //top two bytes of server qid
   uint to;    //top two bytes of mapped qid
};
```

Figure 5: Queue and Qidmap structures.

Qid collision, the mirror problem of fid collision, may occur when multiplexing multiple servers over a single connection to a client. Each server assigns qids to uniquely identify files in its file tree. However, two (or more) different servers may use identical qids to refer different files in their respective hierarchies. A client cannot identify the context of a qid from the information contained in the 9P response, so the mux translates qids from servers to guarantee their uniqueness.

The mux handles qid translation differently than fid translation. First, the mux only translates the *path* field of the qid, and passes the *version* and *type* fields on to a client

unaltered. Second, the mapping makes use of common structure of qid *paths*. Many servers assign qids to files sequentially, starting at 0. Therefore, the most significant bits of a qid's *path* field have relatively low entropy within a server. In contrast, fids chosen by clients do not generally have this property.

The mux maintains a mapping for the top two bytes of every qid it receives from a server, called a `Qidmap`. Each `Qidmap` entry contains the top two bytes of the server qid and the replacement two bytes to ensure qid uniqueness. The mux stores the `Qidmap` for a server in the `Queue` structure for the server's input queue. Figure 5 outlines these structures.

The `mdthread` manages the `Qidmaps`, and ensures uniqueness across all of the servers. When the mdthread receives a response from a server that contains a qid, it looks up the top two bytes of the qid in the `Qidmap` for the server. If the `Qidmap` contains an entry, the `mdthread` replaces the top two bytes of the qid with the replacement bytes. If no mapping exists, the `mdthread` creates a new entry in the `Qidmap`. To ensure uniqueness of the new entry, the mdthread maintains a two-byte counter that contains the value of the replacements bytes of the next mapping. Each time the `mdthread` creates a `Qidmap` entry, it increments the counter. After the `mdthread` makes an entry in a server's `Qidmap`, that entry remains for the entire time the server is attached to the mux.

The qid translation scheme employed by the mux reduces translation overhead from a pure qid to qid mapping in the common case. If the servers connected to the mux use qid assignment schemes with low entropy in the top two bytes of the qid's *path* field, the `Qidmap` for that server will only contain a small number of entries.

3.4.4 Providing a File Tree

```
struct Mhead
{
   Fid *from;  //the from side of the mount point
   Mount *mount; //first directory on the to side of the mount point
   Mhead *hash;  //next Mhead in hash bucket
};

struct Mount
{
   Fid *to;    //fid on the to side of a mount point
   Mount *next;  //next directory in the union mount
};
```

Figure 6: Mhead and Mount structures.

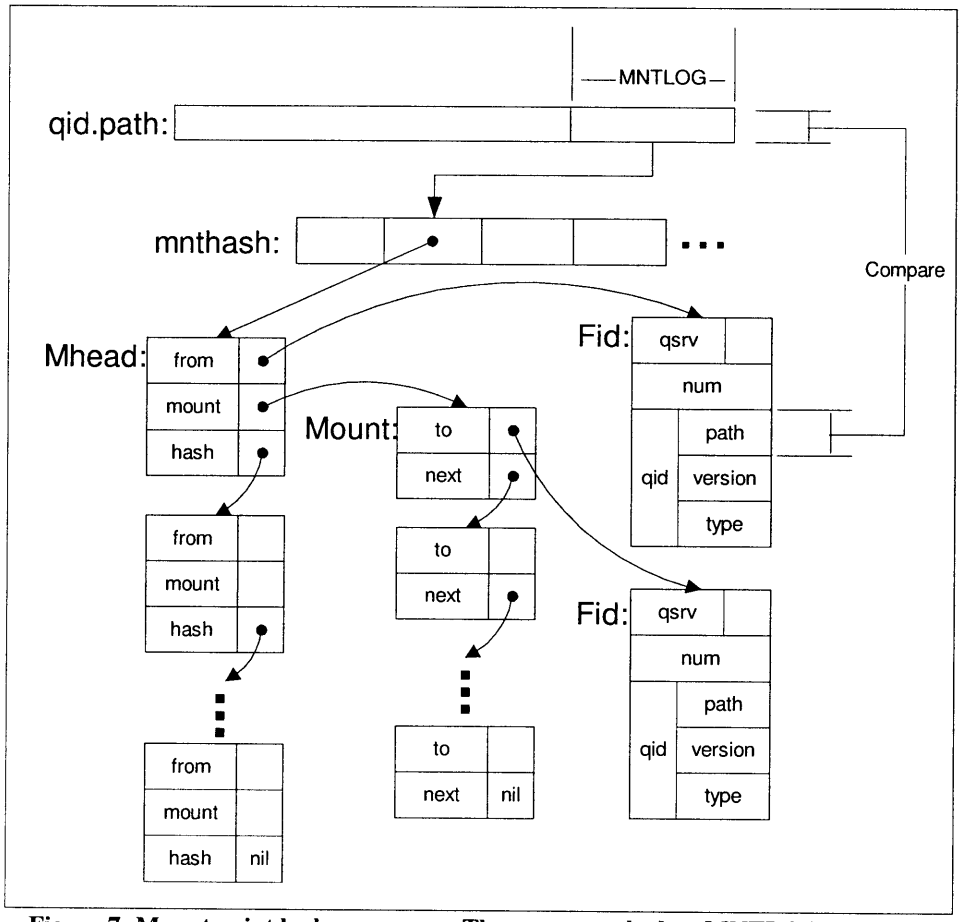

Figure 7: Mount point lookup process. The mux uses the low MNTLOG bits of the qid.path field to index into the mnthash table. The mux follows chain of Mhead structures and compares the qid.path field of the From Fid to find the corresponding mount point. The chain of Mount structures contain the Fids on the *to* side of the mount.

The mux hides the underlying multiplexing from clients and servers. The mux hides the multiplexing of servers from its clients by serving a single file tree. The mux translates client requests that act on this file tree into requests for one or more servers. The mux makes these requests on behalf of its clients, hiding the existence of multiple clients from the servers.

The mux provides name space management tools similar to those exported by the Plan 9 kernel. The mux allows applications to mount the root of a server's file trees at arbitrary points in the name space with a call to `muxmount`. The `muxbind` call rearranges the existing name space. Both of these calls create a mount point in the existing name space. A mount point maps the qid of the old file (called the *from* side of the mount point) to one or more (in the case of union mounts) qid(s) of the mounted file(s) (called the *to* side of the mount point).

For each mount point, the mux stores an `Mhead` structure. As a client traverses the mux's file hierarchy, the mux must check if each file visited is on the *from* side of a mount point. The mux searches through its `Mhead` structures to see if the qid of the file matches an existing mount point. To improve the performance of this lookup, the `Mux` structure contains a hash table (`mnthash`) of `Mheads`. The mux uses as an index into the `mnthash` a fixed number (`MNTLOG`) of the low order bits of the qid's *path* field. Each bucket in the table contains a list of `Mheads`. Each `Mhead` contains a pointer to a `Fid` associated with the *from* side of the mount point and a chain of pointers to `Mount` structures. The `Mount` structures contain a pointer to a `Fid` associated with the *to* side of the mount point. The order of the `Mount` structures in the chain determines the order

of the files in a union mount. Figure 6 illustrates the mount data structures and Figure 7 outlines the mount point lookup process.

The `mdthread` administers these data structures and references them to process requests from clients. To administer the data structures, the mdthread responds to `muxmount` and `muxbind` calls. These procedures wrap the arguments into a special message type called a *muxcall*. The `Msg` structure contains either a pointer to an `Fcall` (for a 9P message) or an `Muxcall` (for muxcall messages). `Muxmount` or `muxbind` send their muxcall to the `mdthread`, which updates the mount data structures accordingly. The mdthread sends a muxcall response to a temporary return queue used by the `muxmount` and `muxbind` calls. These calls block until they receive a response.

The mux uses the data structures described in this section to solve the problems introduced by multiplexing 9P connections. In the next section, we describe the implementation of the `mdthread` and the `fsthread` and explain how the `mdthread` use the data structures presented in this section to manage client requests and server responses.

4 Implementation

This section describes the implementation of the threads that perform the tasks needed for message multiplexing. Section 4.1 describes the implementation of the `mdthread`, which routes messages and maintains the mux name space. Section 4.2 explains the implementation of the `fsthread`, which serves the internal mux file tree.

4.1 Mdthread

The `mdthread` performs the primary logic of the mux. It receives requests from clients, carries out transactions with servers to serve those requests, and sends responses back to clients. The `mdthread` also manipulates the mux data structures to manage the mux's file tree.

4.1.1 Main Loop

The `mdthread` infinitely repeats a main loop. Each time through the loop, the `mdthread` reads in a message from its input queue, processes it, and sends out a message to a client or server. The `mdthread` has two input queues: a client input queue and a server input queue. Each time through the loop, the `mdthread` checks the server input queue first for any outstanding response from servers. When the server input queue is empty, the `mdthread` polls from both the client and server input queues until a it receives a message. This priority biases the mux towards completing outstanding transactions before processing new requests.

When the `mdthread` retrieves a request from a client, it calls a handler corresponding to the message type. If the mux can service the request in-house, the type-specific handler constructs a response and sends the message to the client. If the request requires one or more interactions with servers, the handler saves a pointer to the request's `Msg` structure in an array called `msave`. The handler uses the index of the message in `msave` as the tag field for the first request. This guarantees that all outstanding requests from the mux on servers have unique tags. The handler constructs the rest of the first request and sends it to the appropriate server.

When the `mdthread` retrieves a response from a server, it retrieves the saved request by using the tag of the response to index into the `msave` array. If the response contains a qid, the mdthread performs the qid translation. The `mdthread` calls a handler corresponding to the response type of the saved request. This handler parses the response from the server and performs the next steps in processing the request, which may require sending another request to a server. If servicing the request requires another request, the handler saves the message as before. Since message processing may require storing the results of previous transactions, the `mdthread` can wrap other messages up in the `Msg` structure with the `mwrap` pointer. Thus a message saved in the `msave` array may also store a chain of previous requests and responses. Eventually, the `mdthread` will have all of the information from servers needed to satisfy the request, so the handler constructs a response and sends it to the client.

4.1.2 9P Message Handlers

These handlers correspond to the 9P message types. The main loop of the `mdthread` calls the type-specific handler upon receiving a 9P request.

Version

The version handler closes all outstanding `CFids` for the client. It checks that the version string specifies an accepted version of the 9P protocol. Currently only "9P2000" is supported. If the message contained an invalid version, version sends an `Rerror` message to the client. Otherwise, the handler constructs an `Rversion` message and sends it to the client. The version handler currently ignores the `msize` field.

Auth

The mux currently does not support authentication. The auth handler always sends an `Rerror` to the client. Since authentication occurs once per each 9P session, it is not clear how to provide authentication over a multiplexed 9P connection. Thus, the mux only handles servers that do not require authentication.

Attach

A client attaches to the root of the mux file tree. The `fsthread` serves the root. The attach handler creates a new `CFid` corresponding to the fid specified in the client's attach message and a new `Fid` to point to the root of the file tree. The attach handler sends a walk request to the fsthread that walks the current `Fid` pointing to the root of the name space to ".", creating a second `Fid` that points to the root. When the `mdthread` receives the response from the `fsthread` it generates a response to the client's request and sends it to the client.

Walk

The `mdthread` must keep track of mount points when handling walk requests on behalf of clients. The `mdthread` sends the complete walk message to the server associated with the fid in the client's request. When the server returns a response, the `mdthread` looks through the list of returned qids and determines if any of them correspond to files on the *from* side of a mount point. If so, the `mdthread` sends the walk from the next path element onward to the first server of the *to* side of the mount point. If the server returns an error, the mdthread tries the next server on the *to* side of the mount point. When the `mdthread` runs out of mounted servers, it returns a walk response containing the list of qids so far to the client.

The `mdthread` must also keep track of walking backwards (with the ".." element) across mount points. The mdthread stores a stack of `Mheads` corresponding to crossed mount points in the `CFid` structure. When a walk on the `CFid` crosses a mount point, the mdthread pushes the `Mhead` on the stack. When a walk contains a ".." element at a mount point, the mdthread pops the top entry off the stack to retrieve the *from* side of the mount point.

Read

If the read request corresponds to a file not on the *from* side of a mount point, the `mdthread` forwards the read to the appropriate server and returns the response.

If the request corresponds to a file on the *from* side of a mount point, the `mdthread` issues a read request to the first server on the *to* side of the mount point. When the `mdthread` receives the response, it stores the returned data in a buffer. If

response returned the requested number of bytes, the `mdthread` issues a response containing this buffer to the client. Otherwise, the `mdthread` issues a read request to the next server on the *to* side of the mount point until the total number of bytes requested have been accumulated or the last server in the union directory has responded to the read.

The `mdthread` stores the `Mhead` and the `Fid` corresponding to the last server read along with an offset in the file on that server in the `CFid` structure associated with the read. If the client issues a subsequent read on the `CFid`, the `mdthread` can retrieve the next segment of data from the stored information.

Remove

The `mdthread` must check whether the file affected in a remove message belongs to a mount point. The remove handler looks up the qid corresponding to the fid in the request in the mount table, and if the qid is part of the *from* side of a mount point, the remove handler sends an error to the client. Otherwise, the remove handler forwards the remove request to the server. If the server returns an error, the remove handler forwards this error to the client. Otherwise, the remove handler determines if the file belonged to the *to* side of any mount points. If the file belongs to a mount, the server removes the corresponding `Mount` record from the mount data structure and clunks the mounted fids. The remove handler then sends a remove response to the client.

All remove requests clunk the specified fid, regardless of whether or not the request caused the removal of the file.

Flush

The flush handler searches the `msave` array for a request with the tag specified by `oldtag` in the flush message. If it does not find a corresponding request, the flush handler sends back flush response to the client. If the flush handler discovers a matching message, it sends a flush to the server on which the `mdthread` currently has an outstanding transaction in servicing the original request. This flush request sets `oldtag` to the index of the original client request in the `msave` array. When the flush handler receives the response to this flush request, it discards the message state and responds with a flush response to the client. If the outstanding transaction on the server returns, the flush handler discards the results and continues to wait for the flush response.

Open, Create, Write, Stat, Wstat, Clunk

The handlers for these message types create a new request of the same type, translate the fid and the tag fields, and send the message to the server specified in the `Fid` structure. Upon receiving the response from the server, these handlers fill in the tag field of the response with the tag from the saved request and forward the response to the client. The clunk handler also removes the state associated with the clunked fid.

4.1.3 Muxcall Handlers

These handlers correspond to the muxcall message types. The main loop of the mdthread calls the type-specific handler upon receiving a muxcall request.

Attach

A client call to `muxattach` generates an attach muxcall message and sends it to the `mdthread`. The muxcall attach handler allocates a new `Muxcon` and adds it to the list of active `Muxcons` in the `Mux` structure. The handler constructs a response containing a pointer to the `Muxcon` and sends it to the thread calling the `muxattach` procedure.

Bindmount

The `muxmount` and `muxbind` procedures both generate a bindmount muxcall message and send it to the `mdthread`. The similarity of the mount and bind operations makes it convenient to use a single message type and handler to satisfy both requests. The bindmount handler walks to the old file to obtain a Fid that becomes the *from* side of the mount point. For a mount, the bindmount handler sets up a server thread and attaches to the server file tree to obtain a `Fid` that becomes the *to* side of the mount point. For a bind, the bindmount handler walks to the new file to obtain a `Fid` that becomes the *to* side of the mount point.

After obtaining *from* and *to* `Fids`, the bindmount handler adds an entry in the mount table. If the flag specifies `MREPL`, the handler replaces any existing mounts, and clunks the corresponding `Fids`. If the flag specifies `MBEFORE`, the handler places the new `Mount` structure at the front of the list of existing `Mounts`. An `MAFTER` flag instructs the handler to add the new `Mount` structure after any existing `Mounts`. This ordering determines the search order for union directories. The handler then generates a

response to the bindmount muxcall specifying the mount status. A positive value denotes a successful mount, and a negative value denotes failure.

4.2 Fsthread

The `fsthread` serves a file tree from memory using the RAM based file system `ramfs` [11]. A call to `muxalloc` starts the fsthread. When the fsthread starts, it creates a RAM file for the root directory of the file tree. `Muxalloc` then creates the mdthread, which begins a 9P session with the `fsthread` by performing version and attach transactions. It then enters a loop that reads a request from the `mdthread` from its input queue, serves the request from its file tree, and sends the response to the `mdthread`.

5 Device 9P Servers

To provide access to a device through the mux, the devices must serve 9P. We focus on providing 9P access to the disk and network devices, which are sufficient to run the web server. We use the Plan 9 file server to provide access to files on disk. We alter the network device to export 9P access to the user level.

Once a device provides user-level 9P access through a file descriptor, the mux can present an asynchronous interface to the file tree served by the device. A call to `muxmount` on the file descriptor attaches the device file tree into the mux namespace. Clients can asynchronously access the device with the `muxsend` and `muxrecv` functions, as described in section 3.2.

5.1 File Server

The Plan 9 file server, fossil, currently serves 9P [12]. A client establishes a connection with it to obtain a file descriptor (`fd`). The client writes 9P requests to the `fd` and can retrieve responses from fossil by reading from the `fd`. The client calls `muxmount` to mount the file tree served by fossil into the mux name space. Since the mux currently does not support authentication, fossil must run with authentication disabled.

Fossil runs a user level program and uses multiple kernel threads to serve requests. Fossil creates two new kernel threads to handle I/O on each new 9P connection from a client. One thread reads requests and the second writes responses to the connection. When an I/O process receives a request, it places it on a central message queue. Fossil

creates service threads to read messages from the central queue. The service threads serve cached data immediately and interact with other threads that perform disk I/O to serve disk-bound data. Fossil employs one disk I/O thread for each physical disk.

Fossil places a fixed limit on the number of service threads it creates as well as on the number of outstanding messages in the system. If the incoming load causes fossil to reach either of these limits, new requests will stall until fossil finishes processing a current request.

5.2 Network Device 9P Server

To provide 9P access to the network device at the user level, we provide export, a new system call. A call to export returns a 9P connection to the network device via a pipe.

The kernel implementation of export is derived from the implementation in Inferno operating system, changed to eliminate copies [2], [5]. The implementation consists of a number of kernel threads running in the kernel's address space (kprocs). A main kproc reads 9P requests from the pipe, passing each request to one of a pool of kprocs (exslaves) that run the incoming requests. The exslaves call the network device procedures to perform I/O and then write the 9P response to the pipe. If all exslaves are busy handling requests when a new request arrives, the main server process creates a new exslave. The implementation imposes no explicit limit on the number of exslaves, so incoming requests do not stall waiting for exslaves.

6 Evaluation

To evaluate the design, we analyze the performance of the mux in isolation and also carry out performance testing on a web server that uses the mux to perform asynchronous I/O.

6.1 Mux Performance

In order to achieve performance benefits from asynchronous I/O, the implementation must be efficient. In particular, we measure the latency of 9P transactions handled by the mux. We perform a series of tests to determine how message latency degrades as different factors increase.

In these tests, we consider the scalability of the mux with respect to the number of clients and servers. Additionally, we test the performance of the mux name space of the mux by measuring the latency of messages that walk across mount points and read union directories. For each of these tests, we take the average value over a series of trials, ignoring the first few data points. The latencies of the first several messages suffer, because the newly visited pieces of code are paged into memory. We discount these latencies because they do not accurately reflect the steady state behavior of the system.

6.1.1 Client and Server Scaling

In the client and server scalability tests, we measure the latency of a set of 9P transactions (the message suite) as the number of clients and servers increase. We run three separate tests. The client scalability test measures latency with one server and an increasing number of clients. The server scalability test measures latency with one client

and an increasing number of servers. The combined scalability test measures latency as both the number of clients and servers increase.

Each test sets up a user thread for each client and starts a process running `ramfs` for each server. The client scalability test does not set up a `ramfs` server, but instead uses the internal mux file server to avoid measuring overhead from external servers. For each `ramfs` server, the test creates a directory in the mux namespace and mounts the root of the server at that directory. Clients attach to the mux and sequentially run the message suite on each server in the test. The message suite consists of an instance of every 9P message type except auth and flush.

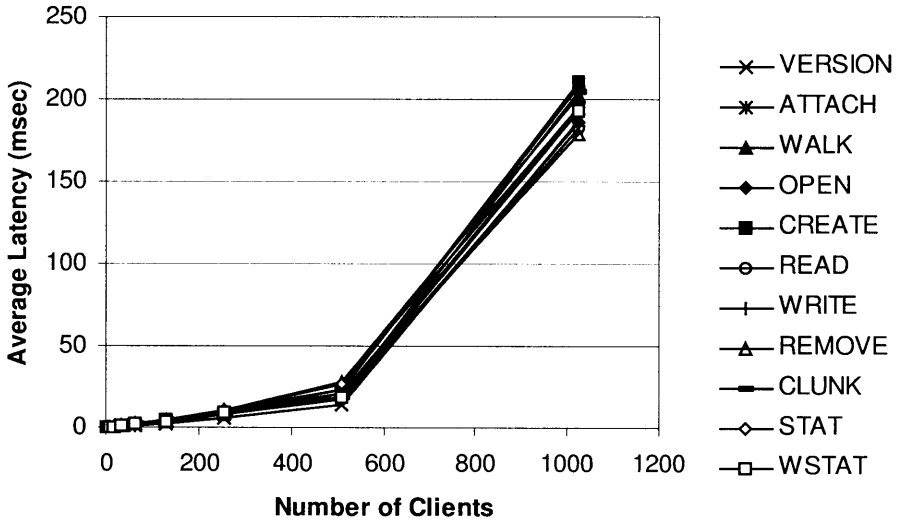

Figure 8: Client scalability. In the test, clients simultaneously perform a message suite. Latency becomes steeply non-linear after the queue size of 512 is exceeded and the mdthread begins to block.

Figure 8 shows the results of the client scalability test. The figure shows nearly linear growth until the 512 clients. After this point, latency increases sharply. This comes from the use of fixed-sized queues in the mux implementation. For these tests, we use a queue size of 512 messages. A thread attempting to send a message on a full queue

will block. Therefore, the central dispatching thread may block if the internal file server's input is full even though the dispatching thread could be processing outstanding responses to earlier messages. If the queue implementation were replaced with one that grew dynamically, we would expect the increase in latency to remain linear.

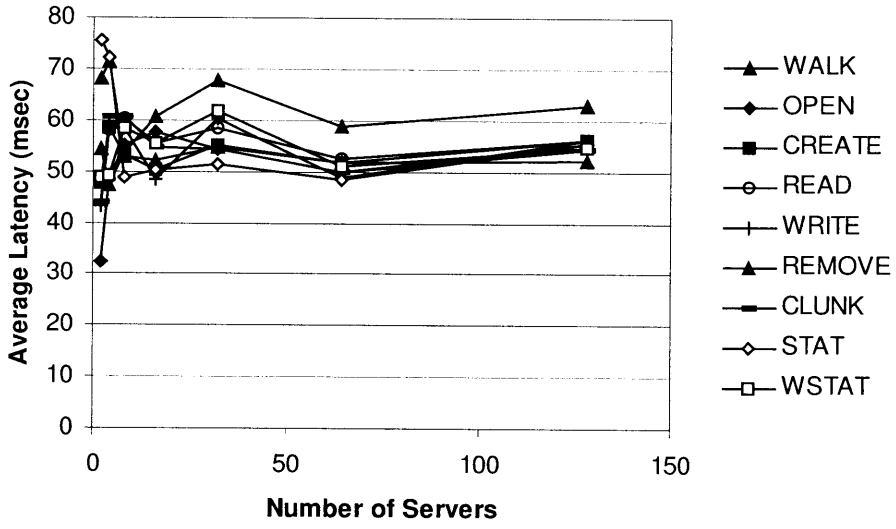

Figure 9: Server scalability. A single client loops over all servers and performs the message suite. The latency is dominated by the server response time.

Figure 9 shows the results of the server scalability test. Since the client executes its script on each server in turn, the latency remains fairly constant as the number of servers increase. Much of the latency comes from the time to send and receive messages from the external server. Indeed, we observed an average time of 40-50 ms to carry out a transaction with the `ramfs` server.

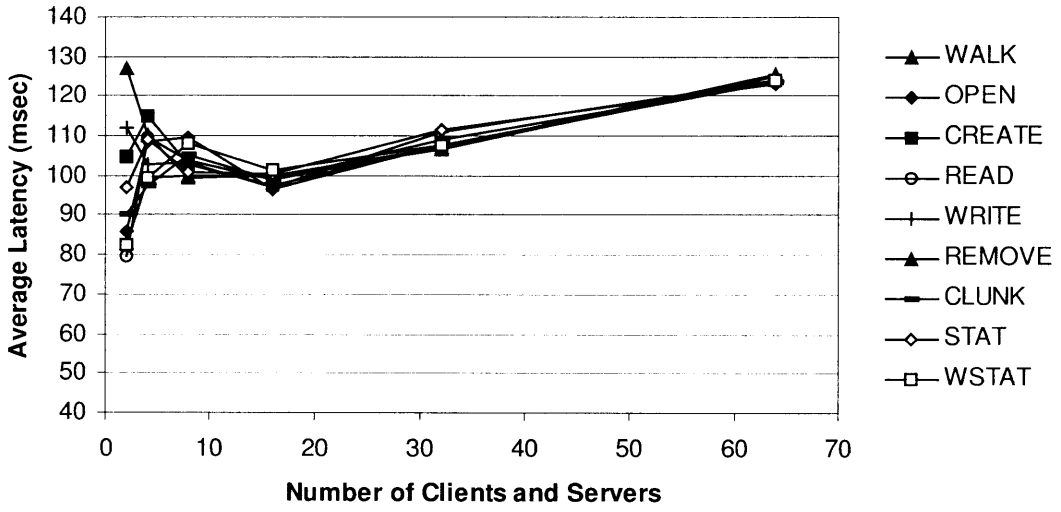

Figure 10: Combined scalability where the number of clients equals the number of servers. Each client loops through the servers performing the message suite. After some initial variation in the latency, the long-term behavior is linear in the number of clients and servers.

Figure 10 shows the results of the combined scalability test. In this test, all clients start by running the test suite on the same server. When a client finishes, it moves on to the next server. For a small number of clients and servers, there is variation in the average message latency. After 16 clients and servers, the latency scales linearly. For small numbers of clients, the system may not reach a steady pipelined state. In this case, some messages may be delayed longer than others depending on the timing of the server responses.

6.1.2 Walk Performance

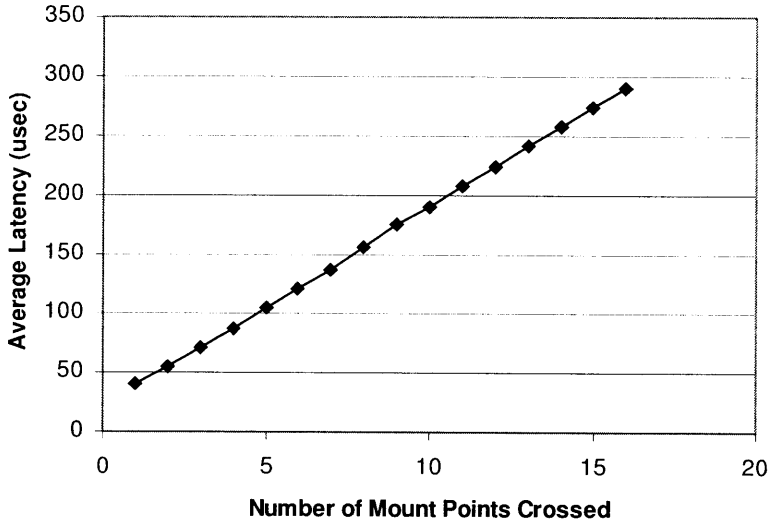

Figure 11: Walk scalability. This test measures walk latency over a series of mount points.

The walk test measures the latency of a single walk message across a series of mount points. Each instance of the test binds a flat hierarchy of directories together into a single chain. A client attaches to the mux and performs a walk to the end of this chain, crossing a mount point for each element in the path. Figure 11 shows the results of this test. We test up to sixteen mount points, which is the maximum number of elements allowed in a single walk message. The linear behavior indicates that the mux imposes only constant overhead for each additional mount.

6.1.3 Union Read Performance

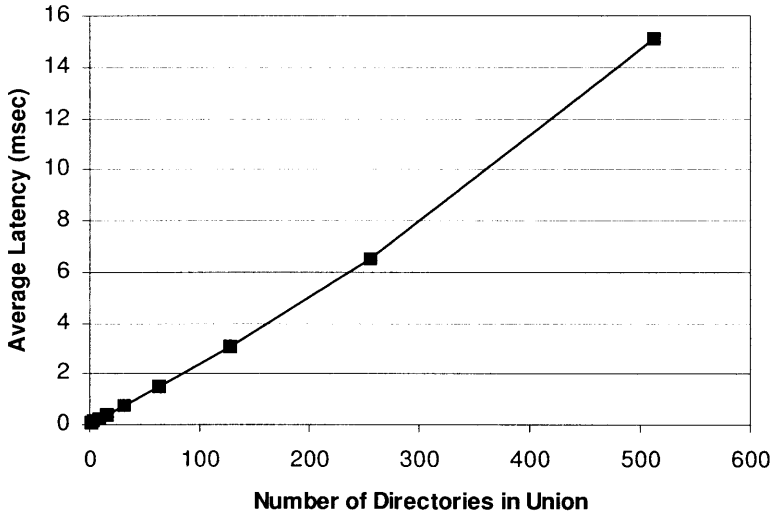

Figure 12: Union read scalability. This test measures the latency of a read message on a union directory consisting of an empty directory repeatedly bound to the end of the union.

The final test of mux performance measures the latency of union reads. This test repeatedly binds an empty directory at the end of a base directory. Thus, a read on the base directory will cause the mux to perform an intermediate read message for each instance of the empty directory in the union. Figure 12 shows the results of the union read test. The graph shows that the read latency grows super-linearly, but with a small factor. It is unclear why this occurs, but for union directories of most sizes used in practice, the growth is essentially linear.

6.2 Web Server Performance

To test the viability of using the mux to provide asynchronous I/O, we perform benchmark tests on a web server that uses asynchronous I/O to overlap network and disk operations with request processing. We start with the Plan 9 multi-process web server, httpd, as a basis for comparison [11]. We modify this web server to use the user thread

library and perform asynchronous I/O through the mux. We present the modifications made to httpd in the next section and the performance comparisons between the modified web server (uhttpd) and httpd in the subsequent section.

6.2.1 Uhttpd Design

To allow the web server to use the user thread library, we replace the I/O system calls (`open`, `read`, `write`, etc.) with library functions that block at the user level. Thus, the thread manager can switch between blocking threads to manage concurrency. To provide the library functions, we use an existing library (`libfs`) that implements I/O procedure calls by carrying out the appropriate 9P transactions on a file descriptor. We modify `libfs` to use `Muxcons` instead of file descriptors.

To implement uhttpd, we replace all instances of I/O system calls in the httpd and network code to use the modified `libfs` calls. Uhttpd begins by allocating a mux and mounting the file trees served by fossil and the network device 9P server into the mux namespace. Next, the web server attaches to the mux and registers the `Muxcon` with `libfs`. When uhttpd performs an I/O operation, `libfs` carries out the appropriate 9P conversation with the mux and any blocking will occur at the user level.

6.2.2 Benchmark Setup

To evaluate the performance of the non-blocking web server, we run a set of benchmark tests on uhttpd and httpd and compare the results. We run the tests with the profiling tools used to measure the Flash web server [7]. The tests use multiple directories each containing 5 MB of data in files varying from 102 bytes to 921 KB for the web server data set. This data set is the same as the one used for SpecWeb99

benchmarks. The workload is a Zipf distribution heavily weighted toward small files. We vary the number of directories served to measure cached versus disk bound workloads and vary the number of clients to measure scalability. We run each test for 5 minutes to allow caches to reach a steady state, wait 5 minutes with no load to stabilize the caches, and then run the tests for 5 minutes and record the results.

We run the benchmarks on a server machine and a client machine connected with copper gigabit Ethernet through a NetGear model GS504T gigabit Ethernet switch. Both machines have Intel PRO/1000 MT network cards. The web server runs on a 3 GHz Pentium 4 CPU and 2 GB of RAM running Plan 9. The client runs Linux 2.6.5 on a 2.4 GHz Pentium 4 CPU and 1 GB of RAM.

6.2.3 Benchmark Results

To evaluate the web server, we run tests to measure throughput as the number of clients scale. All clients run concurrently on the single client machine. We run each throughput test on two different workloads. We measure the in-cache performance by using a 10 MB workload (2 directories). To measure disk bound performance, we use a 3.4 G.B workload (679 directories). Since httpd does not employ a cache, it retrieves data from the file server on every request.

Figure 13 shows the results of throughput tests. The graph shows that uhttpd generates a lower throughput than httpd on the in memory workload, but that the performance of uhttpd degrades more slowly as the number of clients increases. For 64 clients, the latency of uhttpd is 18% worse than httpd.

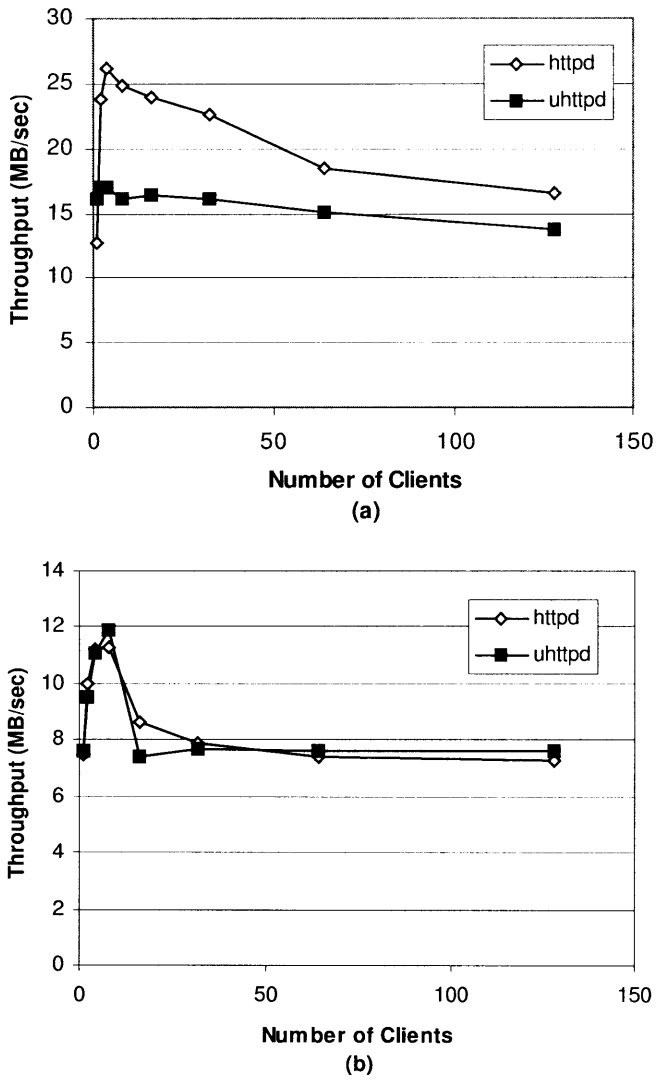

Figure 13: Web server throughput comparison. The throughput for the in memory workload is shown in (a). The disk bound workload is shown in (b).

Figure 14 shows the cumulative distribution of response times for 32 clients. This graph indicates that httpd serves more requests than uhttpd does for the in memory workload. For the disk bound workload, httpd and uhttpd exhibit almost identical characteristics. Interestingly, the steeper slope of the uhttpd distribution function for the

in memory workload indicates that the latency of almost all requests is essentially the same.

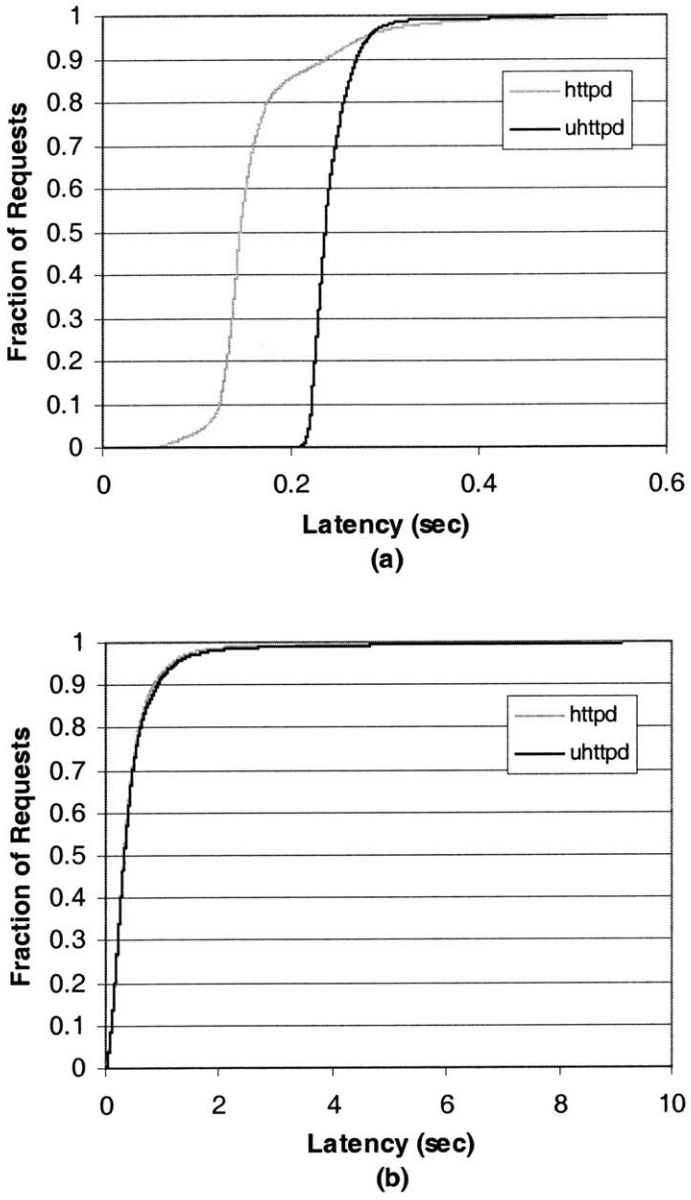

Figure 14: Cumulative distribution of response times for 32 clients. The results for the in memory test are shown in (a). The disk bound test results overlap and are depicted in (b).

6.2.4 Discussion

The results of the benchmarks indicate that the user-threaded web server performs worse than the kernel-threaded implementation on in memory workloads and comparably for disk bound workloads. The performance of the uhttpd server degrades more slowly than httpd as the number of clients increase because the uhttpd does not require a new process for each request.

Neither httpd nor uhttpd is optimized for high performance. In particular, the implementations copy data a handful of times in the process of serving requests. Figure 15 outlines the memory copies made on the data path of a request and response for httpd and uhttpd. Many of these copies could be eliminated with the use of zero-copy I/O mechanisms [8]. Additionally, the user-level copies in uhttpd between the web server memory and the mux I/O processes could be eliminated with a more careful implementation of the asynchronous system call interface. These extra copies are the likely cause of both the throughput and latency gaps between uhttpd and httpd for the in memory workload.

The steep slope of the cumulative distribution function in uhttpd indicates that the mux provides a fair message servicing policy. The queuing in the system provides regularity in request serving.

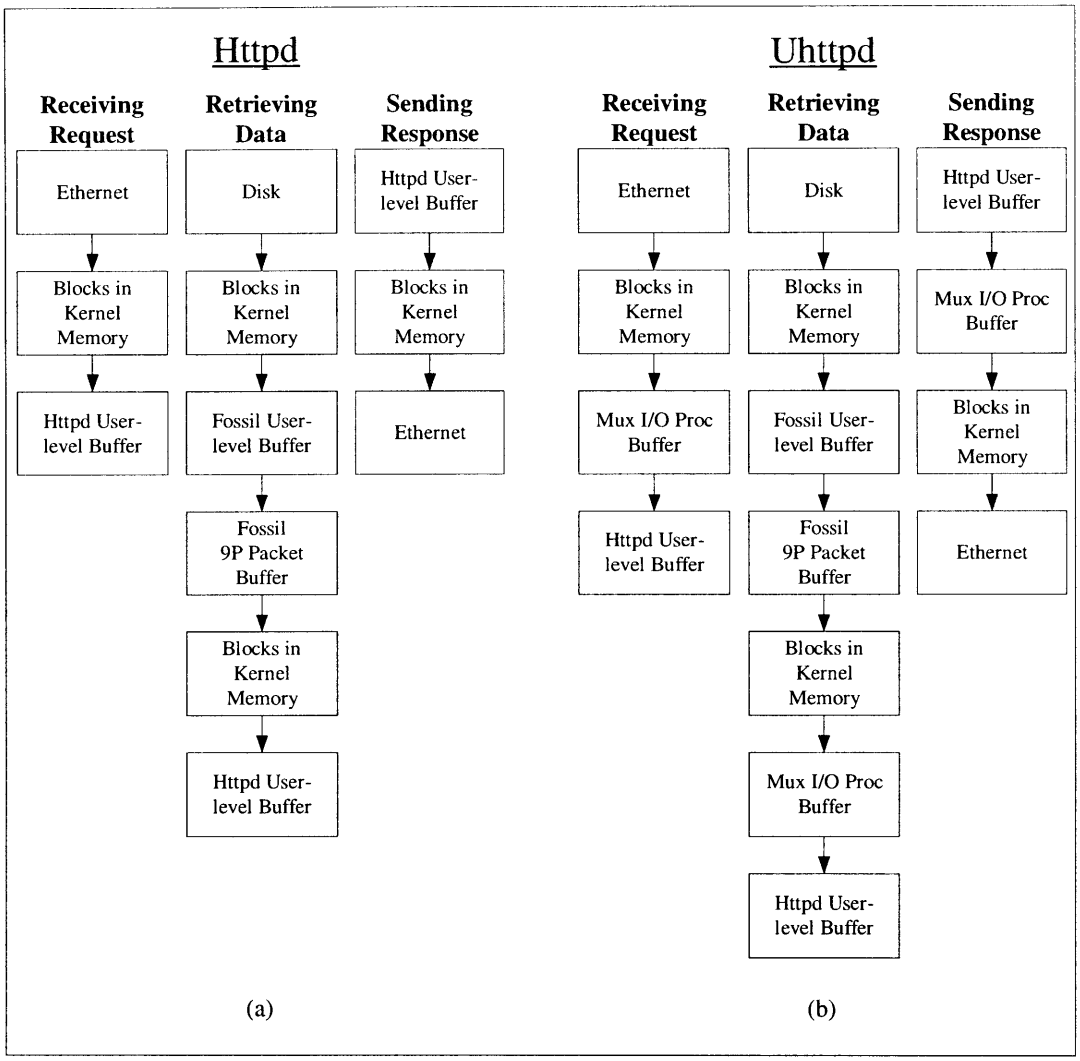

Figure 15: Data copies in request serving. Each arrow in a chain represents a data copy. (a) shows the copies in httpd. The copy path for uhttpd is shown in (b). Uhttpd requires an additional copy in each operation.

7 Conclusions and Future Work

7.1 Summary

The scaling tests performed on the mux indicate that the mux scales well up to hundreds of clients and over a hundred servers. These results lead us to conclude that the mux implementation is correct and demonstrates the feasibility of the message passing design of the mux.

The tests performed on the user-threaded web server indicate that the mux implementation of asynchronous I/O performs comparably with the standard I/O interface. These results demonstrate the feasibility of using the mux to provide asynchronous I/O. The performance benefits of overlapping processing with device operations are not immediately apparent from the tests we performed, but we note that the performance of httpd is largely independent of the kernel scheduling policy. Any ready thread that the kernel chooses to schedule will contribute to the overall throughput numbers. This observation leads us to conclude that an application that requires a more complicated or different scheduling policy than the one employed by the kernel may show performance improvements by using the mux interface.

By providing asynchronous I/O, the mux interface allows fine-grained control over user-level thread scheduling. This control gives programmers the freedom to design applications that rely on more advanced scheduling algorithms than previously possible.

7.2 Future Work

Future work involves moving the user-level mux library into the Plan 9 kernel. Running in the kernel should improve the efficiency of the interface by eliminating the need to export 9P access to devices to the user level. The kernel implementation of the mux would present a single 9P connection to the user-level client. The implementation would still require a user-level library to provide functions for sending and receiving 9P messages that block at the user-level.

In the meantime, future work involves improving efficiency of the user-level mux interface. In particular, eliminating user-level copies of data would improve performance of the uhttpd implementation. As orthogonal issue to the work presented in this thesis, future work could also involve making the underlying httpd web server code more efficient. The number of copies on the data path could be reduced by employing zero-copy I/O techniques [8]. Eliminating the copying overhead would make the context-switching overheads more apparent and should demonstrate better performance of uhttpd on disk bound workloads relative to httpd.

The mux interface provides greater functionality to user threads by allowing blocked threads to be scheduled at the user level. A final avenue for future work involves exploring the benefits of application-specific user-level thread scheduling.

8 References

[1] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. of the Thirteenth ACM Symposium on Operating Systems Priciples,* pages 95-109, October 1991.

[2] S. Dorward, R. Pike, D.L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. Inferno. In *Proc. of the IEEE Compcon 97 Conference,* San Jose, CA, 1997.

[3] D.R. Engler, M.F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles,* pages 251-266, 1995.

[4] FreeBSD Hypertext Man Pages. http://www.freebsd.org/cgi/man.cgi.

[5] Inferno, 4th edition. Distribution. http://www.vitanuova.com/.

[6] D. Mazières. A toolkit for user-level file systems. In *Proc. of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

[7] V.S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proc. of the USENIX 1999 Annual Technical Conference,* pages 199-212, Monterey, CA, June 1999.

[8] V.S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. In *ACM Transactions on Computer Systems,* 18(1):37-66, 2000.

[9] V.S. Pai, and Y. Ruan. Making the "Box" Transparent: System Call Performance as a First-class Result. In *SOSP 2003*. Princeton, NJ, 2003.

[10] R. Pike, D. Presotto, K. Thompson, H. Trickey. Plan 9 from Bell Labs. In *UKUUG Proc. of the Summer 1990 Conf.*, London, England, 1990.

[11] Plan 9 Programmer's Manual, Volume 1, Third Edition, AT&T Bell Laboratories, Murray Hill, NJ, 2000.

[12] S. Quinlan, J. McKie, R. Cox. Fossil, an Archival File Server. http://plan9.bell-labs.com /sys/doc/fossil.pdf.

[13] Sun Microsystems. Solaris 7 Manual.

9 Appendix - 9P

Plan 9's network file protocol, 9P, provides a client with an asynchronous interface to files served by a 9P server. A client sends a T-message (request) to a server and receives an R-messages (responses) from the server. A request and its corresponding response constitute a transaction.

Each request contains a 16-bit *tag*, which clients choose to uniquely identify transactions. The server echoes a request's tag in the corresponding response. To guarantee uniqueness, a client must not send a request with the same tag as an outstanding request. Most request types contain a *fid*, a 32-bit unsigned integer used by the client to reference a file on the server. Clients choose fids and must guarantee their uniqueness. Several response types transmit a *qid* to the client. Assigned by the server, the thirteen-byte qid field uniquely specifies a file in the server's file hierarchy. The first byte of the qid, the *type*, conveys information about the file, such as whether it is a directory, append-only, etc. The next eight bytes, the *path*, uniquely identify the file on the server. The final four bytes, the *version*, contain a version number for the file. Typically, the server increments the version field on each file modification.

In all, 9P has 13 message type pairs (T-message and corresponding R-message) and one unpaired response, an *error* message, for notifying the client when a request results in an error. Figure 16 summarizes these message types.

A 9P message may have one of two formats. The first, called an `Fcall`, is a C structure used for message processing. A 9P message in an `Fcall` gets converted into a machine-independent byte stream for transmission over a 9P connection.

Message	***Description***

`Tversion tag msize version`
 	
`Rversion tag msize version`
	Establishes the version of the 9P protocol (`version`*) and the maximum message size (*`msize`*) that will be used.*

`Tauth tag afid uname aname`
 	
`Rauth tag aqid`
	Used for authentication. `Afid` *is a fid used to authenticate the user named by* `uname` *to the file tree specified by* `aname`*. Returns the qid (*`aqid`*) for the file to be used to carry out an authentication protocol.*

`Tattach tag fid afid uname aname`
 	
`Rattach tag qid`
	Attaches fid *to the root of the file tree specified by* aname *on behalf of a user (*uname*). The* `afid` *is previously established with an auth transaction. Returns the* `qid` *of the root of the file tree.*

`Twalk tag fid newfid nwname nwname*(wname)`
 	
`Rwalk tag nwqid nwqid*(wqid)`
	Walks a fid (`newfid`*) through* `nwname` *path elements (the* `wnames`*) in the file hierarchy starting at the file pointed to by* `fid`*. Returns a list of visited qids (the* `wqids`*)*

`Topen tag fid mode`
 	
`Ropen tag qid iounit`
	Prepares `fid` *for I/O and specifies the type of I/O (*`mode`*) that will be performed. Returns the* `qid` *of the file and the maximum size to be read/written without breaking the message up (*`iounit`*).*

`Tcreate tag fid name perm mode`
 	
`Rcreate tag qid iounit`
	Creates a file (`name`*) with specified permissions (*`perm`*) and opens for specified type of I/O (*`mode`*) in the directory pointed to by* `fid`*. Returns the* `qid` *of the new file and* `iounit` *as in open.*

`Tread tag fid offset count`
 	
`Rread tag count data[count]`
	Reads `count` *bytes of data at* `offset` *from the file pointed to by* `fid`*. Returns the data and number of bytes actually read (*`count`*).*

`Twrite tag fid offset count data[count]`
 	
`Rwrite tag count`
	Write `count` *bytes from* `data` *to* `offset` *in the file pointed to by* `fid`*. Returns number of bytes actually written (*`count`*).*

`Tflush tag oldtag`
 	
`Rflush tag`
	Abort an outstanding message specified by `oldtag`*. When the client receives an* `Rflush`*, it may reuse* `oldtag`*.*

`Tclunk tag fid`
 	
`Rclunk tag`
	Informs the server that `fid` *is no longer needed. After receiving an* `Rclunk`*, the client may reuse* `fid`*.*

`Tremove tag fid`
 	
`Rremove tag`
	Remove the file pointed to by `fid`*. Also clunks* `fid`*.*

`Tstat tag fid`
 	
`Rstat tag stat`
	Return a structure (`stat`*) containing file attributes for the file pointed to by* `fid`*.*

`Twstat tag fid stat`
 	
`Rwstat tag`
	Write file attributes (`stat`*) for the file pointed to by* `fid`*.*

`Rerror tag ename`	*Returns an error string (*`ename`*) on an unsuccessful request.*

Figure 16: 9P message types.