Microkernels Meet Recursive Virtual Machines

Bryan Ford, Mike Hibler, Jay Lepreau,
Patrick Tullmann, Godmar Back, and Stephen Clawson
University of Utah

Microkernels Meet Recursive Virtual Machines

Bryan Ford Mike Hibler Jay Lepreau Patrick Tullmann
Godmar Back Stephen Clawson

Department of Computer Science, University of Utah
Salt Lake City, UT 84112

flux@cs.utah.edu http://www.cs.utah.edu/projects/flux/

Abstract

This paper describes a novel approach to providing modular and extensible operating system functionality and encapsulated environments based on a synthesis of microkernel and virtual machine concepts. We have developed a software-based *virtualizable architecture* called Fluke that allows recursive virtual machines (virtual machines running on other virtual machines) to be implemented efficiently by a microkernel running on generic hardware. A complete virtual machine interface is provided at each level; efficiency derives from needing to implement only *new* functionality at each level. This infrastructure allows common OS functionality, such as process management, demand paging, fault tolerance, and debugging support, to be provided by cleanly modularized, independent, stackable virtual machine monitors, implemented as user processes. It can also provide uncommon or unique OS features, including the above features specialized for particular applications' needs, virtual machines transparently distributed cross-node, or security monitors that allow arbitrary untrusted binaries to be executed safely. Our prototype implementation of this model indicates that it is practical to modularize operating systems this way. Some types of virtual machine layers impose almost no overhead at all, while others impose some overhead (typically 0–35%), but only on certain classes of applications.

1 Introduction

Increasing operating system modularity and extensibility without excessively hurting performance is a topic of much ongoing research [5, 9, 18, 36, 40]. Microkernels [4, 24] attempt to decompose operating systems "horizontally" by moving traditional kernel functionality into servers running in user mode. Recursive virtual machines [23], on the other hand, allow operating systems to be decomposed "vertically" by implementing OS functionality in stackable *virtual machine monitors*, each of which exports a virtual machine interface compatible with the machine interface on which it runs. Traditionally, virtual machines have been implemented on and export existing hardware architectures so they can support "naive" operating systems (see Figure 1). For example, the most well-known virtual machine system, VM/370 [28, 29], provides virtual memory and security between multiple concurrent virtual machines, all exporting the IBM S/370 hardware architecture. Furthermore, special *virtualizable hardware architectures* [22, 35] have been proposed, whose design goal is to allow virtual machines to be stacked much more efficiently.

This paper presents a new approach to OS extensibility which combines both microkernel and virtual machine concepts in one system. We have designed a "virtualizable architecture" that does *not* attempt to emulate an actual hardware architecture closely, but is instead designed along the lines of a traditional process model and is intended to be implemented in software by a microkernel. The microkernel runs on the "raw" hardware platform and exports our software-based virtualizable architecture (see Figure 2), which we will refer to as a *virtualizable process* or *nested process architecture* to avoid confusion with traditional hardware-based architectures. The virtual machine monitors designed to run on this software architecture, which we call *nesters*, can efficiently create additional recursive virtual machines or *nested processes* in which arbitrary applications or other nesters can run.

Although the Fluke architecture does not closely follow a traditional virtual machine architecture, it is designed to preserve certain highly useful properties of recursive virtual machines. These properties are required to different degrees by different nesters that take advantage of the model. For example, demand paging and checkpointing nesters require access to and control over the program state contained in their children, grandchildren, and so on, whereas process management and security monitoring nesters primarily rely on being able to monitor and control IPC-based communication across the *boundary* sur-

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army, under contract number DABT63–94–C–0058. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies of the U.S. Government.

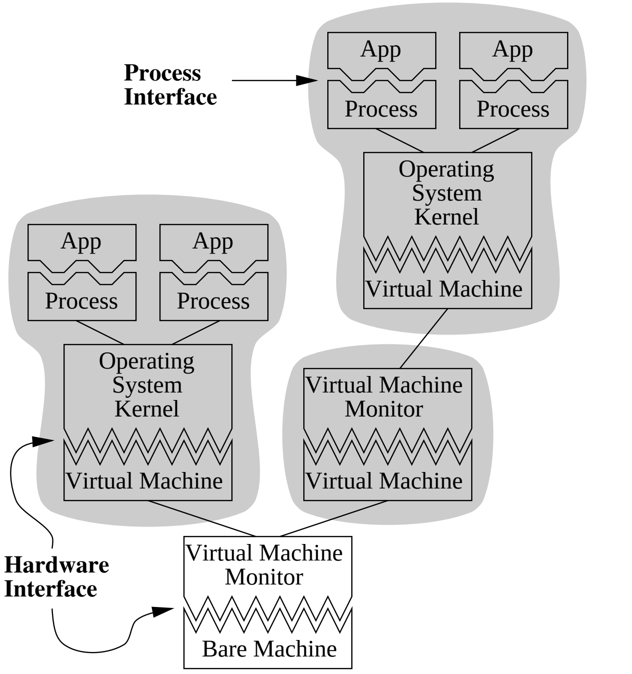

Figure 1: Traditional virtual machines based on hardware architectures. Each shaded area is a separate virtual machine, and each virtual machine exports the same architecture as the base machine's architecture.

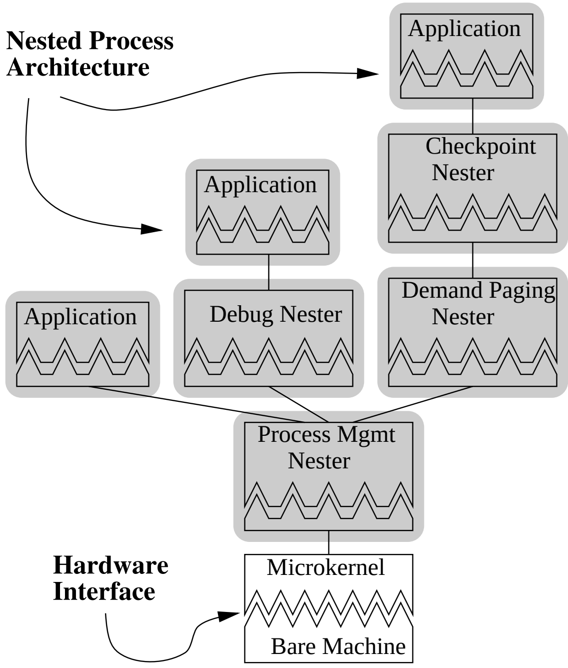

Figure 2: Virtual machines based on an extended architecture implemented by a microkernel. The interface between the microkernel and the bare machine is a traditional hardware-based machine architecture, but the common interface between all the other layers in the system is a software-based nested process architecture. Each shaded area is a separate process.

rounding the nested environment.

Our microkernel's API provides these properties efficiently in several ways. Address spaces are composed from other address spaces using hierarchical memory remapping primitives. For CPU resources, the kernel provides primitives that support hierarchical scheduling. To allow IPC-based communication to short-circuit the hierarchy safely, the kernel provides a global capability model that supports *selective* interposition on communication channels. On top of the microkernel API, well-defined IPC interfaces provide I/O and resource management functionality at a higher level than in traditional virtual machines. These higher-level interfaces are more suited to the needs of modern applications: e.g., they provide file handles instead of device I/O registers.

This nested process architecture can be used to apply existing algorithms and techniques in more flexible ways. Some examples we demonstrate in this paper include the following:

Decomposing the kernel: Some features of traditional operating systems are usually so tightly integrated into the kernel that it is difficult to eliminate them in situations in which they are not needed. A striking example is demand paging. Although it is often possible to disable it in particular situations on particular regions (e.g., using POSIX's `mlock()`), all of the paging support is still in the kernel, occupying memory and increasing system overhead. Even systems that support "external pagers," such as Mach, contain considerable paging-related code in the kernel and most do not allow control over physical memory management, just backing store. Similarly, multiuser security mechanisms are not always needed, since most personal computers are dedicated to the use of a single person, and even process management and job control features may not be needed in single-application systems such as the proverbial "Internet appliance." Our system demonstrates decomposed paging and POSIX process management by implementing these traditional kernel functions as optional nesters which can be used only when needed, and only for the parts of a system for which they are desired.

Increasing the scope of existing mechanisms: There are algorithms and software packages available for common operating systems to provide features such as distributed shared memory (DSM) [10, 32], checkpointing [11], and security against untrusted applications [52]. However, these systems only cleanly support applications running in a single logical protection domain. In a nested process model, any process can create further nested subprocesses which are completely encapsulated within the parent. This design allows DSM, checkpointing, security, and other mechanisms to be applied just as easily to multi-process applications or even complete operating environments. Our system demonstrates this flexibility by providing a checkpointer, implemented as a nester, which can be transparently applied to arbitrary domains such as a single application, a multi-process user environment containing a process manager and multiple applications, or even the entire system.

Composing OS features: The mechanisms mentioned above are generally difficult or impossible to combine flexibly. One might be able to run an application and checkpoint it, or to run an untrusted application in a secure environment, but existing software mechanisms are insufficient to run a *checkpointed, untrusted* application without implementing a new, specialized program designed to provide both functions. A nested process architecture allows one to combine such features by layering the mechanisms, since the interface between each layer is the same. In Fluke, for example, a Unix-like environment can be built by running a process manager within a virtual memory manager, so that the process manager and all of the processes it controls are paged. Alternatively, the virtual memory manager can be run within the process manager to provide virtual memory to an individual process.

We used micro benchmarks to measure the system's performance in a variety of configurations. These measurements indicate a slowdown of about 0–35% per virtual machine layer, in contrast to conventional recursive virtual machines whose slowdown is 20%–100% [7]. Some nesters, such as the process manager, do not need to interpose on performance-critical interfaces such as memory allocation or file I/O, and hence take better advantage of the short-circuit communication facilities provided by the microkernel architecture. These nesters cause almost no slowdown at all. Other nesters, such as the memory manager and the checkpointer, must interfere more to perform their function, and therefore cause some slowdown. However, even this slowdown is fairly reasonable. Our results indicate that, at least for the applications we have tested, this combined virtual machine/microkernel model indeed provides a practical method of increasing operating system modularity, flexibility, and power.

The rest of this paper is organized as follows: In Section 2 we compare our architecture to related work. We describe the key principles upon which our work is based in Section 3, and our software-based virtualizable architecture derived from these principles in Section 4. Section 5 describes the implementation of the example applications and nesters we designed to take advantage of the nested process model. Section 6 describes the experiments and results using the example process nesters. Finally, we conclude with a short reflective summary.

2 Related Work

In this section, we first summarize virtual machine concepts and how our system incorporates them; second, we show how our design relates to conventional process models, and finally, we contrast our system with other microkernel-based systems. We describe related work concerning the details of our design later, in the appropriate sections.

2.1 Traditional Virtual Machines

A *virtual machine simulator*, such as the Java interpreter [25], is a program that runs on one hardware architecture and implements in software a virtual machine conforming to a completely different architecture. In contrast, a *virtual machine monitor* or *hypervisor*, such as VM/370 [28, 29], is a program that creates one or more virtual machines exporting the same hardware architecture as the machine it runs on. Hypervisors are typically much more efficient than simulators because most of the instructions in the virtual machine environment can be executed at full speed on the bare hardware, and only "special" instructions such as privileged instructions and accesses to I/O registers need to be emulated in software. Since the "upper" and "lower" interfaces of a hypervisor are the same, a sufficiently complete hypervisor can even run additional copies of itself, recursively.

Virtual machines have been used for a variety of purposes including security, fault tolerance, and operating system software development [23]. In their heyday, virtual machine systems were not driven by modularity issues at all, but instead were created to make better use of scarce, expensive hardware resources. For example, organizations often needed to run several applications requiring different operating systems concurrently, and possibly test new operating systems, all on only a single mainframe. Therefore, traditional virtual machine systems used (and needed) only shallow hierarchies, implementing all required functionality in a single hypervisor. As hardware became cheap and ubiquitous, virtual machines became less common, although they remain in use in specialized contexts such as fault tolerance [7], and safe environments for untrusted applications [25].

In this paper we revive the idea of using virtual machine concepts pervasively throughout a system, but for the purpose of enhancing OS modularity, flexibility, and extensibility, rather than merely for hardware multiplexing. For these purposes, virtual machines based on hardware architectures have several drawbacks. First, most processor architectures allow "sensitive" information such as the current privilege level to leak into user-accessible registers, making it impossible for a hypervisor to recreate the underlying architecture faithfully without extensive emulation of even unprivileged instructions. Second, a hypervisor's performance worsens exponentially with stacking depth because each layer must trap and emulate all privileged instructions executed in the next higher layer (see Figure 3). Third, since hardware architectures are oblivious to the notion of stacking, all communication must be strictly parent-child; there is no way to support "short-circuit" communication between siblings, grandparents and grandchildren, etc.

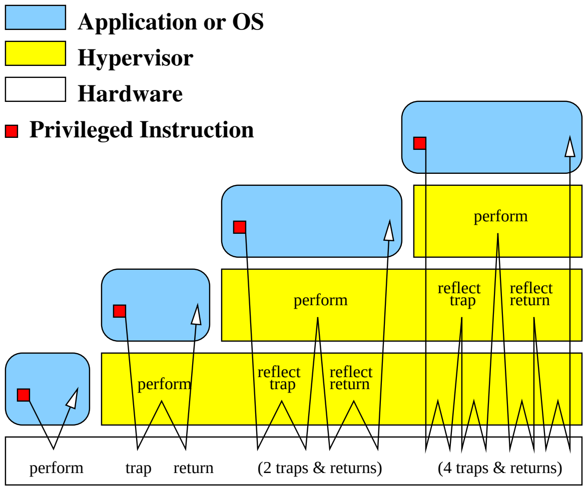

Figure 3: Exponential slowdown in traditional recursive virtual machines caused by emulation and reflection of privileged instructions, accesses to device registers, etc. For n hypervisor layers, at least 2^{n-1} hardware trap/return sequences are required to emulate any privileged instruction executed in the top layer.

2.2 The Nested Process Model

For the above reasons, Fluke does *not* conform exactly to an existing hardware architecture. Instead, we use a software-based architecture which allows us to solve these problems by defining the architecture to avoid them. Although some existing hypervisors export a virtual architecture that differs slightly from the architecture on which they are based, they generally try to minimize such divergence, changing the architecture only enough to address "show-stopping" problems such as double paging. In contrast, Fluke does not attempt to minimize divergence, since running existing operating systems unmodified in our virtual machines is not a goal. Instead, our design goal is to maintain maximum performance in the presence of deep virtual machine layering. The resulting architecture is similar to conventional process models such as Unix's, though with some important differences to make processes recursively virtualizable or "nestable"; hence the term "nested process architecture."

The Cambridge CAP computer [54] implemented a similar high-level architecture in microcode supporting an arbitrarily deep process hierarchy in which parent processes virtualize memory, CPU, and trap handlers for child processes. However, the CAP computer strictly enforced the process hierarchy at all levels, and did not allow communication paths to "short-circuit" the layers as Fluke does. As noted in retrospect by the designers of the system, this weakness made it impractical for performance reasons to use more than two levels of process hierarchy (corresponding to the "supervisor" and "user" modes of other architectures), so the uses of nested processes were never actually explored or tested in this system.

System call emulation, interposition, and stacking have been used in the past to virtualize the activity of a process by interposing special software modules between an application and the actual OS on which it is running. This form of interposition can be used, for example, to trace system calls or change the process's view of the file system [31, 33], or to provide security against an untrusted application [52]. However, these mechanisms can only be applied easily to a single application process and generally cannot be used in combination (only one interposition module can be used on a given process). Furthermore, although file system access and other system call-based activity can be monitored and virtualized this way, it is difficult to virtualize other resources such as CPU and memory.

2.3 Other Microkernel Architectures

Our nested process model shares many of the same goals as those driving microkernel-based systems: flexibility, modularity, extensibility, and the decomposition of traditional OS features into separate, unprivileged modules. In fact, our prototype is essentially a microkernel-based system revolving around a small privileged kernel that exports a minimal set of interfaces to higher-level software. The primary difference of interest in this paper is that our microkernel is designed to support a nested process model efficiently, and the higher-level services on top of the microkernel take advantage of this model to provide a strongly structured system instead of the traditional "flat" collection of client and server processes.

Our work shares many of the extensibility goals of other current kernel-related research [5, 18, 46, 50], but takes a different, often complementary, approach. For example, the Exokernel pushes the "red line" between the privileged kernel and unprivileged code as low as possible so that application code can gain more control over the hardware. In contrast, our work is not primarily concerned with the location of this boundary, but instead with certain fundamental properties the kernel interface must have in order for processes to virtualize each other effectively.

3 Properties of Virtual Machines

Before describing the Fluke architecture in detail, we first present the underlying principles on which it is based. All of these principles derive from a single goal: to preserve the useful properties of a "pure" recursive virtual machine system without incurring the same exponential performance cost. For our purposes, a "pure" virtual machine system is one in which each layer *completely* simulates the environment of the next higher layer, including all instructions, memory accesses, and device I/O [49]. Our work hinges on two primary properties of such systems: state encapsulation and border control. These properties are described briefly in the following sections, and a concrete analysis of how our architecture satisfies these properties is presented later in Section 4.

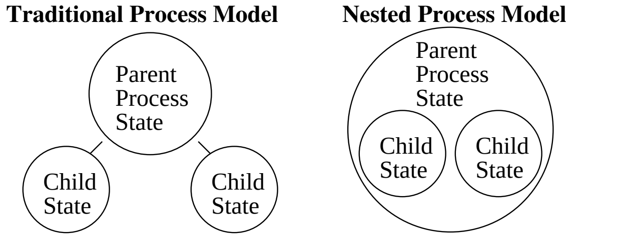

Figure 4: Process state in traditional versus nested processes. In traditional systems, the program state (code, data, heap, stack, threads, etc.) of a child process exists independently of the parent process. In a nested process architecture, the child's state is logically a part of the parent's even though the two run in separate address spaces.

3.1 State Encapsulation

The property of *state encapsulation* is the ability to encapsulate or "nest" one process inside another so that the entire state of a child process and all its descendants is logically a subset of the state of the parent process. This property allows the parent process to treat an entire child process hierarchy as merely a chunk of data that can be moved, copied, deleted, etc., while remaining oblivious to the implementation details of the child. Of primary interest are the following three specific aspects of this property: hierarchical resource management, state visibility, and reference relativity.

3.1.1 Hierarchical Resource Management

In popular systems such as Unix, Windows NT [45], and Mach [1], each process in the system exists independently of all other processes. A child process can outlive the parent process that created it, and retains all of its resources after its parent is destroyed (see Figure 4). By contrast, in a virtual machine system, a virtual machine cannot exist independently of the simulator or hypervisor that created it because the virtual machine's entire state (including the state of any sub-virtual machines it may contain) is merely part of the simulator's program variables. Some operating systems, such as L3 [39], directly support a hierarchical virtual machine-like process model in which a parent process can destroy a child process and be assured that all of the child's descendants will also go away and their resources will be freed. Other systems support a hierarchical *resource pool* abstraction from which the resources used to create processes are allocated; destroying a pool destroys any sub-pools created from it as well as all processes created from these pools. For example, KeyKOS *space banks* [6, 27] serve this purpose, as do *ledgers* in OSF's MK++ [43]. This hierarchical resource management property is a critical prerequisite for many of the applications described in this paper; without it, a parent process cannot even identify all of the state representing a child process subtree, let alone manage it coherently. Note that a traditional "flat" process model can easily be emulated in a system supporting the hierarchical model, as described in Section 5.4, but it is difficult or impossible to emulate a hierarchical model given only a flat collection of processes.

3.1.2 State Visibility

The second aspect of the state encapsulation property is *state visibility*: the ability of a parent process to "get at" the state of a child process subtree rather than merely having control over its lifetime. State visibility is needed by any application that manages child process state, such as demand paging, debugging, checkpointing, process migration, replication, and DSM. Pure virtual machine simulators obviously satisfy this property since all of the state of a virtual machine is merely part of the simulator's variables. Same-architecture hypervisors that "delegate" to the base processor the job of executing unprivileged instructions depend on the processor to reveal the child's full register state through trap frames or shadow registers.

With notable exceptions such as the Cache Kernel [14] and Amoeba [47], most operating systems are not as good about making a child process's state visible to its parent. For example, although most kernels at least allow a process to manipulate a child's user-mode register state (e.g., Unix's `ptrace()` facility), other important state is often unavailable, such as the implicit kernel state representing outstanding long-running system calls the child may be engaged in. While these facilities are sufficient for many applications such as debugging, other applications such as process migration and checkpointing require access to *all* of the child's state.

3.1.3 Relativity of References

While full state visibility is a necessary condition to move or copy a child process, it is also necessary that its references are relative to its own scope and are not absolute references into larger scopes such as the machine or the world. A "reference" for this purpose is any piece of state in the child process that refers to some other piece of state or "object." For example, virtual addresses, unique identifiers, and file descriptors are different forms of references. An *internal reference* is a reference in the child to some other object in the same process; an *external reference* is a reference to an object in an outer scope (e.g., in an ancestor or sibling process). For example, in Figure 5, references 1 and 2 are internal references with respect to process A, while reference 3 and 4 are external references.

In a traditional virtual machine system, external references in a virtual machine are effectively just the addresses of I/O ports or memory-mapped devices simulated by the hypervisor, and internal references are simply pointers within that machine's address space. In both cases, the representation of these references is relative to the virtual machine itself; the contents of the virtual machine can be moved at any time to a different context and all of the references it contains will remain valid.

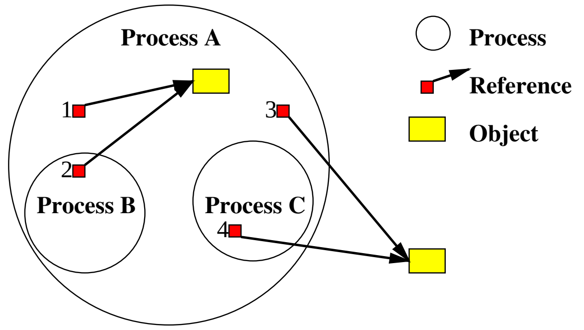

Figure 5: References into, out of, and within nested processes.

This relativity property is the exception rather than the rule in operating systems. Absolute pathnames in Unix serve as a simple example: if a process containing absolute pathnames is migrated from one machine to another, then the new machine must have *exactly* the same file system organization (at least in the parts accessed by the migrated program) or the external references from the program to the file system will become invalid.

On a system such as L3 in which process-internal objects such as threads are addressed using global unique identifiers, even *internal* references in a migrated process will become invalid unless exactly the same global identifiers can be obtained on the target machine. Furthermore, there can be no hope of cloning or replicating a process within a single global identifier namespace, since the global identifiers representing all the internal objects can only refer to one object within that scope. Implementing `fork()` in a single-address-space system [12] is a well-known instance of this problem.

3.2 Border Control

Another primary property of virtual machines is the ability of a hypervisor to monitor and control all communication across the border surrounding a virtual machine without impeding communication within the virtual machine. Barring shared memory channels, the only way for virtual machines to communicate with each other is to trap into the hypervisor, which can handle the request however it chooses. The Clans & Chiefs mechanism in L3 [38] provides border control directly in a microkernel environment. In most microkernels that use a capability model, such as Mach [1] and KeyKOS [6, 27], border control can be achieved through interposition.

Whereas the state encapsulation property allows the parent to control state *inside* the boundary, border control allows the parent to control the child's view of the world *outside* the boundary. As with state encapsulation, the extent to which border control is needed depends on the application in question. For example, Unix's ability to redirect the console output of an entire '`make`' run to a file, including the output of all the subprocesses it forks off, reflects the limited border control provided by the capability-like file descriptor model. However, other applications, such as security monitors which allow untrusted applications to be run safely, require *complete* border control.

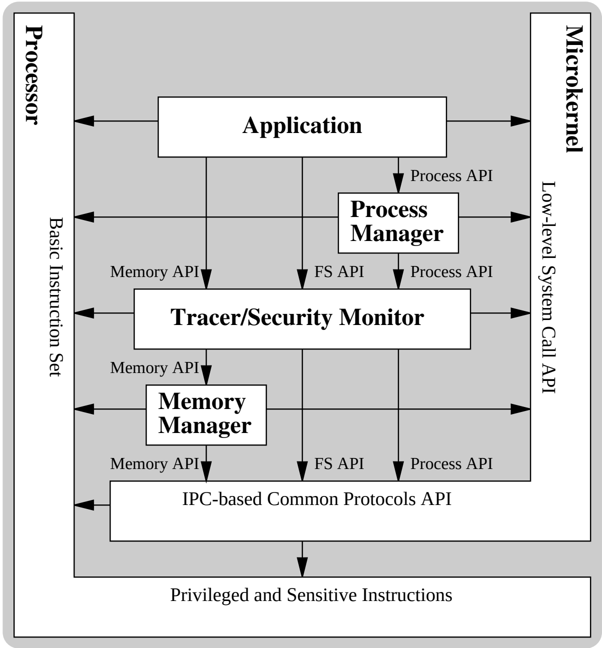

Figure 6: Illustration of the three components of the nested process architecture presented to each layer. The basic instruction set (left) used by all layers is implemented by the underlying processor and never needs to be emulated. The low-level system call API (right) is similarly implemented by the microkernel for all layers, whereas the Common Protocols API (bottom) is initially implemented by the microkernel but may subsequently be interposed on or reimplemented by higher-level layers.

4 Nested Process Architecture

This section describes the nested process architecture used in our system, known as Fluke ("Flux μ-kernel Environment"). The Fluke architecture is the common interface between each of the stackable layers in the system and consists of three components: the basic computational instruction set, the low-level system call API, and the IPC-based *Common Protocols*. As Figure 6 illustrates, the key distinction between these components is that the underlying processor provides the one and only implementation of the basic instruction set used throughout the system, and the microkernel provides the only implementation of the system call API, but the Common Protocols API is independently implemented or interposed on, at least in part, at each layer. The following sections describe the three components of the nested process architecture in detail and explain how this design efficiently supports stacking while preserving all of the critical properties described in the previous section.

```
fluke_type_create(objp)
```
Create a new object of type *type* at virtual address *objp*.

```
fluke_type_destroy(objp)
```
Destroy the object at virtual address *objp*.

```
fluke_type_move(objp, new_objp)
```
Move the object from virtual address *objp* to *new_objp*.

```
fluke_type_reference(objp, ref_objp)
```
Associate the reference object *ref_objp* with the object *objp*.

```
fluke_type_get_state(objp, statep, refp, ...)
```
Return the state of *objp*. State includes simple data passed back in the *type*-specific structure *statep* and zero or more capabilities returned in reference objects.

```
fluke_type_set_state(objp, statep, refp, ...)
```
Loads new state into *objp*. State arguments are identical to those of the `get_state` call.

Figure 7: Example of the Fluke low-level ("system call") API. These are the operations that are common to most object types. Each object type has additional type-specific operations; e.g., IPC operations for ports and lock/unlock operations for mutexes.

4.1 Basic Computational Instruction Set

The lowest level of our nested process architecture is a well-defined subset of the x86 architecture, allowing all processes to run with no instruction emulation on x86 machines. However, other processor architectures could be used just as easily, including purely emulation-based instruction sets such as Omniware [2] or Java bytecode [25]. The only restriction is that application processes must only use the subset of the instruction set that satisfies the properties described in Section 3; otherwise they may not function properly in virtualized environments. The instructions compilers produce generally satisfy these requirements automatically, because they refer only to per-thread register state or per-process memory. However, most architectures have a few "sensitive" yet unprivileged instructions that reveal global information and therefore cannot be virtualized. For example, the x86's `CPUID` instruction identifies the processor on which it is executed, making it impossible to safely migrate applications that rely on this instruction.

4.2 Low-level API

The second part of our nested process architecture is a set of low-level microkernel objects and system calls. These primitives are always implemented by the microkernel directly; therefore it is critical that they be designed to support all of the virtualization properties in Section 3. The reward for carefully designing the low-level API in this way is that it is never necessary for a parent process to interpose on the kernel calls made by child processes; instead, it is always sufficient for the parent process merely to retain control over the resources to which the system calls refer. Figure 7 shows a sample of the Fluke microkernel API [20].

4.2.1 Address Spaces

The Fluke kernel supports an arbitrary number of *address spaces*, much like stripped-down Unix processes or Mach tasks; multiple threads can run in each address space. Unlike most operating systems, address spaces in Fluke are defined relative to other spaces. The low-level API does not include system calls to allocate or free memory; it only provides primitives to remap a given virtual address range from one space into another space, possibly at a different virtual address range or with reduced permissions.

For example, a parent process can use this mechanism to "donate" access to some of its memory pages to a nested child process, for that child to use as its own private storage. This child can in turn donate some of this memory to *its* children, and so on, forming a hierarchy of memory remappings. The parent can change or revoke at any time the permissions the child has to this memory. If a thread in a descendant process attempts to access addresses in its space whose permissions have been revoked, the thread will take a page fault, which the kernel will deliver as an IPC message to a thread in the parent.

This memory remapping mechanism is similar to that of L4 [40], Grasshopper [41], and the "f-maps" in virtual machine systems [22, 23]. Our architecture uses this mechanism to provide the state containment and state visibility properties described in Section 3. Since a child process can only access memory given to it by its parent, the parent can locate and identify all the memory comprising the state of a child (including its descendants) simply by keeping track of which pages of memory it grants to the child. A parent can control the child's use of this memory by manipulating the mappings from its space into the child's.

4.2.2 Kernel Objects

All low-level API calls are operations on a few types of primitive *kernel objects*, such as address spaces and threads. All active kernel objects are logically associated with, or "attached to," a small chunk of physical memory; this is reminiscent of tagged processor architectures such as System 38 [37] and the Intel i960XA [30]. A process can invoke kernel operations on kernel objects residing on pages mapped into its address space by specifying the virtual address of the object as a parameter to the appropriate system call. Since kernel objects are identified by local virtual addresses, this design satisfies the relativity property for user-mode references to kernel objects. In addition, there are system calls that a process can use to determine the location of all kernel objects within a given range of its own virtual address space, as well as system calls to examine and modify the state of these objects [51].

This kernel object design, coupled with the address space remapping mechanism described above, provides the state containment and state visibility properties for kernel object state as well as "plain data," allowing parent processes to identify and gain access to *all* of the vital state comprising a child process. This mechanism is similar to the kernel object caching mechanism in the Cache Kernel [14], except that our mechanism does not impose ordering restrictions on manipulating kernel objects. For example, to take a checkpoint, a checkpointer temporarily revokes its child's access to the memory to be checkpointed, makes a system call to locate the kernel objects within this memory, makes additional kernel calls to extract the vital state from these objects (which includes the state of the child's threads, subprocesses, etc.), saves all the state of both the plain memory and the kernel objects, and finally resumes the child process. This is a very simplified description; the details of the procedure are presented later in Section 5.6. However, this is the fundamental mechanism by which not only checkpointing but other similar applications such as process migration and replication can be implemented transparently in our system.

4.2.3 Thread Objects

Multithreaded processes are directly supported in the low-level API through *thread objects*, which can be created and manipulated just like any other kernel object. Once a new thread object has been created and set up properly, it becomes an active, independent flow of control supervised directly by the kernel, executing instructions in its associated address space.

Unlike in most systems, Fluke threads provide full state visibility: a parent process can stop and examine the state of the threads in a child process at any time, and be assured of promptly receiving *all* relevant child state; i.e., everything necessary to transplant the child nondestructively. One implication of this property, which affects the low-level API pervasively, is that all states in which a thread may have to wait for events caused by other processes must be explicitly representable in the saved-state structure the kernel provides to a process examining the thread.

For example, our low-level API provides a "send request and await reply" IPC primitive as a single system call. Since the server may take arbitrarily long to service the request, and a parent process may try to snapshot the thread's state while the thread is waiting, the kernel must correctly reveal this intermediate state to the parent. In our system the kernel reveals this state by modifying the client thread's registers so that it appears to the parent that the client was calling the low-level API entrypoint that only waits for a reply without sending a message. This technique of adjusting user-level registers to reflect kernel state transitions is used throughout Fluke to provide full kernel state exportability without reducing efficiency by breaking up all system calls into minimal atomic units. The Mach IPC system [16] uses a similar technique, except that user code must explicitly check for interruption after every IPC operation and restart it manually if necessary. Also, various other Mach system calls are not cleanly interruptible at all and therefore do not support full state visibility.

4.2.4 Capabilities

All references between low-level objects in Fluke are represented as kernel-mediated *capabilities* [37]. Each primitive object type contains a number of "capability slots." For example, each thread object contains an address space slot, which refers to the address space in which the thread is to run. Processes can store and manipulate individual capabilities using *reference objects*, which are kernel objects that hold a single capability of any type. System calls are provided to produce capabilities pointing to existing kernel objects, copy capabilities between reference objects and the special-purpose slots in other kernel objects, pass capabilities through IPC, and compare them against each other. A process only refers to a capability by the virtual address of the kernel object in which it is contained; it never has direct access to the capability itself.

Capabilities in our API provide the relativity property (Section 3.1.3) for cross-address-space references, such as references within a child process to objects implemented by its parent. Since only the kernel can access the actual contents of a capability, capabilities can be passed freely between arbitrary processes in our system, allowing communication to short-circuit the process hierarchy when appropriate. This contrasts with CAP [54], where capabilities are used only for communication within a process and all interprocess communication is strictly parent-child and based on traps. Kernel-mediated capabilities satisfy the relativity property because even though the contents of a capability are absolute in the view of the kernel (i.e., it is typically just a pointer in the kernel's address space), from the view of any user-level process, a capability appears relative since the process has no access to the actual pointer stored in the capability. A parent process can locate all of the capabilities contained in a child process, discover to which objects they point, and transparently make substitutions of these capabilities, analogous to "pointer swizzling" done in the persistence and language domains.

Capabilities also provide border control (Section 3.2). Since a parent process determines what capabilities it passes to its child, it can interpose on any of these capabilities, as well as on capabilities subsequently passed into or out of the child through communication channels on which the parent has interposed. This way, the parent can completely monitor and control all communication into or out of a process by intercepting IPC messages. However, the capability model also allows the parent to interpose *selectively* on only a subset of the capabilities provided to a child. For example, a nester that interposes on file system-related operations does not also need to intercept messages to other external services. This contrasts with L3's Clans & Chiefs model [38], where interposition is "all or nothing": if a parent wants to intercept any communication transparently, it must intercept *all* communication.

4.2.5 Scheduling

The final type of resource with which the Fluke kernel deals directly is CPU time. As with memory and communication, the kernel provides only minimal, completely relative scheduling facilities. Threads can act as schedulers for other threads, donating their CPU time to those threads according to some high-level scheduling policy; those threads can then further subdivide CPU time among still other threads, etc., forming a *scheduling hierarchy*. The scheduling hierarchy usually corresponds to the nested process hierarchy, but is not required to do so.

Our scheduling model, which has been prototyped and tested in user space but not yet as part of the Fluke kernel, is detailed in a companion paper [21]. However, only its relative, hierarchical nature is important to the nested process architecture. Other hierarchical schedulers should also work, such as the meter system in KeyKOS [27], lottery/stride scheduling [53], and SFQ scheduling [26].

4.3 High-level Protocols

While our low-level IPC mechanism provides primitive capability-based communication channels, a higher level protocol defines the conventions for communication over these channels. The key distinction between the high-level protocols and the low-level system call API is that *any* layer in the system can provide implementations of the objects defined by these interfaces, whereas only the microkernel can implement the low-level API. High-level objects automatically support the important properties in Section 3 because they are based on a capability model which allows interposition; the disadvantage is that interposition has a significant cost. We have designed our system around this trade-off by placing simple, performance-critical activities in the low-level API while leaving services that are invoked less often to be implemented by high-level objects.

Our high-level interfaces, defined in CORBA IDL, are known as the *Common Protocols* because they are common to each layer in the nesting hierarchy. The Fluke Common Protocols, modeled on existing multiserver systems [8, 33], are designed to support a POSIX-compatible environment, though many of the interfaces are generic enough to support other APIs such as Win32. Table 1 shows the currently defined interfaces and example methods for each.

The most basic interface, the Parent interface, is used for direct parent/child communication, and effectively acts as a first-level "name service" interface through which the child requests access to other services. This is the only interface that *all* nesters interpose on; nesters selectively interpose on other interfaces only as necessary to perform their function. The cost of interposition on the Parent interface is minimal because the child usually makes only a few requests on this interface, during its initialization phase, to find other interfaces of interest. The Parent interface currently provides methods to obtain initial file descriptors (e.g., `stdin`, `stdout`, `stderr`), to find file system, memory, and process managers, and to exit. The relevant details of these interfaces will be presented in the next section as we describe specific applications that use and interpose on them.

`Parent::`	Basic parent-child interface
`get_process`	Get the Process Management interface
`get_mem_pool`	Get the memory pool for this process
`Process::`	POSIX process management interface
`create_child`	Create a new POSIX child process
`exec`	Execute a program
`MemPool::`	Memory Management interface
`create_var_segment`	Create a growable memory segment
`create_sub`	Create a sub-pool of a pool
`FileSystem::`	File system interface
`open`	Open a file
`mkdir`	Create an empty directory
`FileDescription::`	Open file and segment interface
`read`	Read data from a file
`map`	Map a file or memory segment

Table 1: Fluke Common Protocols interfaces and example methods. The top-level Parent interface acts as a name service for obtaining references to process management, memory management and other nesters.

5 System Implementation

In the following sections we overview the kernel, support libraries, and example nesters we have implemented to demonstrate the nested process model. These user-level applications take advantage of the model to provide traditional OS features, namely POSIX process management, demand paging, checkpointing, debugging, and tracing, in a more flexible and decomposed way. Table 2 lists the example nesters and the Common Protocols interfaces they interpose on.

5.1 The Microkernel

To provide the initial, base-level implementation of our nested process architecture, we developed a new microkernel that runs directly on "raw" x86 PC hardware. The initial implementation was designed primarily with portability, readability, and flexibility in mind rather than maximum performance; it is written mostly in machine-independent C and relies heavily on generic facilities provided by the Flux OS Toolkit [19]. The prototype supports kernel preemptibility, full multiprocessor locking, and can be configured to run under either an interrupt or process model. The kernel currently contains built-in console, serial, clock, disk, and network drivers, although it is designed to support out-of-kernel drivers in the future.

Besides implementing the low-level API used by all processes, the microkernel also implements a first-level Common Protocols interface defining the environment presented to the first process loaded (the "root" process). This

Nester	Parent	MemPool	Process	FileSys	File
Debug/Trace	√	√	√	√	√
Process	√		√		
Memory	√	√			√
Checkpoint	√	√	√	√	√

Table 2: Fluke nesters and the interfaces they interpose on. FileSys is the file system interface; File is the file and memory segment interface.

initial interface is sufficient to allow various simple applications as well as nesters to be run directly on top of the microkernel: for example, it includes a basic (physical) memory allocation interface, and a minimal root file system interface which acts like a RAM disk.

Although it should be possible for a traditional monolithic kernel to implement a nested process architecture, we took a microkernel approach for the proof-of-concept, as it would be much more difficult to adapt an existing monolithic kernel because of the large source base and numerous changes that would be required. In addition, a monolithic kernel would benefit less from a nested process model because so much functionality is already hard-wired into the kernel. For example, while our checkpointer example would probably still apply, the decomposed process manager and virtual memory manager wouldn't. Because we chose the microkernel approach, our system takes the well-known "microkernel performance hit" [13] due to the additional decomposition and context switching overhead. This performance impact is made worse by the fact that our microkernel is new and entirely unoptimized.

5.2 The Libraries

In our system, traditional Unix system calls are implemented by the C library residing in the same address space as the application. These C library functions communicate with ancestor nesters and external servers as necessary to provide the required functionality. For example, each process's file descriptor table is managed by its local C library and stored in the process itself, as IPC capabilities to objects managed by file servers.[1] Our system currently provides two different C libraries: a "minimal" library for nesters and simple applications, and a full-blown BSD-based C library to support "real" Unix applications. Similar libraries could be designed to provide compatibility with other APIs and ABIs, such as Linux or Win32.

The Nesting library, linked only into nesters and not ordinary applications, provides the "parent-side" complement to the C library. For example, it contains standard functions to spawn nested subprocesses and to handle or forward a child's Common Protocols requests. Use of this library is optional. Applications can always create nested processes manually in whatever way they desire; the library only provides a "standard" mechanism for creating children and providing Common Protocols-compatible interfaces to them.

[1] The actual files and "open file descriptions," containing seek pointers and other per-open state, are maintained by the file servers; this greatly simplifies some of the traditionally hairy Unix multiserver issues.

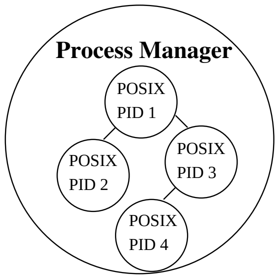

Figure 8: Process manager managing a POSIX-style process hierarchy. Each POSIX process is an immediate child of the process manager in terms of the nested process hierarchy. Process IDs and the POSIX-level parent/child relationships are implemented within the process manager.

5.3 The Debug and Trace Nesters

We have implemented, as a simple nester, a debugger that can be used to debug either ordinary applications or other nesters. The debugger works by handling exceptions that occur in the child process. When a thread in the child faults, the kernel sends an exception RPC to the debugger, which handles the fault by communicating with a remote host running GDB. The debugger restarts the thread by replying to the kernel's exception RPC. Although Mach provides a similar ability to interpose on an exception port, Mach allows a task to change its own exception port reference, potentially allowing a buggy or uncooperative task to escape the debugger's control.

We have also implemented a simple tracer, as a minor modification to the debugger, which can be used to trace the message activity of an arbitrary subprocess. The tracer starts by interposing on the application's Parent interface (as all nesters do). Then, for any request from the child that returns a new IPC capability, the tracer interposes on it. This interposition is completely generic in that the tracer needs to know nothing about the semantics of what it is interposing on; it will work for any IPC-based protocol. Although the tracer does nothing more than record messages that cross the subprocess's border, a security monitor for untrusted applications would interpose in basically the same way.

5.4 The Process Management Nester

Support for POSIX-style processes in our system is provided with the help of a *process manager*, which is the only nester in our prototype system that manages multiple immediate subprocesses at once. Each POSIX process, regardless of its location in the Unix-style process hierarchy, is a direct child of the process manager in the global nested process hierarchy, as illustrated in Figure 8. This allows a POSIX child processes to outlive its POSIX parent, as is the proper behavior for POSIX-style processes. The Process Manager assigns a process ID to each of its child processes, and allows these processes to create other "peer" POSIX processes using `fork()` and `exec()`, which the C library converts into Common Protocols requests to the process manager. The process manager also handles interprocess signals and other POSIX process-related features. As with the other nesters, the Process Manager is an optional component: applications that don't `fork()`, send signals, etc., can be run without it. Furthermore, multiple process managers can be run side-by-side or even arbitrarily "stacked" on top of each other to provide multiple independent POSIX environments on a single machine; each of these environments has its own process ID namespace and POSIX process hierarchy.

The process manager uses the Common Protocols' *MemPool* ("memory pool") interface to manage the memory resources consumed by its children and ensure that all state is properly cleaned up when a POSIX process terminates. A MemPool is an object from which anonymous memory segments can be allocated and mapped by any process holding a capability for the pool. The MemPool interface also provides methods to create and destroy sub-pools; destroying a pool destroys all memory segments allocated from it and, recursively, all sub-pools derived from it. Thus, MemPools provide the hierarchical resource management property (Section 3.1.1) for memory. The process manager uses the MemPool capability from its parent to create a sub-pool for each POSIX process it supervises. When a process terminates or is killed, the process manager simply destroys the appropriate sub-pool; this automatically frees all memory allocated by that process and any nested subprocesses that may have been created. Furthermore, since the subprocess's threads and other kernel objects are merely part of its memory, they too get destroyed automatically when the MemPool is destroyed.

5.5 The Virtual Memory Management Nester

We have implemented a user-level demand paged virtual memory manager which creates a nested environment whose anonymous memory is paged to a swap file. Arbitrary programs can be run in this paged environment, such as a single application, or a process manager supporting an entire paged POSIX environment similar to a traditional Unix system. The memory manager provides paged memory by interposing on the MemPool interface it passes to its child, re-implementing the pool and segment operations itself instead of merely creating a sub-pool and passing the sub-pool to the child as the process manager does.

All memory segments allocated from the memory manager are backed by a swap file and cached in its own address space. The "physical memory" cache used by the memory manager is a memory segment allocated from the memory pool passed in by the memory manager's parent; the swap file is implemented by a Common Protocols file system server.

When a client invokes the `map` operation on a segment implemented by the memory manager, the memory manager uses Fluke kernel address space manipulation primitives to remap the appropriate portions of its own memory into the address space of the client. These mappings correspond to "resident pages" in a conventional, kernel-based virtual memory system. The kernel notifies the memory manager of access violation and non-resident exceptions in the segments it supervises, allowing it to do demand allocation of memory and lazy copy optimization.

In the prototype memory manager, the physical memory cache is divided into fixed-size (4k) pages. All allocations and I/O operations are performed on single pages; no prepaging or other clustering techniques are used. It implements a simple global FIFO page replacement algorithm and uses a single, fixed-size file as its backing store. The current implementation does not maintain "dirty" bits, so pages are always written to backing store.

5.6 The Checkpoint Nester

We have implemented a user-level checkpointer that can operate over a single application or an arbitrary environment, transparently to the target. By loading a checkpointer as the "root" nester immediately on top of the microkernel, a whole-machine checkpointed system can be created. To our knowledge this is the first checkpointer that can operate over arbitrary domains in this way.

Like the memory manager, the checkpointer interposes on the MemPool interface in order to maintain complete control over the state of its child processes. Since the kernel provides primitives to locate and manipulate the low-level objects within a memory region, the checkpointer effectively has direct access to all kernel object state in the child as well as to its raw data. The checkpointer currently uses a simplistic sequential checkpointing algorithm to take a checkpoint: it temporarily revokes all permissions on the memory segments it provides (which also effectively stops all threads in the child since they have no address space to execute in), saves the contents of the child's memory and kernel object state, and then re-enables the memory permissions to allow the child to continue execution. This algorithm, of course, will not scale well to large applications or distributed environments. However, more efficient checkpointers based on well-known single-process algorithms [15, 17] could be implemented in our environment in the same way, and should also work automatically over multi-process domains.

In order to checkpoint kernel objects that contain capabilities, the checkpointer discovers what object each capability points to and replaces it with a simple identifier that is unique within the saved checkpoint image. There are two classes of capabilities that the checkpointer must deal with, corresponding to the two types of references described in Section 3.1.3. To handle capabilities representing internal references (references to other objects within the checkpointed environment), the checkpointer builds a catalog of the objects in the checkpointed environment and uses kernel primitives to look up each capability in this catalog. Capabilities representing external references (references to objects outside the checkpointed environment) will not appear in the catalog, but since any external reference owned by the child environment must have been granted to it by the checkpointer at some point, the checkpointer can recognize it and take a reasonable course of action.

For example, the capabilities representing the `stdin`, `stdout`, and `stderr` file handles are recognized by the checkpointer and, on restart, are reinitialized with the corresponding file handles in the new environment. Thus, all standard I/O file descriptors (including descriptors in nested subprocesses of the application) are transparently routed to the new environment. Similarly, when the child makes a Parent interface request for an external service such as process management, the checkpointer keeps track of the IPC capability returned, so that it can route it to the new environment. IPC capabilities that the checkpointer chose not to interpose on, such as open files other than the standard I/O handles, are replaced with null references. This has similar consequences to an NFS server going down and leaving stale file handles behind. Although our current implementation doesn't interpose on any file system accesses, it could easily recognize `open` calls and save file names, or even whole files, in order to provide a more consistent restart.

The checkpointer is our most comprehensive nester, taking advantage of all of the virtual machine properties described in Section 3 to provide the most complete encapsulation possible. The state encapsulation, visibility, and relativity properties allow the checkpointer to save and restore the state of the child's memory and kernel objects. Additionally, the border control provided by the capability model allows the checkpointer to interpose on whichever interfaces it needs to; our implementation interposes on only those things necessary for a minimal complete checkpoint, comparable to the functionality offered by other user-level checkpointers [42, 48].

There is a large body of work on checkpointers both in and out of the kernel. A few existing operating systems, such as KeyKOS [34] and L3 [39], have implemented checkpointing on a whole-machine basis in the kernel. Similarly, a hypervisor was recently used to provide fault tolerance (replication) on a whole-machine basis on PA-RISC machines [7]. While these features appear practical and useful in some situations, they are inflexibly tied to the machine boundary and cannot easily be used on smaller scopes, such as a process or a group of processes, or on larger scopes, such as networked clusters of machines. The nested process model allows checkpointing and other algorithms to be implemented over more flexible domains.

6 Experimental Results

In order to evaluate the performance effects of nesting processes in our Fluke implementation, we used a set of micro benchmarks designed to reveal operating system performance properties that directly affect real-world applications. Our primary interest in these tests is to show the performance effect of different nesters on various types of applications; thus, we are chiefly concerned with relative slowdown due to nesting rather than the absolute performance of the system. All tests were performed on 200MHz Pentium Pro PCs with 128MB of RAM. The micro benchmarks used are:

memtest: This is a simple memory tester which allocates as much memory as it can with `sbrk` and then makes four passes over it. Each pass writes and then reads back a distinct pattern looking for bit errors. We configured the heap to 4MB for this test. Memtest stresses memory management and is representative of programs that first allocate memory and then repeatedly operate on it.

appel2: This benchmark tests a combination of basic virtual memory primitives as identified by Appel and Li [3]. This test, known as "trap+protN+unprot" in the original paper, measures the time to protect 100 pages, randomly access each page, and, in the fault handler, unprotect the faulting page. Where memtest emphasizes higher-level Common Protocols memory management, appel2 stresses the microkernel's low-level memory remapping mechanism.

forktest: This program exercises the POSIX `fork()` and `wait()` operations of the Process Manager. Forktest creates a parent-child chain four levels deep; each level performs a simple computation and then returns the result in its exit status.

readtest: This test is similar to the `lmbench` [44] `bw_file_read` test. It reads an 8MB file in 8KB chunks accessing every word of the buffer after each read. It is intended to discover the best-case file caching capability of the operating system.

matconn: The final benchmark is a computationally intensive application. It uses Warshall's algorithm to compute connectivity in adjacency matrices, and generates successively larger matrices starting at 2x2 and ending with 128x128.

6.1 Absolute Performance

To provide a baseline for further evaluation, Table 3 presents absolute times for various primitive Fluke microkernel operations. Table 4 shows absolute times for the aforementioned benchmark programs running directly on top of the microkernel with no intervening processes. For reference, we also show micro benchmark performance results for FreeBSD (version 2.1.5). Fluke performs reason-

	Time (μs)
Null system call	2.0
Mutex lock	3.6
Mutex unlock	4.0
Context switch	7.5
Null cross-domain RPC	14.9

Table 3: Absolute performance of microkernel primitives.

Test	Fluke	FreeBSD
`memtest`	929091 μs	914918 μs
`appel2`	5372 μs	3620 μs
`forktest`	N/A	2534 μs
`readtest`	125844 μs	153010 μs
`matconn`	102917 μs	71568 μs

Table 4: Absolute micro benchmark results for Fluke and FreeBSD. Fluke times reflect benchmarks running directly on top of the microkernel's minimal Common Protocols server. FreeBSD times were collected in single-user mode. All times were measured using the Pentium timestamp counter.

ably in most tests even though it is, for the most part, an unoptimized microkernel and FreeBSD is a mature, well-optimized monolithic kernel. There is no time listed for forktest in the Fluke column since the Fluke kernel's Common Protocols server does not implement the Process Management interface.

6.2 Overhead of Interposition

Figure 9 illustrates the overhead associated with interposition. Each benchmark was first run under the "bare" environment created by the microkernel to serve as a baseline. Then the benchmarks were run under one to four levels of the previously described trace nester which interposes on all IPC channels but simply passes data along. In this experiment, readtest is the only benchmark exhibiting measurable slowdown. This is because this test's running time is dominated by data copying since each interposition on the FileDescription read method results in an extra copy as data passes through the tracing nester's address space. These large IPC data transfers are unoptimized in the prototype kernel.

6.3 Performance of Various Nester Hierarchies

Figure 10 illustrates the effect of increasing levels of nesting on the micro benchmarks. The nester stacks were chosen to be representative of "real world" situations. The no-nesters case reflects a real-time or embedded system where only minimal OS services are required. Addition of the process manager (P) and memory manager (M) shows increasing levels of functionality, providing multitasking and traditional virtual memory services. A checkpointer (C) can be included to checkpoint or migrate the entire environment on demand. No checkpoints are actually taken during these tests, but the checkpointer performs all of the interposition and state management activities required to be able to take a checkpoint at any time. The final stack shows the insertion of the tracing nester (T) where it might be used to trace Common Protocols activity.

The matconn and appel2 benchmarks are largely unaffected by nesting. The matconn benchmark is not impacted because it is largely computational. The appel2 result demonstrates the importance of the low-level kernel API directly supporting the virtualization properties described in Section 3. Since nesters do not interpose on the low-level interface, increasing the level of nesting has minimal impact on appel2, even though the nesters may be virtualizing memory. There is a modest (1–4% per level) kernel overhead associated with these nesters if they remap memory through themselves into their children. By mapping a child's memory into its own address space, a nester creates an additional level of mapping hierarchy in the kernel which may need to be traversed when performing virtual-to-physical address translations.

The readtest benchmark is not significantly affected by most nesters since they do not interpose on the file system interface. As in the complete interposition case, there is a significant impact (34%) when the tracing nester is added.

As expected, memtest is only impacted by memory interposition. An interesting observation is that memtest is affected to a much greater degree by the memory manager than by the checkpointer even though both interpose and remap the memory used. The difference is the way in which they remap the memory allocated for a child's segment. The current memory manager provides memory to a segment using single-page sized mappings to make page-level pagein and pageout easy, while the checkpointer uses one large multi-page mapping. The prototype kernel is not yet optimized to deal with the large number of kernel data structures that result from the memory manager's behavior.

The forktest benchmark shows a pattern similar to that of memtest, except that the performance degrades more severely. The greater impact is due to the large ratio of Common Protocols calls to other activity.

6.4 Status

Besides the simple tests and nesters above, our system is just starting to run some larger applications including GNU `make`, `gawk`, `sed`, `bash`, and `gcc`. We expect to make a full, self-hosting public release within the year.

7 Conclusion

In this paper we have presented a novel approach to providing modular and extensible operating system functionality based on a synthesis of microkernel and virtual machine concepts. We have demonstrated the design and im-

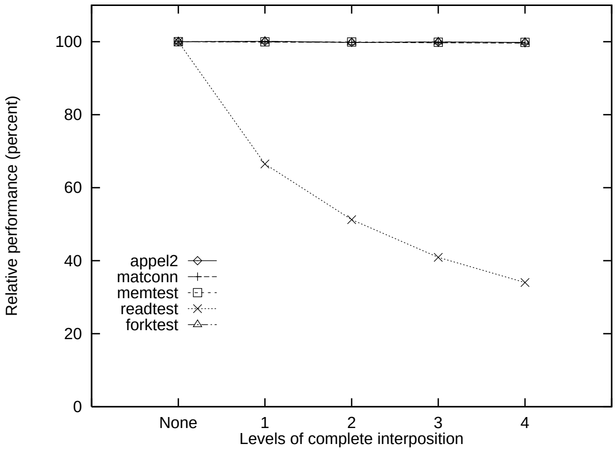

Figure 9: Worst-case overhead due to IPC interposition. Performance is measured relative to the appropriate base (no interposition) case for all tests.

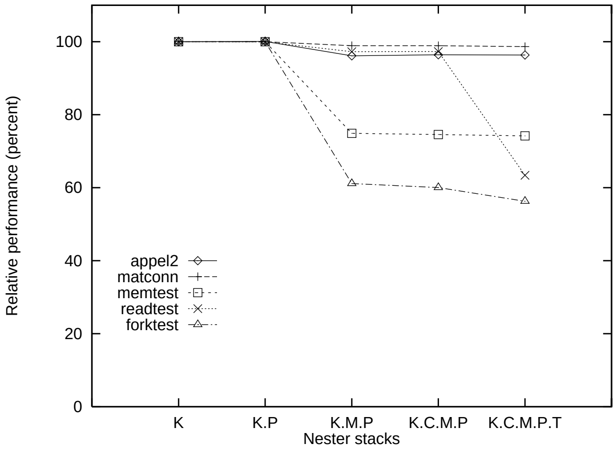

Figure 10: Slowdown due to increasing levels of process nesting. Nester stacks on the horizontal axis were chosen to be representative of actual usage. In each stack, 'K' is the kernel's Common Protocols server, 'P' is the process manager, 'M' is the memory manager, 'C' is the checkpointer, and 'T' is the tracer.

plementation of a microkernel architecture that efficiently supports decomposition of traditional OS services such as process management, demand paging, fault tolerance, and debugging, into cleanly modularized, stackable layers. Our prototype implementation of this model indicates that it is practical to modularize operating systems this way. Initial micro benchmark results are encouraging, showing slowdowns of 0–30% per layer.

Acknowledgements

For their many thoughtful and detailed comments on earlier drafts we thank our patient shepherd Marc Shapiro, the many anonymous reviewers, and Fred Schneider. Jochen Liedtke provided critique and discussion which influenced our kernel design. We thank Steve Smalley and Jeff Turner for their courage in being early guinea pigs and their critical intellectual involvement. Discussions with Jonathan Shapiro also influenced the kernel, while Michael Bushnell and Miles Bader provided key insights on the design of higher-level services. Gary Koob gave us the freedom and trust to try something this new. Finally, we are grateful to all members of the Flux project team for substantial and varied help along the way, including Chris Alfeld, John Carter, Eric Eide, Shantanu Goel, Linus Kamb, Gary Lindstrom, John McCorquodale, Roland McGrath, Bart Robinson, Sai Susarla, Kevin Van Maren, and Teresa Whiting.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of the Summer 1986 USENIX Conf.*, pages 93–112, June 1986.

[2] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and Language-Independent Mobile Programs. In *Proc. ACM SIGPLAN Symp. on Programming Language Design and Implementation*, pages 127–136, May 1996.

[3] A. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, June 1991.

[4] N. Batlivala, B. Gleeson, J. Hamrick, S. Lurndal, D. Price, and J. Soddy. Experience With SVR4 Over Chorus. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 223–241, Seattle, WA, Apr. 1992.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety, and Performance in the SPIN Operating System. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[6] A. C. Bomberger and N. Hardy. The KeyKOS Nanokernel Architecture. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Seattle, WA, Apr. 1992.

[7] T. C. Bressoud and F. B. Schneider. Hyporvisor-based Fault-tolerance. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 1–11, Dec. 1995.

[8] M. I. Bushnell. Towards a New Strategy of OS Design. In *GNU's Bulletin*, Cambridge, MA, Jan. 1994. Also http://www.cs.pdx.edu/~trent/gnu/hurd-paper.html.

[9] P. Cao, E. W. Felten, and K. Li. Implementation and Performance of Application-Controlled File Caching. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 165–177, Monterey, CA, Nov. 1994. USENIX Assoc.

[10] J. Carter, D. Khandekar, and L. Kamb. Distributed Shared Memory: Where We Are and Where We Should Be Headed. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, pages 119–122, May 1995.

[11] K. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

[12] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single Address Space Operating System. Technical Report UW-CSE-93-04-02, University of Washington Computer Science Department, Apr. 1993.

[13] J. B. Chen and B. N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, pages 120–133, 1993.

[14] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 179–193. USENIX Assoc., Nov. 1994.

[15] G. Deconinck, J. Vounckx, R. Cuyvers, and R. Lauwereins. Survey of Checkpointing and Rollback Techniques. Technical Report O3.1.8 and O3.1.12, ESAT-ACCA Laboratory Katholieke Universiteit Leuven, Belgium, June 1993.

[16] R. P. Draves. A Revised IPC Interface. In *Proc. of the USENIX Mach Workshop*, pages 101–121, Oct. 1990.

[17] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The Performance of Consistent Checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47, Oct. 1992.

[18] 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 15th ACM Symp. on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[19] B. Ford and Flux Project Members. The Flux Operating System Toolkit. University of Utah. Postscript and HTML available under http://www.cs.utah.edu/projects/flux/oskit/html/, 1996.

[20] B. Ford, M. Hibler, and Flux Project Members. Fluke: Flexible μ-kernel Environment (draft documents). University of Utah. Postscript and HTML available under http://www.cs.utah.edu/-projects/flux/fluke/html/, 1996.

[21] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996. USENIX Assoc.

[22] R. P. Goldberg. Architecture of Virtual Machines. In *AFIPS Conf. Proc.*, June 1973.

[23] R. P. Goldberg. Survey of Virtual Machine Reseach. *IEEE Computer Magazine*, pages 34–45, June 1974.

[24] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *Proc. of the Summer 1990 USENIX Conf.*, pages 87–96, Anaheim, CA, June 1990.

[25] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems Computer Company, 1996. Available as http://java.sun.com/doc/-language_environment/.

[26] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler For Multimedia Operations. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996. USENIX Assoc.

[27] N. Hardy. The KeyKos Architecture. *Operating Systems Review*, Sept. 1985.

[28] IBM Virtual Machine Facility /370 Planning Guide. Technical Report GC20-1801-0, IBM Corporation, 1972.

[29] IBM Virtual Machine Facility /370: Release 2 Planning Guide. Technical Report GC20-1814-0, IBM Corporation, 1973.

[30] Intel. *i960 Extended Architecture Programmer's Reference Manual*, 1994.

[31] M. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, pages 80–93, Dec. 1993.

[32] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the 1994 Winter USENIX Conf.*, pages 115–132, Jan. 1994.

[33] Y. A. Khalidi and M. N. Nelson. Extensible File Systems in Spring. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, pages 1–14, 1993.

[34] C. Landau. The Checkpoint Mechanism in KeyKOS. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, Sept. 1992.

[35] H. C. Lauer and D. Wyeth. A Recursive Virtual Machine Architecture. In *ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, pages 113–116, Mar. 1973.

[36] C. H. Lee, M. C. Chen, and R. C. Chang. HiPEC: High Performance External Virtual Memory Caching. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 153–164, Monterey, CA, Nov. 1994. USENIX Assoc.

[37] H. M. Levy. *Capability Based Computer Systems*. Digital Press, 1984.

[38] J. Liedtke. Clans and Chiefs. In *Proc. 12. GI/ITG-Fachtagung Architektur von Rechensystemen*, 1992.

[39] J. Liedtke. A Persistent System in Real Use – Experiences of the First 13 Years. In *Proc. of the Third International Workshop on Object Orientation in Operating Systems*, pages 2–11, Dec. 1993.

[40] J. Liedtke. On Micro-Kernel Construction. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, Dec. 1995.

[41] A. Lindstrom, J. Rosenberg, and A. Dearle. The Grand Unified Theory of Address Spaces. In *Proc. of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[42] M. Litzkow and M. Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Proc. of the Winter 1992 USENIX Conf.*, 1992.

[43] K. Loepere et al. MK++ Kernel Executive Summary. Technical report, Open Software Foundation, 1995.

[44] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proc. of 1996 USENIX Conf.*, Jan. 1996.

[45] Microsoft Corporation. *Win32 Programmer's Reference*, 1993. 999 pp.

[46] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A Communications-oriented Operating System. Technical Report 94–20, University of Arizona, Dept. of Computer Science, June 1994.

[47] S. J. Mullender. *Experiences with Distributed Systems*, chapter Process Management in Distributed Operating Systems. Lecture Notes in Computer Science no. 309. Springer-Verlag, 1987.

[48] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proc. of the Winter 1995 USENIX Technical Conf.*, Jan. 1995.

[49] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Fast and Accurate Multiprocessor Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, 3(4), 1995.

[50] C. Small and M. Seltzer. VINO: An Intergrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard University, 1994.

[51] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler. User-level Checkpointing Through Exportable Kernel State. In *Proc. Fifth International Workshop on Object Orientation in Operating Systems*, Seattle, WA, Oct. 1996. IEEE.

[52] D. Wagner, I. Goldberg, and R. Thomas. A Secure Environment for Untrusted Helper Applications. In *Proc. of the 6th USENIX Unix Security Symp.*, 1996.

[53] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, Sept. 1995.

[54] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System*. North Holland, NY, 1979.