Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

Designing a Concurrent File Server

James WHITEHEAD II

Department of Computer Science, University of Oxford,
Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom;

jim.whitehead@cs.ox.ac.uk

Abstract. In this paper we present a design and architecture for a concurrent file system server. This architecture is a compromise between the fully concurrent V6 UNIX implementation, and the simple sequential implementation in the MINIX operating system. The design of the server is made easier through the use of a disciplined model of concurrency, based on the CSP process algebra. By viewing the problem through this restricted lens, without traditional explicit locking mechanisms, we can construct the system as a process network of components with explicit connections and dependencies. This provides us with insight into the problem domain, and allows us to analyse the issues present in concurrent file system implementation.

Keywords. file system, Go, CSP, UNIX, MINIX.

Introduction

Modern operating systems allow multiple users and programs to share the computing resources on a single machine. They also serve as an intermediary between programs and the physical computing hardware, such as persistent storage devices. In the UNIX family of operating systems, disk access is accomplished by interacting with files and directories through a hierarchical file system, using system calls.

Managing access to the file system in a concurrent environment is a challenging problem. An implementation must ensure that a program cannot corrupt the state of the file system, or the data stored on disk. Two programs that are performing operations on different files should not be able to interfere with each other. In addition, a file system implementation must be fast; most programs require some form of disk access.

There are several possible approaches to implementing a shared file system. The UNIX operating system (and its modern relatives Linux and the BSD family) provide a high degree of concurrency, using explicit locking mechanisms. This elaborate system of semaphores, spin locks, mutexes, and interrupt masking is complicated and difficult to understand. In contrast, the MINIX operating system, designed primarily as an educational tool, eschews concurrency in favour of simplicity and readability.

In this paper we investigate the application of process oriented design methodology to the MINIX file server, which provides a well documented sequential reference implementation. Using sound concurrency principles based on the CSP [1] process algebra, we aim to explore a range of designs without the need to introduce explicit locking mechanisms. Through the process of introducing concurrency to an existing file system, we hope to explore the limitations of process oriented design in data-centric applications.

We present a file server architecture that enables file I/O to be performed concurrently alongside the sequential handling of other file system operations. This architecture is based on the MINIX file server, which provides a well documented clean slate from which to de-

rive this design. By decomposing the system into distinct interacting components, we have made it easier to reason about the concurrent properties of (and the interactions between) these components. The result is a file server that is more scalable, while still being easy to understand.

We have created an implementation of this file server using Go[2,3]. Go is a statically typed, garbage collected, compiled programming language that includes support for CSP style concurrency. The language is uniquely placed to allow us to re-use many of the data structures and algorithms from the existing C implementation, while augmenting it with natively supported concurrency. More information about the suitability of Go for process oriented development can be found in [4]. This file server is available as a package for Go called *minixfs* [5]. It allows developers to run instances of the file server in order to interact with MINIX file system disk images.

This paper is organised as follows: Section 1 provides a short introduction to the UNIX file system model, along with a comparison between the UNIX and MINIX implementations. The overall design of the file system architecture is presented in Section 2, as a series of incremental abstractions. In Section 3, we present an implementation of a concurrent block cache and discuss the scope and implications of internal concurrency. Our conclusions and suggestions for future work can be found in Section 4.

1. Background

In this section we describe the UNIX file system model, which is used through this paper. Additionally, we describe two different existing file system implementations: one that is highly concurrent and uses explicit locking, and a sequential version that is designed to be simple to understand.

The UNIX family of operating systems provide access to persistent storage devices through a single hierarchical file system. Each directory can contain both files, which contain raw data, and other directories. This hierarchy is often seen as a tree with a single root node, called the root directory.

Each file comprises metadata, stored in a data structure called an *inode*, and the actual content of the file. This metadata includes the owner of the file, the access permissions, the size of the file, and other similar fields. More importantly, the inode contains a list of the data blocks where the contents of the file may be found on the disk device. Although users identify entries on the file system using path names, such as */var/log/system*, the metadata does not contain this name or any other identifier.

Instead, each directory contains a list of entries that map names to inode numbers, describing the child entries of the directory. In order to convert a path name into an actual inode, the file system must perform a series of lookups for each directory in the path. For the path given above, the root directory must be searched for an entry called *var*, then that directory searched for an entry called *log*, and finally that directory searched for a file called *syslog*.

This method decouples the concept of a file system entry from the name used to access it. The UNIX file system model takes advantage of this by allowing an inode to be accessible by several different path names, simply by having the same inode as a child of multiple directories. In order to track this, each inode contains a count of the number of directories that link to it. This count is used to determine at what point a file is eligible to actually be deleted— as long as the link count is greater than zero, the inode is still a child of some directory and cannot be removed.

There are countless file system operations that are defined by the Single Unix Standard[6] and the POSIX[7] specifications. In this paper we consider a subset of these operations, shown in Table 1. This particular selection of system calls cover the majority of the func-

tionality present in UNIX file systems. The system calls are roughly grouped into hierarchy altering operations, file open/close operations, inode operations, miscellaneous system calls, and file I/O operations.

Table 1. A sampling of the UNIX file system calls

System call	Description
mount(path, device)	Attach a file system to an existing directory node
unmount(path)	Detach a mounted file system
open(path, flags, mode)	Open a file for reading/writing
creat(path, flags, mode)	Create a new file
close(fd)	Close an open file
stat(path)	Get information about a file
chmod(path, mode)	Change the mode of a file
link(path, newpath)	Create a link to a file within a directory
unlink(path)	Remove a link to a file from a directory
mkdir(path)	Create a new directory
rmdir(path)	Remove an existing empty directory
sync()	Synchronize and commit file system
fork()	Fork an existing process
chdir(path)	Change working directory of a process
exit()	Exit the calling process
read(fd, buf, len)	Read the contents of an open file
write(fd, buf, len)	Write contents to an open file
seek(fd, pos, whence)	Move the position marker for an open file

Operating systems are necessarily complex; even a single-user system might have dozens of programs open, with even more running unseen in the background. A file system is a shared resource built on top of shared disk space, in a highly concurrent environment. Therefore, special care must be taken to ensure that the file system cannot be corrupted by multiple programs performing system calls, and that two programs operating on separate files do not interfere with each other. In the next section, we introduce an implementation that allows for a high degree of concurrency by making use of explicit locking mechanisms.

1.1. Highly Concurrent with Explicit Locking

The approach to concurrency in the early implementations of UNIX[8] was to use interrupt masking and semaphores to define small "critical sections" of the program code. The goal of this is to ensure that two processes that are accessing some data structure cannot both enter the critical section at the same time, giving each exclusive access inside the section. This ensures that two concurrent operations can proceed independently until they both attempt to enter competing critical sections. Practically, this means that any number of processes can perform file system operations concurrently and their execution will be serialized in some manner.

Figure 1 shows a slightly simplified example of a critical section in the V6 UNIX kernel[8]. This figure contains a portion of the *free* function, used to return a block to the free block list, allowing it to be re-allocated. Updating this list requires exclusive access, so this function tests a lock variable and suspends itself if the list is not available. At some point in the future this process may be awakened by someone else releasing the lock, at which point it attempts to acquire it again.

The *sleep* function, shown in Figure 2 allows a process to voluntarily yield control of the processor until a corresponding *wakeup* call occurs for the specified parameter, the lock

variable in this case. However, it is possible for the execution of *sleep* to be pre-empted by an interrupt handler that might try to suspend the same process for some other reason, such as the scheduling quantum expiring. In order to prevent this, the procedure calls the *spl6* function to change the priority level of the processor, inhibiting clock interrupts. Once the critical changes to the lock are done, the priority level is restored.

```
free(dev, bnum) {
    register *fp, *bp, *ip;

    fp = getfs(dev);
    while (fp->s_flock)
        sleep(&fp->s_flock, PINOD);

    fp->s_flock++;
    bp = getblk(dev, bno);
    ip = bp->b_addr;
    *ip++ = fp->s_nfree;
    bcopy(fp->s_free, ip, 100);
    fp->s_nfree = 0;
    bwrite(bp)
    fp->s_flock = 0;
    wakeup(&fp->s_flock);
}
```

Figure 1. A portion of the UNIX *free* function

```
sleep(chan, pri) {
    register *rp, s

    s = PS->integ;
    rp = u.ui_procp;
    if (pri >= 0) {
        // Handle signals
    } else {
        spl6();
        rp->p_wchan = chan;
        rp->p_stat = SSLEEP;
        rp->p_pri = pri;
        spl0();
        swtch();
    }
    PS->integ = s;
    return;
}
```

Figure 2. A portion of the UNIX *sleep* function

Both the *free* and the *sleep* functions use critical sections to ensure mutual exclusion, although the context of the function varies the style of protection that is needed. This scheme is brilliant in its simplicity; the relative size of the UNIX kernel compared to modern computer software is a testament to this. Unfortunately, it relies on specific details of the underlying hardware, and this approach is not scalable on machines with multiple processors.

Modern day UNIX derivatives such as Linux and FreeBSD use the same strategy of defining critical sections, but using a combination of semaphores, spin locks, and mutexes instead of interrupt masking. These locking mechanisms are provided by the respective kernels, and are implemented in a portable, efficient, and scalable way.

Figure 3 shows a portion of the *unlazy_walk* function, which is used in path name resolution in Linux. Eleven of the thirty lines shown are explicit lock and unlock operations on five different data structures. In addition, the documentation for this function indicates that it must be called from a specific context, where a set number of locks have already been acquired. In fact, the Linux kernel is littered with comments and text documentation about these locking conventions and mechanisms, which can vary substantially from subsystem to subsystem.

There is no doubt that this strategy is practical and effective, given an efficient implementation. However, it should be clear from these examples that using and understanding this style of locking is not straightforward. It is difficult to develop an intuition about how, why, and where these critical sections are defined.

1.2. No Concurrency

The MINIX operating system was created as an educational tool for studying and experimenting with the internals of operating systems. As a result, the implementation often opts for simplicity over features that would make the operating system more practically useful. One such simplification is the sequential nature of the file server– only a single file system call can be processed at a time.

Although MINIX is designed to be compatible with the UNIX system call interface, the implementation is quite different. Rather than consisting of a single monolithic program

```
static int unlazy_walk(struct nameidata *nd, struct dentry *dentry)
{
    if (nd->root.mnt && !(nd->flags & LOOKUP_ROOT)) {
        spin_lock(&fs->lock);
        goto err_root;
    }
    spin_lock(&parent->d_lock);
    if (!dentry) {
        goto err_parent;
    } else {
        spin_lock_nested(&dentry->d_lock, DENTRY_D_LOCK_NESTED);
        parent->d_count++;
        spin_unlock(&dentry->d_lock);
    }
    spin_unlock(&parent->d_lock);
    if (want_root) {
        path_get(&nd->root);
        spin_unlock(&fs->lock);
    }
    mntget(nd->path.mnt);

    rcu_read_unlock();
    br_read_unlock(vfsmount_lock);
    return 0;

err_child:
    spin_unlock(&dentry->d_lock);
err_parent:
    spin_unlock(&parent->d_lock);
err_root:
    if (want_root)
        spin_unlock(&fs->lock);
    return -ECHILD;
}
```

Figure 3. An example of the locking mechanisms used in the Linux file system

that runs in privileged mode, MINIX is a collection of server processes that run on top of a microkernel. This microkernel only provides a small amount of functionality: address space management, process management and scheduling, and inter-process communication. Device drivers, networking support, and the file system are all provided by server processes that run in user mode. These servers communicate and co-operate with each other to provide the full functionality of the operating system.

The file server consists of a single loop that receives a system call request, performs the required work, and then sends a message with the results. MINIX does support suspending certain types of requests and accepting others: e.g., waiting for keyboard input and operations on pipes between processes. However, these are treated as a special case, and the main pattern is to accept and service requests sequentially. This lack of concurrency helps to simplify the implementation details, but has negative performance consequences.

Any file system request that needs to access a slow device, or consists of a large data transfer may cause the file server to be unresponsive for the duration of that operation. This can manifest itself as a pause that is observable by clients of the file system. For example, if a second user were to attempt to change directory while the file server is handling such a request, the system would appear to hang until the first operation was completed. In practice, this matters little on a single user computer with reasonably fast I/O devices, although it was frustrating on older machines when a background task brought the system to its knees by accessing a diskette drive.

One advantage of the MINIX file server is that it was designed as a sequential system. Our goal is to derive a range of designs that allow concurrency for file system operations. A sequential system with no specific concurrency mechanisms in place provides a clean slate

from which to begin this exploration.

2. Design Overview

In this section, we introduce a middle-ground between the highly concurrent implementations of UNIX, and the simple sequential file server in MINIX. Rather than the design being driven by mutual exclusion for data structures, it consists of a collection of server processes that are each responsible for some portion of the functionality of the file system. By connecting these components together using synchronous message passing over explicit channels, we can construct a functioning file system where the dependencies and relationships between components are explicit.

2.1. A Starting Point

The MINIX file server provides an attractive starting point for our design. The file system logic, data structures, and algorithms are already present, and free from explicit locks. This architecture is depicted in Figure 4. This figure shows three processes interacting with the file system by issuing system calls; the system call interface converts these requests into inter-process communication messages to the file system server, or *FS server*.

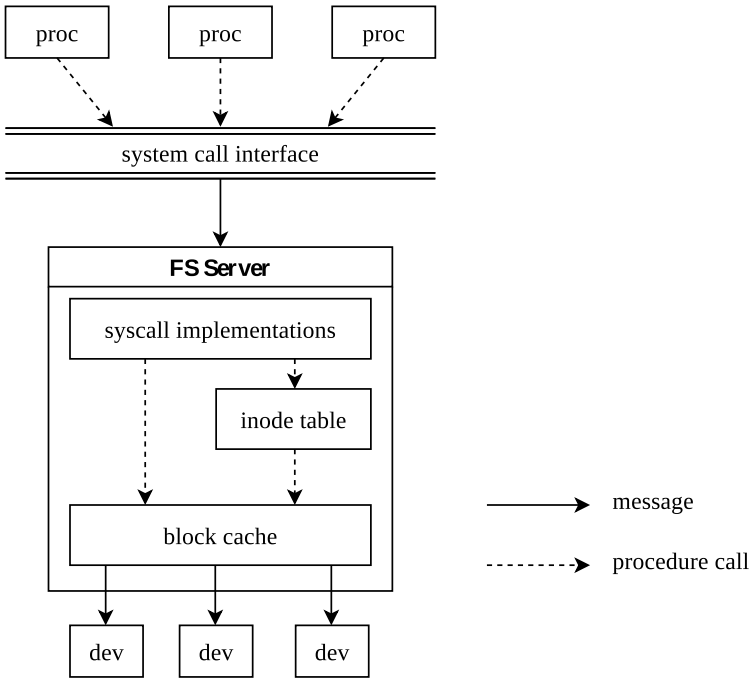

Figure 4. The architecture of the existing MINIX file server

Internally, the FS server maintains two primary data structures: the inode table, and the block cache. A system call request might require multiple accesses to each of these.

MINIX uses the block cache to improve the performance of the file system, by avoiding the need to access the disk devices repeatedly for blocks that have been recently used. When a block is first needed, the server sends a message to the appropriate device driver requesting it. Future requests for the same block will return this cached copy rather than requiring another disk access. When there are no available cache slots, the block that was least recently used will be evicted from the cache and written out to the device. In order to prevent corruption in the case of power loss, the file system also supports the *sync* system call, which will cause any blocks that have been altered to be flushed to the device immediately.

The inode table fulfils a similar role for the inodes that are in use in the file system. When an inode is requested, a copy of the data structure is loaded from the disk via the block cache and kept in memory. This ensures that any inode that is currently being used for a system call will continue to be available, without needing to fetch it from the block. The cache also serves as a reference counting mechanism for inodes, keeping track of the number of times an inode is "in-use" by some part of the file system. For example, the root and working directories for each process must be marked as in-use, preventing them from being unlinked. Similarly, when a file is opened for reading or writing, the file descriptor retains a reference to the inode.

This reference count is quite important in the UNIX file system model, as it determines when a file is eligible for deletion. When a file or directory does not appear anywhere in the hierarchy (i.e., the link count is zero), and is not in use by the file system (i.e., reference count is zero), then it is no longer accessible and should be deleted. This means that a file that has been opened by a process but no longer appears in any directory will continue to be accessible by that process. Several programs use this technique in order to create temporary files that are inaccessible by other processes, by unlinking a file immediately after it has been created.

Both of these data structures exist as logical subcomponents in the existing implementation, being part of the same single program. Accordingly, they inherit the single-threaded nature of the file system and do not address issues of concurrency or mutual exclusion, despite their use of shared data structures.

2.2. Concurrent System Calls

The contents of a file can be accessed after it has been opened using the *open* or the *creat* system calls. If the user has the appropriate permissions to access the file, this establishes a link between the inode and the process, called a file descriptor. This file descriptor contains a pointer to the inode, the permissions with which the file was opened, and the current position within the file. This position normally starts at zero and is incremented during each read or write operation, or can be set explicitly using the *seek* system call. A file descriptor acts like a capability, granting the holder access to the file with a certain set of permissions.

The system calls to read, write, and seek to a position within a file all take a file descriptor as a parameter, instead of a path. Once a file has been opened, these operations on that file can be seen as almost independent from the rest of the file system. We can use this to introduce concurrency, by allowing these operations to be handled by an additional server that sits alongside the FS server.

This *I/O server* will need to interact with the block cache in order to obtain the data from disk. Additionally, it will need to update the information stored in the inode table, to ensure that any changes to the length or the set of data blocks used by the file will be eventually committed. Since both of these caches are data structures in the FS server, this would require the file I/O server to compete with other system call requests, limiting any possible concurrency. In order for this to properly work these components must be accessible as independent server processes.

It is acceptable for these new servers to be sequential, only allowing a single request to be handled at a time. This would still be an improvement over the existing MINIX system, where the smallest atomic operation is the system call. With an independent block cache server, both the I/O and FS servers can process system calls concurrently, with block cache requests being interleaved in some manner.

Another consequence of allowing I/O to occur alongside other system calls is a potential conflict when allocating new inodes or data blocks. When the size of a file or directory is extended, a new data block must be allocated. This is accomplished by fetching the bitmap blocks from the front of the disk, and searching for a spare slot. Since both the file I/O server

and the FS server may need to perform these allocations, we need to ensure that they do not both try to access the bitmap blocks at the same time. We can do this by introducing an allocation server, which is solely responsible for this functionality.

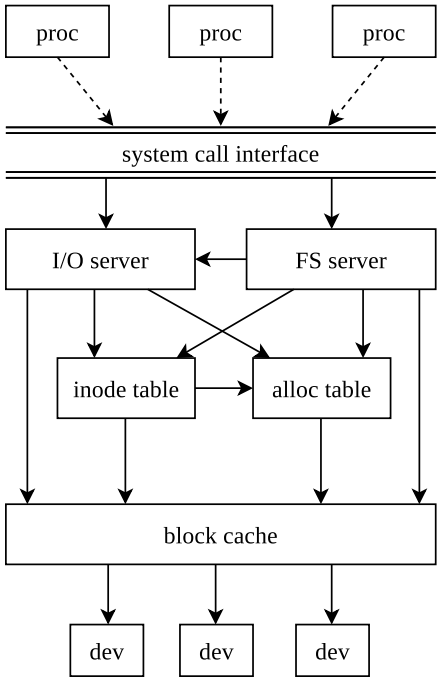

Figure 5. File server that supports concurrent file I/O

Figure 5 shows a process diagram of this architecture, with each rectangle representing an independent server. The arrows between these servers represent messages, with the direction of the arrow showing the initial communication. System call requests from programs are converted by the system call interface into messages for either the I/O server or the FS server as appropriate. In turn, each of these server might need to communicate with the block cache, inode table, or the allocation server.

This figure also shows an additional line of communication from the FS server to the I/O server. This channel is used when the FS server needs to get information about an open file, as described in the following section.

2.3. Inode Data Races

Even if we assume that each of the servers in this design are single-threaded, it isn't clear that the server as a whole is free from race conditions. For example, an I/O system call might make use of the same inode as a normal system call. If both of these servers attempt to access the fields of the inode, we have introduced a classic race hazard. To prevent this, we segment the data structure in such a way that no individual field can be accessed by both servers.

The majority of the operations on inodes are performed by the FS server, so we can assume that it will need to access most of the fields. The I/O server, on the other hand, needs access to the size counter, the list of data blocks, and the modification and access times (so they can be updated). As long as these sets are disjoint, the two servers may access the inode concurrently with a guarantee of mutual exclusion. However, the FS server is still responsible for responding to the *stat* system call, used to obtain information about an inode, including its current size. Instead of returning the value stored on disk or adding a lock for the size field, the FS server can instead ask the I/O server. This separation seems rather natural: if a file is opened and being modified, then only the I/O server would know how large the file is at any

time. The dependency between the *stat* system call and the *write* system call might not be obvious in a lock-based solution, but the communication dependency in our process network is explicit.

2.4. Concurrent File I/O

Once we have an intuition for our server architecture, including the scope of the different components and the dependencies between them, we can look to introduce more concurrency. A policy where the I/O server serialized all incoming requests would ensure there are no conflicts, but would only provide a marginal performance gain. In MINIX, inodes and data blocks cannot be shared between two different files. Therefore, two I/O operations on different files should be safe to perform concurrently. For a single file, we can adopt a straightforward concurrency policy of allowing concurrent read operations, while requiring exclusive access for any writes.

In our implementation, we accomplish this by spawning a new server process the first time an *inode* is opened for reading or writing. This server enforces the concurrent read/exclusive write policy, and continues to run as long as the file is opened by any process.

However, there is a slight complication due to the way that file positions can be shared between processes in UNIX. If one process opens a file and then performs the *fork* system call, the newly created process will have a copy of each of the file descriptors from the original. When one of the processes moves the position in a file descriptor by seeking, reading, or writing, this change should be reflected in the other process as well.

This introduces an additional constraint on our system: any two operations on the same file descriptor must be serialized in order to maintain the semantics of the UNIX file system. So although multiple file operations can be handled concurrently, any two that use the same shared file descriptor cannot. One way to accomplish this is to introduce a new process for each file descriptor to serialize incoming requests. Two separate file descriptors that are connected to the same underlying inode will communicate with the same backend process.

Figure 6 shows a version of the file server where the I/O functionality is provided by a cluster of file descriptor and open file servers. When a file is opened by a process, a file descriptor server is spawned, and the input channel for that server is stored in the inode data structure (in memory, not on disk). Since this field can only be altered when a file is opened or closed, it is free from race hazards due to the sequential nature of the FS server. Now, when the FS server needs to get updated information, it can contact the file descriptor server (and through it the open file server) to obtain it.

The general notion of connecting a file descriptor server to a user process is the approach taken in RMoX[9], an experimental operating system written in occam-π[10]. The technique is used in both the network stack[11] and in several of the file system implementations. Each of these drivers take a different approach to synchronizing between multiple file descriptors that refer to the same file. For example, the *ramdiskfs* driver requires exclusive access to the actual ramdisk device for all read or write operation, while the *hostfs* driver relies on the blocking nature of the host operating system's file operations in order to prevent conflicts.

We have chosen in our design to separate the concerns of the file descriptor from the process that actually interacts with the block cache. This allows us to make the potential concurrency of this "open file" server explicit, and different from that of the necessarily sequential file descriptor. We believe that this architecture is sufficiently general for providing access to resources through a standard POSIX interface.

2.5. Race Conditions in System Calls

The sequential behaviour of the FS server helps to prevent a class of software bugs, called "time-of-check-to-time-of-update" (TOCTOU) race conditions. Although normally viewed

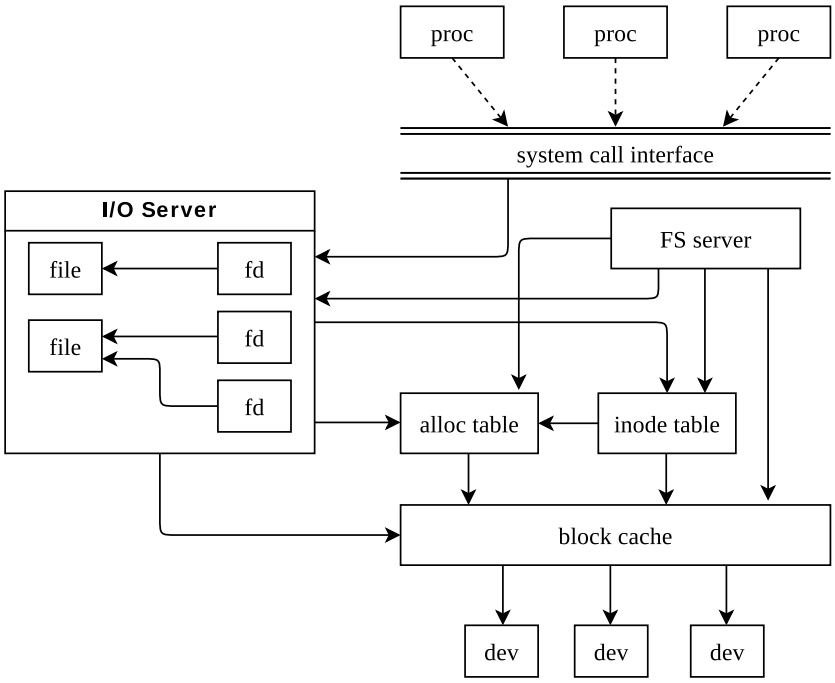

Figure 6. Concurrent I/O server

from a security context in the implementation of privileged user (setuid) programs, this is a class of bugs where a condition is checked and then at some later time, action is taken based upon that check. If the state of the system can change between the check and the update, this could result in incorrect behaviour or worse. An example of this can be seen in the inode link count, used to track the number of times a file is linked in directories on the file system.

The procedure for the *link* system call is as follows:

1. Fetch the inode of the file to be linked
2. Ensure the link count is not already the maximum value
3. Fetch the directory in which the new link will be created
4. Check to see if the target file name already exists
5. Attempt to add the link to the directory
6. If successful, update link count and flag the inode as dirty

The problem is with the second step, where the current value of the link count is checked to ensure it is not already the maximum allowed value. If the file server allowed concurrent system calls, then after this check is performed, another process could come along and add a link to the same file. Afterwards, the original system call would proceed, causing the link the count to overflow. Depending on the inode table implementation, this could result in the file being removed from the file system entirely, and causing the file system to have invalid entries.

One way to solve this sort of problem is to introduce some form of locking, so that the original *link* system call can lock the inode for exclusive access. So long as all of the code that potentially uses the same data structure obey the locking conventions for the object, it is not possible for this race condition to occur. However, this quickly begins to approach the fine-grained layered locking present in the Linux kernel.

A concurrent file system has several different instances of this type of conflict, where some data structure needs to be locked in order to prevent its state from changing during an operation. Although this style of locking can be implemented in a process oriented architecture such as our file server, this introduces *implicit* channels of communications between the different processes, reducing the clarity of the system.

3. Implementation

In this section we present as an example, an implementation of the block cache using the Go programming language.

The file system can access block data that is already in memory several orders of magnitude faster than from disk. This performance is important, as all file system operations involve working with blocks. However, this reliance on the block cache causes the concurrency properties of the cache to propagate upwards through the system. If the cache is only capable of serving a single request at a time, then other requests will have to wait. On the other hand, a concurrent cache would enable the file system to perform multiple operations efficiently. For example, blocks could be fetched from different devices in parallel. Multiple requests could be sent to the device in groups, allowing the disk controller to perform intelligent scheduling or read-ahead.

The primary purpose of the block cache process is to regulate access to the underlying cache data structures, ensuring that they can be updated correctly, free from race conditions. If the block cache is managed by a sequential server process, this is trivial. The real difficulty in making the block cache concurrent is the need to load (and write) blocks to the disk drives. This process may take an extended period of time, during which the cache needs to continue to respond to requests. Arranging for all of this to happen in a way that is correct and safe can be accomplished in several ways; here we describe one possible approach.

Our block cache implementation has the following behaviour on receiving a block request:

- If the block is present in the cache, it should be returned.
- If the block is not present in the cache, it should loaded from disk into an available cache buffer and returned.
- If there are no available cache buffers, a block should be evicted from the cache. This may cause the block to be written out to the disk. When this is finished, the requested block should be loaded from disk and returned.
- If a block is currently being loaded into the cache, the cache should arrange to return the block to the caller when it has completed loading.

We can construct a server that can handle all of these cases appropriately, using channels as the only synchronization and communication primitive. Clients send request messages to the server's input channel and then wait for a response on the output channel.

```
for req := range cache.in {
    switch req := req.(type) {
    case BlockGet:
        resultChan := make(chan *Block)
        cache.out <- resultChan
        cache.get_block(devnum, blocknum, resultChan)
        // other request types here..
    }
}
```

This loop will repeatedly receive a request message from the input channel, and dispatch the appropriate handler for the given request type. Since the loop is entirely sequential, it might appear that only one request can be processed at a time, but this is not the case. Instead of sending the block itself to the output channel, a new channel is created and sent back to the client. This channel acts in a way that is similar to a promise[12] or future[13], as a placeholder for a value that will be available at some point in the future. The client can perform a receive on this channel, which will cause it to block until the value becomes available.

Once this temporary channel has been returned to the caller, the *cache.get_block()* method is invoked in order to retrieve the block. Until this method returns, the server loop will be unable to receive new incoming requests. The strategy here is to perform any manip-

ulation of the cache list data structure during this handling period, with the guarantee that the method has exclusive access during this period.

For the block cache, this involves scanning the cache buffers to see if the desired block is already present. A block that is present may be already loaded, or in the process of being loaded in response to some other client's request. This can be determined by examining the wait list for the block: an empty wait list indicates that the block has already been loaded.

```
func (c *Cache) get_block(devnum int, blocknum int, resultChan chan *Block) {
    bp := c.find_block(devnum, blocknum)
    if bp != nil {
        // the block is in the cache, but might still be loading
        wlist := <-bp.wait
        if wlist.empty() {
            // no one is waiting, so the block is loaded
            bp.wait <- wlist
            resultChan <- bp
            return
        } else {
            // block is loading, add resultChan to the wait list
            wlist.append(resultChan)
            bp.wait <- wlist
            return
        }

    ...
```

This snippet shows the initial part of the *get_block* method, which handles this case. If the block is still being loaded, then the current result channel is added to the wait list and control returns to the server loop.

The mutual exclusion for the wait list here is accomplished using a channel with a buffer size of one. In Go, a send to a channel with space in the buffer is asynchronous, the same as receiving from a channel with a non-empty buffer. So here, receiving from *bp.wait* is the same as locking a mutex for the wait list, while sending the wait list back to the channel is the same as unlocking the mutex. The mutual exclusion for the wait list is essential here, as we will see in the next snippet. This could have been implemented using a server process with channels, a mutex in the data structure, or in several other ways.

```
    } else {
        // the block is not in the cache, and so isn't being loaded
        bp := c.evict_block()
        if bp == nil {
            resultChan <- nil // no available cache slots
            return
```

In this implementation, evicting a block from the cache might involve writing the altered block back to the disk device. This operation is performed synchronously here in order to simplify the example. There is a small chance that all of the cache buffer slots will be in use, so there is nothing that can safely be evicted. In this case, the function just returns `nil`, to indicate that the block could not be loaded successfully. This will eventually lead to a system call failing with an error such as *ENOMEM*.

```
        } else {
            // set block parameters
            bp.devnum = devnum
            bp.blocknum = blocknum

            // create a wait list including this resultChan
            <-bp.wait
            wlist := new_waitlist(resultChan)
            bp.wait <- wlist

            // load the block in the background
            go func() {
```

```
                load_block(bp)

                // block is loaded, notify all waiting channels
                wlist := <-bp.wait
                for _, rchan := range wlist {
                    rchan <- bp
                }
                // release a cleared wait list
                wlist.clear()
                bp.wait <- wlist
            }()
        }
    }
}
```

If the block is not in the cache, and a free cache slot can be obtained, then the process should initiate an asynchronous disk transfer. Before the load is actually requested, the identity parameters (device and block numbers) are set and the result channel is added to the wait list. It is at this stage that a subsequent call to *find_block* would actually locate this data structure, so it is important that this is done before a new incoming request is allowed. Finally, before returning and allowing the server loop to continue, a new goroutine is spawned to load the block from the device. The execution of this function closure will be performed in the background, alongside all of the other goroutines in the program.

When the block load is complete, this worker goroutine acquires the wait list for the block, which might have been updated since it was initially created. The loaded block is then sent over each of the result channels that is present in the wait list. Finally, the list is cleared and then sent back to the channel, unlocking it.

This is just one example implementation of a concurrent block cache. The main server process provides a serialized interface, and arranges for work to be performed asynchronously in order to introduce concurrency. For each component in this file server architecture, there are several possible concurrency policies with varying potential implementations.

4. Conclusions and Future Work

In this paper we have introduced an architecture for a concurrent file server. This server allows for a high level of concurrent file I/O, while serializing all other file system requests. Beginning with the sequential MINIX file server implementation, concurrency is carefully introduced in a controlled manner. Despite the inherently shared nature of file system data structures, this is accomplished without the need for explicit locking mechanisms. Where implicit locks are present, they have a very localized scope. By limiting ourselves to a disciplined model of concurrency, where processes communicate with each other using message passing over explicit channels, we were free to explore a range of possible designs.

Our design represents a compromise between concurrency and tractability, enabling more concurrency than a sequential implementation while making explicit the dependencies between different parts of the system. The importance of the file system's primary data structures, the inode table and the block cache, are underscored in our architecture simply by virtue of these dependencies. This process-oriented style of system design allows us to decouple the concurrency properties of an individual component from the behaviour of the system as a whole. So long as a component maintains the same interface to other components, an implementation could provide any degree of internal concurrency.

This concurrent architecture uses several techniques in order to avoid race hazards, despite the use of shared data structures. Each component in the system is responsible for its internal consistency, which can be accomplished using any of a number of synchronization techniques. Where data is shared between multiple parts of the server, data segmentation is used to ensure mutual exclusion.

We have used the Go programming language as a vehicle to express several concurrency idioms. Providing primitive support for lightweight goroutines and channels, it allows us to explore a range of implementations of CSP-style concurrent systems.

Although our file system design should provide some performance gain on all systems, it has the potential to shine when the workload consists mainly of file I/O. However, if path lookups and inode operations are the dominant file system activity, our server might still exhibit observable pauses in responsiveness. For example, despite being designed for parallel operation, the XFS[14] journalled file system originally suffered poor performance in the presence of repeated inode metadata operations. This was due to the way in which these transactions were serialized to the log stored on disk. More recent versions of the file system introduced a delayed logging system that avoided a layer of locking and moved much of the logging to memory, massively improving the scalability of the file system[15]. It is therefore natural to consider additional opportunities for introducing concurrency, by allowing non-I/O operations to be handled concurrently.

However, the potential for conflicts and interactions between multiple system calls is not entirely clear. We discussed one such conflict in Section 2.5 that occurs from the use of shared inodes. Our experience in designing this file server indicates that there are several instances where a shared data structure must be locked in order to prevent it from changing during the processing of a system call. It would be interesting to example the nature of these conflicts in order to determine opportunities for additional concurrency without resorting to explicit locks.

References

[1] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-131-53271-5.

[2] The Go Programming Language. `http://golang.org/`, March 2011.

[3] The Go Programming Language Specification. `http://golang.org/doc/go_spec.html`, March 2011.

[4] James Whitehead. Serving Web Content with Dynamic Process Networks in Go. In Peter H. Welch, Adam T. Sampson, Jan Baekgaard Pedersen, Jon Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2011*, pages 209–226, jun 2011.

[5] Project minixfs download. `http://www.cs.ox.ac.uk/people/jim.whitehead/go-minixfs-1.0.tar.gz`, July 2012.

[6] OPEN GROUP et al. The Single UNIX Specification, version 2. *System Interface & Headers (XSH)*, (5), 1997.

[7] POSIX.1-2008. The Open Group Base Specifications. Also published as IEEE Std 1003.1-2008, July 2008.

[8] J. Lions. A commentary on the sixth edition UNIX operating system. *Department of Computer Science, The University of New South Wales*, 1977.

[9] Christian L. Jacobsen, Frederick R. M. Barnes, and Brian Vinter. RMoX: A raw-metal occam Experiment. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 269–288, sep 2003.

[10] F.R.M. Barnes and P.H. Welch. occam-pi: blending the best of csp and the pi-calculus, 2006.

[11] A.T. Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, PhD thesis, University of Kent, 2010.

[12] D.P. Friedman and D.S. Wise. *The Impact of Applicative Programming on Multiprocessing*, pages 263–272. Indiana University, Computer Science Department, 1976.

[13] H.C. Baker Jr and C. Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12(8):55–59, 1977.

[14] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, 1996.

[15] XFS Delayed Logging Design. `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/filesystems/xfs-delayed-logging-design.txt`, 2010.