A Stackable File System for Multiserver Environments

Michael Daub

University of Karlsruhe, Germany

Abstract

Microkernels have become an important instrument in tackling the challenges of more and more complex operating systems. With the multiserver approach, they provide us with a high level of security, flexibility, extensibility, robustness and clarity in system design.

In this work, we describe a stackable file system design that is based on the multiserver philosophy. We also discuss performance issues that appear as we migrate from monolithic systems where we can easily pass data buffers by pointers, to a system design with separate address spaces.

Contents

Chapter 1

Introduction

1.1 Motivation

As future operating systems are getting larger and more versatile, system design and maintenance becomes more and more complex. Adding new functionality often affects existing functionality, and security issues are leading to large drawbacks by consuming much time and manpower, and thus financial resources.

In this scenario, microkernel systems become more and more important. By assigning every application its own private address space, system designers gain a high level of fault containment and security. On microkernel systems, threads communicate only via IPC (Inter-Process Communication) through well-defined interfaces, which makes it difficult for one process to affect another process in any harmful way. This way, the system becomes very robust and gains a high level of stability and reliability.

The term "microkernel" already suggests that the kernel itself is very small. It only implements basic abstractions and mechanisms for thread control (including thread communication) and address space management. The operating system itself is stacked on top of the microkernel as several mostly independent server threads working together. With this approach, single components of the system can easily be replaced or adapted to personal needs, giving system designers and administrators a high level of extensibility and flexibility.

As every server thread runs in user mode (only the microkernel itself runs in kernel mode), there is no clean line between system components and applications. Basically, every application can implement some functionality and make it available to every other thread on the system.

This system layout is usually termed a "multiserver environment", where there is no communication with a traditional operating system, but rather with many different servers, each implementing some special kind of functionality.

1.2 Problem definition

The topic of this work is how to expand the idea of multiserver environments to modern and flexible file systems. That is, how to design a virtual file system that consists of several independent components implemented as server threads on a multiserver system. The fact that traditional file system interfaces are designed for monolithic

systems with a common address space for all components becomes a big challenge we have to tackle.

1.3 Thesis overview

In the next chapter, I will give some more background on the idea of stackable file systems and present some steps that were already taken in the past. After that, I will explain in detail what problems occur when trying to implement file systems as multi-server components, and how they can be solved. The functionality will be proven by my experimental implementation of a tiny file system, which is also used to do a wide range of performance measurements. Finally, I will recap my work and give some overview about what we have to keep in mind when using stackable multiserver file systems.

Chapter 2

Background

Moving forward towards a new file system design, I will first give a short review of current file system design and the idea of stackable file systems. In this chapter, I will point out some details on file systems that we can't keep on a multiserver environment, while some other approaches might be very useful to our plans.

2.1 Traditional file systems

On current designs, the operation system provides the user with a well-defined interface for file operations. This file interface hides the logical and physical layout from the user by introducing abstractions like files, directories, file descriptors and file systems.

Of course, on most systems there are multiple file systems, e.g., one for each hard disk, ram disk, exchangeable disk or USB drive. On some systems (like UNIX) this setup is also hidden from the user, etc. In this case, there is a root folder for the complete system and each file system used on the machine is mounted into a hierarchy of subfolders.

All handling behind the interface is embedded into the operating system and is thus transparent to the user.

2.2 The vnode approach

The vnode/vfs [7] framework was introduced by Sun Microsystems to provide a consistent view of a homogenous file system on a set of heterogeneous file systems. It implied the possibility of adding new file systems in a modular manner. It is based on an object-oriented design.

The vnode abstraction represents a file in the UNIX kernel. It is a structure containing a data part and a function pointer part, which represent the basic file system interface. The vnode data part also contains two pointers to file-system-dependent private data and a list of file-system-dependent functions. In the view of object-oriented programming, these pointers to file-system-dependent code can be seen as an implementation of a subclass from the basic vnode class. The basic vnode class defines the global interface for files, and for each physical file system, there's a subclass derived that finally implements its functionality.

2.3 Stackable file systems

The original idea of stackable file systems was to add new functionality to existing file systems [6]. It should be possible to split a complex file system into logically independent parts that can be replaced and adapted in a modular manner. Each layer implements a special functionality and layers not needed for a special call are skipped.

Figure 2.1 shows an example of a possible file system stack. On the lowest level, we have the device driver that gives us access to the physical data storage. One level above, we have a file system implementation, for example FAT. Stacked over the file system level, we have introduced a security level which controls the access to the files. This way, we can implement complex strategies like access rights not only dependent on users but also on applications or even on the current state of the file system (for example, while one file in a directory is opened by a specific application, all files in this directory are locked to other applications). Alternatively, we could imagine an encryption/decryption layer for read and write calls. Of course, on the very top of the stack, we have the client that works with the file.

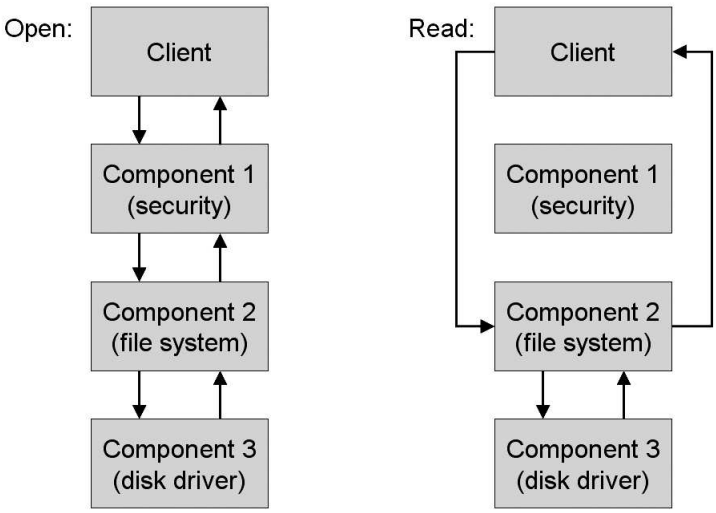

Figure 2.1: Stacked file systems

The important thing that we can see here is that on the open call, all levels of the stack are passed through. But on the read operation we can skip the security level as it is only required for open operations.

2.4 Stackable vnodes

Between 1990 and 1994 there were first efforts at the University of California at Los Angeles (UCLA) and at Sunsoft [4, 5] to redesign the vnode concept moving towards stackable file systems. These frameworks basically represent each file on the system by a stack of vnodes instead of a single vnode. When a file operation is called, the kernel hands the call to the topmost vnode. Each vnode may either process the call completely and return to the caller, or do some processing and pass the call down to the

next vnode on the stack. Of course, it can also hand the call down without doing any processing, just like we can see on the read call in Figure 2.1 where the security level is simply skipped.

Chapter 3

Challenges to multiserver environments

As we have seen on Chapter 2, current file systems are implemented inside the kernel and as part of a monolithic operating system. We have also seen that stacking was implemented as a sequence of function calls from a list of vnodes.

On a multiserver system we have disjunct address spaces, so we can't simply call functions from a vnode list. Further, we need to figure out where we keep our vnodes. As we don't have commonly shared memory among the different file system servers (and we don't want it here - not only for security and complexity reasons, but also because the system design can more easily be extended to distributed file systems when there is no shared memory), the only possibility would be to create an interface server that implements the known file system functionality from monolithic systems, and that calls the corresponding functions on different server threads. But, as we can see later, simply porting the old design to a multiserver environment in this way would result in a vast amount of data transfer overhead which would cause an unacceptable loss of performance.

Instead, we decided to completely redesign the file system layout. The first decision was that, following the multiserver philosophy, each file system and each layer has its own server thread and address space. Each of these servers implements a well-defined file system interface. Single file servers are loosely connected, so when a user works with files, he always works with a set of file servers, either in a direct or in an indirect way.

File servers work together in a client server manner. Each file server may use files on another file server to provide its own files. For example, the FAT layer from Figure 2.1 may use a file on a ram disk server which only exports one single file representing the core memory for the ram disk. The FAT server then implements the functionality for a FAT file system and exports all the files on the ram disk to other clients (which may be applications or additional file servers - the FAT server itself doesn't care about that).

It should be noted that there is no global file server and thus no such thing as a global root directory. Each file server implements its own namespace. However, file servers can be linked together, so that one server can act as a gateway to files on other servers. The user may open a file through this gateway, but use a different file server for working with the file.

3.1 Namespace

On current file systems, the vnode represents a file on the whole system. But on a multiserver system, this is not sufficient. The most important thing we have to know about a file is which server is responsible for implementing that file, so a simple vnode or file descriptor without server information would be useless.

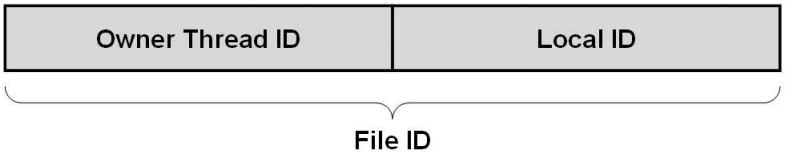

Figure 3.1: Composition of a file ID on multiserver systems

Assuming a file server handles more than one file, it has to identify each file with a local identifier which could be a vnode number or a file descriptor. This can be seen as a namespace local to the server thread. On the other hand we have the system-wide thread ID namespace which provides each server thread with a unique ID. So, we have a hierarchy of namespaces, and we can combine them like in Figure 3.1 to a new namespace. On our file system design, we can use the resulting identifiers as file IDs that contain all information we need to localize a file. Once we want to work with it, we simply take the local ID part from the file ID and send it with a function call to the thread ID part that represents to the file server itself.

3.2 Bypassing server levels on the stack

Now let us take a look at the stacking. On existing stackable file systems, like the stackable vnodes approach, we enter the code for each vnode on the stack and immediately return and jump to the next level, if there is no processing done by the called vnode level.

Now, as our components on the stack have become independent server threads with their own private address space, each call to another server level leads to additional overhead. Not only is address space switching and entering the kernel for a thread switch expensive. As we don't have a shared memory design, we also have to transfer the complete working set to the next level. This means we have to send the whole data on read/write operations. It is clear that this would lead to an unacceptable level of overhead on large data blocks. The only way we can avoid this situation is to let each server level know which server implements the functionality for the next function call, so that this call can be directed to the component handling it. This way we don't waste bandwidth to transfer data to a component that doesn't need to access the data.

Now the question is how we can implement this functionality. If we take a look at the approach from Section 3.1, this can give us a good indication on how to tackle this task. On the namespace considerations, we expanded the local ID with the thread ID of the server that's handling the file because we need this information to be able to work with the file. But if we can add one thread ID, we also can add multiple thread IDs, each associated with a unique functionality, or better said, with one or several functions offered by the file system interface. How many different server pointers we

add depends on the precise file system design we want to use. For example, we can add two additional server pointers, one for read operations and another one for write operations, while the remaining functions are handled by the base server of the file. Figure 3.2 shows how an expanded file ID might look like. Let's call it a file handle. Instead of returning the file ID when opening a file, we now return the file handle.

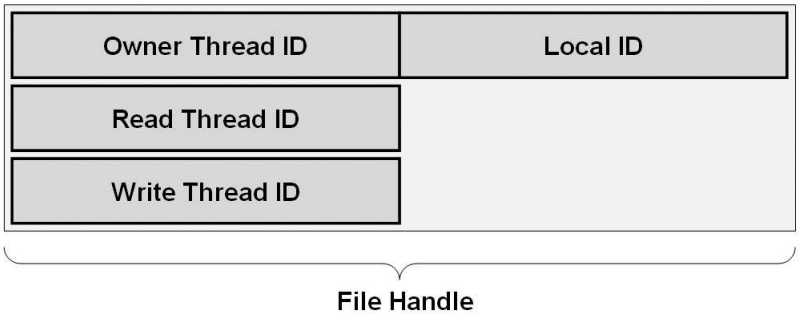

Figure 3.2: Expanding the file ID with additional server information

3.3 Consistency and security

In this section, we will take a closer look on the communication between the server levels.

At the time we open a file there are only two servers on the system that know about this file: The server that hosts the file and the client that opens it. But what happens if the hosting server already works with another file server lower on the stack, and wants the client to send some requests (like read operations) directly to that server? Without additional communication, the server lower on the stack doesn't have any information about how to handle requests to a file that's located on another server.

Another issue might be the question of who may access files on a server. A file server that doesn't implement access control may leave this issue to other server levels. In the beginning, the server implementing access control is granted access to all files. But later, it might want to forward its access rights for a special file to a client that is allowed to work with that file, so that after opening the file we don't have to face any performance loss caused by the intermediate server level. For this reason, we need some mechanism to tell the underlying file server that we want to give a part of our access rights to another thread which is a client to us, but which might also recursively act as another file server level to other threads.

3.3.1 File capabilities

To satisfy these needs, we introduce a capability system. The capabilities can be seen as access rights to a file and reside in the server thread that holds that file. Each request from a client to a file is associated with a capability on the file server. This implies that each capability has two key fields specifying the thread ID of the client that's allowed to work with it and the file ID that the client is using for its requests.

As the file ID of higher server levels (for example the file ID of the files that our client exports to other threads) is not directly associated with the file residing on our file server, we need a third field that maps the file ID specified within the capabilities to a local file, or a subsection of a local file. Figure 3.3 shows a possible layout for such a capability.

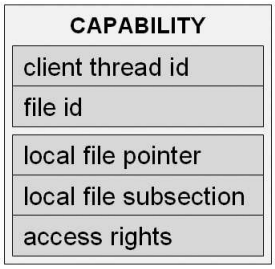

Figure 3.3: Possible layout of access capabilities on stacked file systems

Now, the client can forward its own capability, either in whole or in part, to another thread that becomes a new client to the file server. To do so, the client uses a special function call to instruct the server to create a new capability for another thread that inherits some access rights from the calling client. The server duplicates the capability from the client, sets the new client thread ID and the new file ID (that were part of the function call) and customizes the access rights. Of course, the new thread can't inherit more access rights than those the client already owns. Figure 3.4 illustrates the duplication of a capability.

3.3.2 Working with capabilities

We will demonstrate this capability system on a little example. In Figure 3.5 (a) we have two threads. Thread A is acting as a file server and thread B as a client working with file A:x (file x on server A). As we can see, all communication is going through the capability created on server A.

In the next setup (b), B has decided to become a file server itself and exports a file called y. We can also see that a new client C appeared. Now C is working with B:y in just the same way that B is working with A:x. In this configuration, server A is completely invisible to the client C.

Now let us assume that file B:y is a subsection of A:x. To increase performance, read and write calls that cause much overhead can directly be sent to server A. This is what we see in figure 3.5 (c). Here we have one additional capability field in server A that allows client C to work with B:x that is covered by a section of A:x. As the client only knows file B:x, the new capability maps the calls to a subsection of the local file A:x.

Depending on the concrete implementation, we can also create a mechanism to completely pass an entire file to another server, which would mean in our example that B passes the file it opened on A completely to client C and disappear after that. The result would be a server setup exactly like in Figure 3.5 (a), just that we have thread C at the position of thread B. In this way we can easily implement gateway servers that

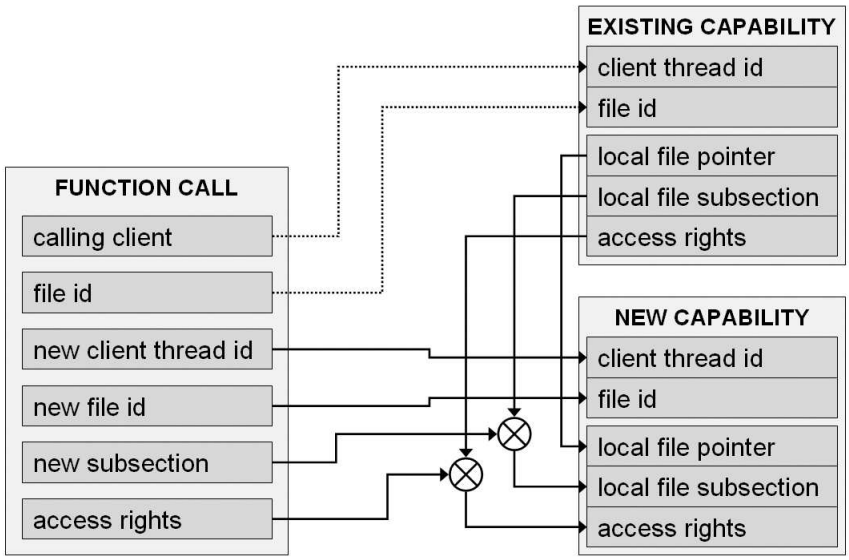

Figure 3.4: Duplication of access capabilities

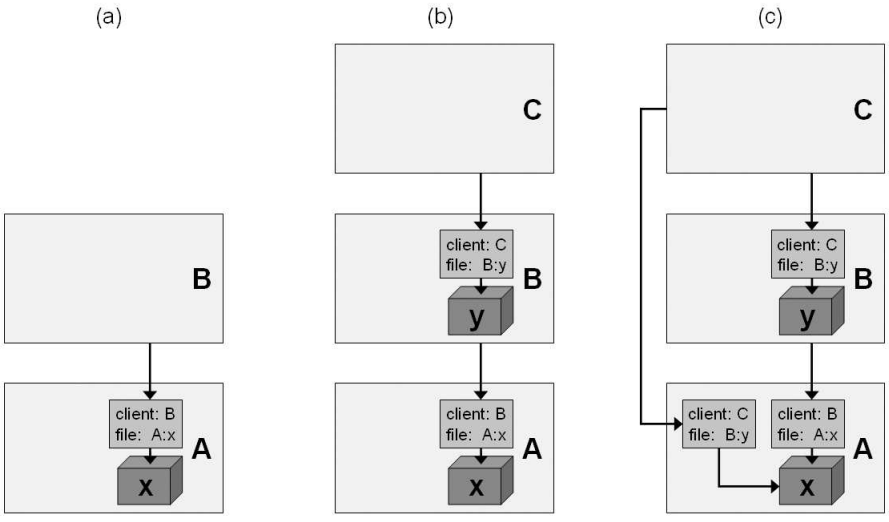

Figure 3.5: Example of using access capabilities

are trusted by the underlying file system and that implement individual strategies to make the files available to a certain set of other threads.

3.4 Data transfer

Data transfer is the most cost intensive part in a multiserver file system. This is why we have to reduce it to a minimum. In Section 3.2 we have already seen how we can prevent unnecessary calls to server levels that implement no functionality. This already saves us a large amount of time. But still, we have to transfer the data between several server threads. Here we have multiple possibilities, and depending on the particular usage scenarios, each of these possibilities has its pros and cons.

First we have to make a distinction into two classes of moving data on a multiserver system. To better understand the differences, we should have a closer look at the way the microkernels on which we build our multiserver systems work.

The basic abstractions exported by microkernels are threads and address spaces. Threads represent the program code we execute, and address spaces provide the protection domain in which we place code and data for the threads. The basic task of microkernels is to manage these abstractions and to provide mechanisms for interaction. Inter Process Communication (IPC) is the mechanism for threads to communicate. Threads use IPC to send information and data to other threads. For address spaces on the other hand, we use mappings to send parts of one address space to another address space. Mappings itself are initiated by threads that send parts of their own address space using special IPC function calls.

3.4.1 IPC

When we send data directly through an IPC function call, the kernel is responsible for copying all the data from the calling thread to the receiving thread. The data is copied across address space borders which might require address space switches and/or TLB cache flushes, depending on the kernel implementation and the processor architecture. It is therefore difficult to give exact theoretical costs for these operations.

Microkernels may also implement multiple ways of direct data transfer through IPC. For instance, the L4 Pistachio [3] microkernel from the University of Karlsruhe implements a set of virtual registers (mapped to physical registers or memory) that are local to each thread and part of the kernel functionality. Sending data through these registers results in high performance since the kernel provides special internal handling for them. This is especially advantagous if we think of recent processor architectures with a larger set of registers residing in the CPU core itself (e.g., the Intel Itanium or the AMD64 architecture). On older architectures, these registers have to be emulated in the core memory, which would lead to additional copy operations and memory access, if we move data from our buffer into these registers.

Another supported method is the usage of string buffers. This is the most intuitive way of data transfer and it is necessary because the amount of memory registers is highly limited. For string buffers, we basically specify the location and the size of a data block that we want to transfer. We also have to specify a receive buffer on the receiving thread in advance. This means that we must know in advance the type and amount of data we will receive. Such a restriction may make it difficult to place data directly into the appropriate local memory, which often results in additional copy operations needed on the receiver side. If we want to receive multiple string buffer

IPCs before we release the first buffer, we will need a memory management or garbage collection subsystem that handles the buffers. We also might waste memory if only a small part of the buffer is used, for example if we use buffers of one megabyte and transfer small packets of one kilobyte.

Sending data through IPC is easy from the view of the client, but it might bring complex buffer handling and copying overhead on the server side. Another drawback is that we have to transfer the complete set of data through each level on the stack.

3.4.2 Mapping

In contrast to the string buffer or virtual register IPCs that we discussed in the previous section, the kernel doesn't touch at all the data when we do mappings. Here we take a window of our own address space and transfer it into the receiver's address space. Mapping utilizes the paging system that is implemented by the MMU (Memory Management Unit) of a CPU. It divides the memory into pages of defined size and alignment. The benefit is that we can move large amounts of data just by changing the address translation within the MMU. This makes it possible to send data to another address space much faster than copying it by accessing each byte in the memory itself.

Using mappings, the server side can itself decide what to do with the data; which part it would touch, and where to copy it. Here we get especially high performance when we just pass the data to the next server level. Also, we don't waste physical memory when we pass multiple levels. With mapping, there is only one physical region that contains the data. Of course, mapping the original memory to other threads with write rights (so that they can modify the data) implies that the original data may be destroyed by one of the underlying server levels.

However, the major drawback in this scenario is the alignment and the size of pages. If we want to transfer a random buffer, we would have to transfer all pages that are used by this buffer. This results in serious security issues when these pages also contain other data, especially when reading from a file. In this case, the client would map the pages to the server with write rights. A malicious server could damage data outside the expected buffer region that contains other data (or code) structures. Another question is the page size that should be supported by the file system. This may vary across different platforms and usage scenarios.

There are a lot of decisions to be made when using mappings for data transfer. And if we want a clean and secure implementation, we need to consider the above facts throughout the client code. Additionally, we might need stub code on the server side that translates buffer information into mapping information.

The mapping approach can result in the highest possible performance, especially on large data blocks and when the kernel and the architecture support several levels of page sizes. But there are several issues that prevent it from being a universal, clean file system implementation.

Mapping works on pages and not on data streams. Therefore, mapping provides a good alternative when we see files exclusively as a sequence of blocks (pages) and not bytes.

Chapter 4

Performance measurements

To get some impression about performance variations on a real system and to verify the functionality, we implemented our research in form of a small stacked file system.

4.1 Experimental setup

We decided to use L4 Pistachio [3] for the experiments. For the interfaces, the IDL4 interface definition language [2] was used to generate optimized stub code for the function calls. The experiment was implemented as a set of three threads. The first thread implements a RAM disk that is exported as a single file. The second thread implements a simple file system based on FAT16. Finally, the third thread is the client that works with files and that does the performance measurements.

On a FAT file system files are divided into clusters. Thus, files may be fragmented and scattered all over the disc. But there may also be files (like static files that were stored in a non-fragmented way after using some system software or after writing to an empty disc) that cover a sequential line of clusters. In this case, the file is present on the physical disc without fragmentation.

In the case of non-fragmented files, the FAT implementation registers appropriate capabilities on the RAM disk server so that the client is able to direct read calls to the RAM disk server itself. This way, the FAT level can be skipped on read calls. For measurement purposes, we additionally implemented the possibility to send the same call to the FAT server that acts as a proxy, simply forwarding the function call to the RAM disk server. This gives us a good comparison between direct calls and stacked calls.

4.2 Four different data transfer designs

The data transfer was implemented in four different variations. For each variation we performed direct reads and indirect reads with different block sizes on both a Pentium 2 system and a AMD Opteron system. The next sections present the results in detail.

4.2.1 IPC using String Buffers to send data

This is the most intuitive way of sending data. We simply let the kernel handle the data transfer. The measurements are given in Figures 4.1 and 4.2. On the Pentium 2, we get

bad performance for very small packet sizes and better performance for larger packet sizes. This is obvious when we think about the fact that we have additional overhead for handling context switches and not only the data transfer itself. The additional handling comes with each call, and it stays constant even if we only handle half the data. When we look at large packet sizes, the ruling part is the time needed for the buffer transfers. Additional handling that doesn't depend on the packet size doesn't have much impact anymore. This is why the throughput levels out at approximately 80MB/s for direct calls.

If we look at the indirect call curve, we can see that we have less throughput. Performance drops by 40-50%. This is also clear because we have additional handling and copying overhead. We can also see a higher performance level for block sizes up to 256KB. In the case for the Pentium 2, at 256KB we can see a little but clear edge on the curve. This appears when the block size exceeds the level 2 cache size which is 256KB on the Pentium 2. In other words, the caches give us some performance boost when copying the same data several times.

The measurements on the Opteron don't look as smooth as on the Pentium 2. This is because modern systems use complex hardware architecture with intermediate buffers and independent behaviour of some components. This makes it difficult to model an exact system that always behaves the same way.

Still, we can clearly see the edge at 512KB (which is the Level 2 cache size here) and the performance boost below 512KB.

4.2.2 IPC using virtual registers to send data

Using virtual registers, we can take advantage of a fast IPC path in the L4 microkernel. However, the memory registers are limited in number, so we only have about 200 bytes for each transfer. With this little memory, it wouldn't make sense to calculate measurements for different buffer sizes. Instead of that, I compared the measurements with those from string buffer IPC at buffer sizes of 200 bytes. The result was a speedup between 12% and 35%, depending on kernel and host platform. But we are still many times slower than the string buffer implementation used with buffer sizes of 16KB and above.

4.2.3 IPC using mappings to send address space

The mapping implementation shows us some interesting results. As we already know, we need some additional handling on the client side to determine the mappings we have to send to the server. We decided to use the fpage abstraction from the L4 microkernel in the interface definition. Fpages support variable page sizes, where each page has the double size of the next smaller pages. Also, the pages have to be aligned to their fpage size. This means each fpage has a size of 2^n and an alignment of 2^n, where $n >= 12$ for the X86 platform.

We can see some "spikes" in the curves that can be interpreted as performance jumps. They appear each time the buffer crosses the former fpage boundaries. When this occurs, we have to send a bigger fpage which results in more mapping overhead. We can also see the caching effects that give us higher throughput on packet sizes around the cache size.

One remarkable thing that we can see on both the Pentium 2 and the Opteron diagrams is that the indirect call only causes a small amount (usually 5-20%) of perfor-

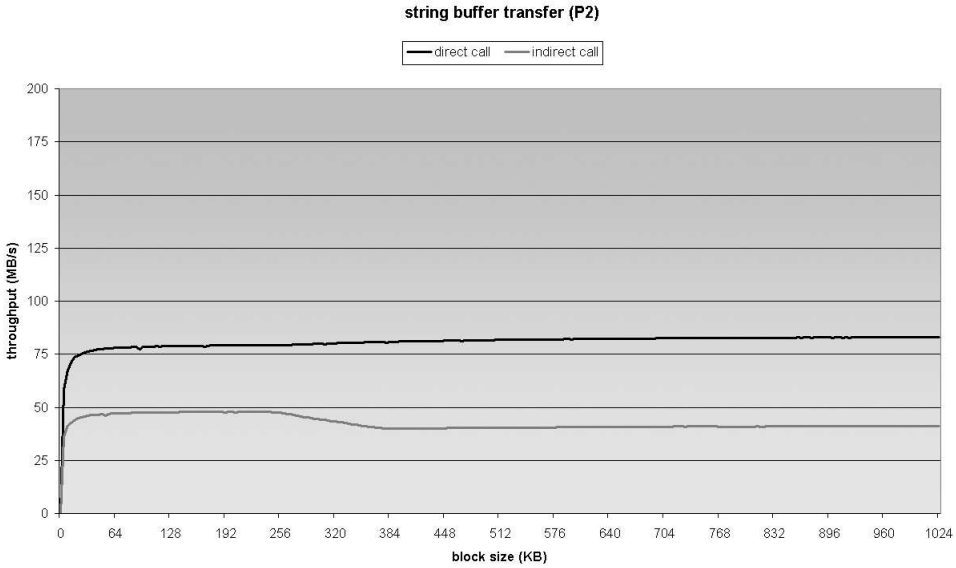

Figure 4.1: String buffer performance on the Intel Pentium 2

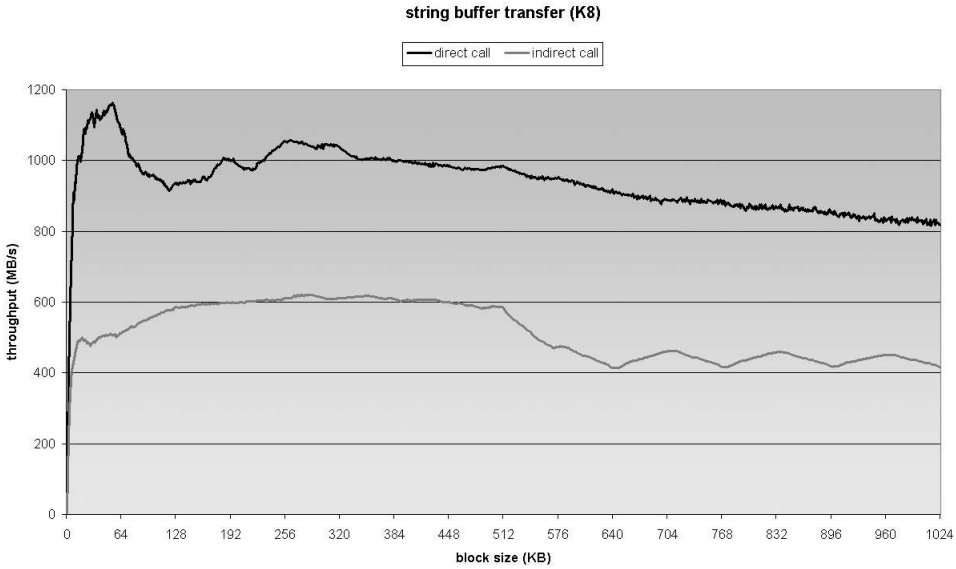

Figure 4.2: String buffer transfer performance on the AMD Opteron

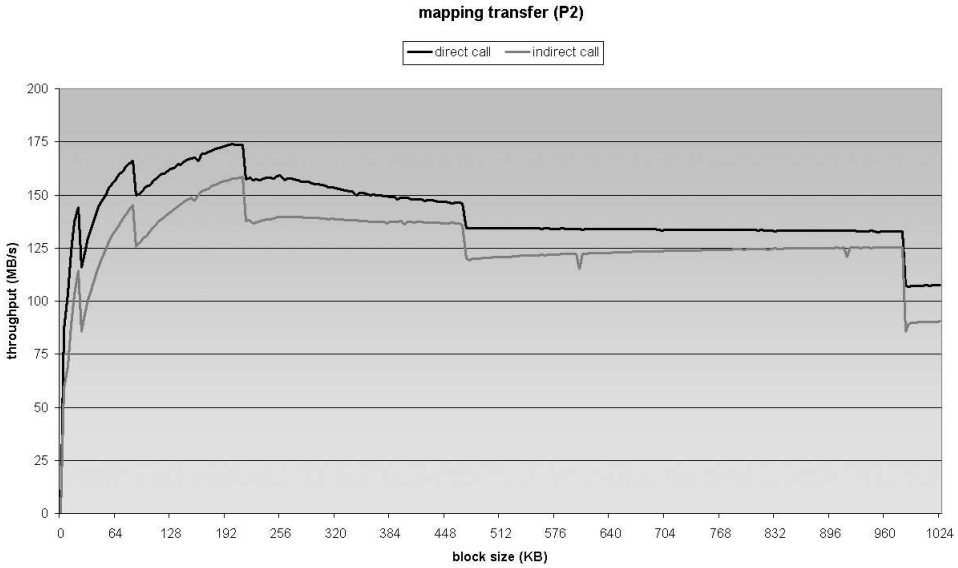

Figure 4.3: Mapping performance on the Intel Pentium 2

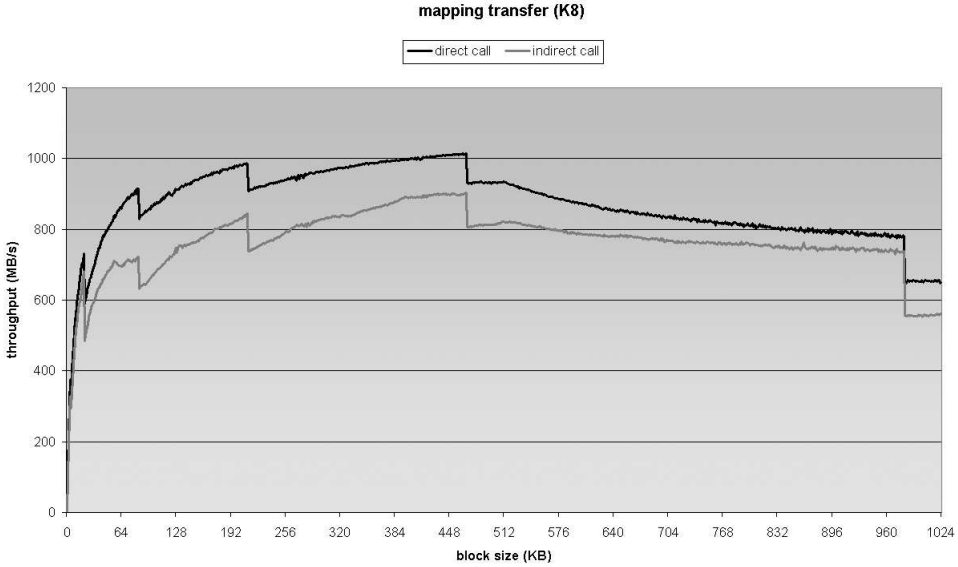

Figure 4.4: Mapping Performance on the AMD Opteron

mance loss. This is because mapping is less cost intensive than transferring the data itself.

4.2.4 Creating a static shared memory window

The final approach to transfer data was to create a shared memory area between client and server. In this implementation, we first do an initialization call to create a shared memory window between client and file server. Data is copied into this memory window on one side and copied out from the window on the other side. This results in one additional copy operation.

We can see again very clearly the impact the cache has on performance. Also notice that on the Pentium, we get even faster copy operations for packet sizes smaller than approximately 350-400KB than with the string buffer IPC implementation. However, for larger packet sizes the string buffer method is slightly faster again. As we can directly specify the target buffer on string buffer method, we would only need one copy operation and some mapping in contrast to two copy operations needed with the memory window method. We thus conclude that the handling and mapping overhead when using adequately small string buffers exceeds the overhead of a second copy operation. For the AMD Opteron, the string buffer IPC always outperforms the memory window approach.

4.3 Analysis

If we compare these implementations (see Figures 4.7 and 4.8), we can say that the mapping method is the best choice when we pass multiple server levels without working much on the data and when the amount of data is sufficiently large. In most situations we might want to use the string buffer method to keep our system simple and universal. Using memory registers for data transfer would only make sense if we really just write small chunks of 200 bytes or less. For some workloads this may usually not be the case.

The memory window approach usually causes additional overhead. The only scenario where it might prove useful is when we load a file, change it, and write it back immediately. In that case, we would create a special memory region that is used for manipulating files. But we have to decide on the memory window size in advance, and we are then bound to that size.

4.4 TLB flushes on the Opteron

The MMU implements paging by translating virtual into physical addresses. Paging information is cached in the Translation Look-aside Buffer (TLB) which is tagged by the virtual addresses. On a context switch, the old virtual addresses become invalid which means we have to flush the TLB and reload it with the mappings for the incoming task.

The AMD Opteron implements a feature called the “TLB Flush Filter”, achieving much the same effect as additional address space identifiers (ASIDs). This makes it possible to keep TLB entries valid even after a context switch. The flush filter is responsible for monitoring the memory regions that contain mapping information cached

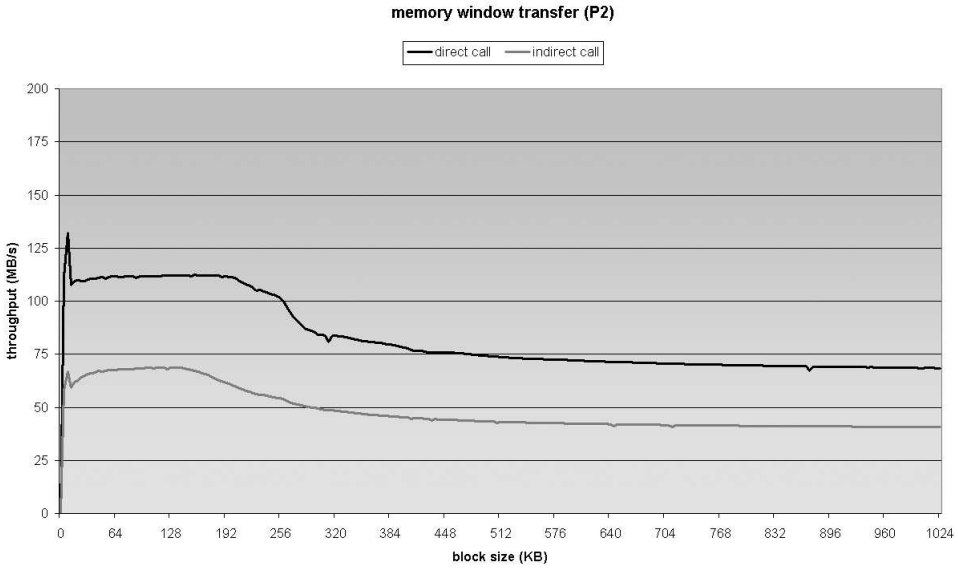

Figure 4.5: Memory window on the Intel Pentium 2

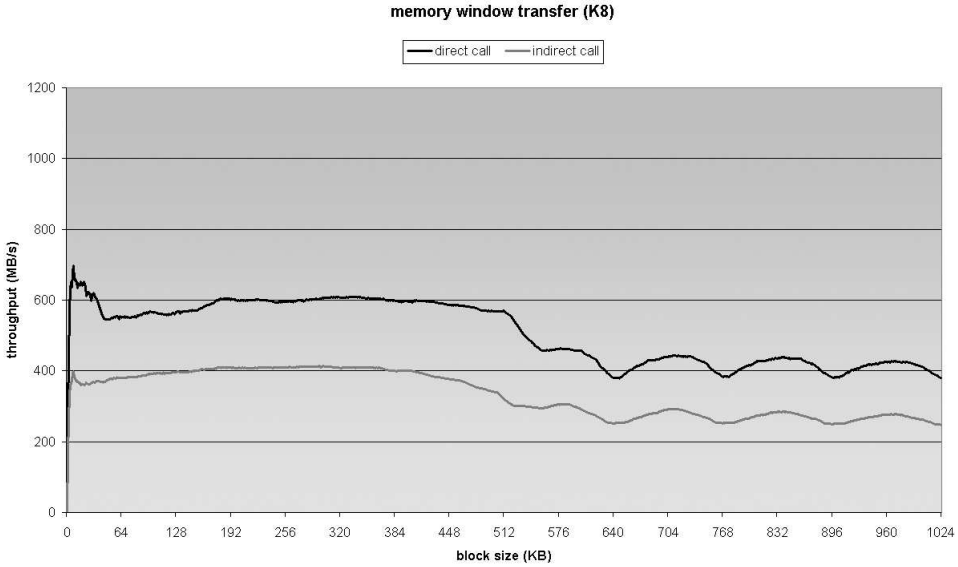

Figure 4.6: Memory window performance on the AMD Opteron

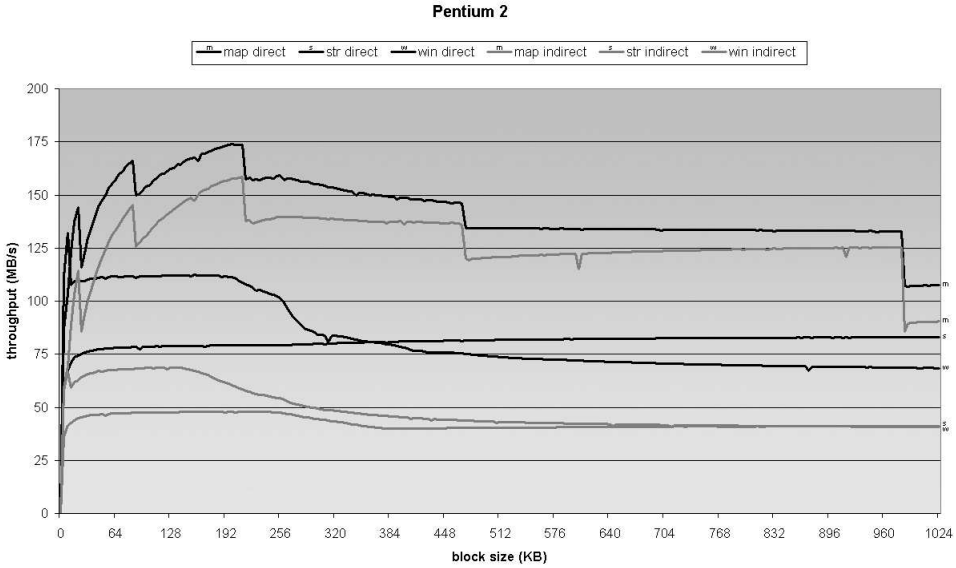

Figure 4.7: Measurements on the Intel Pentium 2

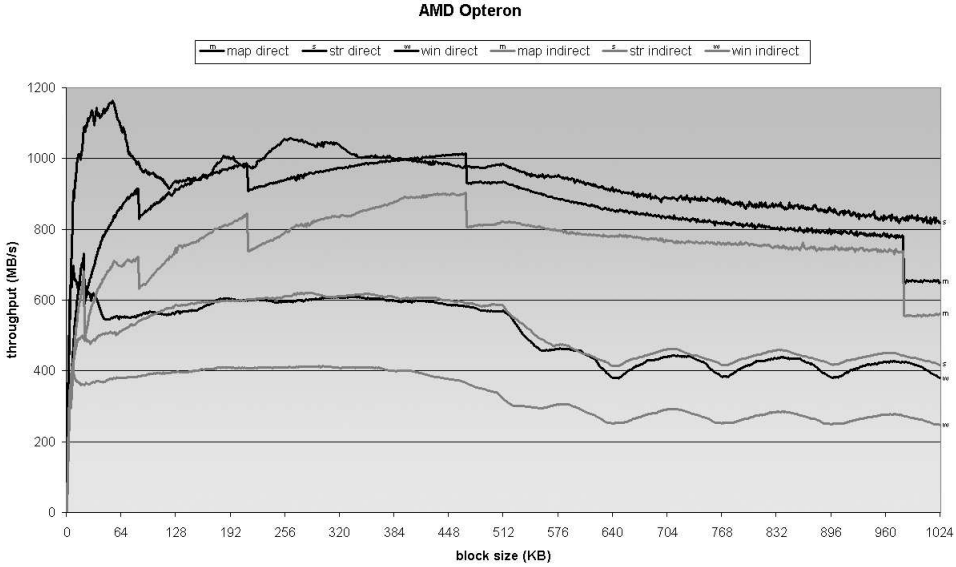

Figure 4.8: Measurements on the AMD Opteron

in the TLB. This way, TLB flushes can be prevented as long as none of these memory regions have been changed.

On each IPC, we face a context switch to the destination thread's address space. Without the TLB Flush Filter each thread would have to re-establish its working set within the TLB. Reloading page table entries is very expensive (in the range of several hundred cycles on modern architectures). With the TLB flush filter, we do not need any reload operations as long as the complete working set fits into the TLB. As such, we can expect higher IPC performance on the Opteron.

During our experiments, the TLB Flush Filter of the Opteron is enabled. But in our setup, there is no visible contrast to runs where the filter is disabled. This has two reasons: First, we are doing measurements on memory-intensive functions, which means that the measurements are ruled by the time we need to transfer the data. The overhead for handling TLB misses is only a fraction of the overhead needed to transfer the data. Second, the code for each of our threads fits into one page. Thus, we are only using a small working set of three pages plus the number of pages needed for the data buffer.

However, on a complex system topology with several server levels and a larger working set (but still fitting into the TLB), the impact of the TLB Flush Filter will be greater on small packet sizes.

Chapter 5

Conclusions and Future Work

Looking at the measurements we can conclude that there is no universal best way to implement a stackable file system on a multiserver environment. We have to take into consideration the scenario we're developing the file system for.

We are able to design a secure, extensible, stackable file system running in a multiserver environment, but we can't give exact predictions on performance. All that we can say is that it will be slower than on a monolithic system. Further experiments with real-life systems must be conducted to determine the exact performance overhead. Another difference to existing file systems is that we need handling on the client side. This could simply be to direct calls to the correct servers listed inside the file handle, but it could also be a calculation of page boundaries.

During the work we came across two completely different ideas on how file systems could be implemented. We conclude this paper with some suggestions of work that can be done in the future.

The first scenario is to use mapping more efficiently and without the drawback of page calculation or accidentally making other private data available to file servers. One promising option would be to use the pager to load files into the local address space. This way, the client has to register files with its pager. We would then need a file server that offers pages of a file, and that is able to move a page aligned window over a file (which would result in the file moving inside a memory window with a static address on the client). The large benefit is that here the file server offers the memory for a file. This way, the same memory can be mapped to multiple clients without copy operations and additional memory usage. This can be incredibly efficient for working with libraries that are used by several threads. The basics to this scenario are explained in [1].

Another scenario would be a file system that is based on asynchronous communication. Function calls don't include data packets but only short signal like information for the task to perform. If a function call reaches a level that needs to work with the data, exactly this server can request the data packet directly from the thread owning it. This way, the whole file system can do fast communication and decide how to move the data most efficiently. For this scenario, the usage of additional helper threads in the same address space as the client might be useful.

Bibliography

[1] Mohit Aron, Luke Deller, Kevin Elphinstone, Trent Jaeger, Jochen Liedtke, and Yoonho Park. The SawMill framework for virtual memory diversity. In *Proceedings of the 6th Asia-Pacific Computer Systems Architecture Conference*, Bond University, Gold Coast, QLD, Australia, January 29–February 2 2001.

[2] Andreas Haeberlen. *IDL4 User Manual*. System Architecture Group, Universität Karlsruhe, April 2003. Available from http://l4ka.org/.

[3] The L4Ka Team. *L4 Kernel Reference Manual (Version X.2)*. Universität Karlsruhe, 2004. Available from http://l4ka.org/.

[4] D.S.H. Rosenthal. Evolving the vnode interface. In *Proceedings of the Summer 1990 USENIX Technical Conference*, pages 107–118, SunSoft, June 1990.

[5] G.C. Skinner and T.K. Wong. Stacking vnodes: A progress report. In *Proceedings of the Summer 1993 USENIX Technical Conference*, pages 161–174, SunSoft, June 1993.

[6] Uresh Vahalia. Section 11.11 - stackable file system layers. In *Unix Internals*, pages 364–267, Prentice-Hall, Inc., ISBN 0-13-101908-2, 1996.

[7] Uresh Vahalia. Section 8.6 - the vnode/vfs architecture. In *Unix Internals*, pages 234–239, Prentice-Hall, Inc., ISBN 0-13-101908-2, 1996.