The SawMill Multiserver Approach

Alain Gefflaut Trent Jaeger Yoonho Park
Jochen Liedtke * Kevin J. Elphinstone † Volkmar Uhlig †
Jonathon E. Tidswell † Luke Deller ‡ Lars Reuther ‡
IBM T.J Watson Research Center
Hawthorne, NY 10532
Email: sawmill@us.ibm.com

1 Introduction

Multiserver systems, operating systems composed from a set of hardware-protected servers, initially generated significant interest in the early 1990's. If a monolithic operating system could be decomposed into a set of servers with well-defined interfaces and well-understood protection mechanisms, then the robustness and configurability of operating systems could be improved significantly. However, initial multiserver systems [4, 14] were hampered by poor performance and software engineering complexity. The Mach microkernel [10] base suffered from a number of performance problems (e.g., IPC), and a number of difficult problems must be solved to enable the construction of a system from orthogonal servers (e.g., unified buffer management, coherent security, flexible server interface design, etc.).

In the meantime, a number of important research results have been generated that lead us to believe that a re-evaluation of multiserver system architectures is warranted. First, microkernel technology has vastly improved since Mach. L4 [13] and Exokernel [6] are two recent microkernels upon which efficient servers have been constructed (i.e., L4Linux for L4 [12] and ExOS for Exokernel [9]). In these systems, the servers are independent OSes, but we are encouraged that the kernel and server overheads, in particular context switches overheads, are minimized. Second, we have seen marked improvements in memory management approaches that enable zero-copy protocols (e.g., *fbufs* [5] and *emulated copy* [3]). Other advances include, improved kernel modularity [7], component model services [8], multiserver security protocols, etc. Note that we are not the only researchers who believe it is time to re-examine multiservers, as a multiserver system is also being constructed on the Pebble kernel [11].

In addition, there is a greater need for multiserver architectures now. Consider the emergence of a variety of specialized, embedded systems. Traditionally, each embedded system includes a specialized operating system. Given the expected proliferation of such systems, the number of operating systems that must be built will increase significantly. Tools for configuring operating systems from existing servers will become increasingly more valuable, and adequate protection among servers will be necessary to guard valuable information that may be stored on such systems (e.g., private keys). This is exactly the motivation for multiserver systems.

In this paper, we define the SawMill multiserver approach. This approach consists of: (1) an architecture upon which efficient and robust multiserver systems can be constructed and (2) a set of protocol design guidelines for solving key multiserver problems. First, the SawMill architecture consists of a set of user-level servers executing on the L4 microkernel and a set of services that enable these servers to obtain and manage resources locally. Second, the SawMill protocol design guidelines enable system designers to minimize the communication overheads introduced by protection boundaries between servers. We demonstrate the SawMill approach for two server systems derived from the Linux code base: (1) an Ext2 file system and (2) an IP network system.

The remainder of the paper is structured as follows. In Section 2, we define the problems that must be solved in converting a monolithic operating system into a multiserver operating system. In Sections 3 and 4, we define the SawMill architecture and the protocol design approach, respectively. In Section 5, we demonstrate some of these guidelines in the file system and network system implementations. In Section 6, we examine the performance of the current SawMill Linux system.

2 Multiserver Issues

An effective multiserver system must: (1) protect its servers from errors or malice in other servers; (2) implement coherent system semantics; and (3) incur min-

*Faculty of Informatics, Universität Karlsruhe, Germany
†School of Computer Science and Engineering, University of New South Wales, Australia
‡Department of Computer Science, Technische Universität Dresden, Germany

imal performance overhead. We define these requirements more precisely in this section.

2.1 Protection

Multiserver systems must preserve the integrity of each server's execution (i.e., protect servers from one another) and protect the integrity and secrecy of the data processed by each server (i.e., ensure the protection of user data). Specifically, we list the following protection requirements:

- Protect the execution integrity of each server:
  - Prevent modification of another server's code
  - Prevent modification of another server's 'control data' (i.e., data that is interpreted for execution, such as the stack)
  - All code and control data must only be obtained from a trusted source
- Protect the secrecy and integrity of user data:
  - Prevent leakage of data to unauthorized subjects
  - Prevent modification of data by unauthorized subjects
  - Protect data from accidental modification by other servers

The first set of requirements prevents malicious or buggy servers from crashing other servers (i.e., protects server integrity). A malicious or buggy server can only cause another server to crash if it modifies the data interpreted by the server in its execution: code, stack, and 'control' data (i.e., memory references and metadata). We distinguish between control data, the data interpreted by the server during execution, and user data, the data being being transferred from the users to the devices or vice versa via the operating system. The server must obtain its code, stack, and control data from a trusted source and limit modification of this data to verifiably correct protocols.

The second set of requirements is designed to protect the user data being processed by the server. In general, there are two types of requirements embodied here. The first two requirements embody traditional access control, including control over overt and covert channels. The third requirement implies that user data transferred among servers must be protected effectively from accidental server modification. That is, as user data is being transferred from one server to another, there must be some protocol to prevent the first server from modifying this user data.

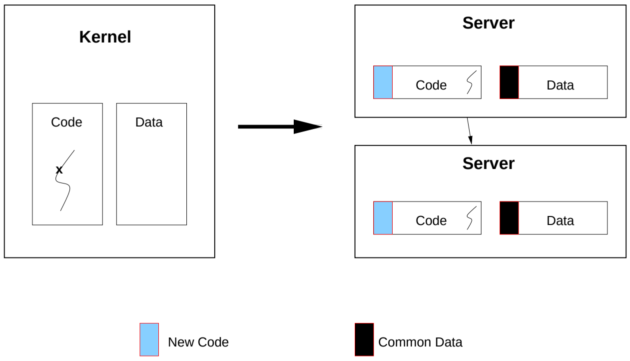

Figure 1: Decomposing a monolithic operating system into a multiserver operating system involves: (1) additional communication between between servers; (2) buffer management for common data in multiple tasks; and (3) the addition of new code to compose a system from independent servers.

2.2 Semantics

In general, the problem of decomposing a monolithic operating system into a multiserver operating system is shown in Figure 1. In a multiserver system, the monolithic system is implemented by a set of user-level system tasks upon a microkernel (not shown). The combination of these tasks define the semantics seen by the user tasks. We define how a multiserver must preserve these semantics below.

- Each system call in the monolithic system must be supported by one or more servers in the multiserver system [1].
- Each system call must be processed as restricted by its atomicity requirements.
- Any server must be able to obtain and enforce system policies (e.g., for security, resource management, etc.).

First, a semantically-equivalent multiserver system must provide the same functionality as the original system. Thus, the decomposed system must be able to respond to all the same system calls as the monolithic system. Second, the atomicity requirements of system calls in the monolithic system must be enforced in the multiserver system. If system call data is distributed among multiple servers, concurrency control mechanisms must account for all those servers to prevent race conditions. Third, a coherent system composed of a set of servers requires that any server be able to obtain and enforce system-wide policies, such as access control and resource management.

[1] In an embedded system, only a subset of the system calls may be necessary, so only these must be supported.

2.3 Performance

The goal of the SawMill multiserver design is to achieve the protection and semantic requirements with no significant performance degradation. Maintaining performance in a multiserver system is non-trivial. Consider Figure 1, in which a single-threaded, monolithic operating system is broken into two single-threaded system tasks. This decomposition creates the potential that local operations in the monolithic system are distributed operations in the multiserver system. Thus, some local operations may now be implemented as IPCs. In order to maintain monolithic system performance, we must reduce both the frequency of IPC operations and the cost of each IPC.

- **IPC Frequency**:
  - IPCs (i.e., context switches) replace procedure calls
  - Additional IPCs may be necessary to maintain the consistency of replicated data
  - Additional IPCs may be necessary to synchronize access to shared data
  - Additional IPCs may be necessary to negotiate resource allocation
  - Cache and TLB locality are reduced by additional context switches
- **IPC Overhead**:
  - Parameters must be marshalled and unmarshalled between servers
  - Parameters need to be transferred between the servers

First, each procedure call that crosses a server boundary in now convert to an IPC, which consists of context switching, marshalling, unmarshalling, copying, and mapping overheads. Clearly, the number of IPCs should be minimized, and the overhead of each IPC and related functionality must be minimized. Also, depending on the protocol chosen, additional IPCs may be necessary to maintain data consistency, synchronize access, and obtain resources. Second, data may need to be transferred across these protection boundaries. While control data transfer can be limited to effective partitioning and caching, significant amounts of user data will be transferred through the servers. This data may be copied, mapped, or shared depending on the protection and protocol semantics.

3 SawMill Architecture

The SawMill architecture for multiserver operating systems is shown in Figure 2. The SawMill architecture consists of three types of components:

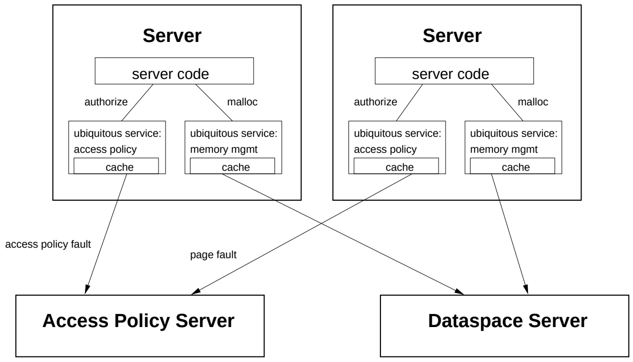

Figure 2: The SawMill approach enables the decomposition of monolithic operating systems into individual *system servers* where *ubiquitous services* manage server data obtained from *resource servers* locally.

- **System servers**: These tasks provide the main functionality of the operating system (e.g., network systems, file systems, etc.)
- **Ubiquitous Services**: These components provide general functionality that may be of use to any system server (e.g., synchronization, access control, naming, communication, etc.).
- **Resource Servers**: These servers manage the core resources for distribution to the system servers (e.g., memory, IRQs, security policy, etc.).

System server code is augmented by libraries, called ubiquitous services, that provide multiserver-aware management of system data. In general, a ubiquitous service obtains resources from the appropriate resource servers which control the distribution of resources among servers. Ubiquitous services manage these resources locally (e.g., caches) to limit communication overhead.

The basic protocol is as follows for memory management [2]. First, the server requests that its memory service library obtain access to its dataspaces (i.e., memory objects). The memory service library then opens the appropriate dataspaces on the appropriate dataspace managers (i.e., resource server). When the server attempts to access this memory, the page fault is directed to memory service library which requests that appropriate dataspace manager service the page fault. Notice that dataspace managers may be stacked, such that a dataspace manager may need to obtain the page from its dataspace manager, and so on. Similar resource faulting approaches are also used for obtaining access control data, names, tasks, and mount points. Other ubiquitous libraries provide multiserver-specific functionality, such as Flick's cross-domain procedure call stubs [8] and synchronization.

4 SawMill Protocols

In order to build an efficient multiserver, system protocols must designed to minimize IPC frequency and overhead (see Section 2.3). We identify these principles for designing efficient multiserver protocols:

- Make system calls directly to the processing server
- Partition server-specific control data
- Share data as widely as possible

First, to reduce the number of IPCs that are necessary to implement a performance-critical system call, clients should communicate directly to the servers that process the system call, where possible. Second, server control data should be partitioned where possible. In cases where the data is shared (i.e., cached read-only in multiple servers), few update messages should be necessary (i.e., few writes or weak consistency). Third, data sharing should be utilized as widely as protection requirements can allow. In particular, the mapping of user data can be avoided in system servers, because the same buffers can be reused for the same user. Also, it may be desirable to share control data read/write between drivers and servers, as long as each can prevent themselves from crashing due to errors or malice.

5 Some Implementation Details

We now examine some interesting examples of applying these concepts to the implementation of the file system and network system. This implementation is based on Linux 2.2.1 code and preserves Linux semantics.

5.1 Protocol Options

First, each system protocol should be designed to require as few IPCs as possible. Consider the protocol options for a multiserver file system in Figure 3. The first protocol simply replaces the inter-server procedure calls with IPCs. In this case, the VFS is called on open to obtain a file handle. On subsequent read/write calls, the client calls the VFS which authorizes the handle and forwards authorized requests onto the PFS. In the second protocol, the VFS obtains a handle from the PFS that the client can use on the PFS directly. Therefore, only the open call is sent to the VFS, but there are potentially several IPCs between the VFS and PFS for name resolution. The third protocol option considers the VFS only as a store for mount point and access control information. This is typical of the resource faulting protocols described in the previous section. In this case, a client emulation library obtains the PFS-mount point mapping, so it only needs to call the VFS for each new mount point it uses [2]. Also, the PFS caches access control data, so it only needs to call the VFS when a request does not hit in the access control data cache.

In the normal case, the third option will result in fewer IPCs. However, due to the complexity in modifications necessary to implement it, the second option is being used currently. Overhead for read/write is the same for both the second and third case, however (see Section 6).

For the network system, an option analogous to the third option is implemented. In this option, a network manager performs the role of the VFS in naming stacks and devices and providing access control information. The network stack corresponds to a PFS.

5.2 Data Partitioning

Fortunately, a significant amount of control data can be partitioned between the servers in both the file system and the network system. However, user data must flow (not necessary be copied) through the servers to reach its destination, so it cannot be partitioned. In the file system, the main control data are the superblocks and the inodes. Superblocks change very infrequently, so caching is not a problem. Inodes are provided by the driver and used read/write by both the PFS and the VFS. However, the inode data used and updated by the VFS and PFS are orthogonal. So, the master copy is stored by the PFS, and the VFS caches its copy and sends updates to the PFS. The PFS does not send updates to the VFS. If the third protocol option were used, little or no sharing of inode data would be necessary between the VFS and PFS.

A problem is that the buffer cache is shared among the PFSs in Linux. Thus, one PFS could disrupt the inode data of another PFS. Clearly, in order to prevent this, a trusted entity (either a buffer cache server or a driver) must partition the buffer cache among PFSs. Since the buffer cache usage of the different PFSs may vary over time, this server will need significant resource management abilities. Currently, our decomposed PFS and driver share their own separate buffer cache, but this is still an open problem.

The network system control data includes device structures, buffer lists, and sk buffs. Since the device structure is used read-only by the network stacks, the drivers maintain the master copy. The network stacks get a copy when they open the device, and the driver notifies them of any changes. There are function pointers in the structure, so these pointers are localized at load time. The buffer lists are partitioned between

[2] An exception is on the processing of symbolic links between file systems, but these can be handled by having each PFS provide the client with the PFS to which its part of the path resolves.

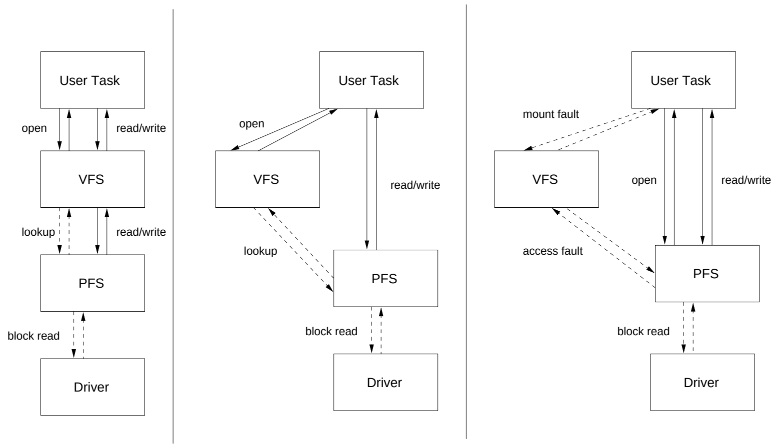

Figure 3: The communication requirements for three file system protocol options. Dashed IPCs are optional. For example, a PFS lookup may not be necessary because the appropriate inodes may be present in the directory cache.

the driver and the stack. For example, when a stack wishes to send a packet, it sends the driver a reference to an sk buff which the driver can queue on its own list. There is no sharing of buffer lists between the stack and driver. sk buffs and packet data are shared as discussed below.

5.3 I/O Management

There are two important issues in I/O management: (1) the protection boundaries that the I/O data crosses (usersystem or server-server within the system) and (2) the type of the data being transferred (user or control data). First, in a traditional UNIX system such as Linux, data is typically copied between the user and the system. In a multiserver system, we have this type of data transfer as well as data transfer between servers in the system. System servers originally shared this data in the monolithic system, so sharing semantics are more natural. Second, we must consider the protection requirements of the type of data that is being transferred. Protection requirements permit user data to be shared among multiple servers, as long as sufficient copy semantics are enforced between the user and system (some form of copy-on-write), but servers must protect themselves from the generation of illegal control data.

First, user data transferred between the user and system is transferred with copy semantics. However, user data buffers are shared among the servers. In effect, the system servers are originators in the fbufs sense [5] of the data, so they maintain write access until the data is transferred to the user or to an untrusted server. Then, a decision must be made whether to downgrade the servers' access to the data to read-only (e.g., using emulated copy [3] or fbuf semantics) and apply copy-on-write for the user, permit the servers to maintain read-write access (e.g., volatile buffers) with copy-on-write for the user, or copy on transfer. Unfortunately, neither emulated copy or fbuf semantics work for the Linux file system. Buffer cache buffers may not be aligned as necessary for emulated copy and Linux copy semantics does not enable the use of fbufs. At present, data is copied between user tasks and the SawMill Linux system servers, although further investigation is ongoing.

For control data, each server must trust the originator of the control data it uses to specify the necessary data, but it must ensure that the control data that it uses does not cause it to crash. For example, the network stack specifies sk buffs for the network driver which the network driver must assume refer to the data that it is to send (if it is within the shared user data region). However, if the sk buff is erroneous (i.e., refers to an illegal memory location), the network driver must catch this. In this case, we prefer an optimistic approach where the network stack and driver share sk buff data and the driver uses the sk buff directly. Any error in the sk buff either results in the wrong data being sent (which the network stack could arrange regardless) or results in an illegal page fault. Since the driver's page fault handler is a local thread, it can recover the driver thread to a consistent state on such an illegal page fault.

6 Performance Summary

In an initial experiment, we compare file system performance using the Iozone reread benchmark [1]. All the data is from a 500 MHz, Pentium III with 64MB RAM. Iozone was run on three systems: (1) Linux 2.2.1; (2) L4Linux derived from Linux 2.2.1; and (3) SawMill Linux derived from Linux 2.2.1. For our analysis we have focused on reread throughput, in which Iozone reads a file twice and measures the through-

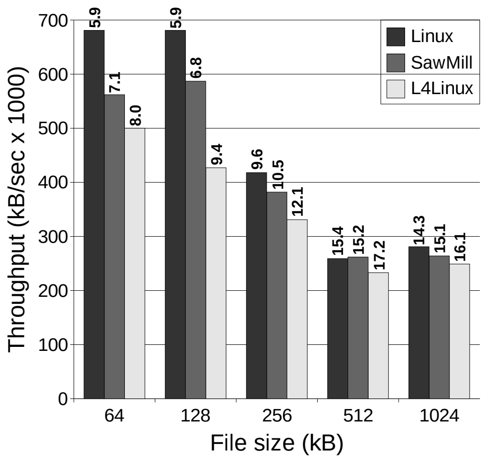

Figure 4: Iozone performance comparison for rereads using 4KB record size on Linux 2.2.1, SawMill Linux (derived from Linux 2.2.1), and L4Linux (derived from Linux 2.2.1)

put of the second read. For our purpose, reread is a harder test than read because the reread operation is faster and thus the componentization overhead is more critical.

Figure 4 shows reread performance for various file sizes using a 4KB record size. For the smaller file sizes (64 and 128 KB), rereads copy data from the L2 cache (file buffer) to the L1 cache (application buffer). Because the copy time is lowest in this case, the performance overhead of IPC is most significant. Beginning with file sizes of 256KB, throughput decreases because the copies are from memory to the L1 cache.

For each 4KB record in the reread, we measured approximately 3000 cycles for bare Linux, 4200 cycles for L4Linux, and 3500 cycles for SawMill Linux. Although the multiserver SawMill clearly performs better than the monolithic L4Linux on the microkernel, SawMill is still 500 cycles slower than bare Linux. We expected an overhead of about 200 cycles: the kernel overhead of the first IPC and the full cost of the second (i.e., including the kernel sysenter/sysexit call).

To our surprise, an important part of the problem seems to be the generated stub code. We expected cheap stubs since the Flick IDL compiler [8] generates C code such that all its operations are inline generated in the user program. However, the code effectively generated by gcc uses about 150 machine instructions for any client stub, mostly useless copies to and from local variables. For the current measurements, we did not hand code stubs but only improved the Flick/gcc coding through simple specialization. However, it is clear that an improved code generation facility has to be developed that generates near-optimal code which we found to be about 30 instructions.

7 Conclusions

In this paper, we describe the SawMill approach to constructing multiserver operating systems. This approach consists of a set of services and design guidelines that enable the development of a multiserver from an existing code base. We demonstrate this approach on the Linux operating system by building multiserver file and network systems. The main complexity lies in the efficient and secure management of data over the disjoint servers, although we have a number of potentially promising approaches. Also, initial performance results indicate that the multiserver overhead is small on the file system and can be further optimized to reduce the impact of multiserver protection barriers. Soon, we will have the network system performance measurements.

References

[1] The IOZone filesystem benchmark, April 2000. Available from http://www.iozone.org.

[2] M. Aron, J. Liedtke, Y. Park, K. Elphinstone, and T. Jaeger. A framework for VM diversity, March 1999. Available from http://www.research.ibm.com/sawmill.

[3] J. Brustoloni and P. Steenkiste. Effects of buffering semantics on I/O performance. In *OSDI'96*. USENIX, October 1996.

[4] T. Bushnell and G. Matzigkeit. The GNU Hurd reference manual, August 1998. Available from http://www.gnu.ai.mit.edu/software/hurd/doc/hurd-toc.html.

[5] P. Druschel and L. Peterson. Fbufs: A high-performacne cross-domain transfer facility. In *SOSP'93*. ACM, December 1993.

[6] D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *SOSP'95*. ACM, December 1995.

[7] B. Ford *et al.* The Flux OSKit: A substrate for OS and language research. In *SOSP'97*. ACM, October 1997.

[8] E. Eide *et al.* Flick : A flexible, optimizing IDL compiler. In *PLDI'97*. ACM, June 1997.

[9] M. Kaashoek *et al.* Application performance and flexibility on Exokernel systems. In *SOSP'97*. ACM, October 1997.

[10] R. V. Baron *et al.* Mach kernel interface manual, 1990. Unpublished manuscript from the School of Computer Science, Carnegie Mellon University.

[11] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble component-based operating system. In *USENIX Technical Conference*. USENIX, June 1999.

[12] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *SOSP'97*. ACM, October 1997.

[13] J. Liedtke. On μ-kernel construction. In *SOSP'95*. ACM, December 1995.

[14] J. M. Stevenson and D. P. Julin. Mach-US: UNIX on generic OS object servers. In *USENIX Technical Conference*. USENIX, Winter 1995.