An Overview of the NetWare Operating System

Drew Major
Greg Minshall
Kyle Powell

Novell, Inc.

Abstract

The NetWare operating system is designed specifically to provide service to clients over a computer network. This design has resulted in a system that differs in several respects from more general-purpose operating systems. In addition to highlighting the design decisions that have led to these differences, this paper provides an overview of the NetWare operating system, with a detailed description of its kernel and its software-based approach to fault tolerance.

1. Introduction

The NetWare operating system (NetWare OS) was originally designed in 1982-83 and has had a number of major changes over the intervening ten years, including converting the system from a Motorola 68000-based system to one based on the Intel 80x86 architecture. The most recent re-write of the NetWare OS, which occurred four years ago, resulted in an "open" system, in the sense of one in which independently developed programs could run. Major enhancements have occurred over the past two years, including the addition of an X.500-like directory system for the identification, location, and authentication of users and services. The philosophy has been to start as with as simple a design as possible and try to make it simpler as we gain experience and understand the problems better.

The NetWare OS provides a reasonably complete runtime environment for programs ranging from multiprotocol routers to file servers to database servers to utility programs, and so forth.

Because of the design tradeoffs made in the NetWare OS and the constraints those tradeoffs impose on the structure of programs developed to run on top of it, the NetWare OS is not suited to all applications. A NetWare program has available to it an interface as rich as that available to a program running on the Unix operating system [RITC78], but runs in an environment that is as intolerant of programming mistakes as is the Unix kernel.

The NetWare OS is designed specifically to provide good performance for a highly variable workload of service requests that are frequent, overlapping, mostly of short-duration, and received from multiple client systems on a computer network. In this capacity, the NetWare OS is an example of a network operating system. General-purpose operating systems, on the other hand, are designed to provide appropriate levels of service across a wide spectrum of differing workloads.

Just as a general-purpose operating system provides primitives required to act as a network operating system, the NetWare OS provides primitives required to develop most programs capable of running on a general-purpose operating system.

The differences between a general-purpose operating system and an operating system like the NetWare OS lie mostly in the handling of tradeoffs between ease of programming and fault tolerance, on the one hand, and execution speed, on the other hand.

General-purpose operating systems normally have a preemptive scheduler. This frees the programmer from involvement in the system's scheduler. This also increases fault tolerance by preventing a misbehaving program from monopolizing the system. The NetWare OS, on the other hand, is non-preemptive by design. This makes the system more responsive under load and simpler (and, therefore, faster) by reducing the number of concurrency issues, such as the locking of data structures, that programs need to manage.

General-purpose operating systems normally use hardware facilities to keep one program from incorrectly accessing storage locations of another program. This increases the fault tolerance in a system by isolating failures to separate regions of the system. The NetWare OS allows all programs to run without hardware protection. This reduces the time required to communicate between programs and between a program and the NetWare OS.

General-purpose, multi-user operating systems normally enforce a strong boundary between the kernel and user-level programs. Additionally, any given instance of a user-level program is often identified with exactly one end user for the purpose of access control. Both of these attributes contribute to the fault tolerance and security of such a general-purpose operating system. In contrast, the NetWare OS makes no formal distinction between kernel and user-level programs. This allows the functionality of the operating system to be easily extended by other programs. Moreover, the NetWare OS does not prevent any program from assuming the identity of any end user for access control purposes. In an environment in which one program may provide service to 1,000 network clients simultaneously, it is more efficient to allow the program to assume different users' identities as it services different requests.

The purpose of this paper is to describe the NetWare OS in greater detail. Section 2 introduces some terms used in the paper and provides an overview of the NetWare system. Section 3 provides more detail on the kernel of the NetWare OS, including a detailed description of the scheduler, one of the keys to the performance of the system. Section 4 describes a software approach to fault tolerance known as "server replication". We conclude with a brief discussion of some future work. For brevity, this paper does not describe the NetWare file system, network protocols and implementation, or other higher level services of the NetWare OS.

2. Overview and Terms

This section defines certain terms and gives an overview of NetWare and the NetWare OS.

2.1. General Terms

An <u>operating system</u> manages the basic resource sharing within a computer system. In addition, an operating system provides a suite of programming abstractions at levels somewhat higher than those provided by the hardware on which it runs [TANE92].

The <u>kernel</u> of an operating system manages the sharing of hardware resources of the computer system, such as the central processing unit (CPU) and main storage. It also provides for interrupt handling and dispatching, interprocess communication, as well as, possibly, the loading of programs into main storage.

The schedulable entities of a system are known as <u>threads</u>. A thread consists of a program counter, other hardware register contents, and other state information relevant to its execution environment.

While processing, we can distinguish between different system states as follows. <u>Interrupt time</u> is the state that occurs after the processor has accepted an interrupt (either hardware or software) and before the software has returned from the interrupt to the normal flow of control. Processing during interrupt time normally occurs on the stack that was active when the interrupt occurred, or on a special stack reserved for interrupt processing. <u>Process time</u> is the normal flow of processing in which the kernel, operating system, and application programs run. Processing during process time occurs on the stack of the currently running thread. There is a third state that we will call software interrupt time that may occur at the end of other kernel activity [LEFF89]. During this time, the kernel may decide to perform certain activities not directly related to the original reason for entering the kernel. During this time, processing occurs on a borrowed or reserved stack.

In the NetWare OS, software interrupt time occurs only when the scheduler has been entered (see section 3.3).

In general, execution is allowed to block only during process time.

An upcall [CLAR85] is a mechanism where a (typically) lower layer of a system can signal an event to a (typically) higher layer in the system by means of a procedure call. In the NetWare OS, these upcalls are known as Event Service Routines (ESRs) or callback routines.

Assume that the main storage of a system is divided into memory objects (e.g., bits or bytes). Assume that the state of execution of a thread at a given time is summarized in an execution context that determines the value of the program counter and other pieces of state. One element of a thread's execution context is the thread's rights to access the different memory objects of the system. These access rights are enforced by the hardware of the system, specifically the virtual memory system, storage keys, or other hardware specific components. In general, these access rights include "read", "execute", "read/write", and "none". (In the NetWare OS, only "read/write" and "none" are actually used.)

Using these concepts, we are in a position to define protection domain. Two memory objects are said to be in the same protection domain if and only if for each execution context X in the system, X has the same access rights to both memory objects. Similarly, two execution contexts are said to be in the same protection domain if and only if for each memory object X in the system, both execution contexts have the same access rights to X.

In the NetWare OS there is a privileged protection domain known as the kernel domain. An execution context whose protection domain is the kernel domain has "read/write" access to all memory objects in the system.

We say that a memory object is in the protection domain of a given execution context (except the kernel protection domain) if and only if the execution context has "read/write" access to the memory object. We say that a memory object is in the kernel protection domain if and only if it is not in any other protection domain in the system. Note that protection domains induce a partition on all the memory objects and on all the execution contexts in the system.

We then speak informally of a protection domain as an equivalence class of execution contexts and the related equivalence class of memory objects.

2.2. NetWare Overview and Terms

A NetWare system (also known as a "NetWare network" or "NetWare LAN") provides for the sharing of services over a network. The two principle components of the system are clients, which request service, and servers, which arbitrate the requests and provide the service.[1] Typical services include file systems, files, printer queues, printers, mail systems, and databases. The network can either be a single subnet, or a collection of subnets interconnected by routers.

In a NetWare system, clients run "off the shelf" operating systems such as DOS, Windows, OS/2, Unix, and the Macintosh operating system. Depending on the specific client and on the service being requested, it may not be necessary to add any additional software to the client in order to participate in a NetWare system. Macintosh and Unix clients, for example, typically come with built-in networked file system client software and thus don't require additional software to obtain file service in a NetWare system.

[1]In certain configurations, both the client and server may run on the same computer.

Servers in a NetWare system run the NetWare operating system, an operating system that has been designed specifically as a platform for running programs that provide data and communications services to client systems. The NetWare OS currently runs on Intel 80x86 systems; there is ongoing work to port the operating system to some of the current Reduced Instruction Set Computer (RISC) platforms.

A NetWare server consists of the NetWare OS kernel and a number of NetWare Loadable Modules (NLMs). An NLM is similar to an "a.out" file in the Unix operating system [RITC78]: it consists of code, data, relocation, and symbol information for a program. An NLM is generated by a linkage editor, which uses as input one or more object files as well as a control file. The object files are generated by assemblers and/or compilers from source files. The relocation information allows an NLM to be loaded at any address in the system. NLMs also contain unresolved external references that are resolved at load time (as described in section 3.1). Finally, NLMs may contain directives that allow them to export procedures for use by other NLMs (again, this is described in section 3.1). In this final sense, NLMs resemble shared libraries [GING89].

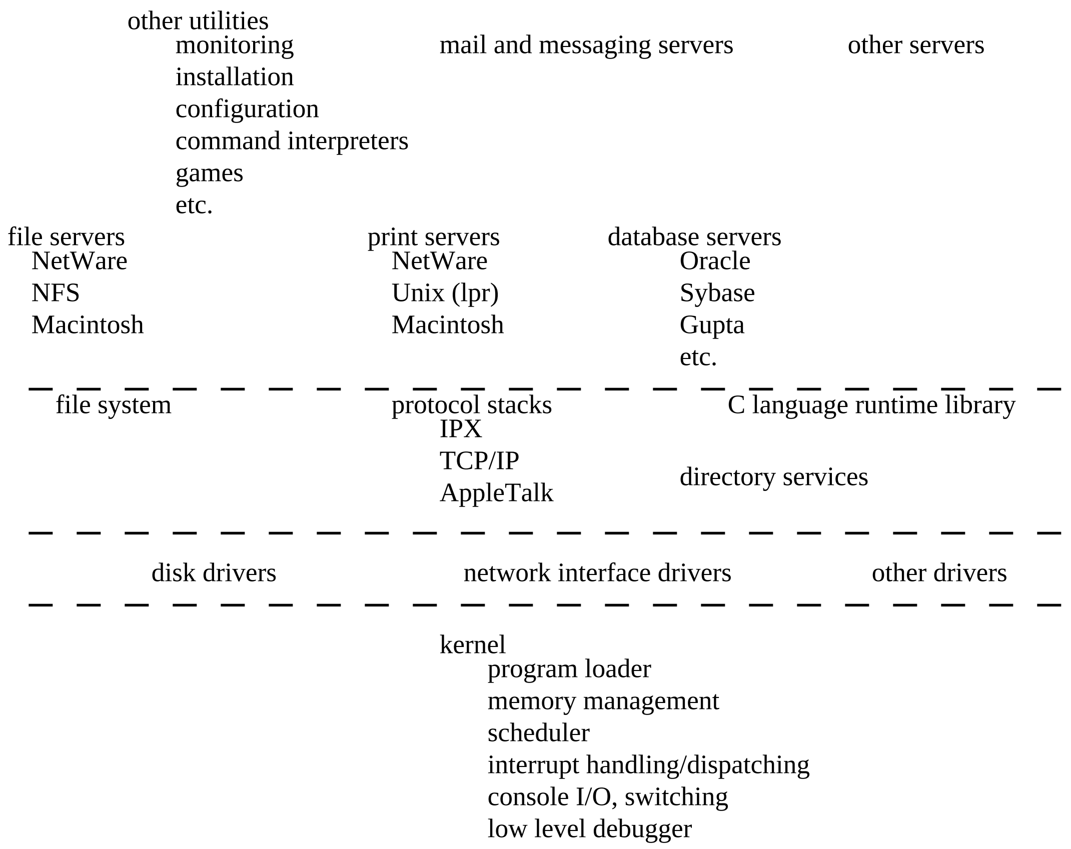
The performance requirements for NLM loading are less stringent than those for the loading of Unix a.out files because, in general, an NLM tends to run for a longer period of time than a Unix process and, therefore, does not need to be loaded and unloaded as frequently as a Unix a.out file.

The NetWare OS does not have the concept of a process. Instead, the basic units are the NLM, a protection domain, and a thread. In part, we do not use the term process because it has such different meanings in different systems. Depending on the system, there may be no memory sharing between processes, or there may be more than one process in the same memory space; there may be exactly one thread in a process, or there may be many threads in a process; there may be exactly one executable in a process, or there may be multiple executables in a process.

Although the NetWare OS has no concept of a process, we use the term "process time" (section 2.1) for what otherwise might be called "thread time".

Conceptually, there is no memory sharing between protection domains. In practice, the kernel domain has read/write access to other protection domains. Additionally, in the current version of the operating system shared memory is used in the implementation of procedure calls between protection domains (described in section 3.1).

The following figure illustrates the conceptual structure of a typical NetWare server. The dashed lines indicate the boundaries between the various layers in the system; these boundaries, like the lines that denote them, are not solid divisions because one NLM may provide service at several levels.



Outside of the kernel, each separate piece in the above figure consists of one or more NLMs. Thus, NLMs can be device drivers, higher level parts of the operating system, server processes, or utilities.

3. The NetWare Kernel

The NetWare kernel includes a scheduler, a memory manager, a primitive file system with which to access other pieces of the system during the process of booting up the system, and a loader that is used to load NLMs. In contemporary usage, the NetWare kernel might be known as a microkernel [GIEN90].

3.1. The Loader

Currently, all NLMs share the same address space, though this address space can be divided up into various protection domains.[2] By default, all NLMs are loaded into and run in the kernel domain. At the time an NLM is loaded, however, the user can specify a protection domain into which to load the NLM. All NLMs resident in a given protection domain have shared memory access to each other's memory. The kernel domain has read/write access to memory in all other protection domains.

The loader maintains a symbol table in order to resolve external references when loading NLMs. When the system first loads, this symbol table contains one entry for each procedure exported by the NetWare kernel.

At load time, any unresolved external references in an NLM will be bound using the symbol table. Any symbol left unbound after load time will cause the NLM load to fail. There are runtime facilities that an NLM can use to test

[2]In the current release of NetWare, only two protection domains are available. This is a restriction that we hope to lift.

the availability of a binding for a given symbol, to register for a notification in the event a given symbol changes its binding, and to determine the binding itself.

In addition to “importing” procedures, an NLM can “export” procedures – that is, make them available to other NLMs. Procedures exported by an NLM will be added to the loader's symbol table when that NLM is loaded and removed when the NLM is unloaded.

When the NLM importing a procedure and the NLM, or NetWare kernel, exporting the procedure are running in the same protection domain, a direct procedure call is made to invoke the procedure.

When an NLM contains a call to a procedure in an NLM that is loaded in a different protection domain, an intra-machine remote procedure call (RPC) [BIRR84] is made to invoke the procedure.

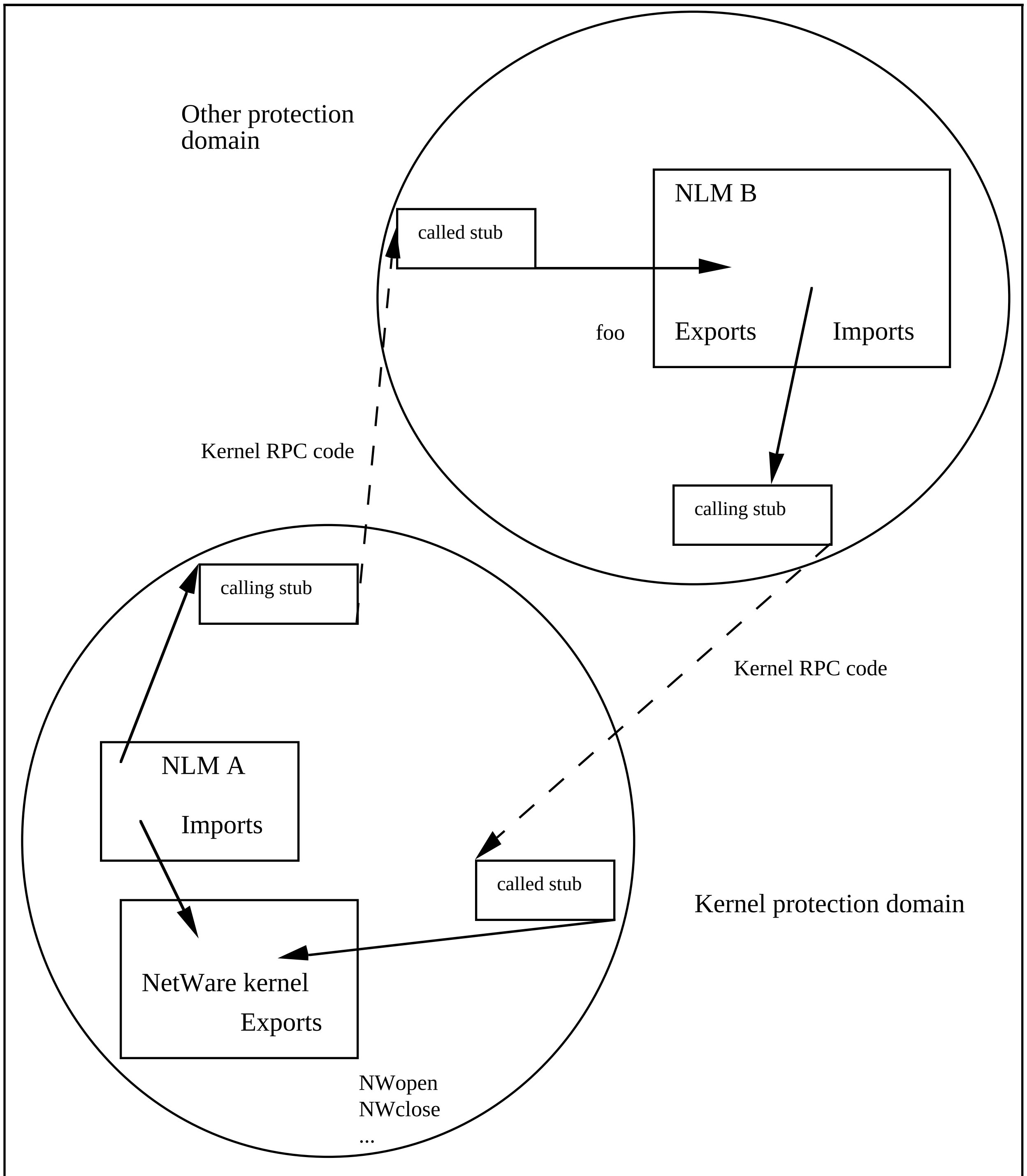
For example, assume that NLM “A” is loaded into a different protection domain than NLM “B”, but NLM “A” contains a call to routine “foo” in NLM “B” (see the following figure).

In a traditional implementation of RPC, a calling stub (user stub in [BIRR84]) linked into NLM A marshals the parameters of the call to foo, and then calls into the kernel to transfer the parameters and control to B. A called stub (server stub in [BIRR84]) linked into NLM B unmarshals the parameters and invokes foo. After foo returns, the called stub marshals any results and then calls into the kernel to return the results and control to the calling stub in A. The calling stub unmarshals the results and returns them to the call point in NLM A that originally called foo.

The NetWare implementation of RPC differs from the traditional RPC model in several respects. The major difference is that the calling and called stubs for exported procedures are automatically generated at load time by the NetWare kernel; they are not linked into either NLM. This isolates developers from knowledge of which procedure calls are direct and which are remote, and allows for increased flexibility in configuring the system. Second, the transfer of parameters and results is done by the kernel, not the calling or called stubs.[3] This process is driven by pseudo-code which describes the types of all the parameters to a given procedure. This gives the kernel the ability to make certain optimizations based on the types of the parameters and on the location of the protection domains involved. Third, when a pointer to a callback routine is passed as a parameter in an RPC, the necessary calling stub, called stub, and kernel data structures are generated at run time. This allows callback routines to be invoked across protection domains.

Load-time creation of RPC stubs and kernel data structures is possible because exported routines are described in an interface description language that allows a programmer to provide a full description of the parameters to each exported routine. These descriptions are similar to the “function prototypes” of ANSI C [HARB91] but also allow for the specification of other attributes of the parameters (in particular, whether the parameter is an input parameter, an output parameter, or is both an input and an output parameter). The NetWare interface description language is similar in function to that of the Common Object Request Broker Architecture [OMG91].

[3]The current implementation of the NetWare kernel makes the calling stack shared between the two protection domains and uses it to transfer the parameters to the called protection domain. Pointers in the parameters result in the kernel making the indicated memory shared between the calling and called protection domains.



In this figure, NLM A is importing one routine from the kernel and another ("foo") from NLM B. NLM B is also importing one routine from the kernel.

The solid lines indicate direct procedure calls. The dotted lines are cross domain procedure calls. The calling and called stubs are created at NLM load time by the NetWare OS loader.

The choice of which NLMs should share protection domains involves a tradeoff between performance and fault tolerance. The use of direct procedure calls gives the best performance, but exposes each NLM to failures occurring in the other NLM. Loading communicating NLMs in separate protection domains gives good fault isolation but incurs the performance overhead associated with the RPC mechanism. By basing the system on a procedure call interface, with load-time stub generation, the NetWare OS allows this tradeoff to be made comparatively late: at load time. In this sense, the NetWare OS is similar to Lipto as described in [DRUS92].

Exposing other interprocess communication mechanisms such as message passing or shared memory to the programmer forces a decision on the binding of program units to protection domains to be made at design time in order to get good performance. Message-passing systems [ACCE86] suffer performance problems when the communicating entities are in the same protection domain. Shared memory systems [LI89], on the other hand, suffer performance problems when the memory sharing is not provided by the hardware. By basing a design on either of these paradigms, future flexibility in system structure is reduced. Basing the NetWare OS on the procedure call paradigm allows the kernel to make use of direct procedure calls when the two communicating NLMs are in the same protection domain, to make use of shared memory to carry out the RPC when that is available, and, in the future, to use message passing when the two NLMs are not in the same address space.

3.2. The NLM Execution Environment Provided by the Kernel

When first loaded, an NLM is given one thread of control. It can then create as many additional threads as it needs. Alternatively, the NLM can destroy its only thread, which leaves the NLM existing as a shared library [GING89].

Each NLM has access to a virtual screen and keyboard. The kernel switches the physical system screen and keyboard (in unison) between the various virtual screens and keyboards in the system.

In the Unix operating system, file descriptors 0, 1, and 2 refer to standard input, standard output, and standard error output respectively [RITC78]. The NetWare OS, on the other hand, does not assign any special meaning to these file descriptors. However, the C runtime environment in NetWare does set up these descriptors (as well as allow for I/O redirection [RITC78]).

The NetWare OS does not provide a separation between user space and kernel space. All NLMs running on a server have the same security privileges (though they may be loaded in different protection domains), and all NLMs have access to the same set of importable procedures. For example, the initialization code in an NLM may be written using higher level programming abstractions, such as a standard C library [PLAU92], while the more performance-critical portions of the NLM may use programming interfaces more tightly coupled to the NetWare OS.

An NLM has associated with it certain resources such as main storage and open files. There are mechanisms by which an NLM can transfer ownership of these resources to other NLMs. By convention, an NLM is expected to release all resources it owns before exiting, as part of a philosophy of having developers aware of which resources they are using (in order to minimize resource usage with long-running NLMs). However, if an NLM exits without releasing all its resources, the resources it had owned are returned to the system, and the system console makes note of the fact that the NLM had not released all its resources.

As a thread executes, its program counter moves between NLMs. This may involve moving between protection domains, as an NLM calls routines in a protection domain external to it. Threads in the NetWare OS combine the Spring concepts of “thread” and “shuttle” [HAMI93].

The ability of threads in the NetWare OS to “follow” procedure calls into other NLMs and protection domains is similar to facilities in other systems ([JOHN93] [HAMI93] [FORD94] [FINL90]).

If a thread is blocked (because of a blocking call to the file system, say, or awaiting network I/O), it goes to sleep until it is unblocked. When a thread is unblocked, it is put on the run list until scheduled for execution (section 3.3 discusses the scheduler in more detail).

If an NLM running in a non-kernel protection domain terminates abnormally (because, for example, of a memory protection check), its protection domain is "quarantined". Because of the non-preemptive nature of the NetWare OS, exactly one thread is executing in the protection domain when it terminates, and that thread is suspended when the failure occurs.[4] Threads are not allowed to call into a quarantined domain; if a thread attempts to call into a quarantined domain, this same mechanism is invoked to suspend the calling thread and quarantine its domain. Previous RPC calls from a quarantined domain are allowed to return to the quarantined domain. Non-suspended threads running in a quarantined domain are allowed to call out from the domain as well as return from the domain.

An abnormal termination in the kernel protection domain is treated as a fatal error.

Because the NetWare OS scheduler is non-preemptive, it is up to each NLM to yield control of the CPU frequently in order that the system remain responsive. Thus, if an NLM has sections that execute for long periods of time without blocking, it will be necessary for the NLM to occasionally call a yield function (exported by the NetWare kernel and described in greater detail in section 3.3). A programmer, in deciding how often to call yield, will attempt increase the "signal to noise ratio" in the program, i.e., make sure that the cost of the yield call is amortized over a significant amount of program processing. This is especially true given that the vast majority of yield calls are, effectively, no-ops (because there is no other thread ready to run). The desire to process a long time between yields conflicts with the goal of having a very responsive system. For this reason, it is important to reduce the cost of doing a null yield in order to encourage the programmer to execute them more frequently. To this effect, in the current NetWare OS, the cost of executing a yield is less than 20 machine instructions if there is nothing else in the queue to run (not including the cost of possibly crossing a protection domain boundary). If there is other work to be done, it takes less than 100 instructions before the new thread is running.

In the case where the NLM calling a yield function is not in the kernel protection domain, an intra-machine RPC must be executed. This significantly increases the cost of the yield call. For example, in the current implementation on an Intel 80486 processor, such a call (with no parameters) adds approximately 50 machine instructions of overhead for the call and the same number for the return. Of each group of 50 machine instructions, however, 49 take approximately two machine cycles each; the other instruction takes approximately 100 machine cycles by itself.

In the future, it is possible to provide preemption in the NetWare kernel but to maintain a non-preemptive execution model within a protection domain. This means that a protection domain could be preempted by the kernel and another protection domain scheduled to run. However, any RPC calls into the preempted domain, or returns back into the preempted domain from previous RPC calls to other protection domains, would need to be blocked until the preempted domain has been rescheduled and itself issued a yield. If this extension to the NetWare kernel were to occur, then the yield calls in a given protection domain would be for sharing the CPU with other threads in that protection domain; other protection domains would not be affected by the frequency of yield calls. A side-effect of this would be to increase the speed of a yield call, because it would not need to cross a protection boundary.

Our experience is that the non-preemptive nature of the NetWare OS is both a blessing and a curse.[5] We end up with a more deterministic system with fewer of the concurrency issues that are so difficult in many systems. For

[4]There are ways in which a thread can arrange to "catch" such a failure, making the failure look like a non-local return. Depending on the mechanism used, the protection domain in which the failure occurred may or may not be quarantined.

[5]For example, the developers of some NLMs have assumed that file system accesses will block often enough to keep the developer from having to explicitly yield the CPU. Because of the fact that the NetWare file system performs so much caching, this assumption of NLMs quite often turns out to be false. In order to allow these NLMs to run but not monopolize the system, the NetWare file system will occasionally, but deterministically, perform a yield for an NLM, even if the file system operation did not need to block.

example, multi-threaded NLMs can avoid locking shared data structures if their use of these data structures does not explicitly yield or call a service that might yield or block.

Additionally, a non-preemptive system, in our opinion, offers significant performance advantages over more traditional preemptive systems. As an example, in the NetWare file system, a read request on a file that has byte range locks set runs through the linked list that makes up the locking structure for the file. In a preemptive operating system, this code section would have to be protected (by disabling interrupts or by locking the linked list). However, in the NetWare OS, the file system code is written with the understanding that there will be no preemption during the processing of the read request. Thus, the code path is both simpler and faster. In a non-preemptive system, the developer has more control of how often his or her program yields control of the system, and can use this control to simplify the program. Additionally, placing yield calls in a program (which can be done late in the program development cycle) has the added benefit of keeping the developer aware of how long the code paths actually are. The developer effort and mindset necessary to program in this environment require a certain amount of education for the developer. While committed to non-preemption, we continue to explore new tools and other means to ease the burden on the NLM developer.

3.3. The Scheduler

The NetWare OS was designed specifically for the purpose of providing service to networked clients. The performance of servicing network requests is very sensitive to the scheduling policy of the server operating system. The NetWare scheduler is therefore tuned to give high performance in this specific environment.

As mentioned above, the NetWare scheduler provides a non-preemptive scheduling paradigm for the execution of threads. A thread that has control of the machine will not have this control taken away from it unless one of the following occurs:

- it has called one of the yield calls (see below)
- it has blocked (awaiting disk I/O, for example)
- it has returned to the scheduler (for threads dispatched to service work objects – see below)

In particular, interrupts (including timer interrupts) do not cause a thread switch until, at least, the next yield call.

In order to be both non-preemptive and responsive, NLMs are explicitly required to offer to relinquish control of the CPU at regular intervals (not to exceed 150 milliseconds). They do this using one of three <u>yield</u> <u>functions</u>:

> ThreadSwitch() informs the scheduler that the thread is in a position to continue performing useful work if no other thread is ready to run. This is the normal form of yield for threads that are processing but want to allow other work to execute.
>
> ThreadSwitchWithDelay() delays the thread until at least a specific number of yields (from other threads) have been processed. This is quite often used by a thread when it is blocked on a spin lock [JONE80] in order to give the thread holding the lock time to release the lock.
>
> ThreadSwitchLowPriority() informs the scheduler that this thread has more work to do, but as a background activity.

Having different yield calls allows the scheduler to determine how to treat the yielding thread, i.e., in which <u>scheduling</u> <u>class</u> the thread belongs and the thread's parameters in that class. An alternative structure would have one yield call and separate calls that a thread could make to set its scheduling class. We are trying to keep threads as lightweight (in terms of state) as possible, so we prefer not to keep a per-thread variable detailing the scheduling class. Additionally, if there were such a state variable per thread, each thread would need to manage this state as it moves between different tasks in the system. This is particularly problematic when coupled with the “work objects” of the NetWare OS (see below), in which a thread may be involved over time in many different parts of the system.

Finally, if the scheduling class were part of a thread's state, we would increase the number of branches in the queuing logic, which may “break” the processor's pipeline. As it is, we know at the entry point to the specific yield routine on which queue we are going to place the thread, so the number of branches is reduced.

There are six basic structures the scheduler uses in scheduling the CPU:

> The delayed work to do list is an ordered (by *pollcount* – see below) list of routines to call, together with a parameter for each routine. Routines called from this list run at software interrupt time.
>
> The work to do list is a first-in, first-out (FIFO) list of routines to call, together with a parameter for each routine. Routines called from this list run at process time.
>
> The run list consists of those threads that are ready to run at any given time. This is a FIFO list. Threads run from this list run at process time.
>
> The hardware poll list is a list of routines to be called to check on the status of different hardware components of the system. In contrast to all the other lists mentioned here, entries on this list remain on the list when their associated routines are called (i.e., they must be explicitly removed). Routines called from this list run at software interrupt time.
>
> The low priority list consists of threads that are ready to run but have indicated that they should be run after everything else in the system has run. This list is also serviced in FIFO order. Threads run from this list run at process time.
>
> Pollcount is a monotonically increasing sequence number that is incremented by one for every yield call made.

ThreadSwitch() will cause the calling thread to be enqueued on the run list if control is taken away from the calling thread.

ThreadSwitchLowPriority() will cause the calling thread to be enqueued on the low priority list if control is taken away from the calling thread.

ThreadSwitchWithDelay() will cause an entry to be created and enqueued on the delayed work to do list.[6] This entry will have its *pollcount* value set to the current pollcount incremented by the value of a system parameter (the so-called “handicap” value, nominally set to 50). This entry will point at a routine that takes as a parameter a thread handle and enqueues that thread on the run queue.

Note that with the exception of the delayed work to do list, all the lists used by the scheduler have a constant insertion and removal time, helping make the scheduler efficient. The delayed work to do list has a variable insertion time, but the removal time is still constant.

There are separate calls to place an entry on the work to do list and on the delayed work to do list. Note that the entries on the work to do list run at process time while entries on the delayed work to do list run at software interrupt time and thus have a more restricted execution environment (for example, they cannot block).

The scheduler basically imposes the following priority on processing in the system (from highest to lowest):

> interrupts (these run at interrupt time)
>
> entries on the delayed work to do list that have a pollcount less than or equal to the current value of pollcount (these run at software interrupt time)

[6]In fact, the entry for the work to do list is created on the stack, so no memory allocation actually occurs.

entries in the work to do list (these run at process time)

threads on the run list[7] (these run at process time)

entries on the hardware poll list (these run at software interrupt time)

threads on the low priority list (these run at process time)

The reality is slightly more complex than the above list because, for example, the scheduler does not allow lower priority tasks to become starved because of higher priority activity. For example, threads on the low priority list are guaranteed to be scheduled at least eighteen times per second – the frequency of the system clock.

The delayed work to do list and the hardware poll list are mechanisms that allow us to trade off responsiveness for higher system capacity by setting certain hardware devices (such as network interface boards or serial line adapters) into a mode in which they do not normally interrupt the system, but rather are serviced periodically by system software. The hardware poll list is a regular, background, polling list. The delayed work to do list can be used to speed up polling a given hardware device when hardware or system loading makes this necessary.

Entries on the work to do list (described in more detail in the next section) are usually invoked to begin processing on a newly received request. Since requests frequently complete without yielding or blocking, calling these entries before processing the run list has the effect of favoring new requests over older, slightly longer running requests. If a request blocks or yields the CPU during its processing, it enters the category of "older, slightly longer running" and, thus, will be scheduled behind newly arriving requests.

The choice between preemption and non-preemption in scheduling is basically a trade off and is based in part on the predicted workload of the system. The NetWare scheduler is optimized for a non-CPU intensive workload. The code path for a typical file system request, for example, is short enough that it contains no yield calls (partly because the code paths do not need to deal with concurrency issues). Additionally, fast thread switch times (on the order of 100 machine instructions) help extend the space of those workloads that perform well in the NetWare OS. Certain CPU intensive workloads, with multiple threads contending for the CPU, might perform better in a preemptive environment.

3.3.1. Work Objects

Earlier versions of the NetWare OS had separate pools of threads dedicated to each different service offered by NetWare (e.g., DOS file service, Unix file service, Macintosh file service, DOS print service). Each pool was created by a particular NLM (in general). As described in the preceding section, the scheduler looks for active threads and schedules them to run. A thread is activated, for example, when an incoming network message arrived on a connection associated with the service provided by that thread.

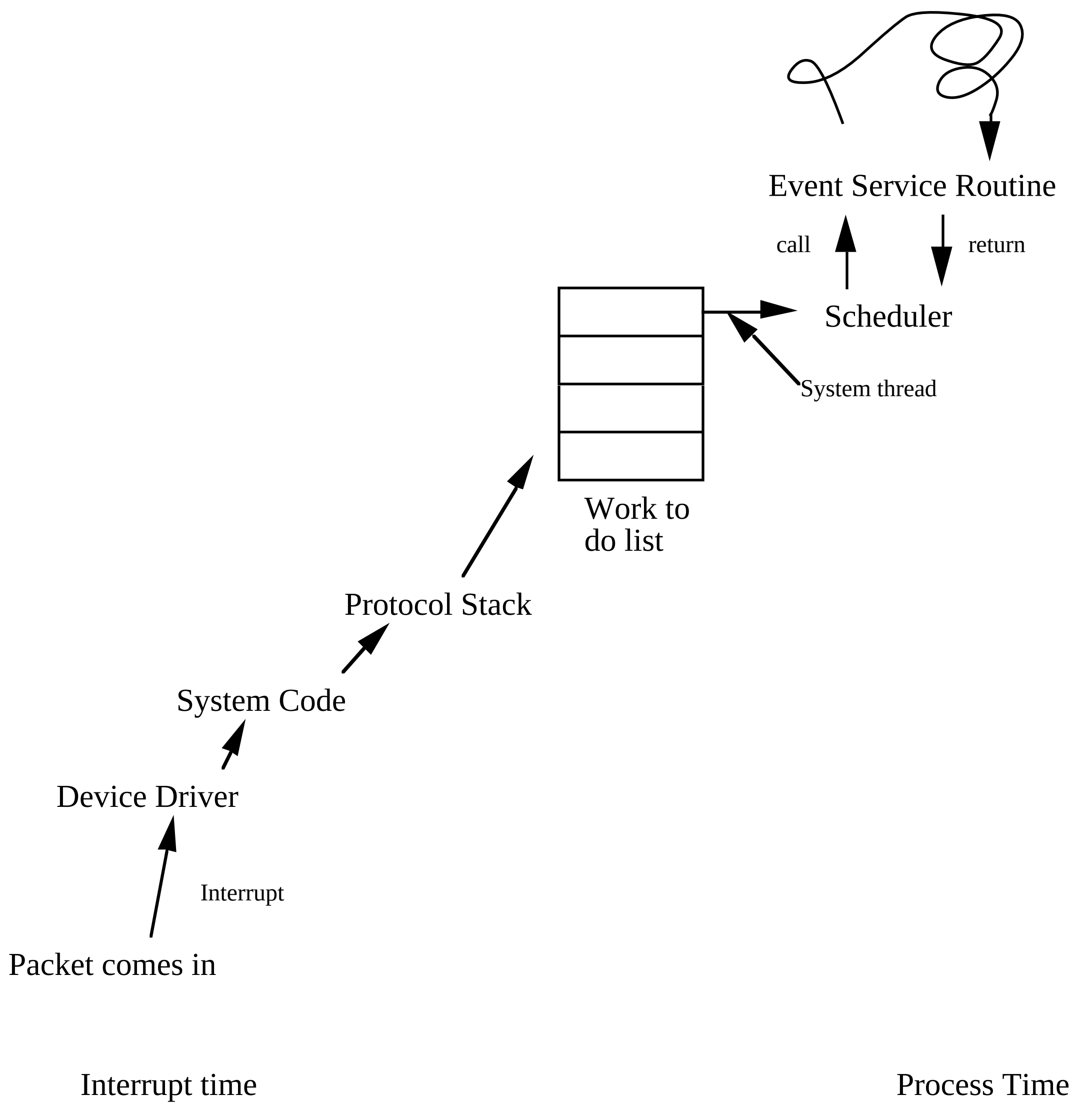
In this environment, each service was responsible for determining how many threads to create, heuristics for when to create new threads or terminate existing threads as the system load changed over time, etc. Implementing these functions, while not difficult, imposed an overhead on the programmer. Additionally, since each thread consumes a certain amount of real memory for such objects as stack space or control blocks, this scheme made inefficient use of memory.

In the most recent version of the NetWare kernel (4.0), a new paradigm for scheduling work has been introduced. This paradigm takes advantage of the fact that NetWare threads carry essentially no state while waiting for a request

[7]Note that threads that have called ThreadSwitchWithDelay() reside on the delayed work to do list but actually have a priority lower than all the threads on the run list. This is because when the entry on the delayed work to do list is actually executed, it will have the effect of enqueuing the thread at the end of the run list.

to service, and so are interchangeable. Thus any thread is often as good as any other thread in performing a service, as long as the call to the service carries the correct parameters.

As mentioned in the previous section, the work to do list consists of a list of tuples, known as “work objects”. Each work object contains the address of a procedure to call, and a parameter to pass to that procedure. As an example of how this works, consider the following figure: a received packet interrupts the system, causing the protocol stack to demultiplex the packet to a given connection. The protocol stack, in the example, builds a work object consisting of the address of an ESR that has been registered for this connection and the address of the incoming packet. The protocol stack then calls into the NetWare scheduler to enqueue the new work object on the work to do list, and then returns from the interrupt.



At process time (during a yield call, or when blocking another thread or because the system was idle when the interrupt occurred) the scheduler examines the work to do list before looking in the run list. In this case, the work to do list is not empty. If the thread running in the scheduler is an otherwise idle system thread (as is often the case), it will call the ESR associated with the first work object directly, with no context switch. Otherwise, a thread will be allocated from a pool of threads managed by the NetWare kernel and will call the ESR. In either case, the ESR will be passed a parameter (in the example, the address of the incoming packet).

The ESR may call other routines, block, and so forth, until it has finished the processing associated with the incoming packet and has returned. The ESR, or any other routine in the processing of the incoming request, may at its option install the extra state that a thread may have.[8] When the ESR returns, its thread runs the kernel scheduling code, possibly picking up and servicing another entry from the work to do list.

Note that in the above example the transition into the protocol stack could also have been called at process time by using work objects.

Moving away from "per-NLM threads" has allowed the system to be run with a much smaller number of worker ("daemon") threads, resulting in more efficient use of memory and increased productivity for NLM programmers.

3.4. Paging

The NetWare kernel does not support paging. It is not clear if the primitives supplied by the kernel today would be sufficient to provide paging via a so-called "external pager" [YOUN87]. The page hardware of the underlying system is, however, used to implement protection domains. Currently, the page hardware is also used to keep certain "per-thread" and "per-protection domain" variables at constant virtual addresses.

4. Server Replication

The NetWare OS has long supported fault tolerance approaches such as the mirroring of disk drives. Recently, however, we have developed a new technology, known as "System Fault Tolerance III" (SFT III) that allows for the mirroring of an entire server. In this configuration, two identical server hardware platforms act as one logical server to their clients. At any time, one of the servers acts as the primary and actually replies to requests from the clients. The second server acts as the secondary by tracking the state of the primary and staying ready to perform a failover (in which it takes over from the primary should the primary fail). No special hardware is involved in performing the mirroring and the possible failover.

4.1. Architecture

In order to accomplish this, we have restructured the system into a so-called "I/O engine" and a "mirrored server engine". The I/O engine contains all code that actually deals with hardware connected to the system, as well as controlling the execution of the mirrored server engine. The mirrored server engine, which actually can run any combination of correctly written, non-hardware-specific NLMs, is the piece that is mirrored between two different systems.

The interface between the I/O and mirrored server engines consists of an "event queue" of events passed from the I/O engine to the mirrored server engine, and a "request queue" moving from the mirrored server engine to the I/O engine. These two queues are the entire interface between the two engines. The mirrored server engine contains a copy of the NetWare kernel complete with scheduler, and so forth, but no code that actually touches any hardware beyond basic CPU registers and the storage allocated to that engine.

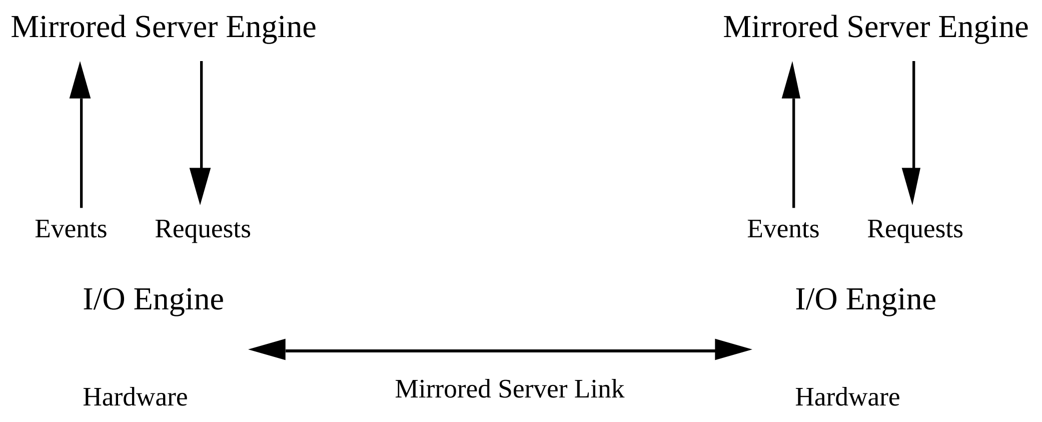
Additionally, the mirrored server engine contains a "virtual" network, with its own network numbers (in the various protocol address spaces), and protocol stacks connected to this virtual network. Note that the network numbers and node numbers on this virtual network are exactly the same for the two mirrored server engines.

The two systems to be mirrored are connected through a high-speed, point-to-point data link running a set of protocols specifically designed to support server replication. The link and the protocols run on the link are known as

[8]This state is related to the C language runtime environment provided by the operating system. Installation of this state takes 24 machine instructions in our Intel 80x86-based implementation.

the “mirrored server link” (MSL). The link itself can be any high-speed data link technology, configured in a point-to-point fashion.[9] The two systems to be mirrored are also connected via the shared internetwork.

Initially, only one of the systems runs, operating in an “un-mirrored” mode. When the second system comes up, the two systems connect over the MSL. The mirrored server engine on the primary server is suspended momentarily, and its entire memory image is transferred to the secondary, where it is loaded into memory, building an exact copy. Once the memory image and associated state such as register contents exist on both machines, both mirrored server engines are started up. The basic architecture is illustrated in the following diagram.



At the moment of starting, the state in both mirrored server engines is identical. Server mirroring works by keeping the state of the two mirrored server engines identical over time, so that if the primary dies (because of a hardware failure, say), the secondary engine will be able to take over (with almost no action required on the part of the clients[10]).

Four basic conditions have made this possible:

1) Networks are inherently unreliable, so any packets awaiting transmission at the primary at the time of failover will either be re-requested by the client or retransmitted by the mirrored server engine.

2) The mirrored server engine is a non-preemptive system, so the execution path inside the mirrored server engine does not require interrupts for its correct functioning.

3) We are able to “virtualize” time, so the actual wall clock time, which will be different at the two mirrored server engines at the same point of execution, is not seen by the mirrored server engines. Thus, the two engines will see the same time at the same points of execution, and thus will operate in exactly the same manner.

4) The event queue is the only input the mirrored server engine has from the rest of the world.

Note that hardware physically attached to a system that is down is unavailable. This includes disk drives. To make sure that we are able to continue file service during such an outage, we mirror the disk drives between the two

[9]A 10 megabit/second ethernet link between the two systems is minimally acceptable as an MSL, but for the initial synchronization, as well as to allow for increased distances between the primary and secondary servers, a more specialized, higher speed link is desirable.
[10]Some network routing protocols converge slowly in certain topologies. To increase the speed at which the clients start communicating with the old secondary/new primary, a message is sent to the clients causing them to reacquire their route to the mirrored server.

servers. There are other protocols for bringing the mirrored disk drives into synchronization, but those are beyond the scope of this paper.

4.2. Dynamics

Every event enqueued to the primary mirrored server engine is reliably transferred over the MSL to the secondary I/O engine to put on the queue for the secondary mirrored server. No other events are queued up for the secondary mirrored server.

When the mirrored server engine has completed all processing possible on its current event and needs new input, it will request a new event from the I/O engine. The two I/O engines maintain sequence numbers on the requests coming down from the mirrored server engines and cooperate to pass the identical events in response to the same request numbers. Thus, the mirrored server engines see the same set of events, and so behave like any state machines that start off in the same state and are given the same set of input stimuli – they make the same state transitions. In fact, the two mirrored server engines execute the exact same code path between events. The two systems are not in close lock step – one of the mirrored server engines will normally be ahead of the other engine. However, the two engines will be in exactly the same state at the point at which they each make a request of their I/O engine.

The mirrored server engine's notion of time is controlled by the incoming events from the I/O engine. The mirrored server engine does not access the CPU timer directly.

In the normal mode of operation, the secondary I/O engine keeps all requests given to it from the secondary mirrored server engine, but does not act on the majority of these requests.[11] The secondary needs to keep track of outstanding requests in order to deal with primary failures. The primary I/O engine executes all requests (with the exception of those directed at the secondary I/O engine). Events generated in the I/O engine as a result of servicing events, or of external events, are collected by the primary I/O engine and communicated to the secondary I/O engine.

As a diagnostic tool we have the capability to capture the request stream generated by the secondary mirrored server engine and compare it with the request stream generated by the primary mirrored server engine. This has been useful in detecting bugs in programs (referencing uninitialized data or data in the process of being updated by the I/O engine).

Additionally, because of our ability to capture the state of a system and capture the event and request streams for that system’s subsequent behavior, we have the (currently unrealized) ability to allow customers in the field to record this data when diagnosing a bug and to forward it to their software vendors for replay and analysis.

4.3. Fault Recovery

The primary and secondary I/O engines constantly communicate over the MSL. If the secondary detects that the primary has failed, it will take over for the primary.[12] While up, the network routing protocol running on the primary had been advertising reachability to the virtual network used by the mirrored server engines. Now, the secondary starts advertising this reachability. As soon as the routing topology converges (which depends on the routing protocol in use as well as on the topology between the connected client systems and the two servers), all connections will continue running as if nothing happened.

When the old primary comes back up, it synchronizes its state with the new primary and takes on the role of secondary.

[11]If the request is, for example, for a disk I/O operation on a disk physically attached to the secondary server, the secondary I/O engine will execute that request.

[12]The secondary, before deciding that the primary is down, attempts to contact the primary across the internetwork. This is done to detect the case where the MSL, and not the primary, has become inoperative.

When applied to file server processes running in the mirrored server engine, this system bears a resemblance to the Harp file system [LISK91]. To our knowledge, however, this is the first system in which a pure software approach to fault tolerance has been applied to such a wide range of services (such as database, mail and messaging, printing, different file service protocols, etc.). Additionally, in SFT III, the knowledge that the system has been replicated is hidden from the specific service processes, being handled by the operating system instead.

The server replication design is described in more detail in [MAJO92].

5. Future Work

As mentioned in section 3.1, currently the system only supports two protection domains. We are working on allowing an arbitrarily large number of protection domains. Currently, portions of the storage allocated to a calling domain are shared with the called domain, a protection exposure which needs to be addressed.

The failure model for protection domains (discussed in section 3.2) will evolve as we get more experience with it.

There is some potential for making use of protection domains in order to run NetWare in a non-shared memory, multiprocessor environment.

Currently, some code paths for interrupt time processing are fairly long. We would like to use work objects to shorten the interrupt time component of these paths. Among other benefits, we expect to get a more robust system by reducing the amount of code that needs to deal with concurrency issues.

Finally, engineering work on NetWare, the product, is ongoing (for example, porting the system to various RISC-based architectures).

6. Acknowledgments

The NetWare kernel and operating system are the result of many people's efforts over the years – too many people to list here. We would like to thank Sam Leffler and the USENIX referees for helpful comments on this paper. Michael Marks provided valuable comments and feedback on earlier drafts of this paper.

7. References

[ACCE86] Accetta, M. J., W. Baron, R. V. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young, "Mach: A New Kernel Foundation for Unix Development", in *Proceedings of the Summer USENIX Conference.* July, 1986.

[CLAR85] Clark, David D., "The Structuring of Systems Using Upcalls", in *Proceedings of the 10th Symposium on Operating Systems Principles.* December, 1985.

[DRUS92] Druschel, Peter, Larry L. Peterson, and Norman C. Hutchinson, "Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto", in *Proceedings of the Twelfth International Conference on Distributed Computing Systems*. June, 1992.

[FINL90] Finlayson, Ross, Mark D. Hennecke, and Steven L. Goldberg, "Vanguard: A Protocol Suite and OS Kernel for Distributed Object-Oriented Environments", in *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, October, 1990.

[FORD94] Ford, Bryan, and Jay Lepreau, "Evolving Mach 3.0 to a Migrating Thread Model", in *Proceedings of the Winter USENIX Conference.* January, 1994.

[GIEN90] Gien, Michel, "Micro-Kernel Design", in *Unix Review*, 8(11):58-63. November, 1990.

[GING89] Gingell, Robert A., "Shared Libraries", in *Unix Review*, 7(8):56-66. August, 1989.

[HAMI93] Hamilton, Graham, and Panos Kougiouris, "The Spring Nucleus: A Microkernel for Objects", in *Proceedings of the Summer USENIX Conference*. June, 1993.

[HARB91] Harbison, Samuel P., and Guy L. Steele Jr., *C, A Reference Manual.* 1991.

[JOHN93] Johnson, David, and Willy Zwaenepoel, "The Peregrine High-Performance RPC System", *Software – Practice & Experience*, 23(2):201-221. February, 1993.

[JONE80] Jones, Anita K., and Peter Schwartz, "Experience Using Multiprocessor Systems – A Status Report", in *ACM Computing Surveys*, 12(2):121-165. June, 1980.

[LEFF89] Leffler, Samuel J., Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System.* 1989.

[LI89] Li, Kai, and Paul Hudak, "Memory Coherence in Shared Virtual Memory Systems", in *ACM Transactions on Computer Systems*, 7(4):321-359. November, 1989.

[LISK91] Liskov, Barbara, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams, "Replication in the Harp File System", in *Proceedings of the 13th Symposium on Operating Systems Principles*. December, 1991.

[MAJO92] Major, Drew, Kyle Powell, and Dale Neibaur, "Fault Tolerant Computer System", in *United States Patent 5,157,663.* October, 1992.

[OMG91] Object Management Group, *The Common Object Request Broker: Architecture and Specification.* 1991.

[RITC78] Ritchie, D. M., and K. Thompson, "The UNIX Time-sharing System", in *Bell System Technical Journal*, 57(6):1905-1929. July-August, 1978.

[TANE92] Tanenbaum, Andrew S., *Modern Operating Systems.* 1992.

[YOUN87] Young, M., A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles.* November, 1987.

Author Information

Drew Major is chief scientist and systems architect at Novell, working with the software development team. He was instrumental in the design and implementation of the NetWare operating system, scheduler, file system, and server replication. He holds a B.S. degree in Computer Science from Brigham Young University.

Greg Minshall is involved in the design of networking and operating systems at Novell. He holds an A.B. degree in Pure Mathematics from the University of California at Berkeley. His e-mail address is minshall@wc.novell.com.

Kyle Powell is a senior systems architect at Novell, having been heavily involved in the design and implementation of the client portion of NetWare, as well as in server replication and the operating system itself. He holds a B.S. degree in Computer Science from Brigham Young University.
