Dynamic Updates and Failure Resilience for the Minix File Server

Thomas Veerman

A Master's thesis
in
Computer Science

Presented to the Vrije Universiteit Amsterdam in Partial Fulfillment of the Requirements for the Degree of Master of Science

May 11, 2009

Dynamic Updates and Failure Resilience for the Minix File Server

Thomas Veerman

APPROVED BY

prof.dr.	**Andrew S. Tanenbaum:** (supervisor)	_____
	Jorrit N. Herder: (second reader)	_____

Contents

List of Figures

Abstract

MINIX 3 is a multi-server operating system based on a microkernel. The reincarnation server provides failure resilience for stateless services by monitoring them at run-time and restarting them if a failure is detected. Also, the reincarnation server provides dynamic updates by replacing stateless services with a new version after, for example, a bug fix.

However, many services are not stateless and cannot rely on the services provided by the reincarnation server. In this thesis we give an overview of existing methods that enable dynamic updates and failure resilience. Then we describe methods to implement dynamic updates and failure resilience for the Minix File Server. Additionally, this thesis presents a comparison of using threads and continuations to implement dynamic updates and failure resilience. Both approaches are capable of implementing all requirements, but our conclusion is that threads are preferable to continuations.

Chapter 1

Introduction

In August 2006 the Minix File Server (MFS) was separated into two layers; an upper layer called the Virtual File Server (VFS) and a lower layer consisting of several File Servers (FSes) [12]. System calls addressed to the file system are handled by the VFS and later dispatched to the FS handling that part of the file system designated by the system call. This model allows the implementation of multiple file systems while maintaining a uniform interface to access files on these file systems. Moreover, separating the file system into several stand-alone processes ensures that a crash of an FS cannot influence other parts of the file system. However, currently when an FS crashes it drags VFS with it.

In this thesis we describe a technique that provides failure resilience for VFS; this technique allows for a transparent restart of an FS when it crashes such that work can continue after it has been restored. We also look at dynamic update for FSes where we try to transparently install a new version of an FS without rebooting the Operating System (OS). These techniques are related to each other as they both involve stopping execution of a running program, save state of the program, start a new copy of the stopped program, and restore state afterwards. In a sense, we can regard dynamic update as a special case of failure; one where the program fails in a controlled way.

Currently only the Minix File System is implemented in a server called MFS (a future release of Minix will include ISOFS). In this document we will be focusing on VFS/MFS only, but our findings will generally be applicable to future FSs as well (e.g., FAT, EXT{2,3}).[1]

The rest of this chapter is organized as follows. In Section 1.1 we will briefly discuss the general architecture of Minix 3 followed by an overview of the structure of VFS in Section 1.2. Then we present a more thorough discussion of dynamic update and failure resilience in Section 1.3 and give an introduction of our VFS designs in Section 1.4. Finally, we give an outline of the rest of this thesis in 1.5.

[1] We use the term FS and MFS interchangeably throughout this document. When discussing in more general terms we use FS, while we use MFS when we focus more on the one implementation that is currently available.

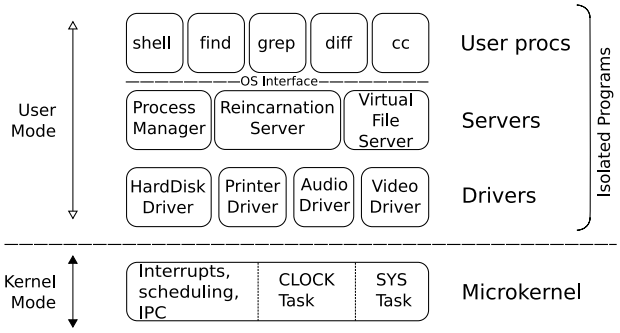


Figure 1.1: The compartmentalized design of MINIX 3.

1.1 The MINIX 3 OS

MINIX 3 is a multi-server operating system designed to be highly dependable [18]. It has a microkernel which does a few very basic tasks (e.g., interrupt handling, process scheduling, and inter-process communication (IPC)) running in kernel mode and on top of that a layer of drivers and servers running in user mode as shown in Fig. 1.1.

The kernel consists of about 4000 lines of code and does only the bare minimum. It programs the MMU and CPU, does interrupt handling, provides IPC, and schedules processes. Additionally, the kernel has two processes compiled into kernel space: the clock task and system task. The clock task is an I/O driver that handles the clock hardware. However, as the kernel routines depend on this driver and it only interfaces with the kernel –user processes cannot access it directly– it is made part of the kernel. Other drivers are all in user space. The system task provides a set of privileged kernel calls that can be used by drivers and servers that require low-level kernel-mode operations. Although the clock and system task are part of the kernel, they are scheduled as separate processes and have their own call stacks.

On top of the kernel runs a layer of drivers that interact with the hardware. They are implemented as independent user-mode processes that are restricted in which privileged operations they can do. Due to their isolated nature they cannot influence other drivers or the kernel in the face of a crash (e.g., by overwriting data structures with corrupt data). Some drivers can even be restarted transparently and continue their operations as if nothing has happened. For example, the network driver can fail and be restarted. The user notices only a very small decrease in network speed (i.e., only noticeable for frequent driver crashes) [17].

Finally, there is an upper layer consisting of POSIX-compliant servers that work together with drivers to provide the functionality found in Unix-like operating systems. Just like drivers, servers are restricted in what they are capable of. The Process Manager (PM) and Virtual File Server together implement the POSIX interface for application programs. PM itself handles process management and signaling. For example, starting and stopping of programs and priority assignment. It also maintains the relation between processes such

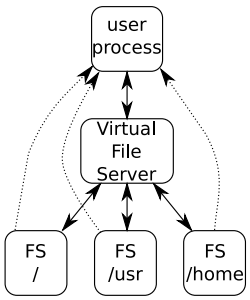


Figure 1.2: File system related system calls go through VFS.

as parent-child relationships and process groups, and allocates and deallocates memory (as of MINIX 3.1.4 memory is handled by a separate VM server). VFS handles the file system in cooperation with one or more File Servers. The Reincarnation Server (RS) has the important role of managing privileged processes (i.e., servers and drivers). It starts and stops servers and drivers and makes sure they are *alive*; it keeps an eye out for crashed and failing processes. The former is achieved as follows. All servers and drivers are marked as children of RS during system initialization. RS then simply waits for `SIGCHLD` signals, which are sent to parent processes when a child exits. The latter is done by sending a heartbeat message to processes and then wait for a response. When a process fails to respond in time, it is regarded as failing and is shutdown and subsequently restarted. Finally, there is the Data Store (DS). It acts as a small database with publish-subscribe functionality. A system process can store information and another system process can subscribe to it. When the stored value changes, all subscribers are notified of the new value.

1.2 The Virtual File System Layer

As mentioned earlier the MINIX File Server is organized in two layers: a top layer consisting of the Virtual File Server and a bottom layer consisting of several File Servers. The FSes each manage a file system on a disk partition. Processes communicate with VFS to do file system related system calls. VFS then talks to the FSes and sends back a reply, as depicted in Figure 1.2.

The figure shows a user process talking to VFS and three FSes that each hold a part of the file system. In this situation the file system on the hard disk is spread out over three partitions; one partition holding the root of the file system, one holding the /usr tree of the file system, and one holding the /home tree. VFS glues the FSes together into one tree. When VFS is done with the request it sends back a message with the result. In some situations the result will not fit in a message. For example, when reading data from disk or asking for the status of a file. When that happens an FS copies the result directly to a buffer inside the process and VFS will only tell how many bytes were read or that the operation was executed successfully.

Now, let us look at an example of how communication works between a

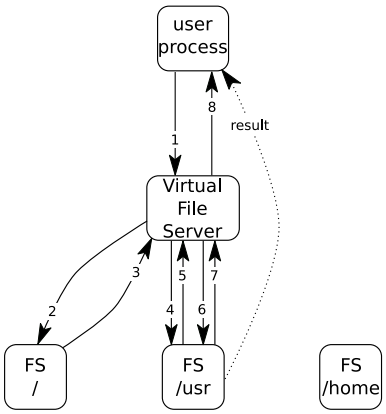


Figure 1.3: Communication flow of file status retrieval.

process, VFS, and the FSes when we retrieve the file status of */usr/bin/cc* in Figure 1.3. The following happens. The process does the system call and VFS starts by looking up which inode belongs to *cc* (1). The string "`/usr/bin/cc`" is checked by VFS to find out whether the path is absolute or relative (i.e., if the path starts or does not start with a /, respectively). If the path is relative the lookup starts at the *current working directory*. Else the lookup starts at the root of the file system.[2] VFS then sends a lookup request to the FS holding the inode from where we start looking; in this case the request is sent to the root-FS (2). The root-FS subsequently parses the string directory by directory until it reaches the file it was looking for or if it finds out the lookup cannot continue on this FS. For example, if we move up in the file system tree to a *higher* FS or down in the tree to a *lower* FS. The former is called *leaving a mount point* and the latter *entering a mount point*. In this case *usr* is located on a lower FS and we will therefore enter a mount point. The root-FS then replies (3) to VFS "enter mount point" along with how much path characters were processed (in this case 1 –the starting /). VFS then looks up in a mount point table which FS belongs to `usr`, and asks it to continue the look up of "`usr/bin/cc`" (4). The usr-FS parses the path up to "`cc`", which results in opening *bin* and looking up the inode in that directory that belongs to *cc*. This inode number is sent back to VFS (5).

VFS now knows that */usr/bin/cc* is on the usr-FS and what its inode is. It then sends a *req_stat* request for that inode to the usr-FS (6). The usr-FS executes the request and copies the result to a buffer in the originating process that did the system call (result). VFS is notified everything went well (7) and in turn replies to the user process the operation was successful (8). System calls are blocking requests, so the reply will unblock the process.

[2]Or to be more precise, at the root of the file system relative to the process. A process can be in a jailed environment where the root of the file system is changed to a confined part of the file system.

1.3 Dynamic Updates and Failure Resilience

Dynamic update is a mechanism that allows software updates and patches to be applied to a running system without loss of service or down-time [6]. Having such a mechanism implemented in software is advantageous, because there is less need to restart the OS (downtime). A study on Windows NT showed that unplanned downtime more than twice as often is caused by software rather than hardware failures; moreover, reboots due to maintenance are responsible for 24% of downtime [34].

In this thesis we present a mechanism that implements dynamic updates for the file system. The mechanism allows an FS to be updated while the rest of the system keeps running.

In MINIX 3, reliability is achieved by fault isolation and failure recovery. Faults are isolated by putting most code in isolated user-mode processes and keep only the bare minimum in a microkernel. If a driver becomes corrupt due to a bug and shows undefined behavior, it can never change data structures of other drivers or the kernel. A driver that has crashed or has become unresponsive is replaced by a fresh copy. This way the system can recover from a failure in, for example, a disk driver. Recently support for protection against block-device driver errors was added to MINIX. The protection achieves end-to-end integrity and is able to detect silent data corruption [19].

There are, of course, situations where MINIX cannot recover from a failure. For example, if VFS or RS fail, there is no way to recover. In the former case there is no file system from which we can read the binary image of a program. Also, VFS plays too big a role in the execution of a program even if the image was somehow stored in memory. In the latter case the (grand) parent of all processes died and all logic to start or stop privileged processes is gone.

To simplify the recovery model some processes are regarded as *trusted* and being stable. No measures are taken to recover from failure. If one of these processes fail, the operating system is likely to come to a halt. On the other hand, processes that are not trusted should never lead to a system wide crash. For example, drivers are not trusted. Everyone can write a driver and do a lousy job. The recovery model of MINIX 3 tries to overcome failing drivers. File Servers, which can be considered as a special class of drivers, are also not trusted, but currently lack a recovery model. If one of them currently fails, they will lead to failure of VFS as well. The recovery model used for drivers cannot be applied to FSes, because FSes are stateful. If an FS crashes it could lose data that was not yet written to disk, possibly leading to file system corruption. Moreover, VFS assumes certain state in an FS which would not be present after a restart. We will present a mechanism for VFS that provides resilience against failing FSes.

1.4 Decoupling VFS and MFS

Asynchronous communication is needed between VFS and FSes to enable VFS to not block when an FS fails and to support request queues that are needed to implement dynamic updates and failure resilience (see Chapter 3).

When communication is asynchronous, the sender of a message does not wait for an answer. This is convenient when the sender only wants to tell the listener

about an event and is not interested in results. In our case, however, the sender is interested in results. This leads to a few problems. For example, when we receive a reply, we have to somehow distinguish between a result of a request sent earlier to an FS and a new incoming request from a user process. Also, as there is no direct relation between sending a request to an FS and receiving a reply, the reply can be a result of any request sent earlier. That is, we have to figure out to which request the reply belongs to.

An advantage of asynchronous communication is that we do not waste time waiting for a reply. Instead, we can do other useful work; we can handle multiple requests concurrently. However, multiple requests can all access the same (global) data structures within VFS. Access to these data structures needs to be protected, or we risk race conditions and potentially deadlocks. We introduce a locking model which provides this protection.

Another consequence of handling requests concurrently is that we risk that system calls are executed interleaved, leading to a different result when they would be executed sequentially. We solve this by serializing all requests. This means that all requests belonging to a system call are executed by an FS before requests of another system call are handled (by the same FS).

In Chapter 4 we describe two designs that provide for this kind of communication; one based on threads and one based on continuations. The threaded design is based on a user-space threading library (Minix currently lacks support for kernel threads) [32]. It has a main thread which receives messages and subsequently spawns a worker thread for each message. The worker thread carries out the work and sends back a reply to the originating process. When it sends a request to an FS it suspends itself waiting for an event. Upon arrival of the reply to the request, the event is fired and the worker thread continues execution. The continuations design is a big finite state machine that has a routine for each state. System calls are divided into sub-requests; each sub-request is separated by an asynchronous communication step. Before and after communication, state must be serialized and deserialized before and after communication, such that the routine belonging to the request's state can continue execution of the system call.

1.5 Outline of the Thesis

The outline of this thesis is as follows. In Chapter 2 we describe related work on dynamic update and failure resilience. We present a classification model where dynamic updating systems are divided into four categories based on the techniques used to implement the systems. Chapter 3 presents techniques to implement both dynamic updates and failure resilience for the VFS. In Chapter 4 we describe how these techniques can be actually implemented in the designs based on threads and continuations, discuss solutions for issues that arise, and compare the approaches. Chapter 5 concludes this thesis.

Chapter 2

Related Work

In this thesis we want to design a VFS/FS model that allows for dynamic updates and is resilient against failing FSes. Before we describe our techniques in Chapter 3, we discuss what others have done. Moreover, based on their work we were able to devise a classification that could aid in the search for a suitable dynamic update mechanism.

2.1 Dynamic Updates

Once in a while software needs to be updated in order to add functionality or fix bugs. Traditionally, the running software is stopped, updated, and subsequently restarted. In some cases the entire OS has to be restarted. This downtime could be very costly or simply intolerable. For example, telephone systems are required to have a downtime of no more than 2 hours within 40 years [28], financial institutions have to be able to do transactions at all times or risk losing revenue, and none-stop systems need to be able to update software without having to interrupt the service they are providing.

Solutions to this problem are based on redundancy and *dynamic updates*. Redundant hardware is used to support high load spikes, but can also be used to replace software. Parts of the system can be shut down for maintenance, while the rest of the system keeps serving requests using the old software [26]. However, these systems are expensive and difficult to build. Also, there are less complex situations that need updating and not having to interrupt service would be a big advantage. For example, it would be convenient not having to reboot your desktop each time the operating system is updated. One way to achieve this is to apply dynamic updates. In literature, dynamic updates are also referred to as *Dynamic (Software) Updating Systems*, *dynamic modification* [14], *on-the-fly program modification*[11], *on-line version change* [15], *live updates* or *hot updates* [2].

Every dynamic updating system makes use of either two base techniques:

1. A new program is started and state of the old program is copied to the new program.

2. Indirection is used with updated pointers.

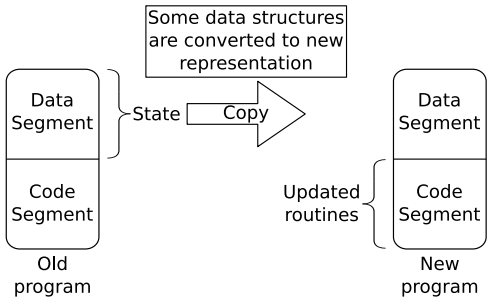


Figure 2.1: State is copied from the old program to the new program.

With the first technique a running program is replaced with a newer version of the same program, as shown in Figure 2.1. Upon update, the running program is suspended, a copy of the new version is started, state is copied from the old program to the new program, and the old program is killed. It is possible that an update not only involves fixing a bug, but also a change in the data structures. During copying of the program state these data structures can be adapted to the new version (e.g., adding or removing fields) using *state transfer functions*.

The second technique revolves around the usage of a table of pointers to procedures. An indirect pointer points to a procedure which is resolved when it is called. Updating the pointer in the table to point to a new version of a procedure results in an updated program. In Figure 2.2 an indirection table is shown.

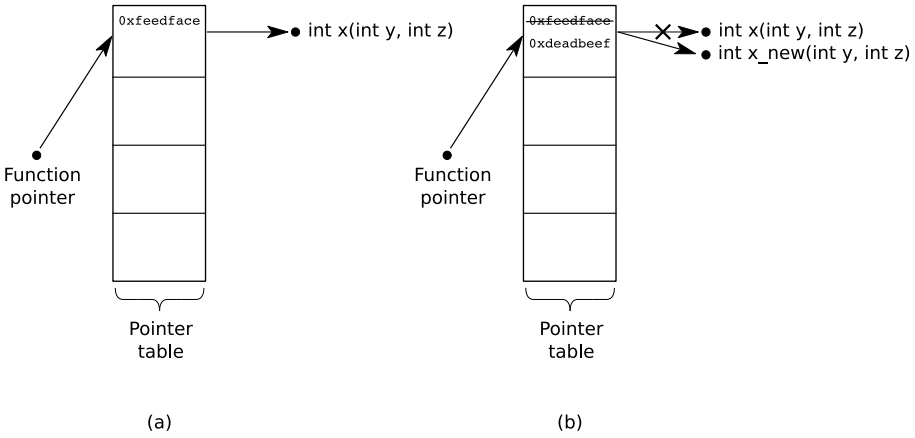


Figure 2.2: Indirection is used with updated pointers.

A program uses an indirect pointer to access procedure `x`. Upon update the pointer in the table is changed and now points to a different routine `x_new`. When a program calls the procedure using the indirect pointer, the new version

will be called instead of the old version. Changes to procedure interfaces are not supported as the program doing the procedure call is not aware of any changes (i.e., function parameters cannot be altered). A variation on this scheme exists that does allow interface changes. All references to the old procedure in the code segment are updated to point to a new version (i.e., no indirect pointers are used). This approach is much more tricky, because you have to cope with calls to the old procedure that still exist on the stack and could return different results than expected.

2.1.1 Classification

In the past software-based updating systems have been classified by different types of software updating systems [28];

- replacement of abstract data types in programs,
- replacement of servers in client-server systems,
- updating of distributed programs that use externally specified communication topologies, and
- programs written in procedural languages.

Also, a classification based on the language (i.e., high level versus C/C++) and domain-specific solutions (client-server, distributed systems, OS kernels) has been used [4].

Here we introduce a classification based on the way the dynamic update mechanisms are constructed:

1. Languages and runtime systems; dynamic updates are supported by a specific language, often backed by a (modified) runtime system.
2. Dynamic updating mechanisms in operating systems; dynamic update mechanisms provided by the operating system.
3. Binary patchers; systems (compilers, linkers, tools) generating binary patches that modify the running program in the code and/or data segments and the stack.
4. Programs with built-in dynamic update mechanisms; programs that have routines facilitating dynamic updates built in.

2.1.2 Languages and runtime systems

This class of dynamic updating systems provides program modifications by using a specific language (often additions to existing languages). In some cases the language uses a runtime system (interpreted languages) or is based on a modified existing runtime system (e.g., Java Virtual Machine).

Disadvantages of these type of dynamic updating systems are the use of a specific language and the use of a runtime system, because it makes them unsuitable for existing programs in a different language and not applicable for operating systems (e.g., updates to the kernel).

Example systems are DYMOS [23], DynamicML [13], and JDRUMS [27]. We will discuss DYMOS and DynamicML.

DYMOS The DYnamic MOdification System is an integrated programming system consisting of a command interpreter, text editor, source code manager, compiler, and runtime support system. The language used is Starmod; a modular, concurrent programming language based on Modula.

In DYMOS, a program is updated by replacing modules or only individual procedures. A new procedure or module is compiled and checked against previous compile results to enforce type checking. Then a loader loads the new module or procedure into the core image, locks access to the to-be-updated procedures to ensure they are not invoked while the update is in progress, updates the Module Access Table (MAT) and Procedure Access Table (PAT) respectively, and unlocks the locked procedures. DYMOS supports interface changes using conversion procedures supplied by the programmer. The PAT is adapted in such a way that the address of an old procedure points to the convert procedure. Subsequently, the convert procedure adapts the call using default parameters and calls the new procedure. Afterwards, the result is adapted back to the old interface and the result is returned. A new address is registered in the PAT for calls to the new procedure.

In order to make sure updates are done at the right time, the update command can be given conditions that have to be met before the update is performed. For example, `update P, Q when P idle`, to update procedures P and Q only when procedure P is not in use (i.e., not on the stack). DYMOS uses the symbol table that is generated after compilation and the state of the running program to determine when it is safe to update P and Q. Optionally, a `delete <procedures>` argument can be appended to designate procedures that are no longer needed and can be removed.

Disadvantages of DYMOS are that it requires source code to be available and it is easy to make an error in the updating process by not supplying the right conditions that have to be met before the update is executed.

DynamicML DynamicML is a language based on Standard ML (a statically compiled functional language) with additions to facilitate dynamic updates.

In DynamicML dynamic updates are done on module-level, with the restriction that modules keep the same interface and type signature. Encapsulated functions, values, and types can be altered in any way. Modules were chosen as a unit of update because modules can be compiled in isolation, therefore they do not require a recompilation of the entire program. Modules are replaced using a modified garbage collector.

In a uniprocessor implementation the data segment is divided into two semi-segments; only one semi-segment at a time is used for the heap. When memory allocation fails, the garbage collector mechanism copies objects still in use from the semi-segment *from* to the semi-segment *to*, resulting in a cleaned up *to* segment. The roles of the semi-segments are then swapped (i.e., *to* becomes the new *from* and vice versa).

Figure 2.3 shows how this mechanism can be used for dynamic updates. While objects are copied to the semi-segment *to*, some objects are processed by conversion functions to update them to the new signature implementation. Although the signature interface remains the same, internally data representation can be significantly changed and might need conversion. When an exception is raised due to a programming error, a rollback is done by simply swapping the

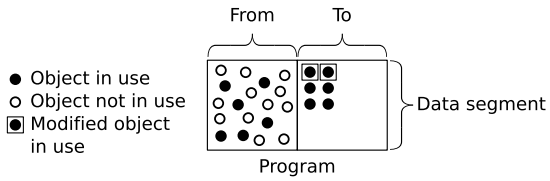


Figure 2.3: DynamicML with the data segment divided in two semi-segments.

semi-segment roles again.

2.1.3 Dynamic updating mechanisms in operating systems

Some operating systems have mechanisms that can be used to provide dynamic updates. In most cases a level of indirection is used for calls to routines. When a routine has to be replaced with a new version, the pointer that points to the old routine is modified to point to the new version.

In general, these kind of dynamic updating systems do not support changes to the routine interface, and often the implementations require specific mechanisms that are not available in other operating systems. However, because dynamic updates are part of the operating system, they can be applied to every program. Example systems are MULTICS [8], Dynamic Type Replacement (DTR) [11], and *D*ynamically *A*lterable *S*ystem (DAS) [14]. We will discuss MULTICS and DTR in more detail.

MULTICS MULTICS is an operating system from the 1960s where a procedure has the ability to use another procedure knowing only its name, without knowledge of its requirements for storage, or which additional procedures it could call upon in turn.

Each procedure call is made indirectly using a pointer in the *linkage segment*. Figure 2.4 shows how this mechanism allows for dynamic updates. A new implementation of a procedure is loaded and then entries for the old implementation in the linkage segment are marked invalid. Invoking the old procedure results in a trap and a transparent relink to the new procedure. Programs that invoked the old procedure right before the update can continue running without a problem as the old code is still available and the interface has not changed.

This mechanism works well, but there is no support for changes to procedure interfaces or data structures.

Dynamic Type Replacement A mechanism similar to MULTICS is Dynamic Type Replacement by Fabry that does support changes to data structures. DTR uses, similar to MULTICS, a level of indirection to provide dynamic updates, embedded in compilers and mechanisms in the operating system.

In DTR, a new code segment for a routine is loaded, and the *capability*[1] in

[1] In a capability-based addressing scheme, pointers are replaced by protected objects called capabilities. These objects can only be created by a privileged process. This allows a strict protection of which parts of the memory a process is allowed to access, making separate

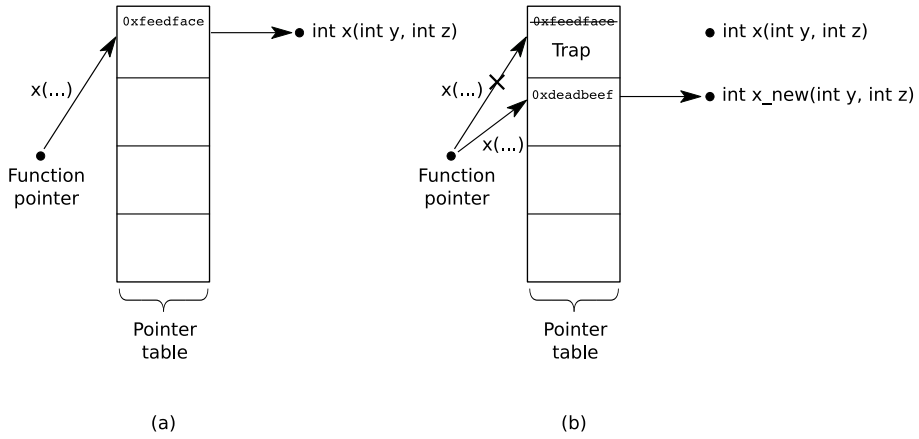


Figure 2.4: Indirection is used with updated pointers. Upon update, the old pointer is replaced by a trap and a new version of the procedure is registered in the pointer table. Invocation of the old version leads to a trap and transparent relink.

the call routine to the old code segment is replaced by a new capability pointing to the new code segment. To support new data structures, every data structure carries a version number. In the indirect call to a routine, a check is done on the version of the data structure in use and the version expected by the new code segment. If they do not match, a conversion routine is called to update the old data structure to the new representation.

Disadvantages of DTR are its lack of support for interface changes and the fact that it needs a capability-based addressing scheme (although Fabry mentions it might be possible to use a MULTICS-like addressing environment). Also, if there are multiple updates and versions the system will have to provide multiple conversion routines for each update.

2.1.4 Binary patchers

Binary patchers are software updating systems that update programs by modifying code and data segments and the stack. The range of type of updates supported among binary patchers differs greatly. Some allow changes to function interfaces, others do not. Changes to global data cannot always be done, and although binary patchers can be applied to any program in general, not every system allows programs to be threaded. Also, programs that heavily rely on state stored in the kernel can prove to be hard to update by some systems.

Below we describe three example systems called Gupta [15], Ginseng [24] and Ksplice [2]. Other binary patchers are OPUS [1] and Hicks [20].

Gupta Gupta is a binary patching system that starts a new process running the new version of a program, and then copies state from the old version to the

address spaces and the accompanying context switches unnecessary.

new version. It uses a normal compiler and linker, requires the program to be linked to a library that provides helping routines, and a wrapping modification shell that starts and stops the programs. Because programs are started by the modification shell, they can be traced with the `ptrace` system call, allowing the shell to inspect and modify the address space and machine registers. A program is replaced by the new version when all routines that have changed are not on the stack.

The dynamic updating process works as follows. A *run* module in the modification shell starts the program. After compiling a new version of the program, the user defines a *change configuration file* listing the functions that have changed. Then the user issues a replace command in the modification shell, so the *replace* module will create a process running the new program. To transfer state, the old program is made to call a library routine to prepare the transition process. The replace module then copies the data and stack of the old program to the new version, followed by the machine registers. To make sure these registers hold correct values in the new program, they are compared with the symbol tables and new offsets are calculated. This is also done for all return addresses on the stack. To support new data structures, the user can write an initialization routine that converts old data structures to new data structures. However, support for this is limited to local data only. Function interface changes are implemented as follows. Suppose functions `a`, `b`, and `c` call `z`, but `z` expects different parameters in the new program. Then either functions `a`, `b`, and `c` can be modified to call `z` with the new parameters and have all 4 functions registered as *changed* in the change configuration file, or make `z` an interprocedure to `new_z` that adapts the parameters to the new version and have only `z` registered as changed. The latter solution suffers from a small overhead and risks taking a long time before it is replaced when it is part of a long running loop.

To copy implicit kernel-state (such as open files) wrappers have to be written for those system calls that store the state in the program. This works for files, but it is not available for network sockets. Also, when a program forks off a child process, the new program will have no relation to that child. Moreover, the new program will carry a new process ID, which might be a problem for interprocess communication.

Ginseng Ginseng is a binary patching system consisting of a compiler, a patch generator, and a runtime system for building updatable software. Programmers are required to annotate *safe update points* in the original code (i.e., when it is safe to do a dynamic update in order not to break the program or produce faulty behavior). Static analysis is used to associate update constraints to each update point, as well as to detect bad programmer-annotated update points.

At compile time, Ginseng rewrites the program to make every function call indirect that will later enable dynamic updates. Also, Ginseng makes use of *type wrapping* that wraps a type to carry a version field, and inserts a *coercion* function that returns the underlying representation wherever the wrapped type is used concretely. The Ginseng patch generator finds out about changed functions and types and provides *state transformer* functions that are run at update time to convert state. The runtime system marks updates as pending, and when a safe point is reached, it checks if a pending update can be performed (that is, if it meets the conditions defined earlier). If it does, it is linked into

Original code	Rewritten code

	`struct T {`
	`  unsigned int version;`
	`  union { struct _T0 data;`
	`  char padding[X]; } udata;`
	`}`

`struct T {`
`  int x;`
`  int y;`
`};`
	`struct __T0 {`
	`  int x;`
	`  int y;`
	`};`

`void foo(int* x) {`
`  *x = 1;`
`}`
	`void __foo_v0(int* x) {`
	`  *x = 1;`
	`}`

	`void * foo_ptr = &__foo_v0;`

	`void __foo_wrap(int* x) {`
	`  (*foo_ptr)(x);`
	`}`

	`struct __T0* __con_T(struct T* abs){`
	`  __DSU_transform(abs);`
	`  return &abs->udata.data;`
	`}`

Table 2.1: Ginseng code rewriting enabling dynamic updates[24].

the updatable program.

To update a program, the user sends a signal to the running program that in turn notifies the runtime system. Initialization code generated by the compiler is run to glue the update into the program by updating function indirection pointers and converting state.

Table 2.1 shows how code is rewritten. Data structures and procedures are renamed and have a version number appended. New data structures are created with the original name that wrap the original data structure and add a version number. `_ptr` variables are used to reference original procedures. New `_wrap` procedures are added as wrappers around the original procedures. Together with the indirect `_ptr` function pointers this enables dynamic updates. Finally, `__con_` procedure are added to unwrap wrapped data structures.

Because of the heavy use of wrapping and indirection pointers, the updatable programs suffer from overhead. It was measured to have an overhead of 0 to 32%.

Ksplice Ksplice is a binary patcher that is able to update *legacy binaries* (unmodified binaries that have not been prepared for dynamic updates) by comparing object code of the running binary and the patched binary. By comparing object code rather than source code, Ksplice avoids issues that could

arise that are not apparent when just looking at the source code. For example, modification to an implicit cast in a prototyped function requires changes to the executable code of all functions that call the prototyped function. Also, some compilers inline code that do not have an explicit `inline` statement. Failing to find all occurrences of the function could lead to data corruption and program instability.

A binary diff reveals all changes in data structures and functions. Adapted code is dynamically linked with the running binary and jump instructions are added at the beginning of obsolete functions. When data structures semantically change between the old and new version, Ksplice allows a programmer to supply additional code to modify data structures in memory. Ksplice generates and loads patches automatically without the need for help from a programmer.

Ksplice was primarily developed to install security patches for the Linux kernel. It supports function interface changes and can modify (global) data structures. The dynamic update is applied to the running version of a binary, instead of starting a new binary and copying a modified data segment (which is impossible for a kernel). However, it does require functions not to be active (i.e., cannot update *non-quiescent* functions), so long running functions will not be updated. To check a function is not active, Ksplice uses Linux's `stop_machine` to grab all CPUs and run a procedure that does a stack inspection to make sure no thread is currently executing the function.+

Java A system for Java that resembles Ginseng a lot was developed by Orso et al [25]. A tool called DUSC (Dynamic Updating through Swapping of Classes) takes a class and rewrites the byte-code so that it is dynamically updatable. To perform a dynamic update, the system waits for a safe point (i.e., no method of the classes that are to be updated is executing), creates new instances of the new classes and copies state of the old instances. Finally it updates reference pointers to point to the new objects. Classes can be added, removed, or updated. Class interface changes are not supported, but can be done anyway by registering the old version as removed and the new version as newly added.

Although this system is based on Java, it requires no changes to the language nor the JVM. However, it does rewrite byte-code and therefore belongs in the *binary patcher* category rather than *Languages and runtime systems*.

2.1.5 Programs with built-in dynamic update mechanisms

Programs belonging to this category employ techniques that are entirely enclosed by the program itself. They do not make use of tools that aid in the update[2], specific languages that are tailored for dynamic updates, or mechanisms built into the operating system. Programs that make use of the client-server technique typically belong in this category. Also, as we will see below, design patterns can be used to facilitate dynamic updates.

We will discuss Dynamic C++ classes [21] in more detail. An other example that belongs to this category is dynamic updates in the K42 research kernel [6, 5].

[2]Except tools that aid the user in notifying the program that there are updates and what should be updated.

Dynamic C++ classes Dynamic C++ classes allow run-time updates of an executing C++ program at the class level. The system works with standard C++ compilers and the use of proxy templates. Using two-level indirection, method invocations are forwarded to the correct implementation. Interface changes are not supported because of the use of proxy templates. The templates remain fixed and only forward invocations. Internally, new class implementations can invoke newly introduced private methods. There are no state transfer mechanisms; older versions of objects remain in use. The templates have an *invalidate* method that marks all older versions as invalid. If an invalidated, old version is then invoked, an exception is raised. It is up to the caller to recover from that (e.g., construct a new object of the new version). Obviously, this method is to be used carefully.

The advantage of this technique is that it can be applied to any program written in C++. However, only classes that are made *dynamic* are supported. Also, long running dynamic objects are not replaced.

2.2 Failure Resilience

There exist many techniques that limit the scope of a failing component in a system such that only a part of the system fails. The failing component can then be restarted and continue service or the complete system is restarted. For example, in some systems drivers are isolated and can be restarted and continue their service [30, 17]. However, there are also situations where a failing component can cause data corruption that might not be reparable. For example, when an FS fails, it could cause data corruption on disk, leading to an irreparable file system.

Existing approaches based on hardware use redundant components to provide fault tolerance [3] or memory protection by using capability-based addressing [10] or separate address spaces such that processes can only access memory appointed to them. Virtual Memory (VM) can provide protection by marking memory regions read-only. Memory protection only limits the scope of a failed component, but it does not help recovery.

A different way to achieve failure resilience is based on using high-level, type-safe languages and runtime systems. These languages and systems can prevent fault situations from occurring at runtime. Also techniques that check code correctness can be applied.

Virtualization can be used to limit the amount of code that crash the system. In this situation a machine has multiple operating systems running in a virtual environment. When one of the virtual machines crashes, the others can continue and even take over the job of the failed machine (this only works when the involved virtual machines run in absolute lockstep). Of course, when an error occurs in a driver of the host machine and the system crashes, all virtual machines are dragged along with it.

Finally, some systems wrap operations in transactions that are committed atomically. When an error occurs, the transaction is aborted and modifications are undone [16, 33].

Systems that implement techniques described above only try to limit the damage a failing component can do. We are interested in mechanisms that

can also recover a failed component and restore it to the state it was in before the failure. One such system is described in [30] where they introduce *shadow drivers*. A shadow driver is a kernel agent that improves reliability for a single device driver. When a driver fails, its shadow restores the driver to a functioning state in which it can process I/O requests made before the failure. While the driver recovers, the shadow driver services its requests. When the driver has restarted, the shadow driver replays requests that alter the internal state, so it returns to the state it was in before the crash. Afterwards it resumes pending requests. They use Nooks [31] to isolate the driver from the kernel to make sure it cannot damage other components except itself upon crashing.

A related technique is *checkpointing* where snapshots of the state of a component are taken and communication logged. After a failure a component is restarted and state restored by retrieving and restoring to saved state and then replay stored messages [29].

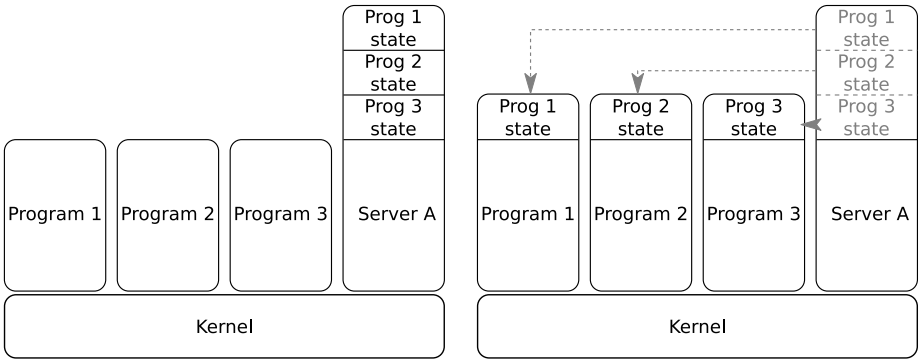
Another interesting technique is provided by the Rio (RAM I/O) file cache [7] where they make system memory safe for persistent storage. Normally memory is regarded as unsafe and data is periodically written to safe storage (e.g., to disk or tape). However, the time window between storing data in cache and writing it through to stable storage is still vulnerable to crashes. Making the window smaller (i.e. wait a shorter time before write through) decreases file I/O performance. The Rio fail cache overcomes this problem by performing a warm boot. The system is rebooted upon crash and during the boot a memory dump is written to the swap file.[3] After a full boot a tool in user-space inspects the memory dump and writes dirty cache to disk. While this technique is certainly useful, it is not interesting for us as we want to avoid having to reboot the system.

In [9] a new OS design is introduced where client-specific state is partitioned into isolated per-program memory regions referred to as Server State Regions (SSRs). An SSR is created when a client first binds to a server, and a new SSR is created for every server to which a client binds. SSRs are destroyed when clients unbind from a server. SSRs are created from the clients memory pool, but the client cannot touch the memory.

For example, when a client binds to a server and initiates a service request, the kernel enters a mapping for the SSR into the virtual memory translation tables of the server before dispatching the request to it. The server can then access the SSR and use it to store and manage client-specific state required to handle the request. When the server sends a response back to the process, the kernel removes the server's access to the SSR.

When a server crashes in this scheme, state is preserved because state is stored at the client and not at the server. Only SSRs that are mapped in during a crash are affected and could potentially be corrupted. When a server is restarted, it is responsible for detecting corrupted SSRs and repairing them. If it fails to do so, an error message is propagated to the client.

[3] It must be noted that they implemented this technique on a system that does not reset memory on boot. The technique will not work on x86 hardware.



(a) Normal situation (b) Using SSRs

Figure 2.5: Storing client-specific state at the client using Server State Regions

Chapter 3

Dynamic Updates and Failure Resilience

As we have pointed out in the introduction of this document, the design of VFS/MFS must address the following key requirements:

- MFS can be replaced, on the fly, with a new version (dynamic update).
- VFS must be able to cope with a crash of an FS (failure resilience).

When an FS crashes VFS must detect that this has happened and the FS must be recovered. It is very likely that the failed FS still had state stored that is not yet written to disk, potentially causing file system corruption. Part of the recovery process is recovering this lost state. However, when a program crashes it is generally hard to tell what exactly happened. A program can crash, for example, when it executes a faulty instruction or references memory it is not allowed to access (i.e., writes to read-only memory). Also, programs often exit with an error value when they encounter an error situation they cannot recover from (e.g., when a `malloc` fails). In the case of MFS, it panics in situations when there is no other option than to fail. These situations are often a result of sanity checks on parameters (e.g., null pointer checks) and are unlikely to cause state corruption.

In this document we will assume that crashes do not cause state to become corrupt. However, we will provide a means to validate recovered state. If after verification it turns out that state is not valid, a file system check of the data on disk and in memory is done to salvage as much as possible. Checking data in memory may increase the chance of recovery of certain data, as it might be missing or corrupt on disk, but still be available and intact in memory.

3.1 General observations

Analysis of the current code leads to a few observations. Each user process (application) can make at most only one call to VFS at any time and will be blocked waiting for the result. VFS can handle only one system call at a time. That is, it receives a system call, handles the call (e.g., sends zero or more

messages to FSes), and sends a reply with the result.[1] The Process Manager (PM) can send messages to VFS to inform it about, for example, `exit`ing or `fork`ing processes. These messages are sent asynchronously and therefore it is not mandatory to immediately send back a reply to PM.

In MFS, the only FS currently available in Minix, state consists of two data structures: the inode table and block cache. Also, MFS is capable of handling only one request at a time (e.g., only one `read`, `write`, or a lookup as opposed to multiple concurrent `read`s or `write`s).

The rest of this section describes mechanisms that implement Dynamic Updates and Failure Resilience.

3.2 Dynamic Updates

Initially we were aiming for a generic dynamic update technique that could be implemented by every stateful program, but soon we reduced that goal to design a technique specifically for VFS. Consequently, we were not looking for a technique that would be provided by the operating system.

The use of a binary patcher would certainly be useful, but for us they have too many drawbacks. For example, due to the nature of MFS, we have to be able to alter global data structures (e.g., adding a field to the `fproc` table). Also, it is desirable to avoid having versions of functions and data structures, because it complicates the update process. Finally, binary patchers can cause significant overhead and slow down performance.

It was not feasible to use a specific language with support for dynamic updates built-in (using a runtime system), because we wanted to adapt an existing program written in C. Also, VFS has to be fast and the use of a runtime system is likely to impose a significant performance penalty.

The method to implement dynamic updates for VFS we describe below, is classified as a program with built-in update mechanisms.

In a sense we can regard dynamic updates as a special case of failure.[2] In both situations the program will stop running, start a new copy, and reset program state to the moment before the failure or update. However, as a result of a crash the file system on disk might be corrupt or the inode cache or block cache in memory is in an inconsistent state. With dynamic updates we do not have this problematic situation. Therefore, a safer method to restore state can be used by writing all data to disk and reopen inodes afterwards.

Also, between updates internal data structures might change. Versioning could be used to convert data structures between versions, but that leads to a host of other issues. For example, for how many versions should conversion routines be included? A possibility is to support only one version difference and combine that with version chaining (i.e., let multiple successive FS versions update one after another). That would make the update procedure much more complex, because you need to figure out which version you are running and

[1] Exception to this scheme being reading/writing from/to pipes leads to suspension of a process until the other side of the pipe has connected and using select to wait for an active file descriptor.

[2] This holds only for a subset of dynamic updating systems. Some programs never stop running, but dynamically load additional code.

provide all intermediate versions one way or another. However, since it is highly unlikely that the file system format on disk changes often, a much easier solution would be to write data to disk and then let the new FS read the data back in. In that way there is no need for conversion routines at all.

Dynamic updates can be achieved as follows. The interface between VFS and MFS is adapted by adding a new request telling the MFS to restart. VFS issues this request and returns to its main loop to process other messages. When MFS receives this request, it does a `sync` to write the inode table to the block buffer and the block buffer to disk, followed by an exit. In order to initiate a dynamic update, the user issues a `service update <label> <path to binary>` command. RS copies the binary to memory (to a buffer on the heap), sends the update request to VFS, waits for the FS that `<label>` refers to, to stop, and finally it executes the new binary. The path to the binary can be arbitrary. That is, the binary does not need to be installed in `/sbin`. Also, by first copying the binary to memory, we do not risk the binary being unavailable during the update. This resembles the driver recovery scheme in MINIX 3 for failed block device drivers, where a copy of the block driver is held in memory and a modified `exec` call is used to start a binary from memory instead of disk[17].

Upon startup of the updated FS, it sends an FS_READY message to VFS notifying it is alive. VFS subsequently performs a *semi-mount* to remount the new MFS into the file system. The semi-mount differs from a normal mount in the sense that it does not need to do every sanity check a mount operation normally does, and after a successful mount it updates the vnode table to reflect the new MFS endpoint. For example, the procedure already has a proper minor device number (cf., bogus input from a mount command), it knows that the minor device is not in use by a block-special file, and it knows the vnode which the partition is mounted on is valid. To finish the operation, VFS sends a list of inode numbers to MFS for it to open in order to synchronize the inode tables. This way it is unnecessary to save the inode table externally before a restart. As we will show in Section 3.3.1, it is possible to store state in a shared memory region. This way, the buffer cache will be available to the updated FS, speeding up the reopening of inodes as it is not necessary to read them from disk.

Saving the inodes to disk and later reopening them also works for anonymous pipes. These pipes are all handled by the root MFS. Although they do not have a name like named pipes, they do have an inode number and block cache entry assigned. Upon shutdown of the root MFS, this inode and buffered data are simply written to disk. When the MFS comes back up, the pipe and data are made available again by opening the inode from disk. When the pipe is no longer in use, the inode is returned to the free inode list and disk space is available again.[3]

Finally, while an FS is being updated, requests targeted at that FS need to be queued. Afterwards, when the FS has finished updating, the queued requests are executed.

[3]It might be a good idea to introduce a separate pipe FS that handles all anonymous pipe operations. As we will see later, this could improve concurrency as the root-FS has less work to do. Such a project will be part of future work.

3.3 Resilience to FS crashes

The problem you face with a crashed FS is losing state (data written to cache and modified inode data) and leaving the file system on disk in an inconsistent state due to partial writes. Also, if VFS is not aware that an FS has crashed, it will try to send messages to a non-existing endpoint and panic.

A possible solution to the first problem could be using a journalling file system. However, we want our solution to be unrelated to the file system format used on disk. It is possible that in the future MINIX will support other file systems that also have no journal to recover from. So instead, we somehow need to restore state from the crashed FS and copy it to a newly started FS. After the state has been recovered, the partially executed request is reissued to fix any inconsistencies.

After an FS has crashed, VFS needs to detect that this has occurred. When an FS crashes, RS is signalled and spawns a new copy. Upon startup the FS sends an FS_READY message to VFS, which will subsequently suspend it because it has no work for it. At this point VFS has to find out that an FS has failed and which one specifically. To be able to do that RS should register the endpoint of an FS with DS and VFS should subscribe to it when it mounts an FS. The key is the initial endpoint of the FS and the value is the actual endpoint of the FS. When an FS starts for the first time, the key and value are the same. After an FS crashes or restarts, the value becomes the new endpoint of the newly started FS. When the value changes, VFS will be notified by DS and know *that* an FS has crashed and *which* one.

3.3.1 Shared memory regions

After an FS has been restarted, the state of the crashed FS needs to be restored. Virtual Memory (VM) can provide an elegant solution. Upon startup, an FS stores the inode table and block cache in shared memory and publishes the name of the region in DS. VFS tells the suspended FS it should restore state from a dead FS by sending a *recovery* request. The suspended FS then asks DS for the name of the memory region and subsequently maps in the shared memory after deleting its own shared memory region. This avoids the need to copy data and providing temporary access privileges to other process's address space (e.g., keeping the dead FS in memory until the inode table and block cache have been copied). This approach may also be used as an optimization for dynamic updates (e.g., to keep the buffer cache in memory). The current implementation of VM in MINIX lacks support for these shared memory regions and for `mprotect` (to mark parts of memory as read-only to protect it from being overwritten upon failure). Implementing these features is part of future work.

As we pointed out at the beginning of this chapter, we assume that the state is not corrupted by the crash. Nevertheless, it is important to check this is true. Otherwise we risk continuing to work with corrupt data and make the problem worse. When state turns out to be corrupt, the user should be notified and a file system check can be done in an attempt to fix inconsistencies.

State can be verified by computing checksum values over the inodes and blocks in the block cache after each successful request. During recovery, the checksum values are compared to the recovered state. If verification fails, sub-

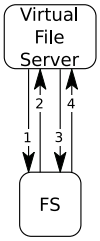


Figure 3.1: VFS/FS communication with transactions. 1) send request. 2) {ok, error} result value, reply message, possible auto-commit flag. 3) commit request to store result of previous request. 4) commit acknowledgement.

sequent system calls from processes that had inodes opened on the failed FS are forced to fail, by closing the associated vnodes in the vnode table, file descriptors, and unlock locks on file descriptors that are no longer opened.

3.3.2 Transactions

In most cases, a request failed to be executed completely, when an FS crashed. It is up to VFS to reissue the failed request. In order for VFS to do this, all requests should be queued (which is also necessary for dynamic updates). Each FS has its own queue. The head request is send and subsequently removed from the queue when the reply from the FS is received. When an FS fails (and thus no reply is received), VFS can resend the head request after the FS has been restored.

However, how do we know at what stage of the request the FS crashed? We can distinguish 3 crash scenarios:

- Crash before state is modified.
- Crash after state is modified.
- Crash after state is modified and written to disk.

The first case is no problem as we can simply reissue the request. In the second case it is very likely important data was changed leading to a different outcome when we retry the failed operation. The last situation is actually not that bad, because after all, it is exactly what we want. In all cases VFS has no way of telling which scenario occurred and retrying the failed request is not guaranteed to give the correct result.

Upon closer inspection of the code of MFS it turned out that 60% of the requests are idempotent[4] and can be reissued without a problem (see Tables 3.1 and 3.2). The remaining calls can be made idempotent by transforming them into transactions that need a *commit* call before they are actually executed.

[4] "An expression raised to the square or any higher power it gives itself as the result, it may be called idempotent." - Benjamin Peirce (1809-1880) in 1870 in American Journal of Mathematics (1881)

Idempotent	Non-idempotent
breadwrite	create
chmod	link
chown	mkdir
flush	mknod
fstatfs	mountpoint
ftrunc	newnode
getdents	putnode
inhibread	rename
lookup	rmdir
newdriver	slink
rdlink	unlink
readsuper	
readwrite	
stat	
sync	
unmount	
utime	

Table 3.1: Idempotent and non-idempotent VFS-MFS requests

Figure 3.1 shows how this would work. VFS sends a request to an FS which executes the request just like normal, except that it writes the result to temporary data structures. The FS then sends an OK message to VFS, which holds the result of the request. In turn, VFS sends COMMIT and the FS then commits the changes, which must be an atomic operation. After committing, the FS returns a COMMITTED message. In case an FS knows that a request is idempotent, it sets an auto-commit flag in the OK message, so part of the communication can be skipped as optimization. An OK message could also hold an error value. In that case no state has changed and the transaction is implicitly aborted.

If an FS crashes before the request is committed, no important data have been changed yet. The commit either fails or succeeds, but never partly because it is atomic. But what if the FS fails after committing the request and is not able to send a message to VFS to acknowledge it has committed the request? VFS will think the commit failed and will reissue the request, which will lead to a wrong outcome as the requests that need transactions are not idempotent. This problem can be solved by having VFS adding IDs to requests. The ID is recorded in the atomic commit. When an FS fails after committing and VFS finally reissues the failed request after the FS is recovered, the FS can tell if it already executed this request by checking the last committed request ID. In that case it sends ALREADY_COMMITTED.

Adding transactions and request IDs demands quite a few changes to the current VFS/MFS protocol. Also, the current message formats for the requests are for a large part congested and we do not want to add more message types to the system. However, we can make a few optimizations, so changes will be minimal [32]. The ID value can stay small as it is only used to differentiate between the current and previous non-idempotent request. In the current

Request	Possible issues on retry
create	inode erroneously marked in use, entry already exists
link	entry already exists on retry
mkdir	see create
mknod	see create
mountpoint	inode is already marked as mountpoint
newnode	inodes erroneously marked as in use
putnode	deallocate not allocated inode
rename	old name non-existent, new name already existing
rmdir	entry non-existent
slink	inodes erroneously marked as in use, entry already exists
unlink	see rmdir

Table 3.2: Problems that can arise when a non-idempotent request is issued for the second time.

VFS/MFS protocol the request type and result of a request are stored in the `m_type` field, which is an 32-bit integer. The request type is a small number and the result value is either OK (0) or a small, negative error value. It is therefore possible to split the `m_type` field in three parts; a 16 bit field holding the request result, a 15 bit field holding the request ID, and a 1 bit auto-commit flag. The VFS/FS protocol needs an additional request type to signal an FS to commit a transaction.

We should note that our transaction protocol only protects against crashes. We cannot tell whether an FS really committed a request or not (i.e., it reports it committed the request, but due to a bug it actually did not). We can distinguish the following scenarios:

1. FS commits request properly and returns COMMITTED.
2. FS does nothing and returns COMMITTED.
3. FS commits garbage and returns COMMITTED.
4. Same as 1-3, but FS does not return COMMITTED.

The second and third scenario can also occur with idempotent requests where the commit step is omitted. For example, reading data from a file is an idempotent request. The FS copies the read data directly to the process that issued the request and sends OK (and how many bytes were read) to VFS when done. However, VFS has absolutely no guarantee that the data was really copied. If the FS fails to acknowledge it committed the request (successfully or not), we end up in a denial-of-service scenario, because VFS will never continue to the next request. In this document we assume all operations execute successfully or fail due to a crash. Faulty behavior and Byzantine failures certainly deserve attention, but that remains future work.

3.3.3 Data structures temporarily in use

It must be noted that virtually each request requires an FS to read an inode from disk, do some work, and finally release the inode. If this release is not done,

it stays in use indefinitely and the system will not be able to cleanly unmount (and dirty inodes are never written to disk). For example, VFS sends a request to an FS, the FS opens and closes an inode to do its job, and sends a reply to VFS. If the FS crashes while the request is not yet finished (i.e., inode is still opened), and VFS retries the request after the FS has been recovered, the inode stays opened even when the request has been successfully executed. As a workaround we add a counter to the inode data structure telling how often an inode is in temporary use. Under normal circumstances these counters should be zero after the request has finished, except when an FS crashes. On a following inode release or unmount, we can tell that this inode should be released one more time (i.e., counter-value times more).

The failure resilience technique described above resembles Server State Regions (SSRs) by moving the state to a separate part of memory, using shared memory. The main difference is that the FS is responsible for the integrity of all state, whereas with SSRs only the mapped-in SSRs are vulnerable. However, some errors can take a while before they are noticed, so that multiple SSRs can still be corrupted. Even the ones not mapped-in at the moment the server crashes. Another difference is that SSRs are based on a server with multiple clients, where it makes sense to store client specific data at the client itself. In our case there are multiple servers (FSes) and just one client (VFS). It is not necessarily a problem, but that is not how the technique is intended to be used.

3.4 Request queueing

While an FS is not available due to a dynamic update or a failure, VFS cannot handle system calls designated to that FS. Instead, requests belonging to those system calls have to be put on a queue associated with that FS, so they can be handled later when the FS is available again. When a request is at the head of the queue, it is send to the FS. When the FS sends a reply, the request is removed from the queue and the next request in line can be send.

During a dynamic update, requests are simply appended to the queue. The same happens during a failure. However, also a recovery request is prepended to the queue and send to the (newly started) FS (i.e., handled with priority). When recovery has completed, the recovery request is removed from the queue and the next request (which was the head request before the recovery and probably caused the FS to crash) is send for the second time (i.e., before and after the crash).

3.5 Asynchrony

All communication between VFS and FSes is currently synchronous; VFS sends a message and blocks until a reply arrives. However, communication between an FS and the disk driver is also synchronous. When a disk driver crashes and an FS waits eternally for a reply, VFS is also unable to continue as it will not receive a reply from the FS. By making the communication between VFS and FSes asynchronous, VFS will not block on a failure of FS any more and can continue sending requests to other FSes. As a side effect, this could improve VFS's performance as multiple FSes can handle requests concurrently.

Asynchronous communication is also necessary to implement request queues (which, in turn, are needed for dynamic updates and failure resilience). As VFS appends requests to the queues, it cannot block for a reply (otherwise it would have no more than one request on a queue). A system call handler adds a request to the queue and suspends. When a reply to its request comes in, the handler is woken up, and it continues its work.

Using asynchronous communication, VFS can send requests to all FSes and does not block when an FS fails. However, this has considerable consequences for the structure of VFS. In the next chapter, two approaches, one based on threads and one based on continuations, will be explained in detail.

Chapter 4

Decoupling VFS and MFS

In this chapter we discuss techniques that enable the key requirements mentioned in the previous chapter:

- Make MFS dynamically updatable.
- Make VFS resilient to MFS crashes.
- Improve performance of VFS by introducing concurrency.

We have already explained how the first two requirements can be achieved using generic techniques. In this chapter we describe solutions that are required to implement these techniques and also make it possible to provide concurrency. We present and compare two approaches: a design based on threads and a design based on continuations.

Both approaches share a few properties; we will describe these in Section 4.1. In Section 4.2 we describe a way in which global variables are protected such that they cannot be accessed by multiple system calls concurrently. Section 4.3 describes both designs in detail. Finally, we present a comparison in Section 4.4.

4.1 General Request Handling

In general, when a process makes a system call to VFS, we can distinguish the following steps:

1. A process makes a system call to VFS,
2. VFS sends the request to an FS where the actual work is carried out,
3. FS returns a reply to VFS, and
4. VFS reports result to the originating process.

Accordingly, the structure of VFS will be as follows (see Figure 4.1). Either PM or a user process sends a request or makes a system call to VFS, respectively. VFS subsequently stores the sent message in the `fproc` entry of the process that made the call (either directly or indirectly through PM) or in some cases handles a PM-request immediately. A system call handler processes the request and when communication with an FS is required, a request is put on the queue

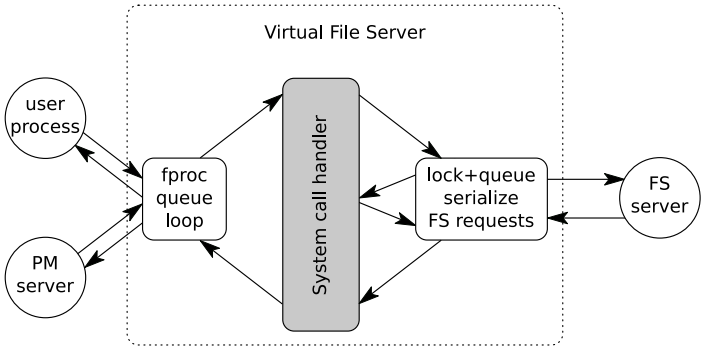


Figure 4.1: General code structure.

associated with that FS. While on the queue, the process is in a suspended state. When the FS sends a reply back, the main loop of VFS looks up to which request the reply belongs and continues the system call. When the system call is completed a reply is sent to the user process or PM.

Requests sent to VFS (on the left side in the figure) will be described in detail in Section 4.1.1. Communication between VFS and FSes (on the right side in the figure) is discussed in Section 4.1.2.

4.1.1 Processing Requests to VFS

In the current, synchronous VFS design, a message is received in the main loop and dispatched to the system call handler (e.g., `do_chmod` for the `chmod` request). If VFS has to communicate with an FS, it sends a message and waits for a reply. This is repeated until VFS is done processing the system call and it sends a reply to the originating process (see Figure 1.3).

In an asynchronous design, this is not possible as it is not guaranteed we will even receive a reply to our request, when we explicitly wait for a reply. We can receive no reply (because an FS might have crashed), replies for other requests, or even new requests from user processes. Instead, all messages, including replies from FSes are received in the main loop and processed from there. Consequently, we have to receive messages as fast as possible and store them somewhere, otherwise we will not be able to receive results from FSes.

As we have pointed out in Chapter 3, currently, a process can make at most only one call to VFS at any time. Or to be more precise, a process can make only one system call at a time. For example, a process blocked on a call it made to PM, cannot simultaneously make a system call to VFS. We can use this property to associate messages with a specific process and store them in the corresponding `fproc` table entry. Also replies from FSes are stored in the `fproc` entry of the original process that made the system call.

Approach	Requests	Local to VFS	On behalf of other process (signalled)
Handled immediately	fork, setgid, setsid, setuid	yes	no
Queued	exec	no	no
Queued or flagged	dumpcore, exit, unpause	no	yes
Special handling	reboot	no	no

Table 4.1: PM-VFS requests

Messages from PM

Messages from PM are an exception to this scheme (see Table 4.1 and Table A.2 in Appendix A). Some messages ask VFS to do just a little bookkeeping (`fork`, `setgid`, `setsid`, and `setuid`). They are executed *local to VFS* (require no FS communication) and can be handled immediately. Others require quite a bit of work on VFS's part and also might have to be postponed while an FS is not available (`dumpcore`, `exec`, `exit`, `reboot`, and `unpause`).

All requests from PM are stored in the `fproc` table entry of the originating process, except for requests that belong to *Handled immediately*; they are handled immediately and require no queueing. Processes that made a system call to PM cannot have requests outstanding at VFS at the same time and therefore there should be room to store PM-requests.

However, and this makes it tricky, requests belonging to *Queued or flagged* can be caused by a signal from another process (e.g., when the super user forces a program to quit). It is possible that the process which is signalled to `exit` could already be busy making a system call to VFS. In that case, there is no room left to store a message in the `fproc` table. *Queued or flagged* requests are handled as follows. In case the targeted process is not busy making a system call to VFS, they can be stored in the `fproc` entry. Otherwise they are stored as flags in the `fproc` entry, which are checked the moment the reply to that request comes in. `unpause` requires special attention as it is used to unblock a process that is waiting for a reply from an FS (so it makes no sense to wait for a reply and then unblock it). Instead of waiting for a reply, the main loop checks on each iteration for this flag and unpauses the process if it is set (the process receives EINTR).

`reboot` is special among PM-requests; it iterates over all processes exiting them, unmounts all mount points, and finally a reply is send back to PM. While the system is being shut down by `reboot`, other processes should not be able to make system calls (note that this was impossible in the synchronous design where only one system call at a time is handled). This is solved by setting an *exiting* flag in the process table that is checked each time a message comes in. If the flag is set, new calls from a flagged process are discarded. Also, while an FS is not available due to a dynamic update or a failure, `reboot` has to be postponed until the FS returns (and is recovered). The latter is also solved by setting a `reboot` flag and having the main loop check for it on each iteration. When the flag is set and all FSes are available, the reboot is executed.

4.1.2 Serializing requests to MFS

We have to be careful that system calls are not executed in an interleaved fashion, such that the outcome is different from what it would be if the system calls were executed in succession. For example (see Figure 4.2), an `unlink` and `stat` operation on the same file should not lead to the outcome where the `stat` request reports that the file has a zero link count. Either `stat` is executed before `unlink` and reports valid data, or it is executed after the unlink and returns an error message, stating that the file does not exist. In other words, it is necessary to serialize requests.[1]

OK		OK	
User A: stat(x)	User B: unlink(x)	User A: stat(x)	User B: unlink(x)
1. n = lookup(x)			1. n = lookup(x)
2. stat(n)			2. unlink(n)
	3. n = lookup(x)	3. n = lookup(x)	
	4. unlink(n)	4. report errno	

Not OK	
User A: stat(x)	User B: unlink(x)
1. n = lookup(x)	
	2. n = lookup(x)
	3. unlink(n)
4. stat(n)	

Figure 4.2: Possible execution orders of stat and unlink on the same file.

This is solved by putting requests on a queue that is associated with the FS it is targeted at. The queue is handled in FIFO order. We assume an FS can handle only one request at a time (which is currently true for MFS) and therefore this queue is enough to ensure serialized execution. However, not all system calls require communication. System calls directed at VFS can be categorized in four categories (see Table 4.2).

System calls with a path name argument *always* have to do a lookup first and are subsequently serialized. That way they do not alter global data that could influence other concurrent calls.

System calls that do not have an argument (or nothing file related) do not change global data in a way they can pose a threat to other calls (i.e., umask sets the creation mode of new files for a specific process and (f)sync writes the cache to disk).

The indirect calls from PM either make no changes to global data (i.e., set process specific values –`setsid`, `setgid`, `setuid`), atomically increase usage counters for file descriptors (`fork`), cancel a request for a specific process (`unpause`), or issue requests that end up on the queues or are harmless to other concurrent requests (`exit`, `dumpcore`, `reboot`, `exec`). The latter calls, except `exec`, actually lower usage counters for file descriptors that could be shared

[1] There is a fourth possibility where user B first looks up file x, then user A does the same and a stat on file x, and finally user B unlinks x. This way a correct result is achieved. However, interleaving allows for incorrect answers, while non-interleaving does not.

Category	System calls
System calls with a path name argument	mkdir, access, chmod, open, creat, mknod, chdir, chroot, unlink, utime, truncate, chown, mount, unmount, rename, link, slink, rdlink, stat, lstat
System calls with a file descriptor argument	lseek, read, write, close, fchdir, fchmod, fchown, pipe, fstat, fstatfs, ftruncate, dup, dup2, fcntl, select[2], getdents, ioctl
System calls with other or no arguments	umask, sync, fsync
Indirect through PM	setsid, setgid, setuid, fork, exit, dumpcore, unpause, reboot, exec

Table 4.2: System call categories directed at VFS

among more than one process. However, a process can never close a file descriptor for another process, so the file descriptor will not cease to exist while another process is still using it.

A problem arises with the order of requests for system calls with a file descriptor argument. When two system calls arrive at VFS that operate on the same file descriptor and `filp` object, the result can be subject to race conditions. For example, processes A and B share a file descriptor fd and process A writes 20 bytes to fd while B does a file position seek. The write request is on the queue and advances the file pointer x with 20 bytes when the write went okay to $x+20$ (actually, the FS tells how many bytes were written). In the mean time the seek request advances the file pointer with 10 bytes. That is, it takes the current position and adds 10 bytes ($cur + 10$). If these system calls were to be executed sequentially, the seek operation would result in $x+20+10 = x+30$. However, the write request has not yet finished, so the seek sets the file position to $x + 10$. When the write request finishes, the file position is set to $x + 20$, a different result. This is possible because the seek command requires no communication with an FS and consequently is not serialized by a queue. Serializing all requests prevents these problems.

The two approaches achieve serialization differently. Without going into much detail (the designs are discussed in detail in Section 4.3), in the threaded design a worker thread has to obtain an exclusive lock on a `vmnt` and releases it when it is done. As soon as the mutex is free, the next thread obtains a lock on it. Multiple threads trying to obtain a lock on the mutex constitute a queue. Internally, this is handled as a FIFO queue by the threading library to prevent starvation. While waiting on obtaining a lock, a thread is suspended.

In the continuations design each `vmnt` has a queue associated. A queue consists of a linked list of `fproc` entries. The head of the queue has an implicit, exclusive lock on the virtual mount point.

4.2 Locking model

In VFS there are multiple global variables; `vmnt` table, `vnode` table, `filp` table, `lock` table, `select` table, and `dmap` table. These variables need to be protected against concurrent access by multiple system calls, because otherwise we risk race conditions. In order to provide protection, each system call handler has to obtain an exclusive lock on a mount point, as all global variables can be linked one way or another to a `vmnt`. By granting exclusive access and serializing the requests, we can guarantee correct results. This rather coarse-grained locking model is required and enough to ensure protection. Making the locking model more fine-grained, will make the implementation much more complex, but gain little benefit. The time spent by VFS working on a system call is negligible compared to the time spent by FSes[32].

In general, a system call handler obtains an exclusive lock on a `vmnt` for exactly the time it requires exclusive access (possibly spanning multiple calls to an FS). By making sure only one request at a time is allowed exclusive access to a `vmnt` until it has completely finished its operations, it is not necessary to explicitly lock other data structures as they are implicitly locked.

A lookup (known as pathname resolution in the POSIX standard) requires special attention. A lookup might involve multiple FSes and can use any FS as starting point. If a lookup would lock all FSes it visits, we risk a deadlock situation when multiple lookups are done concurrently. For example, two lookups where one lookup starts at FS A and one at FS B and the former tries to enter FS B and the latter FS A. Both have to wait until the other has unlocked the `vmnt`, which is not going to happen. Therefore, when a lookup leaves a mount point and enters another one, it is removed from the queue of the former and added to the tail of the queue of the latter mount point. When the lookup is finished, the lock on the `vmnt` is kept, in order to maintain exclusive access such that the original system call can be fully executed.

In the time window between moving from mount point to mount point, anything could happen to the path the lookup is trying to resolve. For example, the access mode might get changed or a component of the path might be unlinked or renamed. If the lookup was atomic, this situation could never occur. The POSIX standard is unclear on this subject as to what to do. A possible workaround would be to add a lookup-mutex that is locked by the lookup procedure, followed by locking all `vmnt`s. The lookup can then take place safely. This mechanism is known as a barrier and would kill performance. We have decided that this is too minor a problem and leave it as is.

4.3 Threads vs. Continuations

In this section we describe the two approaches based on threads and continuations that implement the key requirements. The threaded design is based on previous work by David van Moolenbroek [32], who wrote a threading library that works in user-space and is none-preemptive. He used a very fine-grained locking model (separate locks for most global variables), which as mentioned earlier, is much too complex and not necessary in our design. The continuations design is based on work by Philip Homburg, who wanted to use continuations to prevent VFS from hanging after a driver crash (see Section 3.5)[22]. His

approach differs from ours as we handle incoming messages from and sending replies to processes centrally.

Threads	**Continuations**
Obtain lock on `vmnt`	Explicitly serialize state on FS calls
(May cause temporary thread suspension)	Add to `vmnt` queue to grab lock
`SENDA(FS request);`	`if(head of queue) SENDA(FS request);`
Suspend thread to await reply	Explicit return to main loop to await reply
Main loop does `RECV(ANY, &m_in);` if(sender == FS) lookup process that sent last request	
`vmnt` lock is still held	No interleaving possible; can remove request from queue
Resume thread by `fire_event()` with result	Call next continuation do_foo_x with result
(asynchronous)	(synchronous)
Continue thread execution	Deserialize continuation state and continue
Thread must explicitly unlock `vmnt` when done	Replace request at head of queue to keep lock
Next thread can send message on release of `vmnt` lock (another thread may be unblocked and get the lock)	(after callback) if(queue not empty) `SENDA(next FS request)`

Table 4.3: Overview of threads and continuations.

In Table 4.3 we present an overview on how both approaches work. In Section 4.3.1 we discuss the threaded design in detail, followed by the continuations design in Section 4.3.2.

4.3.1 Threaded design

In this section we describe a threaded design of VFS that provides dynamic updates and failure resilience. This threading model is based on a main thread which receives messages and subsequently spawns a worker thread for each message. The worker thread carries out the work and sends back a reply to the originating process. During operation, the worker thread can send a number of asynchronous messages (`SENDA()`) to FSes and suspend itself (waiting for an event). When the main thread receives a message from an FS, it looks up the worker thread the message belongs to and schedules it to run again by firing the event the worker thread is waiting for (`fire_event()`).

For example, to do a `chmod("/usr/bin/cc", mode)` we do the following. First, we have to find the inode of ***cc***. The lookup routine obtains a lock on the root-FS and issues a lookup request to this FS by calling `sendrec`. Subsequently, `sendrec` asynchronously sends the message to the FS and schedules to wait for the reply by calling `event_wait(&fp->fp_event)`. Afterwards, control is

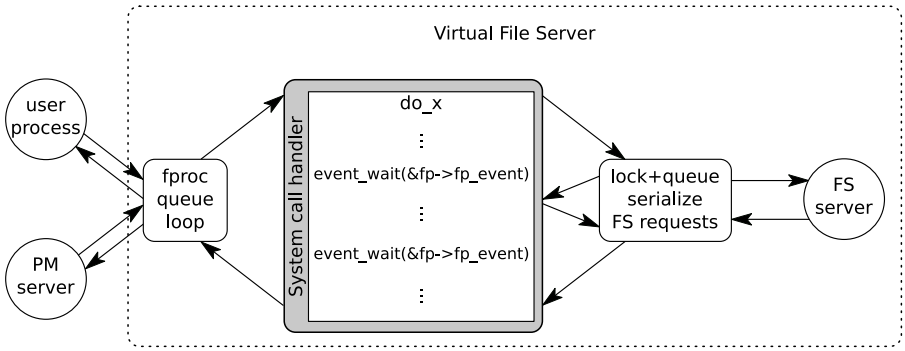


Figure 4.3: Code structure threads.

switched to another thread. Each thread has part of the stack allocated to it, so there is no need to explicitly serialize and deserialize state. When the main thread receives the reply from the FS, it figures out which worker thread handles the `chmod` call, and calls `event_fire(&rfp->fp_event)` to fire the event. At this point, the worker thread is scheduled to run again. Once running, the result of the lookup request is that we have to enter the /usr-FS and continue the lookup there, using the same method described above. When it has finally found the inode belonging to ***cc*** and permissions are all right, the worker thread can issue the actual `chmod` request to the /usr-FS. Subsequently, it waits for the event that a reply is received and reports the result to the user process when it is finished.

Under heavy load it is unfeasible to spawn a thread for each process, because that could easily exhaust stack space (or lead to excessive memory usage if virtual memory is used). The threading library allows to use fewer worker threads than processes by using a `store_t` data structure, that stores all data a thread needs to be able to start running. This enables VFS to use MAX_THREADS and create a queue of pending work that is processed by available worker threads. A worker thread is available when it has completely finished a system call (i.e., a reply is sent back to the originating process).

As described in our initial observations, a process can have at most one system call outstanding at once. This makes it possible to bind a thread to a process and store associated data in the `fproc` entry. At some point in the future MINIX is going to have kernel threads and processes will be able to do multiple system calls. For VFS to support multiple system calls from a single process, it should keep an `fthread` table inside the `fproc` entry with room for n threads.[3] Multiple worker threads for the same process do not compete with each other as they are serialized. Hence there is not need to apply mutual exclusion for shared data structures. While the table is full, subsequent system calls should be ignored until room is available again. A reasonable value for n and other design changes that are required, are to be considered in the future.

A worker thread is reserved specifically for requests from PM. This ensures

[3]System processes in MINIX maintain state in static data structures to prevent bugs.

messages from PM are always handled, no matter how many threads are busy (although PM does not require *immediate* response, it does expect *fast* response). Requests that are on behalf of other processes, are handed over to worker threads associated with that process. The other requests are handled by the PM-thread.

As we pointed out in the locking model in Section 4.2 it is not necessary to use a fine-grained locking model as all variables are protected by exclusive access to a `vmnt`. There is, however, one exception when using threads. During development of the threading library and implementation of the library in VFS, the author tried to minimize memory usage on the stack, and found that during `exec` calls memory usage peaked at 4 kilobytes whereas 1 kilobyte would suffice for the other system calls [32]. This was caused by a 4 kilobyte buffer put on the stack by the `patch_stack` routine.[4] This was solved by making this buffer static, which essentially makes it a global variable. Consequently, this buffer needs protection against concurrent access. A global `exec` mutex was introduced to provide exclusive access and covers all code of the `exec` implementation. We extend our locking model by letting a thread obtain a lock on the `exec` mutex when it executes the `exec` system call.

Serialization

We pointed out in Section 4.1.2 that it is necessary to serialize *all* requests, even when they do not need communication with an FS. To accomplish this in the threaded design, a thread must obtain a lock on a `vmnt`.

A thread releases the lock it obtained, when it has fully executed the system call. Consequently, no interleaving can take place.

4.3.2 Design based on Continuations

In the current version of VFS, each system call handler consists of a `do_foo` routine and some helper routines. Requests to an FS are carried out by `req_foo` routines that will return the result of the request. A `do_foo` routine can consist of multiple requests to FSes. For example, let us consider `chmod("/usr/bin/cc", mode)` again. We first need to find the inode of ***cc***. We start at the root of the file system, move to ***/usr*** which happens to be mounted on a different FS, and finally find the inode of ***cc***. This required two messages. If everything went well and permissions to alter are all right, we can execute the actual call. To do this call asynchronously, we need to split *chmod* in three parts:

1. do_chmod: start of call → lookup inode.
2. do_chmod_1: inode found → execute mode change.
3. do_chmod_2: mode changed → handle result.

As a message is sent asynchronously, the call will return immediately. When the FS is done processing the request, it sends a reply to VFS. VFS handles the message as every other message, so it will only start processing it when it

[4] `patch_stack` is called when the file-to-be-executed is in fact an interpreted script. In that case, the interpreter to which the script references should be executed with the path name of the script as argument instead.

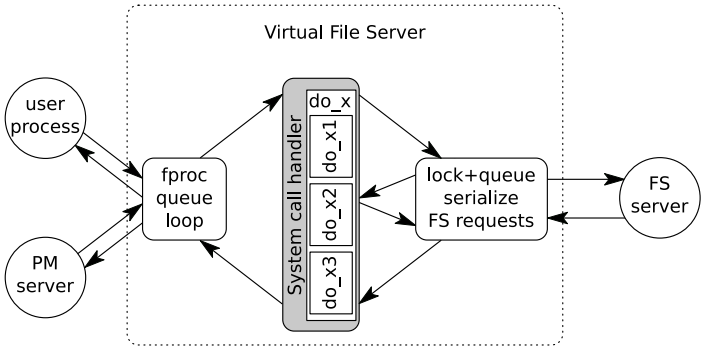


Figure 4.4: Code structure continuations.

has returned to its main loop (all messages are handled centrally). VFS detects the reply from the FS, looks up which call and process the reply belongs to, and continues the call at the next part (note that this is the same for threads). Clearly, the structure of `do_foo` with some helping routines cannot be used anymore.

In the continuations design, a system call to VFS is separated into sub-requests where each sub-request sends a message asynchronously to an FS. With each call, progress has to be recorded, so VFS can lookup where it has to continue. Each step should have an accompanying routine. See lines 13-18 of Listing B.2 in Appendix B for an example on how this is done.

Before a message is sent, all state that is required to continue the system call after the reply from the FS is received, has to be serialized. After receiving the reply, that state is deserialized again. For example, see lines 62 and 63 (serializing) and line 80 (deserializing). This data is stored in a *request state* table and a pointer to the table entry is recorded in the `fproc` entry to which the state belongs. For continuations to be able to handle multiple system calls from one process (to support multithreaded programs in the future), each `fproc` entry should keep a list of pointers to the request state table entries, instead of just one pointer. This way it can keep track of multiple requests per process.

By using a statically sized table there is a limit to how many system calls we can handle at the same time. When that limit is reached, new incoming system calls are suspended. One entry in the table is reserved for PM, so it can always be serviced (cf., a dedicated thread in the threaded design).

Serialization

Similar to the threaded design, all requests need to be serialized here as well. Requests are explicitly serialized by appending them to a queue that is associated with the FS. When the system call requires additional communication with the same FS, the request at the head of the queue is overwritten with the new request and resend (overwritten, because at this point the head request is not yet removed from the queue). Only the head request is being handled until it is finished. When the system call has completed its job, the request is

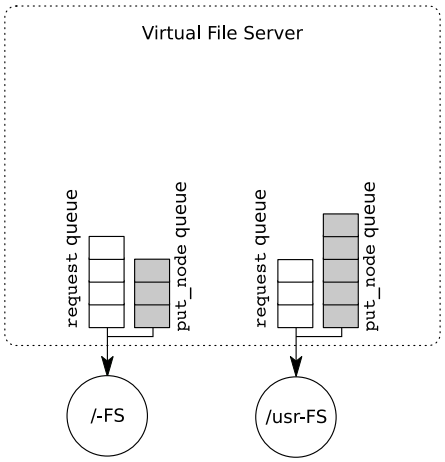


Figure 4.5: Each FS has its own request queue and put_node queue associated. The request queue consists of a list of pointers to `fproc` entries. The put_node queue consists of a list of pointers to `vnode`s and is handled with priority.

removed from the head of the queue, allowing the next request on the queue to be executed. This way interleaving is impossible.

To make sure all requests are properly serialized (also taking into account the system calls that do not require communication) each `do_foo` routine starts with a dummy step (see line 14). However, when no message is send to an FS, VFS will never receive a reply in the main loop to continue the system call. This is solved by having the `get_work` procedure in the main loop check if the head of a queue is a dummy request. If one exists, it is handled first, otherwise it will block on an incoming message.

One of the most often executed requests is `req_putnode` and needs special treatment because of this. That is, if we would treat this (sub)request like any other request, the number of states a system call can have, would explode (note that each sub-request is separated by a communication step). It would make the code extremely complex and probably slow. Moreover, `put_vnode` in VFS is a `void` function; the request always returns OK, so we are not interested in results. To solve this problem we introduce a putnode queue specifically for `req_putnode` requests. If the queue is empty and a request is put on the queue, it is executed asynchronously right away. If it is not empty, the request is simply added to the queue. In both cases the function returns almost immediately. Of course we now risk race conditions due to inodes not being put yet and new requests for the MFS coming in. This is remedied by making sure that the putnode queue, which is associated with the FS a request is targeted at, is empty. If it is not empty, then that queue should be handled first; it has priority over the normal request queue.

Request interdependencies

Not all requests to an FS immediately lead to results that can be used by VFS. For example, in the `do_chmod` routine the first step consists of looking up the inode of the targeted file. Often this involves several FSes. The lookup routine sends a message, checks if it needs to change to another FS (crossing mount points), and issues another message to that FS. However, it just set the call state to CHMOD_LOOKUP with accompanying parameters. So after sending one message by the lookup routine, VFS will continue with `do_chmod_lookup` while the request might not be finished yet. Somehow, we need to keep "stuck" in the lookup routine before returning to the chmod routine.

This is solved by setting the call state to DO_LOOKUP and add a *next_state* field to the request state table. The sole purpose of the routine associated with the DO_LOOKUP state, is to keep issuing `req_lookup` requests until the result is OK or an error message, after being set up properly by helping routines. When it receives the result it is looking for, the current state can be changed into *next_state* and the current request at the head of the queue is *restarted*.

4.4 Comparison

Regarding our key requirements (see beginning of this chapter), both designs are not that different. Initially we thought that the threaded design would have much more difficulty handling failures. However, after realizing we can simply resend a failed request after restoration, there is no need to keep much state in the threaded design. Also, if a request keeps failing and we have to abort the complete call, both approaches have the availability over the same data to undo changes.

There are, however, a few important differences. The threaded design must obtain an exclusive lock explicitly, whereas continuations obtains this lock implicitly. Another difference is that continuations must insert additional requests at the head of the queue (overwriting the former head) to keep this lock. Both designs must explicitly unlock; threads unlock the mutex and continuations remove the head of the queue and send the next request, if there is one. Continuations must serialize and deserialize state before and after communication. Threads just store state on the stack. Finally, to serialize all requests, threads simply obtain a lock, while continuations must use a dummy step. Because this dummy step does not involve communication, the main loop must check on each iteration for the existence of a dummy step on one of the queues.

By using threading, we can stick for a large part to the current design of VFS which gives us a clear advantage; we will not have to rewrite big portions of the code to make it adhere to a new design. Moreover, most changes can be hidden inside a communication layer, keeping existing code fairly readable. The changes that are most visible in the code, are locking and unlocking `vmnt`s. This is certainly not the case with continuations. All of VFS requires a major overhaul, which will not result in very readable code. Also, VFS is already complex as it is, continuations would make it really hard for future developers to understand what is going on where, and how to make adjustments.

Both approaches introduce asynchrony to decouple VFS and MFS. However, the code becomes much harder to test and debug. Both designs can introduce

	Threading	Continuations
Code readability	+	−−
Complexity	+	−
Engineering effort	+	−−
Performance	+	+
Support for kernel threads	+	+
Memory footprint	−	−
"Debuggability"	−	−

Table 4.4: Property comparison

timing issues which are hard to solve. The threaded design keeps much of the normal program flow intact, whereas continuations jump from routine to routine each time communication is necessary. This leads to an explosion of states in which a system call can occur. Technically the same operations are executed, but it is much harder to keep an overview of how all involved routines cooperate.

We pointed out that our coarse-grained locking model is sufficient in order to protect access to global variables in VFS. Fine-grained locking (i.e., lock variables such as `filp`s and `vnode`s individually) does not offer us any advantages. Moreover, with continuations it is not even feasible. As there is only one thread of execution, it does not make sense to use locks. These locks would merely function as a means to choose a job from a list of jobs (i.e., pick one that is not locked). If implemented, we would just try to mimic threads in a very complicated way. Fine-grained locking would cause for even more subdivisions (similar to `put_node` requests), as each access to a global variable is separated.

A comparison of the code of the two approaches can be found in Appendix B where we show example implementations of the `chmod` and `read` system calls and the lookup procedure. The first thing one notices, is the amount of code required for continuations. Due to the separation of code into multiple functions, structure is lost. The code is less elegant and there is a lot of additional code that deals with serializing and deserializing state. Reduced code readability and more lines of code, are bound to contain more bugs. The threaded design does require some additional code for locking and unlocking `vmnt`s, but not near as much as continuations do. Also, multiple worker threads are easier to grasp compared to a single thread of execution in the continuations design, that constantly switches between subsets of requests.

Comparison of properties We have compared both designs in Table 4.4. Code readability suffers greatly when using continuations, while the threaded design remains almost the same as the synchronous version of VFS. Continuations require a major overhaul of the current VFS, while making it threaded will take considerably less time as there are fewer changes to make.

Performance-wise both designs are very similar. The locking model and communication protocol are the same for both. More importantly, they achieve the same level of concurrency. The time VFS spends working on a system call, is negligible compared to the time spent by FS and the disk driver [32].

When kernel threads are implemented, VFS needs to be altered in such a way that it can handle multiple system calls from a single process. Both designs

can be easily adapted to this requirement.

The threaded design and continuations design have a larger footprint than the current, synchronous VFS. Each thread in the threaded design requires 1 kbyte of memory on the stack and continuations need memory to serialize and deserialize state.

Due to concurrency, both designs are hard to debug as they introduce timing issues.

Chapter 5

Summary and Conclusion

The method we describe in this document to implement dynamic updates for VFS, classifies as a program with built-in update mechanisms. Our technique avoids versioning by writing dirty cache to disk, start a copy of the new version of the binary, and then reopen inodes that were open, returning it to the previous state. Because we start a fresh copy of the binary and restore state from disk, we have the freedom to change virtually every aspect of the FS.

Resilience to failures is achieved by storing state externally in shared memory. Upon failure, a new copy of the FS is started, the shared memory region used by the failed FS is mapped in, and the state is verified. State verification involves computing checksums over the inodes and blocks in the block cache and comparing these with the stored checksums. When inconsistencies are found, a file system check is done and the problem has to be reported to the user.

The above can be implemented using synchronous communication. However, we need asynchronous communication to decouple VFS and MFS. Asynchronous communication makes it possible to queue requests and handle them when an FS is available. This way, VFS can process all system calls up to the point when communication is needed with an unavailable FS, instead of having VFS block upon update or failure of an FS (either a failure of the FS itself or indirectly through a disk driver crash). Also, asynchronous communication allows VFS to instruct multiple FSes to handle requests concurrently. To prevent these requests from interleaving and accessing global variables simultaneously, they need to obtain a lock on a `vmnt` and be serialized.

We have introduced two designs to implement asynchronous communication; one based on threads and one based on continuations. The threaded design takes less effort to implement, is easier to grasp, and the code is less complex and more structured than the continuations design. Therefore, we believe that this design is preferable to the design based on continuations.

Current Status and Future Work We have implemented a prototype version of dynamic updates in the synchronous version of VFS. It works, but has some serious drawbacks compared to the technique we describe in this document. The synchronous VFS has a very simple queueing mechanism, which is much less flexible in its ability to suspend requests (i.e., we had to suspend a process upon an incoming request, as we had no way of telling beforehand *if* the request was going to communicate, and with *which* FS that would be, instead

of suspending a request when it cannot be handled).

In order to be able to implement our failure resilience technique, we need proper support for Virtual Memory in Minix. Currently, we lack support for `mprotect` and shared memory.

A threaded version of VFS was implemented by David van Moolenbroek [32]. We implemented part of a prototype of the continuations design. It required a lot of effort to implement just a few requests, which led us to redesign VFS using threads.

Based on the above conclusions, our implementation of VFS using threads is scheduled for the near future.

We stated in Section 1.1 FSes are considered a special case of drivers, that are not trusted. Recently, support was added to Minix to provide end-to-end integrity and silent data corruption protection for block-device drivers. Ideally something similar would be devised for FSes, as currently VFS has no way of verifying whether FSes do what they say they do (see transaction protocol in Section 3.3).

We apply asynchrony to decouple VFS and MFS. As a side effect, VFS's performance is improved due to concurrency. There are, however, more ways to speed up certain operations. For example, we could cache the results of lookups (directory entries). Consequently, an inode stays opened and thus in memory, even when it is not used by any process. When the `vnode` table fills up, the oldest cache entries are removed, allowing for new inodes to be opened.

As we mentioned in Section 3.2, it is probably useful to introduce a pipe-FS that handles all anonymous pipe operations. These operations are currently handled by the root-FS, creating a bottleneck (e.g., many lookups start at or pass through the root-FS, too). A pipe-FS could relieve the load of the root-FS. Other FSes can be simplified as they do not have to be able to handle these pipes anymore.

Bibliography

[1] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: online patches and updates for security. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005. USENIX Association.

[2] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *2009 ACM SIGOPS EuroSys Conference on Computer Systems*, April 2009.

[3] Wendy Bartlett, Ieee Computer Society, and Lisa Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing*, 1:2004, 2004.

[4] A. Baumann. *Dynamic Update for Operating Systems.* PhD thesis, University of New South Wales, Sydney, Australia, Aug. 2007.

[5] A. Baumann, J. Appavoo, R. W. Wisniewski, D. Da Silva, O. Krieger, and G.t Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proc. of USENIX'07*, pages 279–291, June 2007.

[6] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, Apr. 2005.

[7] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The rio file cache: Surviving operating system crashes. In *Proc. 7th Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, 1996.

[8] R. C. Daley and J. B. Dennis. Virtual memory, processes, and sharing in multics. *Commun. ACM*, 11(5):306–312, 1968.

[9] F. M. David, J. C. Carlyle, E. M. Chan, P. A. Reames, and R. H. Campbell. Improving dependability by revisiting operating system design. In *Proc. 3rd Workshop on Hot Topics in Dependability*, pages 58–63, June 2007.

[10] R. S. Fabry. Capability-based addressing. *Commun. ACM*, 17(7):403–412, 1974.

[11] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 470–476, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[12] B. Geröfi. Minix vfs: Design and implementation of the minix virtual file system. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, August 2007.

[13] S. Gilmore, D. Kírlí, and C. Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, 1997.

[14] H. Goullon, R. Isle, and K. P. Lohr. Dynamic restructuring in an experimental operating system. *IEEE Trans. Softw. Eng.*, 4(4):298–307, 1978.

[15] D. Gupta and P. Jalote. On-line software version change using state transfer between processes. *Softw. Pract. Exper.*, 23(9):949–964, 1993.

[16] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in quicksilver. *ACM Transactions on Computer Systems*, 6:82–108, 1988.

[17] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *Proc. 37th Conf. on Dependable Systems and Networks*, pages 41–50, June 2007.

[18] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Construction of a highly dependable operating system. In *Proc. 6th European Dependable Computing Conf.*, pages 3–12, Oct. 2006.

[19] J. N. Herder, D. C. van Moolenbroek, R. Appuswamy, B. Wu, B. Gras, and A. S. Tanenbaum. Dealing with driver failures in the storage stack. In *Submitted for publication 4th Latin-American Symposium on Dependable Computing*, Sept. 2009.

[20] M. Hicks. *Dynamic Software Updating*. PhD thesis, University of Pennsylvania, 2001.

[21] Gísli Hjálmtýsson and Robert Gray. Dynamic c++ classes: a lightweight mechanism to update code in a running program. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 1998. USENIX Association.

[22] Philip Homburg. Asynchronous VFS. Available in src.20080414.asynchvfs branch of the Minix SVN repository.

[23] I. Lee. *Dymos: a dynamic modification system*. PhD thesis, The University of Wisconsin - Madison, 1983.

[24] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. *SIGPLAN Not.*, 41(6):72–83, 2006.

[25] A. Orso, A. Rao, and M. Harrold. A technique for dynamic updating of java software. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 649, Washington, DC, USA, 2002. IEEE Computer Society.

[26] D. Pescovitz. Monsters in a box. *Wired*, 8(12):341–437, 2000.

[27] T. Ritzau and J. Andersson. Dynamic deployment of java applications. In *Java for Embedded Systems Workshop*, May 2000.

[28] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, 1993.

[29] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3:204–226, 1985.

[30] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Trans. on Comp. Syst.*, 24(4):333–360, November 2006.

[31] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. on Comp. Syst.*, 23(1):77–110, Feb. 2005.

[32] D. van Moolenbroek. Multimedia support for minix 3. Master's thesis, Vrije Universiteit, September 2007.

[33] Matthew J. Weinstein, Jr. Thomas W. Page, Brian K. Livezey, and Gerald J. Popek. Transactions and synchronization in a distributed operating system. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*, pages 115–126, New York, NY, USA, 1985. ACM.

[34] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Networked windows nt system field failure data analysis. In *PRDC '99: Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing*, page 178, Washington, DC, USA, 1999. IEEE Computer Society.

Appendix A

Request listing

This appendix lists the requests that VFS sends to MFS and PM to VFS.

A.1 VFS-MFS requests

Table A.1 gives an overview of the requests sent by VFS to FSes. The handling of these messages upon failure of an FS is discussed in Section 3.3.2.

Request	Operation
breadwrite	fetches disk block, reads/writes data from/to those blocks
chmod	gets inode, sets mode to predetermined value
chown	gets inode, sets owner to predetermined value
create	allocates free inode, creates directory entry, and initializes inode
flush	writes dirty blocks to disk, flags blocks as clean after write, invalidates blocks (i.e., need to be fetched from disk after new read)
fstatfs	retrieves info on the file system
ftrunc	wipes part of inode
getdents	copies entries of a directory to user buffer
inhibread	sets a flag to stop read ahead
link	adds an entry to a directory
lookup	looks up the inode belonging to a pathname
mkdir	creates inode (see create), adds . and .. entries
mknod	creates inode (see create)
mountpoint	looks up if an FS can be mounted on an inode, marks inode as mountpoint afterwards
newdriver	updates endpoint value with predetermined value
newnode	allocates free inode
putnode	marks inode not in use, deallocates inode
rdlink	reads contents of a soft link
readsuper	reads superblock for valid file system
readwrite	fetches/acquires disk block, reads/writes data (on partial write acquired blocks are reused and overwritten)

rename	adds new directory entry, removes old directory entry
rmdir	removes directory entry
slink	creates inode, adds directory entry
stat	reads inode data
sync	writes dirty inodes to block cache, see flush
unlink	see rmdir
unmount	writes root inode to disk, see flush
utime	updates file dates

Table A.1: VFS-MFS requests

A.2 PM-VFS requests

Table A.2 shows an overview of requests PM sends to VFS. How these messages are handled upon dynamic update or failure of an FS is discussed in Section 4.1.1.

Request	Operation
dumpcore	writes dump of program image in memory to disk and stops process
exec	executes program
exit	stops process
fork	fork a child process
reboot	exits all processes and unmounts all mount points
setgid	set group ID for process
setsid	make a process session leader
setuid	set user ID for process
unpause	interrupt pending request

Table A.2: PM-VFS requests

Appendix B

Design comparison

In this section we show two examples of system calls and the lookup procedure when implemented using the designs described in this document. All examples of the threaded design are for a large part based on results by [32].

We will not explain the code in detail. That is left to the reader as an exercise.

B.1 chmod

The `chmod` system call changes the permission of a file (read, write, and execute settings for the owner, group members, and everyone else). Arguments for the call are a path name and the desired permissions mode. First the path name is resolved to a vnode, then permissions are verified to make sure the user is allowed to modify the file permissions (owner and super user are allowed), and finally a request is sent to the FS to make the change when everything is okay.

The examples shown here, differ slightly from the actual implementations. The `fchmod` call provides a file descriptor which can easily be resolved to a `vnode`. Normally, the `chmod` and `fchmod` calls are combined and a test is done to find out which call was made to do either a lookup for a `vnode` or just take it from the file descriptor. We have left out the implementation of `fchmod` to make the example more readable.

Listing B.1: chmod system call using threads

```
1  PUBLIC int do_chmod() {
2    char fullpath[PATH_MAX];
3    struct chmod_req req;
4    struct lookup_req lookup_req;
5    struct node_details res;
6    struct vmnt *vmp;
7    struct vnode *vp;
8    int r, ch_mode;
9
10   if(fetch_name(fullpath, m_in.name, m_in.name_length, M3) != OK)
11     return(err_code);
12
13   /* Fill in lookup request fields */
```

```
14    lookup_req.path = fullpath;
15    lookup_req.lastc = NULL;
16    lookup_req.flags = EAT_PATH;
17
18    /* Request lookup */
19    if ((r = lookup(&lookup_req, &vmp, &res)) != OK) return r;
20
21    req.fs_e = res.fs_e;
22    req.inode_nr = res.inode_nr;
23
24    /* Find vnode, if there is an entry for it. */
25    vp = find_vnode(req.fs_e, req.inode_nr);
26
27    /* Only the owner or the super_user may change the mode of a file. No
28     * one may change the mode of a file on a read-only file system. */
29    if (vp->v_uid != uid && uid != SU_UID)
30      r = EPERM;
31    else
32      r = read_only(vp);
33
34    if(r != OK) {
35      unlock_vmnt(vmp);
36      return r;
37    }
38
39    /* Clear setgid bit if file is not in caller's grp. */
40    if (fp->fp_effuid != SU_UID && vp->v_gid != fp->fp_effgid)
41      m_in.mode &= ~I_SET_GID_BIT;
42
43    /* Fill in request message fields.*/
44    req.uid = fp->fp_effuid;
45    req.gid = fp->fp_effgid;
46    req.rmode = m_in.mode;
47
48    /* Issue request */
49    r = req_chmod(&req, &ch_mode);
50
51    /* Update and unlock the corresponding vnode, if any. */
52    if (vp != NIL_VNODE) {
53      if (r == OK) vp->v_mode = ch_mode;
54      put_vnode(vp);
55    }
56
57    /* Unlock the file system if necessary */
58    unlock_vmnt(vmp);
59
60    return r;
61  }
```

The threaded design does not differ much from the original synchronous version. The lookup procedure obtains a exclusive lock on the `vmnt` and tries to find the `inode` belonging to the pathname. It performs some sanity checks and subsequently issues the `chmod` request to the FS, followed by an unlock of the `vmnt` lock that was obtained by the lookup procedure. Except for the code

related to locking the data structures, the procedure is exactly the same as the original version.

Now let us look at the equivalent using continuations.

Listing B.2: chmod system call using continuations design

```
1  PUBLIC int do_chmod() {
2    int r;
3    struct fproc *rfp;
4    struct vmnt *vmntp;
5
6    rfp = get_proc();                /* Get fproc entry to original proc */
7    vmntp = get_vmnt();               /* Get pointer to vmnt entry of FS */
8    rfp->REQ_STATE.call_nr = call_nr;                  /* System call nr */
9
10   if(rfp->REQ_STATE.call_state == NULL_STATE)
11     rfp->REQ_STATE.call_state = CHMOD_START;
12
13   switch(rfp->REQ_STATE.call_state) {
14     case CHMOD_START:  r = dummy(CHMOD_INIT, ROOT_FS_E); break;
15     case CHMOD_INIT:   r = do_chmod_init();              break;
16     case CHMOD_LOOKUP: r = do_chmod_lookup();            break;
17     case CHMOD_MODE:   r = do_chmod_mode();              break;
18   }
19
20   if(r != SUSPEND) {           /* An error occured or we have a result */
21     vmntp->m_state = MS_IDLE;                          /* Reset vmnt and */
22     rfp->REQ_STATE.call_state = NULL_STATE;           /* request state */
23     dequeue_req(vmntp, rfp);              /* Remove request from queue */
24     revive(rfp->fp_endpoint, r);                       /* Unsuspend proc */
25   }
26
27   /* Execute next request on queue */
28   while(WORK_LEFT(vmntp)) restart_req(vmntp, TRUE);
29
30   return r;
31 }
32
33 PUBLIC int dummy(int call_state, int fs_e) {
34   struct fproc *rfp;
35   struct vmnt *vmntp;
36
37   rfp = get_proc();                /* Get fproc entry to original proc */
38   vmntp = find_vmnt(fs_e);          /* Get pointer to vmnt entry of FS */
39
40   rfp->REQ_STATE.call_state = call_state;         /* Where to continue */
41   vmntp->dummy_requests++;           /* Increase dummy request counter */
42   rfp->fp_message = m_in;             /* Store message for later usage */
43   enqueue_req(fs_e, rfp);            /* Put dummy request on the queue */
44
45   suspend(XFS);                 /* Suspend process on FS communication */
46   return SUSPEND;
47 }
48
49 PUBLIC int do_chmod_init() {
```

```
50    struct fproc *rfp;
51    int r;
52
53    rfp = get_proc();               /* Get fproc entry to original proc */
54
55    /* Fetch path name from user space (or message) and store it in request
56     * state for this process */
57    if (fetch_name(rfp, rfp->fp_message.name, rfp->fp_message.name_length,
58                        rfp->fp_message.pathname) != OK) {
59      return(err_code);
60    }
61
62    rfp->REQ_STATE.CHMOD.rs_mode = rfp->fp_message.mode;
63    rfp->REQ_STATE.next_state = CHMOD_LOOKUP; /* State to go to */
64
65    /* Request lookup */
66    r = lookup(rfp, 0 /*flags*/, FALSE /*!use_realuid*/);
67    if (r != OK) return r;
68
69    return SUSPEND;
70  }
71
72  PUBLIC int do_chmod_lookup() {
73    struct fproc *rfp;
74    int r;
75    uid_t uid;
76    gid_t gid;
77    struct vnode *vp;
78
79    rfp = get_proc();               /* Get fproc entry to original proc */
80    vp = rfp->REQ_STATE.LOOKUP.rs_vnode;              /* Resolved vnode */
81    uid = rfp->fp_effuid;
82    gid = rfp->fp_effgid;
83
84    /* Only the owner or the super_user may change the mode of a file. No
85     * one may change the mode of a file on a read-only file system. */
86    if (vp->v_uid != uid && uid != SU_UID)
87      r = EPERM;
88    else
89      r = read_only(vp);
90
91    /* If error, return inode. */
92    if (r != OK)  {
93      put_vnode(vp);
94      rfp->REQ_STATE.LOOKUP.rs_vnode = NULL; /* Reset */
95      return(r);
96    }
97
98    /* Clear setgid bit if file is not in caller's grp. */
99    if (uid != SU_UID && vp->v_gid != gid)
100     rfp->REQ_STATE.CHMOD.rs_mode &= ~I_SET_GID_BIT;
101
102   /* Issue request. Note that we don't have to enqueue. We simply replace
103    * the head of the queue (previous lookup) with a new request. */
```

```
104   req_chmod(vp->v_vmnt, vp->v_inode_nr, rfp->REQ_STATE.CHMOD.rs_mode);
105   return SUSPEND;
106 }
107
108 PUBLIC int do_chmod_mode() {
109   struct fproc *rfp;
110   struct vmnt *vmntp;
111   int r;
112   struct vnode *vp;
113
114   rfp = get_proc();                /* Get fproc entry to original proc */
115   vmntp = find_vmnt(fs_e);          /* Get pointer to vmnt entry of FS */
116   r = m_in.RESULT;
117
118   vp = rfp->REQ_STATE.LOOKUP.rs_vnode; /* Result of last lookup */
119   rfp->REQ_STATE.LOOKUP.rs_vnode = NULL; /* Reset */
120
121   if (r == OK) vp->v_mode = m_in.RES_MODE;
122   put_vnode(vp);
123
124   return r;
125 }
```

In Listing B.2 `dummy` puts a dummy request on the queue of the root-FS to serialize the request. The root-FS is used as we do not know at this point, on which FS the file is located. Then `do_chmod_init` retrieves the path name argument from the originating process and issues a lookup request. When the lookup finally results in a `vnode` we arrive at `do_chmod_lookup` which checks permission to change the file mode. Then it issues a `chmod` request to the FS holding the file. When the FS sends the reply we finalize in `do_chmod_mode` by registering the new mode in the `vnode`, putting the `vnode` retrieved by the lookup process, and telling the originating process everything went well.

B.2 lookup

The lookup procedure in VFS is very complicated. The procedure is fed a pathname and some other details and it has to find the FS and inode number that hold the designated file. The lookup can start from any FS, move to other FSes, and visit an FS multiple times.

Listing B.3: lookup using threads

```
1  PUBLIC int lookup(lookup_req, vmpp, node, pathrem)
2  lookup_req_t *lookup_req;
3  struct vmnt **vmpp;
4  node_details_t *node;
5  char **pathrem;
6  {
7    struct vmnt *vmp, *old_vmp;
8    struct vnode *start_node;
9    struct lookup_res res;
10   int r = 0;
11   int symloop = 0;
```

```
12    int path_processed = 0;
13
14    /*Make a copy of the request so that the original values will be kept*/
15    struct lookup_req req = *lookup_req;
16    char *fullpath = lookup_req->path;
17
18    /* Clear pathrem */
19    if (pathrem != NULL) *pathrem = NULL;
20
21    /* Better safe than sorry */
22    node->inode_nr = 0;
23    *vmpp = NIL_VMNT;
24
25    /* Empty (start) path? */
26    if (fullpath[0] == '\0') {
27      if (pathrem != NULL) *pathrem = fullpath;
28      return ENOENT;
29    }
30
31    /* Set user and group ids according to the system call */
32    req.uid = (call_nr == ACCESS ? fp->fp_realuid : fp->fp_effuid);
33    req.gid = (call_nr == ACCESS ? fp->fp_realgid : fp->fp_effgid);
34
35    /* Set the starting directories inode number and FS endpoint, if the
36     * calling function hasn't already filled them in. */
37    if (req.flags != ADVANCE) {
38      start_node = (fullpath[0] == '/' ? fp->fp_rd : fp->fp_wd);
39      req.start_dir = start_node->v_inode_nr;
40      req.fs_e = start_node->v_fs_e;
41      vmp = start_node->v_vmnt;
42    } else {
43      start_node = NULL;
44      req.flags = EAT_PATH_OPAQUE;
45      vmp = find_vmnt(req.fs_e);
46    }
47
48    /* Is the process' root directory on the same partition?
49     * If so, set the chroot directory too. */
50    if (req.fs_e == fp->fp_rd->v_fs_e)
51        req.root_dir = fp->fp_rd->v_inode_nr;
52    else
53        req.root_dir = 0;
54
55    req.skip_chars = path_processed;
56    req.symloop = symloop;
57
58    /* Lock the vmnt */
59    lock_vmnt(vmp);
60
61    /* Issue the request */
62    r = req_lookup(&req, &res);
63
64    /* While the response is related to mount control set the
65     * new requests respectively */
```

```
66    while(r == EENTERMOUNT || r == ELEAVEMOUNT || r == ESYMLINK) {
67      /* At this point, a symlink may have been copied back into the path
68       * buffer by FS, and FS may already have processed a part of it. If
69       * not, it will simply have advanced over the old contents of the
70       * buffer. In any case, the number of chars processed as returned by
71       * the FS is the new correct offset into our buffer. */
72      path_processed = res.char_processed;
73
74      /* Remember the current value of the symloop counter */
75      symloop = res.symloop;
76
77      /* Remember for which vmnt we currently hold a lock */
78      old_vmp = vmp;
79
80      if(r == ESYMLINK){         /*Symlink encountered with absolute path*/
81        start_node = fp->fp_rd;
82        vmp = start_node->v_vmnt;
83      } else if(r == EENTERMOUNT) {          /* Entering a new partition */
84        start_node = NIL_VNODE;
85        /* Start node is now the mounted partition's root node */
86        for (vmp = &vmnt[0]; vmp != &vmnt[NR_MNTS]; ++vmp) {
87          if (vmp->m_fs_e != NONE
88                  && vmp->m_mounted_on->v_inode_nr == res.inode_nr
89                  && vmp->m_mounted_on->v_fs_e == res.fs_e) {
90                    start_node = vmp->m_root_node;
91                    break;
92          }
93        }
94        if (start_node == NIL_VNODE) {
95          res.inode_nr, res.fs_e);
96          if (old_vmp != NIL_VMNT) unlock_vmnt(old_vmp);
97          return ENOENT;
98        }
99      } else {                                      /* Climbing up mount */
100       /* Find the vmnt that represents the partition on which we
101        * "climb up". */
102       if ((vmp = find_vmnt(res.fs_e)) == NIL_VMNT) {
103         if (old_vmp != NIL_VMNT) unlock_vmnt(old_vmp);
104         return ENOENT;
105       }
106       /* Start node is the vnode on which the partition is mounted */
107       start_node = vmp->m_mounted_on;
108     }
109     /* Fill in the request fields */
110     req.start_dir = start_node->v_inode_nr;
111     req.fs_e = start_node->v_fs_e;
112
113     /* Is the process' root directory on the same partition?*/
114     if (start_node->v_dev == fp->fp_rd->v_dev)
115       req.root_dir = fp->fp_rd->v_inode_nr;
116     else
117       req.root_dir = 0;
118
119     /* Fill in the current offset into the path name */
```

```
120       req.skip_chars = path_processed;
121       req.symloop = symloop;
122
123       /* Unlock the previously locked vmnt, if any */
124       if (old_vmp != NIL_VMNT) unlock_vmnt(old_vmp);
125
126       /* Acquire a lock to the next vmnt */
127       lock_vmnt(vmp);
128
129       /* Issue the request */
130       r = req_lookup(&req, &res);
131     }
132
133     if (r == ENOENT && pathrem != NULL) {
134       path_processed = res.char_processed;
135       *pathrem = &fullpath[path_processed];
136     } else if (r != OK && vmp != NIL_VMNT) {
137       unlock_vmnt(vmp);
138       vmp = NIL_VMNT;
139     }
140
141     /* Fill in response fields */
142     node->inode_nr = res.inode_nr;
143     node->fmode = res.fmode;
144     node->fsize = res.fsize;
145     node->dev = res.dev;
146     node->fs_e = res.fs_e;
147     node->uid = res.uid;
148     node->gid = res.gid;
149     *vmpp = vmp;
150
151     return r;
152   }
```

The lookup procedure in the threaded design had to be adapted in order to make it work properly with concurrent threads. When a thread does a lookup and later another one, the file system could have been altered in the mean time (i.e., part of the path residing at another FS was changed). In practice, this happens when the first lookup tries to find a directory and a second lookup tries to find the inode of a file within that directory. In the original, synchronous version this is not a problem, but it is with threads. As a solution the lookup procedure was adapted by making the second lookup continue with the results of the first lookup. That is, it takes the inode of the directory as starting point instead of the working directory or root directory.

The code above can be divided in three sections;

- the first section prepares the lookup and issues the first request,
- the second section consists of a while loop that keeps trying to find the inode until an error occurs or the inode is found, and
- the last section unlocks the `vmnt` in case of an error or wraps the result in a data structure for the calling function.

Differences to the original version are the modification to continue a lookup with the results of a previous lookup and the locking of a `vmnt` in the first section, the (un)locking and error handling of `vmnt`s in the second section, and finally unlocking a `vmnt` in case of an error in the last section (note that a `vmnt` is not unlocked if the lookup went well).

Now let us look at the same lookup using continuations. Again we have simplified the example a little to make the code more readable. In reality there are a few helping routines that help pick the starting `vnode`. For example, to start the lookup using the results of the previous lookup (i.e., to lookup a file within a directory).

Listing B.4: lookup using continuations

```
1  PUBLIC int lookup(rfp, startnode, flags, use_realuid, next_state)
2  struct fproc *rfp;
3  struct vnode *startnode;
4  int flags;
5  int use_realuid;
6  int next_state;
7  {
8    /* Resolve a pathname relative to startnode. */
9    int r;
10
11   /* Empty path? */
12   if(rfp->REQ_STATE.LOOKUP.rs_fullpath[0] == '\0') return ENOENT;
13
14   rfp->REQ_STATE.LOOKUP.rs_path_off = 0; /* path characters processed */
15   rfp->REQ_STATE.LOOKUP.rs_dir_ino = startnode->v_inode_nr;
16
17   /* Is the process' root directory on the same partition?
18    * If so, set the chroot directory too. */
19   if (rfp->fp_rd->v_dev == startnode->v_dev)
20       rfp->REQ_STATE.LOOKUP.rs_root_ino = rfp->fp_rd->v_inode_nr;
21   else
22       rfp->REQ_STATE.LOOKUP.rs_root_ino = 0;
23
24   /* Set user and group ids according to the system call */
25   rfp->REQ_STATE.LOOKUP.rs_uid =
26   (use_realuid ? rfp->fp_realuid : rfp->fp_effuid);
27   rfp->REQ_STATE.LOOKUP.rs_gid =
28   (use_realuid ? rfp->fp_realgid : rfp->fp_effgid);
29   rfp->REQ_STATE.LOOKUP.rs_flags = flags;
30   rfp->REQ_STATE.LOOKUP.rs_vnode = NULL;
31
32   /* Number of symlinks seen so far */
33   rfp->REQ_STATE.LOOKUP.rs_symloop = 0;
34
35   rfp->REQ_STATE.call_state = DO_LOOKUP;
36   rfp->REQ_STATE.next_state = next_state;
37
38   /* Enqueue and optionally start request */
39   enqueue_req(startnode->v_vmnt, rfp);
40
41   return SUSPEND;
```

```
42  }
43
44  PUBLIC int cnt_lookup()
45  {
46    int r, i = 0;
47    struct vmnt *vmntp;
48    struct fproc *rfp;
49    lookup_res_t res;
50
51    rfp = get_proc();
52    vmntp = get_vmnt();
53    r = m_in.RESULT;
54    cpf_revoke(vmntp->m_ctx.mc_gid); /* Revoke grant */
55    vmntp->m_state = MS_IDLE;
56
57    /* Fill in response according to the return value */
58    res.fs_e = m_in.m_source;
59    switch (r) {
60      case OK:
61        res.inode_nr = m_in.RES_INODE_NR_A;
62        res.fmode = m_in.RES_MODE_A;
63        res.fsize = m_in.RES_FILE_SIZE_A;
64        res.dev = m_in.RES_DEV_A;
65        res.uid= m_in.RES_UID_A;
66        res.gid= m_in.RES_GID_A;
67        break;
68      case EENTERMOUNT:
69        res.inode_nr = m_in.RES_INODE_NR_A;
70        res.char_processed = m_in.RES_OFFSET_A;
71        res.symloop = m_in.RES_SYMLOOP_A;
72        break;
73      case ELEAVEMOUNT:
74        res.char_processed = m_in.RES_OFFSET_A;
75        res.symloop = m_in.RES_SYMLOOP_A;
76        break;
77      case ESYMLINK:
78        res.char_processed = m_in.RES_OFFSET_A;
79        res.symloop = m_in.RES_SYMLOOP_A;
80        break;
81      default:
82        break;
83    }
84
85    r = lookup_restart(vmntp, rfp, r, &res);
86    while(WORK_LEFT(vmntp)) restart_req(vmntp, TRUE);
87    return r;
88  }
89
90  PUBLIC int lookup_restart(vmntp, rfp, r, resp)
91  struct vmnt *vmntp;
92  struct fproc *rfp;
93  int r;
94  lookup_res_t *resp;
95  {
```

```
 96  /* The lookup request replies with EENTERMOUNT, ELEAVEMOUNT, ESYMLINK,
 97   * OK, or an error message (ENOTDIR, ENOENT, EACCESS, etc). In the latter
 98   * two cases we're done; the others require more lookup requests. */
 99
100    struct vnode *dir_vp = 0, *vp;
101    struct vmnt *vmp;
102
103    rfp->REQ_STATE.LOOKUP.result = OK;
104    while (r == EENTERMOUNT || r == ELEAVEMOUNT || r == ESYMLINK) {
105      /* Update new path offset and symbolic link loop counter */
106      rfp->REQ_STATE.LOOKUP.rs_path_off = resp->char_processed;
107      rfp->REQ_STATE.LOOKUP.rs_symloop += resp->symloop;
108
109      if(rfp->REQ_STATE.LOOKUP.rs_symloop > SYMLOOP_MAX) {
110        r = ELOOP;
111        break;
112      }
113
114      if(r == ESYMLINK) {      /* Symlink encountered with absolute path */
115        dir_vp = rfp->fp_rd;
116      } else if(r == EENTERMOUNT){   /* Entering a new partition */
117        /* Start node is now the mounted partition's root node */
118        /* Find that 'child' vmnt */
119        for (vmp = &vmnt[0]; vmp != &vmnt[NR_MNTS]; vmp++) {
120          if (vmp->m_mounted_on->v_inode_nr == resp->inode_nr &&
121              vmp->m_mounted_on->v_fs_e == resp->fs_e) {
122            dir_vp = vmp->m_root_node;
123            break;
124          }
125        }
126      } else {                                        /* Climbing up mount */
127        vmp = find_vmnt(resp->fs_e); /* Find 'parent' vmnt */
128
129        /* Start node is the vnode on which the partition is
130         * mounted */
131        dir_vp = vmp->m_mounted_on;
132      }
133
134      /* Set the starting directory inode number */
135      rfp->REQ_STATE.LOOKUP.rs_dir_ino = dir_vp->v_inode_nr;
136
137      /* Is the process' root directory on the same partition?
138       * If so, set the chroot directory too. */
139      if (rfp->fp_rd->v_dev == dir_vp->v_dev)
140        rfp->REQ_STATE.LOOKUP.rs_root_ino = rfp->fp_rd->v_inode_nr;
141      else
142        rfp->REQ_STATE.LOOKUP.rs_root_ino = 0;
143
144      if (dir_vp->v_vmnt != vmntp) {           /* Changing mount points? */
145        dequeue_req(vmntp, rfp);          /* Then clean up request queue */
146        vmntp->m_state = MS_IDLE;
147        enqueue_req(dir_vp->v_vmnt,rfp);            /* Queue new request */
148      } else                             /* Staying at current mount point */
149        restart_req(vmntp, FALSE);                     /* Retry request */
```

```
150
151       return SUSPEND;
152
153     }
154
155     /* Store lookup result so orig. call can deal with the problem if there
156      * is any. (For example, clean up data structures.) */
157
158     if(r != OK) /* Encountered problems while looking up inode? */
159       rfp->REQ_STATE.LOOKUP.result = r;
160     else {
161       /* We found the inode; store result in request state */
162       /* Check vnode already in use */
163       vp = find_vnode(resp->fs_e, resp->inode_nr);
164       if (vp != NIL_VNODE) vp->v_ref_count++;
165       else {
166         /* Find free vnode */
167         vp = get_free_vnode();
168
169         /* Fill in the free vnode's fields */
170         vp->v_fs_e = resp->fs_e;
171         vp->v_inode_nr = resp->inode_nr;
172         vp->v_mode = resp->fmode;
173         vp->v_size = resp->fsize;
174         vp->v_uid = resp->uid;
175         vp->v_gid = resp->gid;
176         vp->v_sdev = resp->dev;
177         vp->v_vmnt = vmntp;
178         vp->v_dev = vmntp->m_dev;
179         vp->v_ref_count = 1;
180
181         rfp->REQ_STATE.LOOKUP.rs_vnode = vp;
182     }
183
184     rfp->REQ_STATE.call_state = rfp->REQ_STATE.next_state;
185     rfp->REQ_STATE.next_state = NULL_STATE;
186
187     return restart_req(vmntp, FALSE);  /* Continue orig. request */
188   }
```

The lookup works as follows. The lookup starts by calling `lookup` (through the use of helping routines) where the lookup state is prepared and the first lookup request is issued. The function calling lookup also has to provide the next state the request arrives at, after finishing the lookup (`next_state`).

The reply of the FS will be picked up by the main loop of VFS which catches all incoming messages. When a message comes from an FS, VFS knows it is a reply to a request sent earlier. VFS then looks up the `fproc` entry that sent the message and is then able to figure out what the original system call was (e.g., `open`, `chmod`, `close`, etc). Normally, the system call is the key to a function pointer in a table and the original `do_foo` routine is called. However, `lookup` is an exception, because it cannot be the original call. Therefore VFS checks if the reply was for a lookup and calls `cnt_lookup` or the `do_foo` routine accordingly.

`cnt_lookup` wraps the result in a data structure and *restarts* the lookup procedure by calling `lookup_restart`. `lookup_restart` corresponds to the second section in the threaded version (which is a while loop, hence the restart). In case the file is not yet found, `lookup_restart` removes the lookup request from the queue and adds it to the new queue, if necessary (e.g., this is necessary when the lookup has to continue on another FS, but not when we encounter a symbolic link which resides on the same FS). The lookup request is then restarted. If the FS did find the file or an error occurred, the result is wrapped in a data structure and the function belonging to `next_state` is called (using a huge switch statement in `restart_req`).

B.3 read

As final example we will show the implementations of the `read` system call. This system call provides a file descriptor, a buffer where the data should end up, and the number of bytes to read.

Listing B.5: read using threads

```
1  PUBLIC int do_read()
2  {
3    return(read_write(READING));
4  }
5
6  PUBLIC int read_write(rw_flag)
7  int rw_flag; /* READING or WRITING */
8  {
9  /*Do read(fd, buffer, nbytes) or write(fd, buffer, nbytes) call*/
10   register struct filp *f;
11   register struct vnode *vp;
12   struct vmnt *vmp;
13   off_t bytes_left;
14   u64_t position;
15   unsigned int off, cum_io;
16   int op, oflags, r, chunk, who_e, block_spec, char_spec;
17   int regular, partial_pipe = 0, partial_cnt = 0;
18   mode_t mode_word;
19   struct filp *wf;
20   phys_bytes p;
21   struct dmap *dp;
22
23   /* Request and response structures */
24   struct readwrite_req req;
25   struct readwrite_res res;
26
27   /* For block spec files */
28   struct breadwrite_req breq;
29
30   if(m_in.nbytes < 0) return(EINVAL);
31
32   /* Get the filp, and lock the vmnt */
33   f = get_filp(m_in.fd);
34   if(f == NIL_FILP) return(err_code);
```

```
35
36    if(((f->filp_mode)&(rw_flag == READING ? R_BIT : W_BIT)) == 0){
37      r = f->filp_mode == FILP_CLOSED ? EIO : EBADF;
38      return(r);
39    }
40
41    if (m_in.nbytes == 0)
42      return(0);  /* so char special files need not check for 0 */
43
44
45    position = f->filp_pos;
46    oflags = f->filp_flags;
47    vp = f->filp_vno;
48    vmp = vp->v_vmnt;
49    r = OK;
50
51    lock_vmnt(vmp);
52
53    if (vp->v_pipe == I_PIPE) {
54      /* fp->fp_cum_io_partial is only nonzero when doing partial writes */
55      cum_io = fp->fp_cum_io_partial;
56    } else
57      cum_io = 0;
58
59    op = (rw_flag == READING ? VFS_DEV_READ : VFS_DEV_WRITE);
60    mode_word = vp->v_mode & I_TYPE;
61    regular = mode_word == I_REGULAR || mode_word == I_NAMED_PIPE;
62    char_spec  = (mode_word == I_CHAR_SPECIAL  ? 1 : 0);
63    block_spec = (mode_word == I_BLOCK_SPECIAL ? 1 : 0);
64
65    if(char_spec) {                          /* Character special files. */
66      dev_t dev;
67      dev = (dev_t) vp->v_sdev;
68      r = dev_io(op, dev, who_e, m_in.buffer, position,m_in.nbytes,
69                 oflags);
70      if (r >= 0) {
71        cum_io = r;
72        position = add64ul(position, r);
73        r = OK;
74      }
75    } else if(block_spec) {                      /* Block special files. */
76
77      /* Fill in the fields of the request */
78      breq.rw_flag = rw_flag;
79      breq.fs_e = vp->v_bfs_e;
80      breq.blocksize = vp->v_blocksize;
81      breq.dev = vp->v_sdev;
82      breq.user_e = who_e;
83      breq.pos = position;
84      breq.num_of_bytes = m_in.nbytes;
85      breq.user_addr = m_in.buffer;
86
87      /* Issue request */
88      r = req_breadwrite(&breq, &res);
```

```
 89
 90      position = res.new_pos;
 91      cum_io += res.cum_io;
 92
 93    } else {                                /* Regular files (and pipes) */
 94      if(rw_flag == WRITING && block_spec == 0) {
 95        /* Check for O_APPEND flag. */
 96        if(oflags & O_APPEND) position = cvul64(vp->v_size);
 97
 98        /* Check in advance to see if file will grow too big.*/
 99        if(cmp64ul(position,vp->v_vmnt->m_max_file_size - m_in.nbytes)> 0){
100          unlock_vmnt(vmp);
101          return(EFBIG);
102        }
103      }
104
105      /* Pipes are a little different. Check. */
106      if (vp->v_pipe == I_PIPE) {
107        r = pipe_check(vp, rw_flag, oflags, m_in.nbytes, position,
108                       &partial_cnt, 0);
109        if (r <= 0) {
110          unlock_vmnt(vmp);
111          return(r);
112        }
113      }
114
115      if (partial_cnt > 0) {
116        /* So that we don't have to deal with partial count in the FS
117         * process. */
118        m_in.nbytes = MIN(m_in.nbytes, partial_cnt);
119        partial_pipe = 1;
120      }
121
122      /* Fill in request structure */
123      req.fs_e = vp->v_fs_e;
124      req.rw_flag = rw_flag;
125      req.inode_nr = vp->v_inode_nr;
126      req.user_e = who_e;
127      req.seg = D;
128      req.pos = position;
129      req.num_of_bytes = m_in.nbytes;
130      req.user_addr = m_in.buffer;
131      req.inode_index = vp->v_index;
132
133      /* Truncate read request at size (not for special files). */
134      if((rw_flag == READING) &&
135        cmp64ul(add64ul(position, req.num_of_bytes), vp->v_size) > 0){
136        /* Position always should fit in an off_t (LONG_MAX). */
137        req.num_of_bytes = vp->v_size - cv64ul(position);
138      }
139
140      /* Issue request */
141      r = req_readwrite(&req, &res);
142
```

```
143       if (r >= 0) {
144         if (ex64hi(res.new_pos))
145           panic(__FILE__, "read_write: bad new pos", NO_NUM);
146
147         position = res.new_pos;
148         cum_io += res.cum_io;
149       }
150     }
151
152     /* On write, update file size and access time. */
153     if (rw_flag == WRITING) {
154       if (regular || mode_word == I_DIRECTORY) {
155         if (cmp64ul(position, vp->v_size) > 0) {
156           if (ex64hi(position) != 0)
157             panic(__FILE__,"read_write: file size too big",NO_NUM);
158           vp->v_size = ex64lo(position);
159         }
160       }
161     } else {
162       if (vp->v_pipe == I_PIPE) {
163         if (cmp64ul(position, vp->v_size) >= 0) {
164           /* Reset pipe pointers */
165           vp->v_size = 0;
166           position = cvu64(0);
167           wf = find_filp(vp, W_BIT);
168           if (wf != NIL_FILP) wf->filp_pos = cvu64(0);
169         }
170       }
171     }
172
173     f->filp_pos = position;
174     unlock_vmnt(vmp);
175
176     if (r == OK) {
177       if(partial_pipe) {
178         partial_pipe = 0;
179         /* partial write on pipe with */
180         /* O_NONBLOCK, return write count */
181         if(!(oflags & O_NONBLOCK)) {
182           fp->fp_cum_io_partial = cum_io;
183           suspend(XPIPE);   /* partial write on pipe with */
184           return(SUSPEND);  /* nbyte > PIPE_SIZE - non-atomic */
185         }
186       }
187       fp->fp_cum_io_partial = 0;
188       return cum_io;
189     }
190
191     return r;
192   }
```

The threaded implementation starts by obtaining a lock on the `vmnt` object pointed to by the file descriptor (file descriptor → `filp` → `vnode` → `vmnt`). Then specific steps are taken for the different types of files (character special,

block special, and normal files). The rest of the code does not differ much from the original version, except that in case of an error or the read request executed successfully, the `vmnt` object is to be explicitly unlocked. This resembles how `vnode`s are supposed to be put after another function created it or increased its usage counter and the `vnode` is no longer used.

Using continuations the implementation would look as follows.

Listing B.6: read using continuations

```
1  PUBLIC int do_read()
2  {
3  /* Perform the read(fd, buffer, nbytes) system call. */
4    return(do_rw(READING));
5  }
6
7  PUBLIC int do_rw(rw_flag)
8  int rw_flag;
9  {
10   int r, fs_e;
11   struct fproc *rfp;
12   struct filp *f;
13
14   rfp = get_proc();
15   rfp->REQ_STATE.call_nr = call_nr;
16
17   if(rfp->REQ_STATE.call_state == NULL_STATE) {
18     if((f = get_filp(m_in.fd)) == NIL_FILP)
19       return(err_code);
20     else
21       fs_e = f->filp_vno->v_fs_e;
22     rfp->REQ_STATE.call_state = RW_START;
23   }
24
25   switch(rfp->REQ_STATE.call_state) {
26     case RW_START: r = dummy(RW_INIT, fs_e); break;
27     case RW_INIT: r = read_write(rw_flag); break;        /* Initialize */
28     case RW_BLOCK: r = fnlz_block_rw(); break;    /* Finalize block rw */
29     case RW_NORMAL: r = fnlz_normal_rw(); break; /* Finalize normal rw */
30     case RW_PIPE: r = fnlz_pipe_rw(); break;       /* Finalize pipe rw */
31     default:
32       panic(__FILE__, "Unknown state for do_rw", NO_NUM);
33   }
34
35   if(r != SUSPEND) {           /* An error occured or we have a result */
36     vmntp->m_state = MS_IDLE;                          /* Reset vmnt and */
37     rfp->REQ_STATE.call_state = NULL_STATE;             /* request state */
38     dequeue_req(vmntp, rfp);              /* Remove request from queue */
39     revive(rfp->fp_endpoint, r);                       /* Unsuspend proc */
40   }
41
42   return r;
43 }
44
45 PUBLIC int read_write(rw_flag)
46 int rw_flag; /* READING or WRITING */
```

```
47  {
48  /*Do read(fd, buffer, nbytes) or write(fd, buffer, nbytes) call*/
49    struct filp *f;
50    struct vnode *vp;
51    struct vmnt *vmntp;
52    struct dmap *dp;
53    u64_t position;
54    unsigned int cum_io, cum_io_incr;
55    int op, oflags, r, chunk, block_spec, char_spec, regular;
56    mode_t mode_word;
57    phys_bytes p;
58
59    if (m_in.nbytes < 0) return(EINVAL);
60
61    if ((f = get_filp(m_in.fd)) == NIL_FILP) return(err_code);
62
63    if(((f->filp_mode)&(rw_flag == READING ? R_BIT : W_BIT)) == 0)
64      return(f->filp_mode == FILP_CLOSED ? EIO : EBADF);
65
66    if (m_in.nbytes == 0)
67      return(0);  /* so char special files need not check for 0 */
68
69    position = f->filp_pos;
70    oflags = f->filp_flags;
71    vp = f->filp_vno;
72
73    if (vp->v_pipe) {
74      if (fp->fp_cum_io_partial != 0)
75        panic(__FILE__,"read_write: fp_cum_io_partial not cleared",NO_NUM);
76
77      r = rw_pipe(rw_flag, who_e, m_in.fd, f, m_in.buffer, m_in.nbytes);
78      return r;
79    }
80
81    r = OK;
82    cum_io = 0;
83    op = (rw_flag == READING ? VFS_DEV_READ : VFS_DEV_WRITE);
84    mode_word = vp->v_mode & I_TYPE;
85    regular = mode_word == I_REGULAR;
86    char_spec = (mode_word == I_CHAR_SPECIAL ? 1 : 0);
87    block_spec = (mode_word == I_BLOCK_SPECIAL ? 1 : 0);
88
89    if (char_spec) {                          /* Character special files. */
90      dev_t dev;
91      int suspend_reopen;
92      suspend_reopen = (f->filp_state != FS_NORMAL);
93
94      dev = (dev_t) vp->v_sdev;
95      r = dev_io(op, dev, who_e, m_in.buffer, position,
96                 m_in.nbytes, oflags, suspend_reopen);
97      if (r >= 0) {
98        cum_io = r;
99        position = add64ul(position, r);
100       r = OK;
```

```
101       }
102
103       /* On write, update file size and access time. */
104       if (rw_flag == WRITING) {
105         if (regular || mode_word == I_DIRECTORY) {
106           if (cmp64ul(position, vp->v_size) > 0) {
107             if (ex64hi(position) != 0) {
108               panic(__FILE__,"read_write: size too big", NO_NUM);
109             }
110             vp->v_size = ex64lo(position);
111           }
112         }
113       }
114
115       f->filp_pos = position;
116       if (r == OK) return cum_io;
117     } else if (block_spec) {                    /* Block special files. */
118       fp->REQ_STATE.call_state = RW_BLOCK;
119
120       /* Store parameters in process structure. */
121       fp->REQ_STATE.BLOCK_RW.rs_filp = f;
122       fp->REQ_STATE.BLOCK_RW.rs_dev = vp->v_sdev;
123       fp->REQ_STATE.BLOCK_RW.rs_pos = position;
124       fp->REQ_STATE.BLOCK_RW.rs_count = m_in.nbytes;
125       fp->REQ_STATE.BLOCK_RW.rs_buffer = m_in.buffer;
126       fp->REQ_STATE.BLOCK_RW.rs_rw_flag = rw_flag;
127
128       /* Enqueue and optionally start request */
129       vmntp = find_vmnt(vp->v_bfs_e);
130       enqueue_req(vmntp, fp);
131
132       return SUSPEND;
133
134     } else {                                          /* Regular files */
135       if (rw_flag == WRITING) {
136         /* Check for O_APPEND flag. */
137         if (oflags & O_APPEND) position = cvul64(vp->v_size);
138       }
139
140       fp->REQ_STATE.call_state = RW_NORMAL;
141
142       /* Store parameters in process structure. */
143       fp->REQ_STATE.NORMAL_RW.rs_filp = f;
144       fp->REQ_STATE.NORMAL_RW.rs_position = position;
145       fp->REQ_STATE.NORMAL_RW.rs_rw_flag = rw_flag;
146       fp->REQ_STATE.NORMAL_RW.rs_buffer = (vir_bytes)m_in.buffer;
147       fp->REQ_STATE.NORMAL_RW.rs_count = m_in.nbytes;
148
149       /* Enqueue and optionally start request */
150       enqueue_req(vp->v_vmnt, fp);
151       return SUSPEND;
152
153     }
154
```

```
155   return r;
156 }
157
158 PUBLIC int rw_pipe(rw_flag, usr, fd_nr, f, buf, req_size)
159 int rw_flag; /* READING or WRITING */
160 endpoint_t usr;
161 int fd_nr;
162 struct filp *f;
163 char *buf;
164 size_t req_size;
165 {
166   int r, oflags, partial_pipe;
167   size_t size;
168   struct vnode *vp;
169   u64_t position;
170
171   oflags = f->filp_flags;
172   vp = f->filp_vno;
173   position = cvu64((rw_flag == READING) ?
174                                   vp->v_pipe_rd_pos : vp->v_pipe_wr_pos);
175
176   r = pipe_check(vp, rw_flag, oflags, req_size, position, 0);
177   if (r <= 0) {
178     if (r == SUSPEND)
179       pipe_suspend(fp, rw_flag, fd_nr, buf, req_size);
180     else
181       fp->REQ_STATE.call_state = NULL_STATE;
182     return(r);
183   }
184
185   size = r;
186   partial_pipe = (size < req_size) ? 1 : 0 ;
187
188   /* Truncate read request at size. */
189   if((rw_flag==READING) && cmp64ul(add64ul(position,size),vp->v_size)>0){
190     /* Position always should fit in an off_t (LONG_MAX). */
191     size = vp->v_size - cv64ul(position);
192   }
193
194   fp->REQ_STATE.call_state = RW_PIPE;
195
196   /* Store parameters in process structure. */
197   fp->REQ_STATE.PIPE_RW.rs_filp = f;
198   fp->REQ_STATE.PIPE_RW.rs_position = position;
199   fp->REQ_STATE.PIPE_RW.rs_rw_flag = rw_flag;
200   fp->REQ_STATE.PIPE_RW.rs_buffer = (vir_bytes) buf;
201   fp->REQ_STATE.PIPE_RW.rs_count = size;
202   fp->REQ_STATE.PIPE_RW.rs_partial_pipe = partial_pipe;
203   fp->REQ_STATE.PIPE_RW.rs_fd_nr = fd_nr;
204   fp->REQ_STATE.PIPE_RW.rs_req_size = req_size;
205
206   /* Enqueue and optionally start request */
207   enqueue_req(vp->v_vmnt, fp);
208
```

```
209   return SUSPEND;
210 }
211
212 PUBLIC int fnlz_block_rw(void) {
213   int r;
214   off_t position;
215   struct fproc *rfp;
216   struct filp *f;
217   struct vnode *vnop;
218   struct vmnt *vmntp;
219
220   vmntp = get_vmnt();                                  /* Look up vmnt */
221   rfp = get_proc();               /* Get fproc entry to original proc */
222   cpf_revoke(vmntp->m_ctx.mc_gid);                     /* Revoke grant */
223   r = m_in.RESULT;
224
225   if(r >= 0) {
226     f = rfp->REQ_STATE.BLOCK_RW.rs_filp;
227     f->filp_pos = make64(m_in.RES_FD_POS_LO_A, m_in.RES_FD_POS_HI_A);
228     r = m_in.RES_FD_CUM_IO_A;
229   }
230
231   return r;
232 }
233
234 PUBLIC int fnlz_normal_rw(void)
235 {
236   int r;
237   off_t position;
238   struct fproc *rfp;
239   struct filp *f;
240   struct vnode *vnop;
241   struct vmnt *vmntp;
242
243   vmntp = get_vmnt();                                  /* Look up vmnt */
244   rfp = get_proc();               /* Get fproc entry to original proc */
245   cpf_revoke(vmntp->m_ctx.mc_gid);                     /* Revoke grant */
246   r = m_in.RESULT;
247
248   if(r >= 0) {
249     f = rfp->REQ_STATE.NORMAL_RW.rs_filp;
250     vnop = f->filp_vno;
251     position = m_in.RES_FD_POS_LO_A;
252
253     /* On write, update file size. */
254     if (rfp->REQ_STATE.NORMAL_RW.rs_rw_flag == WRITING &&
255                                             position > vnop->v_size){
256       vnop->v_size = position;
257     }
258     f->filp_pos = cvul64(position);
259     r = m_in.RES_FD_CUM_IO_A;
260   }
261
262   return r;
```

```
263 }
264
265 PUBLIC int fnlz_pipe_rw(void)
266 {
267   int r;
268   off_t position;
269   size_t cum_io, cum_io_incr, req_size;
270   struct fproc *rfp;
271   struct vmnt *vmntp;
272
273   int rw_flag;  /* READING or WRITING */
274   int fd_nr, partial_pipe, oflags;
275   vir_bytes buf;
276   struct filp *f;
277   struct vnode *vp;
278
279   vmntp = get_vmnt();                                  /* Look up vmnt */
280   cpf_revoke(vmntp->m_ctx.mc_gid);                     /* Revoke grant */
281   rfp = get_proc();               /* Get fproc entry to original proc */
282
283   r = m_in.RESULT;
284   position = m_in.RES_FD_POS_LO_A;
285   cum_io_incr = m_in.RES_FD_CUM_IO_A;
286
287   /* Return on error*/
288   if (r < 0) return r;
289
290   /* rfp->fp_cum_io_partial is only nonzero when doing partial writes */
291   cum_io       = rfp->fp_cum_io_partial;
292   buf          = rfp->REQ_STATE.PIPE_RW.rs_buffer;
293   rw_flag      = rfp->REQ_STATE.PIPE_RW.rs_rw_flag;
294   f            = rfp->REQ_STATE.PIPE_RW.rs_filp;
295   partial_pipe = rfp->REQ_STATE.PIPE_RW.rs_partial_pipe;
296   fd_nr        = rfp->REQ_STATE.PIPE_RW.rs_fd_nr;
297   req_size     = rfp->REQ_STATE.PIPE_RW.rs_req_size;
298
299   oflags = f->filp_flags;
300   vp = f->filp_vno;
301
302   cum_io += cum_io_incr;                       /* Update cumulative io */
303   buf += cum_io_incr;                         /* Update buffer pointer */
304   req_size -= cum_io_incr;                /* Calculate remaining bytes */
305
306   /* On write, update file size and access time. */
307   if(rw_flag == WRITING) {
308     if (position > vp->v_size) vp->v_size = position;
309   } else if (position >= vp->v_size) {
310     /* Reset pipe pointers */
311     vp->v_size = 0;
312     vp->v_pipe_rd_pos = 0;
313     vp->v_pipe_wr_pos = 0;
314     position = 0;
315   }
316
```

```
317   if(rw_flag == READING)
318     vp->v_pipe_rd_pos = position;
319   else
320     vp->v_pipe_wr_pos = position;
321
322   if (partial_pipe) {
323     /* partial write on pipe with O_NONBLOCK, return write count */
324     if (!(oflags & O_NONBLOCK)) {
325       /* partial write on pipe with req_size > PIPE_SIZE, non-atomic */
326       rfp->fp_cum_io_partial = cum_io;
327       pipe_suspend(rfp, rw_flag, fd_nr, (char *)buf, req_size);
328       return SUSPEND;
329     }
330   }
331
332   rfp->fp_cum_io_partial = 0;
333   return cum_io;
334 }
```

As the system call requires a file descriptor, we can put the dummy request on the queue for the FS that actually holds the inode. Upon return the read operation is prepared by `read_write`, which makes sure the arguments make sense and figures out what type of file we try to read from. Subsequently it prepares the request state needed for the request that is to be sent. When an FS replies with the result, the wrapper calls the function that finalizes the request (i.e., to finish the block read, pipe read, or normal read).

What can be seen from this example is that not only the amount of code has almost doubled compared to the threaded version, it has also become unclear. The structure is gone. This makes the code hard to understand for future developers, hard to modify, and consequently bug prone.