An Implementation of UNIX™ on an object-oriented operating system

Yousef A. Khalidi **Michael N. Nelson**

Sun Microsystems Laboratories, Inc.
Mt. View, CA 94043 USA

Abstract

This paper describes an implementation of UNIX on top of an object-oriented operating system. UNIX is implemented without modifying the underlying mechanisms provided by the base system. The resulting system runs dynamically-linked UNIX binaries and utilizes the services provided by the object-oriented system.

1 Introduction

In this paper we describe an implementation of a UNIX system built using Spring, an experimental object-oriented operating system developed by our research group at Sun Microsystems Laboratories. The UNIX implementation presented here is a subset of SunOS 4.1 and runs most SPARC International SCD 1.1 compliant programs.

1.1 Motivation.

A problem that we faced once we built our operating system kernel and a set of core system services was how to proceed with building an application base. Other new systems when faced with the same problem have either built an application base, ported an application base, or provided the ability to run binaries from another system such as UNIX. We chose to use the last approach because we wanted to start using our system to build interesting applications using the base object model provided by Spring, without having to first implement or port a window system, an editor, and a compiler. Therefore, we decided to implement a UNIX subsystem to be able to leverage the vast majority of existing UNIX applications. Moreover, building the UNIX system on Spring served as a proof of the viability of the Spring design and as a way to exercise the system.

This paper is organized as follows: in the remainder of this section we give a brief overview of the Spring system. Section 2 lists the design goals of the implementation. Section 3 gives an overview of the architecture while 4 describes the implementation in detail. The implementation status is presented in section 5. A comparison to other related work is presented in Section 6. Finally, conclusions and several possible extensions to this work are listed in Section 7.

1.2 Spring Operating System

Spring is a distributed, multi-threaded, extensible operating system that is structured around the notion of *objects*. A Spring object is an abstraction that contains state and provides a set of methods to manipulate that state. The

description of the object and its methods is an *interface* that is specified in an *interface definition language*. The interface is a strongly-typed contract between the implementor (*server*) and the *client* of the object.

A Spring *domain* is an *address space* with a collection of *threads*. A given domain may act as the server of some objects and the clients of other objects. The implementor and the client can be in the same domain or in a different domain. In the latter case, the representation of the object includes an unforgeable nucleus *door* (or *handle*) that identifies the server domain.

Since Spring is object-oriented it supports the notion of *interface* inheritance. Spring supports both notions of *single* and *multiple* interface inheritance. Interface inheritance is an important factor in making Spring extensible. An interface that accepts an object of type *foo* will also accept an instance of a subclass of foo. For example, the *address_space* object has a method that takes a *memory_object* and maps it in the address space. The same method will also accept *file* and *frame_buffer* objects as long as they inherit from the memory_object interface.

The Spring kernel supports basic cross domain invocations and threads, low-level machine-dependent handling, as well as basic virtual memory support for memory mapping and physical memory management. A Spring kernel does not know about other Spring kernels—all remote invocations are handled by a *network proxy* server. In addition, the virtual memory system depends on external pagers to handle storage and network coherency.

A typical Spring node runs several servers in addition to the kernel (Figure 1). These include the domain manager and the virtual memory manager; a name server; a file server (that also acts as a default system pager); a linker domain that is responsible for managing and caching dynamically linked libraries; a network proxy that handles remote invocations; and a tty server that provides basic terminal handling as well as frame-buffer and mouse support.

The Spring file system supports cache coherent files. File objects inherit from the memory_object interface and therefore can be memory mapped. The file system uses the virtual memory system to provide data caching and uses the operations provided by the virtual memory manager to keep the data coherent. It consists of two types of file servers, one that stores data on local disks and handles cache coherency for local files, and another that utilizes virtual memory to provide caching for read and write operations and to cache file attributes for remote files. The file system also acts as the system pager.

1.3 Spring Naming

One particularly important component of the Spring architecture is the Spring naming model. In this section we describe the Spring naming model, and then in Section 4 we describe how we emulate the UNIX file system naming model on top of the Spring model.

The Spring naming service allows any object to be associated with any name. A name-to-object association is called a *name binding*. Each name binding is stored in a context. A *context* is an object that contains a set of name bindings in which each name is unique. An example of a context is a UNIX file directory. An object can be bound to several different names in possibly several different contexts at the same time.

Since a context is like any other object, it can also be bound to a name is some context. By binding contexts we can create a *naming graph*—a directed graph with nodes and labelled edges where the nodes with outgoing edges are contexts. The UNIX file system is a naming graph that is frequently restricted to a tree. We can use more complex names for referring to an object in a naming graph. Given a context in some naming graph, we can use a sequence of names to refer to an object; the sequence of names defines a path in the naming graph to navigate the resolution process. Such a sequence of names is called a *compound name*. UNIX path names are an example of compound names.

Each domain has a context object that implements the per-domain name space. Each per-domain name space shares a set of bindings with other domains. Thus all domains have part of their name space in common, but they can also customize their name space as appropriate. Our naming system is based on the architecture described in [2] and shares the per-process view feature of the Plan-9 naming system [3].

2 Design Goals

We started this effort to provide a Unix subsystem with several goals in mind:

- **No modifications to Spring.** Spring was designed as an open extensible system. A major goal was to implement UNIX using existing Spring primitives and services without modifying the base system.
- **Support for dynamically linked executables.** Dynamically-linked executables that run on SunOS 4.1 should run without modifications on our system.
- **Security.** Applications must not be able to violate UNIX protections.
- **Interoperability.** Interoperability between native Spring applications and UNIX programs and libraries is a design goal. In particular, Spring applications should be able to use UNIX libraries (e.g. Xlib), Spring applications should be able to start UNIX programs, and UNIX programs should be able to exec Spring applications.
- **Performance.** Degradation in performance due to our UNIX subsystem should be minimal. The performance of applications should be a function of the underlying Spring software and hardware, and there should be a minimal performance penalty imposed by the UNIX subsystem.

Providing a complete implementation of UNIX and support for statically linked UNIX binaries were not goals of this project. We felt that we had neither the resources nor the need for such functionality. It is worth noting, however, that it is possible to provide complete UNIX support, including running statically linked binaries with our design (see §7).

3 Overall architecture/design overview

In implementing UNIX on Spring, we wanted to use the services already provided by the underlying system. Spring provides a powerful naming architecture, a distributed coherent file system, a flexible virtual memory system and support for several devices. We did not want to rewrite any of these functions. Moreover, we wanted the resulting Unix subsystem to be “clean” and free from copyright restrictions.

Figure 1. Major system components of a Spring node

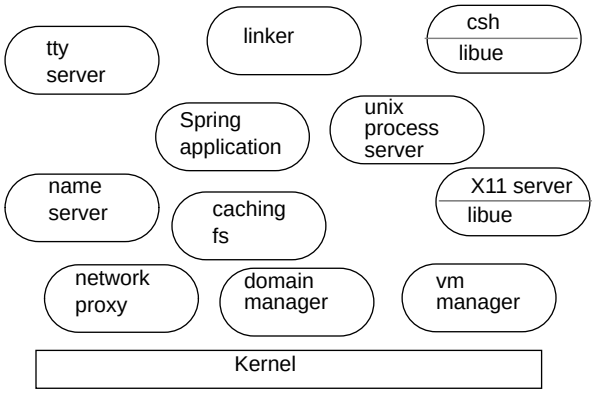
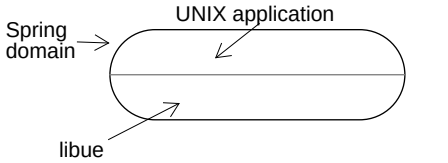


Figure 2. UNIX application on Spring



dynamically-linked with each UNIX application, contains:

- stubs for man (2) system calls
- a list of fd->object translations
- *unix_process* object
- a helper thread to handle signal delivery
- *libc* except for man (2) system calls

The implementation consists of two components: a library (*libue.so*) that is dynamically linked with each UNIX binary, and a set of UNIX-specific services exported via Spring objects implemented by a server domain (*UNIX process server*). The main criterion used to decide whether a certain function belongs to *libue.so* or the server is simple: as long as security is not compromised a function belongs in *libue.so*. The UNIX process server on the other hand implements functions that are not part of the Spring base system and which cannot reside in *libue.so* due to security reasons.

3.1 libue.so

The *libue.so* library encapsulates some of the functionality that normally resides in a monolithic UNIX kernel. In particular, it delivers signals forwarded by the UNIX process server (§4.3), and keeps track of the association between

UNIX file descriptor numbers (fd's) and Spring objects. It also contains a *helper_thread* which is used in delivering signals and in program start-up (§4).

The library maintains a data structure called the *fd_table* that consists of an array indexed by fd numbers returned by the open (2) call. Each element of the array contains a pointer to an object that is a subclass of the C++ class *descriptor*. This class defines virtual methods for reading, writing, stating, selecting, asynchronous IO, and IO controls. The implementation of the *descriptor* base class provides generic implementations for these methods that are sufficient for most subclasses. Subclasses of *descriptor* can override this generic support by defining their own implementations of the appropriate virtual methods. Subclasses of descriptor are listed in Table 1.

TABLE 1. Descriptors maintained by *libue.so*

descriptor subclass	Spring object encapsulated
file_descriptor	file
tty_descriptor	tty
pipe_descriptor	unix_pipe
pty_descriptor	slave_pty
fb_descriptor	frame_buffer
io_descriptor	io.sequential_io
kbd_descriptor	keyboard
ms_descriptor	mouse
socket_descriptor	unix_socket

For each man (2) system call, we implemented a library stub. In general, there are three kinds of calls:

1. Calls that simply take as an argument an fd, parse any passed flags, and invoke a Spring service (e.g. read (2), write (2) and mmap (2)). Most of file system and virtual memory operations fall in this category.
2. Similar to (1) but eventually call out to a UNIX-specific service in the Unix process server. Examples include pipe (2) and kill (2).
3. Calls that change the local state without calling out to any other domain. dup (2), some fcntl (2) and many signal handling calls fall into this category (main exceptions are kill (2) and killpg (2)).

We do not change *libc* or any other library. Instead when a program is exec'd (§4.2), *libue.so* is dynamically linked with the application image in place of *libc*. *libue.so* contains most of *libc* except that we remove the man (2) system calls from *libc* and substitute our stubs instead.

3.2 UNIX process server

TABLE 2. UNIX-specific objects

object	inherits from	example methods
unix_process	—	get_pid(), get_socket()
unix_pipe	io.sequential_io	read(), write()
unix_socket	io.sequential_io	connect(), read()
master_pty	io.sequential_io	start_output()
slave_pty	tty.tty	flush_output()

The main functions of the UNIX process server are to maintain the parent-child relationship among processes, to keep track of process and group ids, to provide sockets and pipes, and to forward signals. The objects that the server implements that are used to provide this functionality are listed in table 2. This section describes these objects and their implementation.

The UNIX process server implements one *unix_process* object for each UNIX domain. This object is passed to each domain as part of the fork() operation (see §4.1). The *unix_process* object represents the identity of each UNIX process and encapsulates its process id, user id, and the resources held by the process. When a call arrives on this object, the server knows which process made the call and proceeds accordingly. For example, if the call is a *send_signal*() method, the server can decide whether or not the caller has the permission to send a signal to the destination process. Similarly, if the call allocates a socket or a tty, the server associates the allocated resource with the calling process. Methods on this object fall into four categories: methods to get/set ids of process/parent/group; methods for sending and handling signals (§4.5); process control methods (fork, wait and exit; §4.1); and methods to obtain sockets, pipes, and ptys (see below).

The UNIX process server implements one *unix_pipe* object for each UNIX pipe in the system. *libue.so* obtains unix_pipe objects from the UNIX process server by invoking the get_pipe() method on its *unix_process* object. *Unix_pipe* inherits from *io.sequential_io* and does not add any more methods. (The Spring interface *io.sequential_io* provides methods to read and write a sequential stream). In the current implementation, data read and written to

pipes pass through the UNIX_server (see §7 for a possible alternative implementation).

Sockets are implemented by the UNIX process server via *unix_socket* objects. There is one *unix_socket* object for each socket in the system. These objects inherit from the Spring io.sequential_io interface and add several socket-specific methods. Socket objects are obtained by calling the get_socket() method on the *unix_process* object. Local connections go through the UNIX process server, while remote connections go through the network proxy. The current implementation supports SOCK_STREAM and SOCK_DGRAM types in PF_UNIX and PF_INET domains. Sockets and pipes share the same underlying implementation.

Pseudo ttys are implemented with the *master_pty* and *slave_pty* objects. The *master_pty* object provides the master side of a pty. This object inherits from the Spring io.sequential_io interface and adds methods such as stop_output and enable_packet_mode that are required to implement the semantics of a UNIX master pty. The *slave_pty* object provides the slave side of a pty. It acts just like a tty so it inherits from the Spring interface tty.tty. Master and slave ptys are obtained by *libue.so* from the *unix_process* object methods get_master_pty() and get_slave_pty() respectively.

4 Implementation of major system components

4.1 Fork

Most of the work of forking a domain is done within *libue.so* but some help is required from the UNIX process server. Note that since our current implementation is based on SunOS 4.x, we assume a single-threaded UNIX application. We refer to this thread as the *main_thread*. Forking a UNIX domain on Spring goes through the following steps:

1. The *main_thread* which invoked the *fork* system call goes to sleep after waking up the *helper_thread*.
2. The *helper_thread* wakes up, saves the current register state of the *main_thread* in the static structure *fork_regs*, creates a new domain, and contacts the UNIX process server to inform it that this domain is forking.
3. The UNIX process server makes a copy-on-write copy of the parent domain's memory into the child domain and returns a *unix_process* object for the child.
4. The *helper_thread* packages up the file descriptors, invokes the child with these file descriptors and the child's *unix_process* object, and wakes up the *main_thread.*
5. The *main_thread* wakes up and returns from the fork system call.

The newly created child domain begins executing in the start-up code in *libue.so*. The thread that is executing in this start-up code is the child's *main_thread.* The start-up sequence for a forked child is the following:

1. The child's *main_thread* unmarshals the file descriptors and the *unix_process* object, creates the *helper_-thread*, creates a *signal_handler* object (see §4.3) and passes it to the UNIX process server via the *unix_process* object, and does other miscellaneous initialization.
2. The *main_thread* wakes up the helper thread and then goes to sleep.
3. The *helper_thread* restores the *main_thread's* registers from the *fork_regs* structure where they were saved by the parent before its address space was copied and resumes the *main_thread.*
4. The *main_thread* wakes up and returns 0 from the fork system call.

4.2 Exec

Execing a new domain can be done entirely within *libue.so*. Our current implementation of *exec* is simple but not as efficient as possible (see §7 for a discussion of more efficient ways of implementing *exec*). Execing a new UNIX domain is done by creating a new domain, initializing its address space, dynamically linking the program image (more about this later), packaging up the current domain's file descriptors, and then invoking the new domain. Once the new domain is invoked, the domain that performed the exec is destroyed. When the newly exec'd domain begins execution it merely unmarshals the file descriptors and its *unix_process* object, creates the *helper_thread* and the *signal_handler* object, registers the *signal_handler* object with the UNIX process server, and then calls the main program.

Creating a new UNIX domain during exec involves dynamically linking together the new image. In Spring

there is a separate domain that performs dynamic linking. When a UNIX domain execs it dynamically links the new image by calling the dynamic linker domain which returns a set of <*memory_object, address, length*> tuples for the new image. One of these memory objects will be *libue.so* which was linked in place of *libc.so*[1]. These memory objects are then mapped into the new domain's address space at the given address for the given length. These memory objects along with memory objects for stacks and heap comprise the UNIX domain's address space.

Unfortunately it is not sufficient to merely replace *libc.so* with *libue.so.* The reason is that the standard UNIX start-up code in drt0.o that is linked in with each UNIX binary contains system call traps to dynamically link the image on UNIX. We need to replace this start-up code with Spring UNIX emulation start-up code. We do this by inserting a special *startup.so* shared library as the first dynamic library to be linked into the image; this requires special support from our dynamic linker domain[2]. This *startup.so* contains the normal Spring crt0.o and drt0.o code with some additions for UNIX emulation. The final step that this special start-up code does is call into an initialization function in *libue.so* which does things such as unmarshal the file descriptors. Thus a UNIX emulation domain is not started at the entry point given in the binary but rather at an entry point in the special *startup.so.*

Our implementation of *exec* has to be able to start native Spring domains as well as UNIX domains. In order to do this we have to know whether a program that we are starting was compiled for Spring or UNIX. We make this possible by putting a magic number right after the a.out header in each program binary that is compiled for Spring. If a program binary has this magic number then we realize that it is a Spring domain.

We start Spring domains from UNIX emulation by invoking the *start_spring_domain* method on the current process's *unix_process* object. We have to involve the UNIX server because we need someone to deal with UNIX signals and Spring domains don't understand UNIX signals. Once the *start_spring_domain* method returns, the domain that invoked it destroys itself because it is no longer needed.

The implementation of the *start_spring_domain* method on the UNIX server starts the Spring domain running and records the fact that the new domain is a Spring domain. If any signals are sent to the Spring domain the UNIX server takes the default action. For example, if a SIGINT is sent, then the UNIX server will kill the Spring domain and if SIGTSTP is sent then the UNIX server will stop the Spring domain. Thus Spring domains can be controlled from their UNIX parent just like normal UNIX domains.

4.3 Starting a UNIX Process from Spring

The previous section discussed how we start Spring domains from UNIX domains and UNIX domains from UNIX domains, but we have not discussed how we start UNIX domains from Spring domains. In our initial implementation we had a special program called *unix_init* that could be started from Spring. This program would start a *csh* as the first UNIX program and then other UNIX programs could be started from the *csh.* However, in order to get full interoperability between UNIX programs and native Spring programs we decided that it was desirable to be able to start any UNIX program from any Spring domain.

We use the magic number described in the previous section to help us start UNIX programs from Spring domains. The standard Spring library code that is responsible for starting new domains looks at the magic number of the program that it is starting. If it doesn't have the magic number, then it is assumed to be a UNIX program. In this case the program is linked with the special *startup.so* shared library at the front. Once the program is linked it is started like any other Spring domain. Thus the only special support that we have in the standard Spring library for UNIX emulation is a couple of lines of code in the routine that starts new domains.

When a UNIX domain that was started from Spring begins running, the start-up code in *libue.so* discovers by looking at its environment that it was started from a Spring domain instead of a UNIX domain. Once it discovers this it performs all necessary initialization to turn this domain into a true UNIX domain. This includes doing things such as

1. *Libc.so* is replaced by *libue.so* by merely having a symbolic link from *libc.so* to *libue.so.* Thus when the dynamic linker tries to link *libc.so* it will end up actually linking *libue.so.*

2. The only special support for UNIX emulation domains is that the Spring dynamic linker allows an extra shared library to be inserted at the front of the list of shared libraries linked in with an image. The dynamic linker itself knows nothing about UNIX emulation, it just knows how to handle an extra shared library.

contacting the UNIX process server and informing it of the new domains existence.

4.4 File Operations

The Spring base system supports file system objects and operations that are analogous to the UNIX file system. Thus, it is easy to emulate basic file system operations such as read, write, and stat. The main complexity with emulating the file system calls are handling naming issues, selecting, and asynchronous IO.

4.4.1 Basic Operations

As we mentioned in §3.1 *libue.so* maintains an *fd_table* that contains one entry for each UNIX file descriptor. Each of these entries points to an object that is a subclass of *descriptor*. Entries are added to this table by UNIX system calls such as *open*, *pipe*, and *dup*. When one of the basic operations on a UNIX file descriptor such as read, write, or fstat is invoked, the appropriate method on the descriptor object pointed to by the given *fd_table* entry is invoked. For example if the read system call is made with file descriptor *fd*, the *read* method on the descriptor object pointed to by *fd_table[fd]* is invoked.

4.4.2 Naming

The Spring naming model and the UNIX file system naming model differ in two important ways. One basic difference is that whereas the UNIX file system can only name files, directories, and devices, the Spring naming system can name all types of objects. Thus in order to allow UNIX programs to live in the Spring world we have to decide if an object being resolved is of an acceptable type to UNIX. All operations except for *open* will work on any type of object. However, *open* will currently work only on a subset of Spring objects. Currently we use a simple policy for determining if an object is acceptable to *open*:

- If the name of the object begins with “/dev” then we discern its type from its name (e.g. “/dev/mouse” refers to an object of type *mouse*) and get the desired object from the Spring name space. If we can find the corresponding Spring object, then the object is acceptable.
- Otherwise, unless the name resolves to a Spring *context* object or a Spring *file* object, the object is deemed unacceptable to a UNIX program.

A more general solution to this problem that will allow other types of objects to be accessed by UNIX programs is discussed in §7.

The other basic difference between UNIX and Spring naming is that Spring naming model does not support “..”. The reason is that a Spring context can be bound in any number of contexts so there is no notion of a “parent” directory like there is in UNIX. In order to handle the lack of support for “..” in the Spring naming system we keep the current directory as an absolute path name. Thus all the *chdir* system call does is change the path name that is kept by *libue.so*. When we encounter a relative path name in a system call we merely append this path name to the current directory path name. If a path name has any “..” entries in it, we modify the path name to remove these entries. For example if the current directory were “/foo/bar” and we were given the relative name “../lah” we would produce the absolute path name “/foo/lah”.

The disadvantage of keeping the current directory as an absolute path name is that we don’t have the same semantics as UNIX when someone changes the current working directory through a symbolic link. In UNIX “..” will go up towards the real root on the current file system, but in our world “..” will just take the last component off the current directory string. In practice we haven’t found this to be a problem.

4.4.3 Select

The *select* system call is implemented entirely in *libue.so* using Spring threads. We use threads for two purposes: waiting for a descriptor to become ready and for time-outs. When a user invokes select, we poll all the descriptors on which they are selecting to see if any are ready. If none are ready then we create a thread for each descriptor and have this thread wait until the descriptor’s Spring object is ready. When a thread returns from the wait it marks the descriptor as ready. If a user invokes *select* repeatedly, we only start threads on those descriptors for which there is not already a thread waiting.

We also use a thread for time-outs. We have one time-out thread for each domain. Before *select* goes to sleep waiting for a descriptor to become ready the time-out thread is made to go to sleep for the given time-out value. If it wakes up, it wakes up the sleeping thread that is doing the select.

4.4.4 Asynchronous IO

Asynchronous IO is implemented using callback objects. When a user puts a descriptor in asynchronous IO mode a callback object that is implemented by *libue.so* is installed with the implementor of the descriptor's Spring object. When the object becomes ready, the object manager invokes the callback object, *libue.so* handles the callback and sends a SIGIO signal to the current domain.

4.5 Signals

There are two types of signal system calls: those that send signals (e.g. *kill*) and those that modify the process's signal state (e.g. *sigsetmask*). In our implementation of UNIX we are able to handle the second type of system call locally without crossing into a different domain. Thus most signal system calls which would have required a kernel trap in standard UNIX are merely procedure calls in Spring.

The signal calls that send signals obviously cannot be done without leaving the current domain. There are two parts to the signal mechanism: requesting that a signal be sent to a process and handling the signal request at the signalled process. Requesting that a signal be sent involves the UNIX process server and handling the signal involves the signalled domain.

4.5.1 At the UNIX process server

The *kill* call invokes the process' *unix_process* object requesting that a signal be delivered to a particular process. The UNIX process server checks that the sending process can signal the destination process and then forwards the signal request to the destination process by invoking the *deliver_signal* method on the *signal_handler* object of the recipient.

The UNIX process server does not deliver SIGKILL to the destination process. When SIGKILL is received by the UNIX process server, the UNIX process server terminates the given process.

4.5.2 At the signalled process

When a signal arrives at the signalled process via the *deliver_signal* method on the *signal_handler* object implemented by the signalled domain, *libue.so* must determine what action to take. Possible actions are:

- Ignore the signal.
- Block the signal.
- Kill the process.
- Handle the signal.

The first three actions are easy. The interesting action is handling the signal. In order to handle signals we once again use the *helper_thread* that we used for *fork*. Note that the thread that is invoking the *signal_handler* object is a new thread which we will call the *signal_thread*. In order to deliver a signal the following steps are followed:

1. The *signal_thread* stops the *main_thread*, gets the *main_thread's* registers and stores them on its stack, and then continues the *main_thread* with modified registers so that it will begin executing in a routine that will call the signal handler.
2. The *main_thread* starts executing and then calls the signal handler.
3. When the signal handler returns, the main thread gets its old registers off of the stack, stores them in a global structure, wakes up the *helper_thread*, and goes to sleep.
4. The *helper_thread* wakes up and resumes the *main_thread* with its old registers.

We also deliver signals at other times. For example, if a signal that has been blocked is suddenly enabled, then it will be delivered immediately. This is done by having routines such as *sigsetmask* check the list of pending signals before they return. If they find a pending signal that needs to be delivered then they call the signal handler directly; there is no need to save state on the stack or use the *helper_thread* because the *main_thread* can just call the signal handler itself.

4.6 Virtual memory

UNIX virtual memory calls translate easily into calls on Spring's *address_space* object, and the UNIX process server is not involved in handling these calls. In general, the Spring virtual memory system is a super-set of UNIX virtual memory operations. An interesting exception is copy-on-write. The UNIX mmap(2) call with MAP_PRIVATE flag establishes a pseudo copy-on-write mapping, since any modifications made to the source memory object are visible to the process that establishes the private mapping as long as such writes are made before the private copy is written. Spring virtual memory on the other hand

provides true copy-on-write (modifications to the source memory object or to the private copy are not visible to the other). We implemented UNIX's MAP_PRIVATE using the copy-on-write implementation of Spring despite the difference in semantics and did not encounter any applications that cared about the difference in the map semantics.

5 Implementation Status

The implementation of the UNIX process server consists of 7000 lines of C++, while *libue.so* is implemented using 14500 lines of code. The effort took approximately 1 person-year to complete. Around 60% of SunOS 4.1 system calls are implemented. The main exceptions are *ptrace*, System V IPC and stream calls, and calls such as *sigstack*, *audit*, *mknod*, and *mount*. Despite these omissions we run most SunOS binaries without modifications, including X/NeWS, emacs, vi, csh, make, and various compilers.

As we described before, we used the Spring file system, virtual memory, dynamic linking, networking, and device drivers, and we did not have to re-implement any of these basic operating system services for the Unix system.

6 Related Work

In this section we compare our system to two other implementations of UNIX on kernelized systems: MACH 3.0 with the BSD4.3 Single Server [4] and CHORUS/MiX V.4 [5].

6.1 MACH 3.0 with the BSD4.3 Single Server

The BSD4.3 Single Server is a MACH task that contains an implementation of BSD4.3 [4]. An emulation library is loaded into the address space of UNIX processes (using virtual memory inheritance starting from /etc/init). A system call typically traps into the MACH kernel and is redirected back into the emulation library of the trapping process. The emulation library then sends a message to the BSD server which in turn executes the actual UNIX call. The BSD server shares two pages with each UNIX process, which are used to communicate some information between the server and its client.

Unlike the centralized BSD server, our UNIX process server only provides support for redirecting signals, keeps track of basic relationships among UNIX processes and provides support for pipes and local sockets. Moreover, it is not involved at all in implementing other UNIX calls, such as file system and virtual memory operations. As we mentioned before, we rely on native Spring servers for such things as the file system, virtual memory, dynamic linking, networking protocols, and naming support. However, unlike the BSD server implementation, our implementation does not currently support statically-linked executables (but see §7).

Our implementation does not rely on sharing memory between the UNIX process manager and UNIX processes; we believe that our implementation works better on NUMA machines. Our file system moreover provides consistent shared files across the network, and in general all the servers in our system can be located on more than one machine.

6.2 CHORUS/MiX V.4

MiX V.4 is a subsystem built on top of the CHORUS kernel [5]. MiX V.4 is composed of a set of servers that communicate through CHORUS IPC. The most important MiX V.4 server is the Process Manager (PM) through which client program UNIX calls are directed. Other servers include the File Manager (FM) and the Streams Manager (StM).

The CHORUS implementation is perhaps closer to ours in that the implementation of the various UNIX functionality is split among several CHORUS servers. An important difference, however, is that unlike our implementation all UNIX process calls in MiX V.4 have to pass through the PM on their way to their respective servers. In addition, the MiX V.4 implementation is tuned for running the various MiX servers in the supervisor address space [6] (although MiX V.4 servers are fully independent and can run in independent user spaces). We do not plan on moving our servers into supervisor space.

We share with MiX V.4 the support for network-wide shared files and the general distributed nature of the implementation. However, we only implement a subset of SunOS 4.1 whereas MiX V.4 is a complete implementation of SVR4.

7 Conclusions & Future work

We implemented a UNIX subsystem on top of an non-UNIX object-oriented operating system. As a result we were able to run a large number of existing applications on Spring. The implementation showed the flexibility of our system since we were able to achieve our goals without changing Spring. The implementation exercised the underlying system and forced us to complete some missing functionality of Spring.

We believe that a fundamental reason for our successful effort was the decision to provide an implementation at the man (2) system calls without rewriting any UNIX libraries. In doing so, we confined our effort to a (relatively) well-defined interface that *libc* and other libraries used.

There are several ways in which we can extend this work:

- **Implement the rest of the system calls.** SunOS provides a rich set of system calls. Although we only implemented a subset of the system calls, we were able to run most programs. As we gain more experience with the system we may add some of the missing functionality.
- **Provide SunOS 5.0 multi-threaded application interfaces**. The current implementation is tailored toward supporting SunOS 4.1 calls and libraries. Our work was developed in parallel with SunOS 5.0. which is based on SVR4 and provides a multi-thread application architecture [1]. Now that the 5.0 work is done we plan to port to its interfaces. We do not expect that this will be difficult since the Spring system and UNIX process server are already multi-threaded, and so is most of *libue.so*. Modifications will mainly be the extensions made in SunOS 5.0 to the signal model [1].
- **Handle statically-linked binaries.** The current implementation cannot run statically linked executables. For our purposes, we believe that it is not worth the effort to provide binary compatibility as most UNIX applications are dynamically linked. Moreover we believe that the use of dynamic linking will increase in the future. Our architecture does not preclude providing such functionality, however. Spring provides the ability to field domain traps and convert the traps into invocations on callback objects. One can provide support for statically-linked binaries by establishing a callback object on each UNIX domain (where the implementation of the callback object resides in the UNIX process itself). The implementation can then field the callbacks, decode the trap information and then use the *same libue.so* code to execute the calls.
- **Move pipes and pty implementation out of the UNIX process server.** Currently, obtaining pipes and ptys, as well as moving data through them, is done through the UNIX process server. We can get better performance by separating the functionality of setting up the connection from data movement. Such an implementation would use the UNIX process server for setting up the initial connection, but would copy the data directly between UNIX processes.
- **More efficient exec.** The current implementation of exec requires that a new domain be created whenever a process execs. Another option would be to do the exec in place; that is, replace the current domain's address space with the exec'd domains address space. This would require that a portion of code always be in each domain that can be used for this purpose.
- **Allow access to non-UNIX type objects.** Currently, UNIX domains can only access those types of objects that exist on UNIX. For example, if a stream object that were bound somewhere in the name space were opened by a UNIX domain and it wasn't a file the open would fail. We should be able to allow UNIX domains access to generic Spring objects as long as they support the io.sequential_io Spring interface.
- **Extend UNIX semantics**. An interesting opportunity made possible by the Spring system is to extend UNIX semantics to a distributed system. New functionality such as remote fork and exec operations, and network-wide coherent mapped memory can be added without much additional effort. Spring object invocation is location-independent, and all Spring services are distributed in nature. The ability to share memory and files across the network in a coherent fashion is already provided by Spring virtual memory and file systems.

Acknowledgments

We would like to acknowledge the efforts of Graham Hamilton and Sanjay Radia in helping to get the UNIX system running on Spring. We would like to also acknowledge the efforts of all Spring contributors whose work made this project a success.

8 References

[1] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, *SunOS Multi-thread Architecture*, USENIX Winter 1991, pp. 65-79.

[2] Sanjay Radia, “Names, Contexts, and Closure Mechanisms in Distributed Computing Environments”, Ph.D. thesis, Technical report UW/ICR 90-01, Department of Computer Science, University of Waterloo, 1989.

[3] R. Pike, D. Presotto, K. Thompson, and H. Trickey, “Plan 9 from Bell Labs”, Proceedings of the Summer 1990 UKUUG Conference, July 1990, pp. 1-9.

[4] D. Golub, R. W. Dean, A. Forin, and R. F. Rashid, “Unix as an Application Program”, Proceedings of Summer 19990 USENIX Conference, June 1990, pp. 87-95.

[5] N. Batlivala, et al., “Experience with SVR4 Over CHORUS”, Proceedings of USENIX Workshop on Micro-kernels and Other Kernel Architectures, April 1992, pp. 223-241.

[6] R. W. Dean and F. Armand, “Data Movement in Kernelized Systems”, Proceedings of USENIX Workshop on Micro-kernels and Other Kernel Architectures, April 1992, pp. 243-261.