CHAPTER SEVENTEEN

Another Level of Indirection

Diomidis Spinellis

"**ALL PROBLEMS IN COMPUTER SCIENCE CAN BE SOLVED BY ANOTHER LEVEL OF INDIRECTION,"** is a famous quote attributed to Butler Lampson, the scientist who in 1972 envisioned the modern personal computer. The quote rings in my head on various occasions: when I am forced to talk to a secretary instead of the person I wish to communicate with, when I first travel east to Frankfurt in order to finally fly west to Shanghai or Bangalore, and—yes—when I examine a complex system's source code.

Let's start this particular journey by considering the problem of a typical operating system that supports disparate filesystem formats. An operating system may use data residing on its native filesystem, a CD-ROM, or a USB stick. These storage devices may, in turn, employ different filesystem organizations: NTFS or ext3fs for a Windows or Linux native filesystem, ISO-9660 for the CD-ROM, and, often, the legacy FAT-32 filesystem for the USB stick. Each filesystem uses different data structures for managing free space, for storing file metadata, and for organizing files into directories. Therefore, each filesystem requires different code for each operation on a file (`open`, `read`, `write`, `seek`, `close`, `delete`, and so on).

From Code to Pointers

I grew up in an era where different computers more often than not had incompatible filesystems, forcing me to transfer data from one machine to another over serial links. Therefore, the ability to read on my PC a flash card written on my camera never ceases to amaze me. Let's consider how the operating system would structure the code for accessing the different filesystems. One approach would be to employ a `switch` statement for each operation. Consider as an example a hypothetical implementation of the `read` system call under the FreeBSD operating system. Its kernel-side interface would look as follows:

```
int VOP_READ(
        struct vnode *vp,        /* File to read from */
        struct uio *uio,         /* Buffer specification */
        int ioflag,              /* I/O-specific flags */
        struct ucred *cred)      /* User's credentials */
{
    /* Hypothetical implementation */
        switch (vp->filesystem) {
        case FS_NTFS:               /* NTFS-specific code */
        case FS_ISO9660:            /* ISO-9660-specific code */
        case FS_FAT32:              /* FAT-32-specific code */
     /* [...] */
     }
}
```

This approach would bundle together code for the various filesystems, limiting modularity. Worse, adding support for a new filesystem type would require modifying the code of each system call implementation and recompiling the kernel. Moreover, adding a processing step to all the operations of a filesystem (for example, the mapping of remote user credentials) would also require the error-prone modification of each operation with the same boilerplate code.

As you might have guessed, our task at hand calls for some additional levels of indirection. Consider how the FreeBSD operating system—a code base I admire for the maturity of its engineering—solves these problems. Each filesystem defines the operations that it supports as functions and then initializes a `vop_vector` structure with pointers to them. Here are some fields of the `vop_vector` structure:

```
struct vop_vector {
        struct vop_vector *vop_default;
        int (*vop_open)(struct vop_open_args *);
        int (*vop_access)(struct vop_access_args *);
```

and here is how the ISO-9660 filesystem initializes the structure:

```
struct vop_vector cd9660_vnodeops = {
        .vop_default =          &default_vnodeops,
        .vop_open =             cd9660_open,
        .vop_access =           cd9660_access,
        .vop_bmap =             cd9660_bmap,
        .vop_cachedlookup =     cd9660_lookup,
        .vop_getattr =          cd9660_getattr,
        .vop_inactive =         cd9660_inactive,
        .vop_ioctl =            cd9660_ioctl,
        .vop_lookup =           vfs_cache_lookup,
        .vop_pathconf =         cd9660_pathconf,
        .vop_read =             cd9660_read,
        .vop_readdir =          cd9660_readdir,
        .vop_readlink =         cd9660_readlink,
        .vop_reclaim =          cd9660_reclaim,
        .vop_setattr =          cd9660_setattr,
        .vop_strategy =         cd9660_strategy,
};
```

(The `.field = value` syntax is a nifty C99 feature that allows fields of a structure to be initialized in an arbitrary order and in a readable way.) Note that although the complete `vop_vector` structure contains 52 fields, only 16 are defined in the preceding code. As an example, the `vop_write` field is left undefined (getting a value of `NULL`) because writing to files is not supported on ISO-9660 CD-ROMs. Having initialized one such structure for every filesystem type (see the bottom of Figure 17-1), it is then easy to tie this structure to the administrative data associated with each file handle. Then, in the FreeBSD kernel, the filesystem-independent part of the `read` system call implementation appears simply as (see Figure 17-1):

```
struct vop_vector *vop;

      rc = vop->vop_read(a);
```

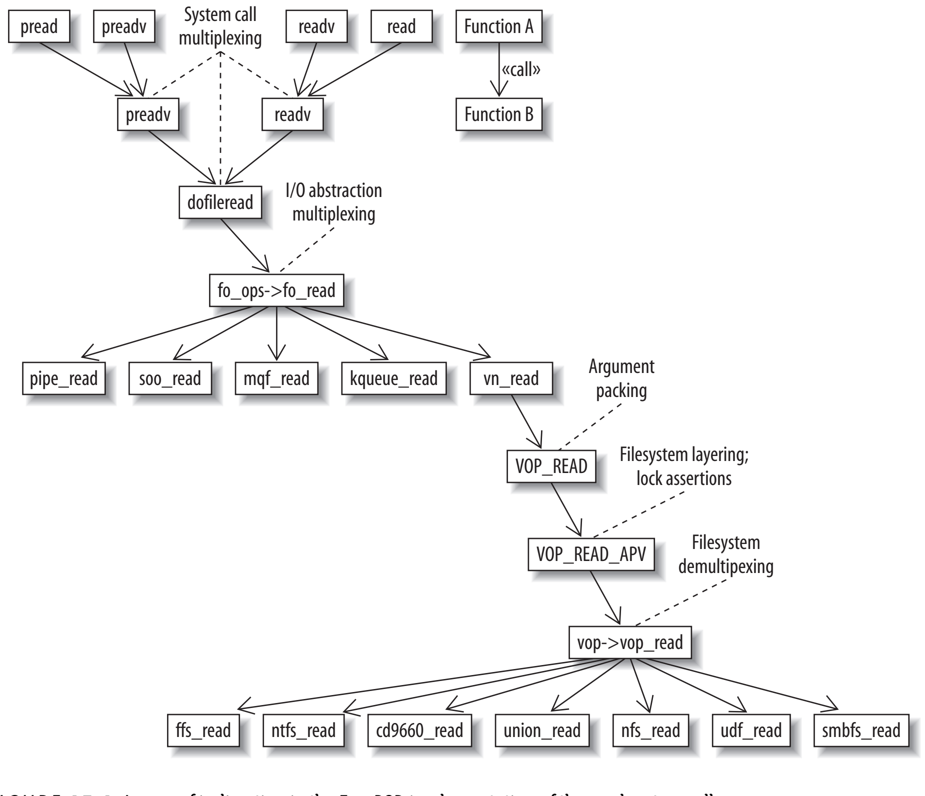

FIGURE 17-1. Layers of indirection in the FreeBSD implementation of the read system call

So, when reading from a CD containing an ISO-9660 filesystem, the previous call through a pointer would actually result in a call to the function `cd9660_read`; in effect:

```
    rc = cd9660_read(a);
```

From Function Arguments to Argument Pointers

Most Unix-related operating systems, such as FreeBSD, Linux, and Solaris, use function pointers to isolate the implementation of a filesystem from the code that accesses its contents. Interestingly, FreeBSD also employs indirection to abstract the read function's arguments.

When I first encountered the call `vop->vop_read(a)`, shown in the previous section, I asked myself what that `a` argument was and what happened to the original four arguments of the hypothetical implementation of the `VOP_READ` function we saw earlier. After some digging, I found that the kernel uses another level of indirection to layer filesystems on top of each other to an arbitrary depth. This layering allows a filesystem to offer some services (such as translucent views, compression, and encryption) based on the services of another underlying filesystem. Two mechanisms work cleverly together to support this feature: one allows a single bypass function to modify the arguments of any `vop_vector` function, while another allows all undefined `vop_vector` functions to be redirected to the underlying filesystem layer.

You can see both mechanisms in action in Figure 17-2. The figure illustrates three filesystems layered on top of one another. On top lies the *umapfs* filesystem, which the system administrator mounted in order to map user credentials. This is valuable if the system where this particular disk was created used different user IDs. For instance, the administrator might want user ID 1013 on the underlying filesystem to appear as user ID 5325.

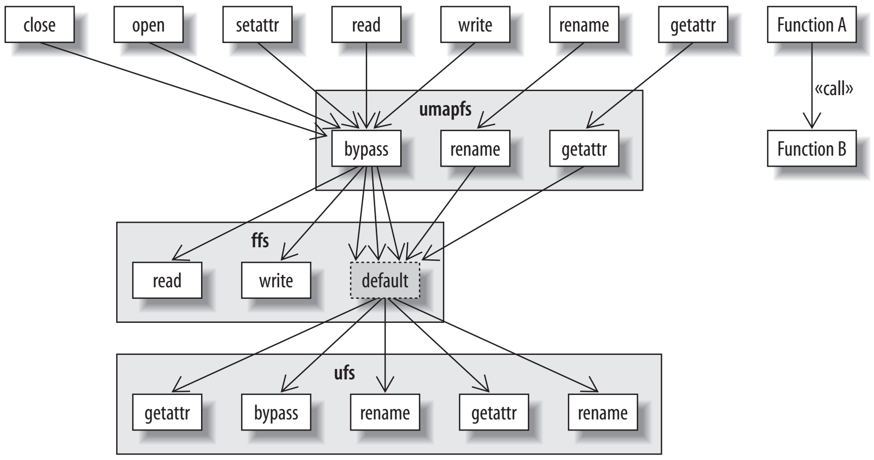

FIGURE 17-2. Example of filesystem layering

Beneath the top filesystem lies the Berkeley Fast Filesystem (*ffs*), the time- and space-efficient filesystem used by default in typical FreeBSD installations. The *ffs* in turn, for most of its operations, relies on the code of the original 4.2 BSD filesystem implementation *ufs*.

In the example shown in the figure, most system calls pass through a common bypass function in *umapfs* that maps the user credentials. Only a few system calls, such as `rename` and `getattr`, have their own implementations in *umapfs*. The *ffs* layer provides optimized implementations of `read` and `write`; both rely on a filesystem layout that is more efficient than the one employed by *ufs*. Most other operations, such as `open`, `close`, `getattr`, `setatr`, and `rename`, are handled in the traditional way. Thus, a `vop_default` entry in the *ffs* `vop_vector` structure directs all those functions to call the underlying *ufs* implementations. For example, a `read` system call will pass through `umapfs_bypass` and `ffs_read`, whereas a `rename` call will pass through `umapfs_rename` and `ufs_rename`.

Both mechanisms, the bypass and the default, pack the four arguments into a single structure to provide commonality between the different filesystem functions, and also support the groundwork for the bypass function. This is a beautiful design pattern that is easily overlooked within the intricacies of the C code required to implement it.

The four arguments are packed into a single structure, which as its first field (`a_gen.a_desc`) contains a description of the structure's contents (`vop_read_desc`, in the following code). As you can see in Figure 17-1, a `read` system call on a file in the FreeBSD kernel will trigger a call to `vn_read`, which will set up the appropriate low-level arguments and call `VOP_READ`. This will pack the arguments and call `VOP_READ_APV`, which finally calls `vop->vop_read` and thereby the actual filesystem read function:

```
struct vop_read_args {
        struct vop_generic_args a_gen;
        struct vnode *a_vp;
        struct uio *a_uio;
        int a_ioflag;
        struct ucred *a_cred;
};

static __inline int VOP_READ(
        struct vnode *vp,
        struct uio *uio,
        int ioflag,
        struct ucred *cred)
{
        struct vop_read_args a;

        a.a_gen.a_desc = &vop_read_desc;
        a.a_vp = vp;
        a.a_uio = uio;
        a.a_ioflag = ioflag;
        a.a_cred = cred;
        return (VOP_READ_APV(vp->v_op, &a));
}
```

This same elaborate dance is performed for calling all other vop_vector functions (stat, write, open, close, and so on). The vop_vector structure also contains a pointer to a bypass function. This function gets the packed arguments and, after possibly performing some modifications on them (such as, perhaps, mapping user credentials from one administrative domain to another) passes control to the appropriate underlying function for the specific call through the a_desc field.

Here is an excerpt of how the *nullfs* filesystem implements the bypass function. The *nullfs* filesystem just duplicates a part of an existing filesystem into another location of the global filesystem namespace. Therefore, for most of its operations, it can simply have its bypass function call the corresponding function of the underlying filesystem:

```
#define VCALL(c) ((c)->a_desc->vdesc_call(c))
int
null_bypass(struct vop_generic_args *ap)
{
    /* ... */
      error = VCALL(ap);
```

In the preceding code, the macro VCALL(ap) will bump the *vnode* operation that called null_bypass (for instance VOP_READ_APV) one filesystem level down. You can see this trick in action in Figure 17-3.

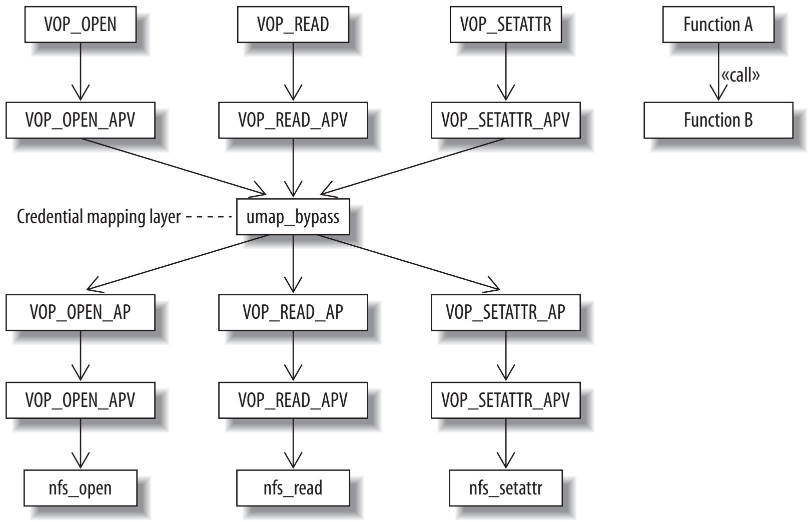

FIGURE 17-3. Routing system calls through a bypass function

In addition, the vop_vector contains a field named default that is a pointer to the vop_vector structure of the underlying filesystem layer. Through that field, if a filesystem doesn't implement some functionality, the request is passed on to a lower level. By populating the bypass and the default fields of its vop_vector structure, a filesystem can choose among:

- Handling an incoming request on its own
- Bypassing the request to a lower-level filesystem after modifying some arguments
- Directly calling the lower-level filesystem

In my mind, I visualize this as bits sliding down the ramps, kickers, and spinners of an elaborate pinball machine. The following example from the read system call implementation shows how the system locates the function to call:

```
int
VOP_READ_APV(struct vop_vector *vop, struct vop_read_args *a)
{
    [...]
      /*
 * Drill down the filesystem layers to find one
 * that implements the function or a bypass
 */
while (vop != NULL &&
            vop->vop_read == NULL && vop->vop_bypass == NULL)
                vop = vop->vop_default;
      /* Call the function or the bypass */
        if (vop->vop_read != NULL)
                rc = vop->vop_read(a);
        else
                rc = vop->vop_bypass(&a->a_gen);
```

Elegantly, at the bottom of all filesystem layers lies a filesystem that returns the Unix "operation not supported" error (`EOPNOTSUPP`) for any function that wasn't implemented by the filesystems layered on top of it. This is our pinball's drain:

```
#define VOP_EOPNOTSUPP  ((void*)(uintptr_t)vop_eopnotsupp)

struct vop_vector default_vnodeops = {
        .vop_default =          NULL,
        .vop_bypass =           VOP_EOPNOTSUPP,
}

int
vop_eopnotsupp(struct vop_generic_args *ap)
{
        return (EOPNOTSUPP);
}
```

From Filesystems to Filesystem Layers

For a concrete example of filesystem layering, consider the case where you mount on your computer a remote filesystem using the NFS (Network File System) protocol. Unfortunately, in your case, the user and group identifiers on the remote system don't match those used on your computer. However, by interposing a *umapfs* filesystem over the actual NFS implementation, we can specify through external files the correct user and group mappings. Figure 17-3 illustrates how some operating system kernel function calls first get routed through the bypass function of *umpafs*—`umap_bypass`—before continuing their journey to the corresponding NFS client functions.

In contrast to the null_bypass function, the implementation of umap_bypass actually does some work before making a call to the underlying layer. The vop_generic_args structure passed as its argument contains a description of the actual arguments for each *vnode* operation:

```
/*
 * A generic structure.
 * This can be used by bypass routines to identify generic arguments.
 */
struct vop_generic_args {
        struct vnodeop_desc *a_desc;
        /* other random data follows, presumably */
};

/*
 * This structure describes the vnode operation taking place.
 */
struct vnodeop_desc {
        char    *vdesc_name;            /* a readable name for debugging */
        int      vdesc_flags;           /* VDESC_* flags */
        vop_bypass_t    *vdesc_call;    /* Function to call */

        /*
         * These ops are used by bypass routines to map and locate arguments.
         * Creds and procs are not needed in bypass routines, but sometimes
         * they are useful to (for example) transport layers.
         * Nameidata is useful because it has a cred in it.
         */
        int     *vdesc_vp_offsets;      /* list ended by VDESC_NO_OFFSET */
        int     vdesc_vpp_offset;       /* return vpp location */
        int     vdesc_cred_offset;      /* cred location, if any */
        int     vdesc_thread_offset;    /* thread location, if any */
        int     vdesc_componentname_offset; /* if any */
};
```

For instance, the vnodeop_desc structure for the arguments passed to the vop_read operation is the following:

```
struct vnodeop_desc vop_read_desc = {
        "vop_read",
        0,
        (vop_bypass_t *)VOP_READ_AP,
        vop_read_vp_offsets,
        VDESC_NO_OFFSET,
        VOPARG_OFFSETOF(struct vop_read_args,a_cred),
        VDESC_NO_OFFSET,
        VDESC_NO_OFFSET,
};
```

Importantly, apart from the name of the function (used for debugging purposes) and the underlying function to call (VOP_READ_AP), the structure contains in its vdesc_cred_offset field the location of the user credential data field (a_cred) within the read call's arguments. By using this field, umap_bypass can map the credentials of *any* vnode operation with the following code:

```
        if (descp->vdesc_cred_offset != VDESC_NO_OFFSET) {
                credpp = VOPARG_OFFSETTO(struct ucred**,
                    descp->vdesc_cred_offset, ap);
                /* Save old values */
                savecredp = (*credpp);
                if (savecredp != NOCRED)
                        (*credpp) = crdup(savecredp);
                credp = *credpp;
                /* Map all ids in the credential structure. */
                umap_mapids(vp1->v_mount, credp);
        }
```

What we have here is a case of data describing the format of other data: a redirection in terms of data abstraction. This *metadata* allows the credential mapping code to manipulate the arguments of arbitrary system calls.

From Code to a Domain-Specific Language

You may have noticed that some of the code associated with the implementation of the read system call, such as the packing of its arguments into a structure or the logic for calling the appropriate function, is highly stylized and is probably repeated in similar forms for all 52 other interfaces. Another implementation detail, which we have not so far discussed and which can keep me awake at nights, concerns locking.

Operating systems must ensure that various processes running concurrently don't step on each other's toes when they modify data without coordination between them. On modern multithreaded, multi-core processors, ensuring data consistency by maintaining one mutual exclusion lock for all critical operating system structures (as was the case in older operating system implementations) would result in an intolerable drain on performance. Therefore, locks are nowadays held over fine-grained objects, such as a user's credentials or a single buffer. Furthermore, because obtaining and releasing locks can be expensive operations, ideally once a lock is held it should not be released if it will be needed again in short order. These locking specifications can best be described through preconditions (what the state of a lock must be before entering a function) and postconditions (the state of the lock at a function's exit).

As you can imagine, programming under those constraints and verifying the code's correctness can be hellishly complicated. Fortunately for me, another level of indirection can be used to bring some sanity into the picture. This indirection handles both the redundancy of packing code and the fragile locking requirements.

In the FreeBSD kernel, the interface functions and data structures we've examined, such as `VOP_READ_AP`, `VOP_READ_APV`, and `vop_read_desc`, aren't directly written in C. Instead, a domain-specific language is used to specify the types of each call's arguments and their locking pre- and postconditions. Such an implementation style always raises my pulse, because the productivity boost it gives can be enormous. Here is an excerpt from the `read` system call specification:

```
#
#% read         vp      L L L
#
vop_read {
        IN struct vnode *vp;
        INOUT struct uio *uio;
        IN int ioflag;
        IN struct ucred *cred;
};
```

From specifications such as the above, an *awk* script creates:

- C code for packing the arguments of the functions into a single structure
- Declarations for the structures holding the packed arguments and the functions doing the work
- Initialized data specifying the contents of the packed argument structures
- The boilerplate C code we saw used for implementing filesystem layers
- Assertions for verifying the state of the locks when the function enters and exits

In the FreeBSD version 6.1 implementation of the *vnode* call interface, all in all, 588 lines of domain-specific code expand into 4,339 lines of C code and declarations.

Such compilation from a specialized high-level domain-specific language into C is quite common in the computing field. For example, the input to the lexical analyzer generator *lex* is a file that maps regular expressions into actions; the input to the parser generator *yacc* is a language's grammar and corresponding production rules. Both systems (and their descendants *flex* and *bison*) generate C code implementing the high-level specifications. A more extreme case involves the early implementations of the C++ programming language. These consisted of a preprocessor, *cfront*, that would compile C++ code into C.

In all these cases, C is used as a portable assembly language. When used appropriately, domain-specific languages increase the code's expressiveness and thereby programmer productivity. On the other hand, a gratuitously used obscure domain-specific language can make a system more difficult to comprehend, debug, and maintain.

The handling of locking assertions deserves more explanation. For each argument, the code lists the state of its lock for three instances: when the function is entered, when the function exits successfully, and when the function exits with an error—an elegantly clear separation of concerns. For example, the preceding specification of the `read` call indicated that the `vp` argument should be locked in all three cases. More complex scenarios are also possible. The following code excerpt indicates that the `rename` call arguments `fdvp` and `fvp` are always unlocked, but the argument `tdvp` has a process-exclusive lock when the routine is called. All arguments should be unlocked when the function terminates:

```
#
#% rename        fdvp    U U U
#% rename        fvp     U U U
#% rename        tdvp    E U U
#
```

The locking specification is used to instrument the C code with assertions at the function's entry, the function's normal exit, and the function's error exit. For example, the code at the entry point of the rename function contains the following assertions:

```
ASSERT_VOP_UNLOCKED(a->a_fdvp, "VOP_RENAME");
ASSERT_VOP_UNLOCKED(a->a_fvp, "VOP_RENAME");
ASSERT_VOP_ELOCKED(a->a_tdvp, "VOP_RENAME");
```

Although assertions, such as the preceding one, don't guarantee that the code will be bug-free, they do at least provide an early-fail indication that will diagnose errors during system testing, before they destabilize the system in a way that hinders debugging. When I read complex code that lacks assertions, it's like watching acrobats performing without a net: an impressive act where a small mistake can result in considerable grief.

Multiplexing and Demultiplexing

As you can see back in Figure 17-1, the processing of the read system call doesn't start from VOP_READ. VOP_READ is actually called from vn_read, which itself is called through a function pointer.

This level of indirection is used for another purpose. The Unix operating system and its derivatives treat all input and output sources uniformly. Thus, instead of having separate system calls for reading from, say, a file, a socket, or a pipe, the read system call can read from any of those I/O abstractions. I find this design both elegant and useful; I've often relied on it, using tools in ways their makers couldn't have anticipated. (This statement says more about the age of the tools I use than my creativity.)

The indirection appearing in the middle of Figure 17-1 is the mechanism FreeBSD uses for providing this high-level I/O abstraction independence. Associated with each file descriptor is a function pointer leading to the code that will service the particular request: pipe_read for pipes, soo_read for sockets, mqf_read for POSIX message queues, kqueue_read for kernel event queues, and, finally, vn_read for actual files.

So far, in our example, we have encountered two instances where function pointers are used to dispatch a request to different functions. Typically, in such cases, a function pointer is used to demultiplex a single request to multiple potential providers. This use of indirection is so common that it forms an important element of object-oriented languages, in the form of dynamic dispatch to various subclass methods. To me, the manual implementation of dynamic dispatch in a procedural language like C is a distinguishing mark of an expert programmer. (Another is the ability to write a structured program in assembly language or Fortran.)

Indirection is also often introduced as a way to factor common functionality. Have a look at the top of Figure 17-1. Modern Unix systems have four variants of the vanilla read system call. The system call variants starting with p (pread, preadv) allow the specification of a file position together with the call. The variants ending with a v (readv, preadv) allow the specification of a vector of I/O requests instead of a single one. Although I consider this proliferation of system calls inelegant and against the spirit of Unix, applications programmers seem to depend on them for squeezing every bit of performance out of the Web or database servers they implement.

All these calls share some common code. The FreeBSD implementation introduces indirection through additional functions in order to avoid code duplication. The function `kern_preadv` handles the common parts of the positional system call variants, while `kern_readv` handles the remaining two system calls. The functionality common in all four is handled by another function, `dofileread`. In my mind, I can picture the joy developers got from factoring out the code common to those functions by introducing more levels of indirection. I always feel elated if, after committing a refactoring change, the lines I add are less than the lines I remove.

The journey from our call to a read function in our user-level program to the movement of a disk head to fetch our data from a platter is a long and tortuous one. In our description, we haven't considered what happens above the kernel layer (virtual machines, buffering, data representation), or what happens when a filesystem handles a request (buffering again, device drivers, data representation). Interestingly, there's a pleasant symmetry between the two ends we haven't covered: both involve hardware interfaces (virtual machines, such as the JVM at the top, and real interfaces at the bottom), buffering (to minimize system calls at the top, and to optimize the hardware's performance at the bottom), and data representation (to interact with the user's locale at the top, and to match the physical layer's requirements at the bottom). It seems that indirection is everywhere we care to cast our eyes. In the representative chunk we've looked at, nine levels of function calls, two indirections through function pointers, and a domain-specific language provided us with a representative view of its power.

Layers Forever?

We could continue looking at more code examples forever, so it is worth bringing our discussion to an end by noting that Lampson attributes the aphorism that started our exploration (all problems in computer science can be solved by another level of indirection) to David Wheeler, the inventor of the subroutine. Significantly, Wheeler completed his quote with another phrase: "But that usually will create another problem." Indeed, indirection and layering add space and time overhead, and can obstruct the code's comprehensibility.

The time and space overhead is often unimportant, and should rarely concern us. In most cases, the delays introduced by an extra pointer lookup or subroutine call are insignificant in the greater scheme of things. In fact, nowadays the tendency in modern programming languages is for some operations to always happen through a level of indirection in order to provide an additional measure of flexibility. Thus, for example, in Java and C#, almost all accesses to objects go through one pointer indirection, to allow for automatic garbage collection. Also, in Java, almost all calls to instance methods are dispatched through a lookup table, in order to allow inheriting classes to override a method at runtime.

Despite these overheads that burden all object accesses and method calls, both platforms are doing fine in the marketplace, thank you very much. In other cases, compilers optimize away the indirection we developers put in our code. Thus, most compilers detect cases where calling a function is more expensive than substituting its code inline, and automatically perform this inlining.

Then again, when we're operating at the edge of performance, indirection can be a burden. One trick that developers trying to feed gigabit network interfaces use to speed up their code is to combine functionality of different levels of the network stack, collapsing some layers of abstraction. But these are extreme cases.

On the other hand, the effect that indirection has on the comprehensibility of our code is a very important concern, because over the last 50 years, in contrast to the dizzying increases in CPU speeds, the ability of humans to understand code hasn't improved much. Therefore, the proponents of agile processes advise us to be especially wary when introducing layering to handle some vague, unspecified requirements we imagine might crop up in the future rather than today's concrete needs. As Bart Smaalders quipped when discussing performance anti-patterns: "Layers are for cakes, not for software."