42

Crash Consistency: FSCK and Journaling

As we've seen thus far, the file system manages a set of data structures to implement the expected abstractions: files, directories, and all of the other metadata needed to support the basic abstraction that we expect from a file system. Unlike most data structures (for example, those found in memory of a running program), file system data structures must **persist**, i.e., they must survive over the long haul, stored on devices that retain data despite power loss (such as hard disks or flash-based SSDs).

One major challenge faced by a file system is how to update persistent data structures despite the presence of a **power loss** or **system crash**. Specifically, what happens if, right in the middle of updating on-disk structures, someone trips over the power cord and the machine loses power? Or the operating system encounters a bug and crashes? Because of power losses and crashes, updating a persistent data structure can be quite tricky, and leads to a new and interesting problem in file system implementation, known as the **crash-consistency problem**.

This problem is quite simple to understand. Imagine you have to update two on-disk structures, A and B, in order to complete a particular operation. Because the disk only services a single request at a time, one of these requests will reach the disk first (either A or B). If the system crashes or loses power after one write completes, the on-disk structure will be left in an **inconsistent** state. And thus, we have a problem that all file systems need to solve:

THE CRUX: HOW TO UPDATE THE DISK DESPITE CRASHES

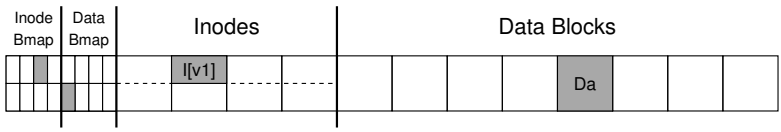
The system may crash or lose power between any two writes, and thus the on-disk state may only partially get updated. After the crash, the system boots and wishes to mount the file system again (in order to access files and such). Given that crashes can occur at arbitrary points in time, how do we ensure the file system keeps the on-disk image in a reasonable state?

In this chapter, we'll describe this problem in more detail, and look at some methods file systems have used to overcome it. We'll begin by examining the approach taken by older file systems, known as **fsck** or the **file system checker**. We'll then turn our attention to another approach, known as **journaling** (also known as **write-ahead logging**), a technique which adds a little bit of overhead to each write but recovers more quickly from crashes or power losses. We will discuss the basic machinery of journaling, including a few different flavors of journaling that Linux ext3 [T98,PAA05] (a relatively modern journaling file system) implements.

42.1 A Detailed Example

To kick off our investigation of journaling, let's look at an example. We'll need to use a **workload** that updates on-disk structures in some way. Assume here that the workload is simple: the append of a single data block to an existing file. The append is accomplished by opening the file, calling `lseek()` to move the file offset to the end of the file, and then issuing a single 4KB write to the file before closing it.

Let's also assume we are using standard simple file system structures on the disk, similar to file systems we have seen before. This tiny example includes an **inode bitmap** (with just 8 bits, one per inode), a **data bitmap** (also 8 bits, one per data block), inodes (8 total, numbered 0 to 7, and spread across four blocks), and data blocks (8 total, numbered 0 to 7). Here is a diagram of this file system:



If you look at the structures in the picture, you can see that a single inode is allocated (inode number 2), which is marked in the inode bitmap, and a single allocated data block (data block 4), also marked in the data bitmap. The inode is denoted I[v1], as it is the first version of this inode; it will soon be updated (due to the workload described above).

Let's peek inside this simplified inode too. Inside of I[v1], we see:

```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

In this simplified inode, the `size` of the file is `1` (it has one block allocated), the first direct pointer points to block `4` (the first data block of the file, Da), and all three other direct pointers are set to `null` (indicating

that they are not used). Of course, real inodes have many more fields; see previous chapters for more information.

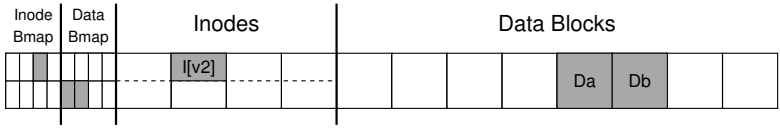
When we append to the file, we are adding a new data block to it, and thus must update three on-disk structures: the inode (which must point to the new block as well as have a bigger size due to the append), the new data block Db, and a new version of the data bitmap (call it B[v2]) to indicate that the new data block has been allocated.

Thus, in the memory of the system, we have three blocks which we must write to disk. The updated inode (inode version 2, or I[v2] for short) now looks like this:

```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

The updated data bitmap (B[v2]) now looks like this: 00001100. Finally, there is the data block (Db), which is just filled with whatever it is users put into files. Stolen music perhaps?

What we would like is for the final on-disk image of the file system to look like this:



To achieve this transition, the file system must perform three separate writes to the disk, one each for the inode (I[v2]), bitmap (B[v2]), and data block (Db). Note that these writes usually don't happen immediately when the user issues a `write()` system call; rather, the dirty inode, bitmap, and new data will sit in main memory (in the **page cache** or **buffer cache**) for some time first; then, when the file system finally decides to write them to disk (after say 5 seconds or 30 seconds), the file system will issue the requisite write requests to the disk. Unfortunately, a crash may occur and thus interfere with these updates to the disk. In particular, if a crash happens after one or two of these writes have taken place, but not all three, the file system could be left in a funny state.

Crash Scenarios

To understand the problem better, let's look at some example crash scenarios. Imagine only a single write succeeds; there are thus three possible outcomes, which we list here:

- **Just the data block (Db) is written to disk.** In this case, the data is on disk, but there is no inode that points to it and no bitmap that even says the block is allocated. Thus, it is as if the write never occurred. This case is not a problem at all, from the perspective of file-system crash consistency[1].

- **Just the updated inode (I[v2]) is written to disk.** In this case, the inode points to the disk address (5) where Db was about to be written, but Db has not yet been written there. Thus, if we trust that pointer, we will read **garbage** data from the disk (the old contents of disk address 5).

  Further, we have a new problem, which we call a **file-system inconsistency**. The on-disk bitmap is telling us that data block 5 has not been allocated, but the inode is saying that it has. This disagreement in the file system data structures is an inconsistency in the data structures of the file system; to use the file system, we must somehow resolve this problem (more on that below).

- **Just the updated bitmap (B[v2]) is written to disk.** In this case, the bitmap indicates that block 5 is allocated, but there is no inode that points to it. Thus the file system is inconsistent again; if left unresolved, this write would result in a **space leak**, as block 5 would never be used by the file system.

There are also three more crash scenarios in this attempt to write three blocks to disk. In these cases, two writes succeed and the last one fails:

- **The inode (I[v2]) and bitmap (B[v2]) are written to disk, but not data (Db).** In this case, the file system metadata is completely consistent: the inode has a pointer to block 5, the bitmap indicates that 5 is in use, and thus everything looks OK from the perspective of the file system's metadata. But there is one problem: 5 has garbage in it again.

- **The inode (I[v2]) and the data block (Db) are written, but not the bitmap (B[v2]).** In this case, we have the inode pointing to the correct data on disk, but again have an inconsistency between the inode and the old version of the bitmap (B1). Thus, we once again need to resolve the problem before using the file system.

- **The bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2]).** In this case, we again have an inconsistency between the inode and the data bitmap. However, even though the block was written and the bitmap indicates its usage, we have no idea which file it belongs to, as no inode points to the file.

[1] However, it might be a problem for the user, who just lost some data!

The Crash Consistency Problem

Hopefully, from these crash scenarios, you can see the many problems that can occur to our on-disk file system image because of crashes: we can have inconsistency in file system data structures; we can have space leaks; we can return garbage data to a user; and so forth. What we'd like to do ideally is move the file system from one consistent state (e.g., before the file got appended to) to another **atomically** (e.g., after the inode, bitmap, and new data block have been written to disk). Unfortunately, we can't do this easily because the disk only commits one write at a time, and crashes or power loss may occur between these updates. We call this general problem the **crash-consistency problem** (we could also call it the **consistent-update problem**).

42.2 Solution #1: The File System Checker

Early file systems took a simple approach to crash consistency. Basically, they decided to let inconsistencies happen and then fix them later (when rebooting). A classic example of this lazy approach is found in a tool that does this: **fsck**[2]. `fsck` is a UNIX tool for finding such inconsistencies and repairing them [M86]; similar tools to check and repair a disk partition exist on different systems. Note that such an approach can't fix all problems; consider, for example, the case above where the file system looks consistent but the inode points to garbage data. The only real goal is to make sure the file system metadata is internally consistent.

The tool `fsck` operates in a number of phases, as summarized in McKusick and Kowalski's paper [MK96]. It is run *before* the file system is mounted and made available (`fsck` assumes that no other file-system activity is on-going while it runs); once finished, the on-disk file system should be consistent and thus can be made accessible to users.

Here is a basic summary of what `fsck` does:

- **Superblock:** `fsck` first checks if the superblock looks reasonable, mostly doing sanity checks such as making sure the file system size is greater than the number of blocks allocated. Usually the goal of these sanity checks is to find a suspect (corrupt) superblock; in this case, the system (or administrator) may decide to use an alternate copy of the superblock.
- **Free blocks:** Next, `fsck` scans the inodes, indirect blocks, double indirect blocks, etc., to build an understanding of which blocks are currently allocated within the file system. It uses this knowledge to produce a correct version of the allocation bitmaps; thus, if there is any inconsistency between bitmaps and inodes, it is resolved by trusting the information within the inodes. The same type of check is performed for all the inodes, making sure that all inodes that look like they are in use are marked as such in the inode bitmaps.

[2] Pronounced either "eff-ess-see-kay", "eff-ess-check", or, if you don't like the tool, "eff-suck". Yes, serious professional people use this term.

- **Inode state:** Each inode is checked for corruption or other problems. For example, `fsck` makes sure that each allocated inode has a valid type field (e.g., regular file, directory, symbolic link, etc.). If there are problems with the inode fields that are not easily fixed, the inode is considered suspect and cleared by `fsck`; the inode bitmap is correspondingly updated.
- **Inode links:** `fsck` also verifies the link count of each allocated inode. As you may recall, the link count indicates the number of different directories that contain a reference (i.e., a link) to this particular file. To verify the link count, `fsck` scans through the entire directory tree, starting at the root directory, and builds its own link counts for every file and directory in the file system. If there is a mismatch between the newly-calculated count and that found within an inode, corrective action must be taken, usually by fixing the count within the inode. If an allocated inode is discovered but no directory refers to it, it is moved to the `lost+found` directory.
- **Duplicates:** `fsck` also checks for duplicate pointers, i.e., cases where two different inodes refer to the same block. If one inode is obviously bad, it may be cleared. Alternately, the pointed-to block could be copied, thus giving each inode its own copy as desired.
- **Bad blocks:** A check for bad block pointers is also performed while scanning through the list of all pointers. A pointer is considered "bad" if it obviously points to something outside its valid range, e.g., it has an address that refers to a block greater than the partition size. In this case, `fsck` can't do anything too intelligent; it just removes (clears) the pointer from the inode or indirect block.
- **Directory checks:** `fsck` does not understand the contents of user files; however, directories hold specifically formatted information created by the file system itself. Thus, `fsck` performs additional integrity checks on the contents of each directory, making sure that "." and ".." are the first entries, that each inode referred to in a directory entry is allocated, and ensuring that no directory is linked to more than once in the entire hierarchy.

As you can see, building a working `fsck` requires intricate knowledge of the file system; making sure such a piece of code works correctly in all cases can be challenging [G+08]. However, fsck (and similar approaches) have a bigger and perhaps more fundamental problem: they are *too slow*. With a very large disk volume, scanning the entire disk to find all the allocated blocks and read the entire directory tree may take many minutes or hours. Performance of `fsck`, as disks grew in capacity and RAIDs grew in popularity, became prohibitive (despite recent advances [M+13]).

At a higher level, the basic premise of `fsck` seems just a tad irrational. Consider our example above, where just three blocks are written to the disk; it is incredibly expensive to scan the entire disk to fix problems that occurred during an update of just three blocks. This situation is akin to dropping your keys on the floor in your bedroom, and then com-

mencing a *search-the-entire-house-for-keys* recovery algorithm, starting in the basement and working your way through every room. It works but is wasteful. Thus, as disks (and RAIDs) grew, researchers and practitioners started to look for other solutions.

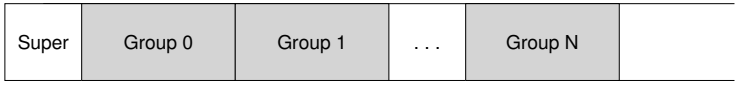
42.3 Solution #2: Journaling (or Write-Ahead Logging)

Probably the most popular solution to the consistent update problem is to steal an idea from the world of database management systems. That idea, known as **write-ahead logging**, was invented to address exactly this type of problem. In file systems, we usually call write-ahead logging **journaling** for historical reasons. The first file system to do this was Cedar [H87], though many modern file systems use the idea, including Linux ext3 and ext4, reiserfs, IBM's JFS, SGI's XFS, and Windows NTFS.

The basic idea is as follows. When updating the disk, before overwriting the structures in place, first write down a little note (somewhere else on the disk, in a well-known location) describing what you are about to do. Writing this note is the "write ahead" part, and we write it to a structure that we organize as a "log"; hence, write-ahead logging.

By writing the note to disk, you are guaranteeing that if a crash takes places during the update (overwrite) of the structures you are updating, you can go back and look at the note you made and try again; thus, you will know exactly what to fix (and how to fix it) after a crash, instead of having to scan the entire disk. By design, journaling thus adds a bit of work during updates to greatly reduce the amount of work required during recovery.

We'll now describe how **Linux ext3**, a popular journaling file system, incorporates journaling into the file system. Most of the on-disk structures are identical to **Linux ext2**, e.g., the disk is divided into block groups, and each block group has an inode and data bitmap as well as inodes and data blocks. The new key structure is the journal itself, which occupies some small amount of space within the partition or on another device. Thus, an ext2 file system (without journaling) looks like this:



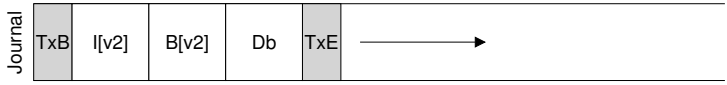
Assuming the journal is placed within the same file system image (though sometimes it is placed on a separate device, or as a file within the file system), an ext3 file system with a journal looks like this:

The real difference is just the presence of the journal, and of course, how it is used.

Data Journaling

Let's look at a simple example to understand how **data journaling** works. Data journaling is available as a mode with the Linux ext3 file system, from which much of this discussion is based.

Say we have our canonical update again, where we wish to write the 'inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk again. Before writing them to their final disk locations, we are now first going to write them to the log (a.k.a. journal). This is what this will look like in the log:



You can see we have written five blocks here. The transaction begin (TxB) tells us about this update, including information about the pending update to the file system (e.g., the final addresses of the blocks I[v2], B[v2], and Db), as well as some kind of **transaction identifier** (**TID**). The middle three blocks just contain the exact contents of the blocks themselves; this is known as **physical logging** as we are putting the exact physical contents of the update in the journal (an alternate idea, **logical logging**, puts a more compact logical representation of the update in the journal, e.g., "this update wishes to append data block Db to file X", which is a little more complex but can save space in the log and perhaps improve performance). The final block (TxE) is a marker of the end of this transaction, and will also contain the TID.

Once this transaction is safely on disk, we are ready to overwrite the old structures in the file system; this process is called **checkpointing**. Thus, to **checkpoint** the file system (i.e., bring it up to date with the pending update in the journal), we issue the writes I[v2], B[v2], and Db to their disk locations as seen above; if these writes complete successfully, we have successfully checkpointed the file system and are basically done. Thus, our initial sequence of operations:

1. **Journal write:** Write the transaction, including a transaction-begin block, all pending data and metadata updates, and a transaction-end block, to the log; wait for these writes to complete.
2. **Checkpoint:** Write the pending metadata and data updates to their final locations in the file system.

In our example, we would write TxB, I[v2], B[v2], Db, and TxE to the journal first. When these writes complete, we would complete the update by checkpointing I[v2], B[v2], and Db, to their final locations on disk.

Things get a little trickier when a crash occurs during the writes to the journal. Here, we are trying to write the set of blocks in the transaction (e.g., TxB, I[v2], B[v2], Db, TxE) to disk. One simple way to do this would be to issue each one at a time, waiting for each to complete, and then issuing the next. However, this is slow. Ideally, we'd like to issue

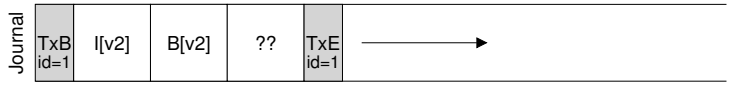
ASIDE: **FORCING WRITES TO DISK**

To enforce ordering between two disk writes, modern file systems have to take a few extra precautions. In olden times, forcing ordering between two writes, A and B, was easy: just issue the write of A to the disk, wait for the disk to interrupt the OS when the write is complete, and then issue the write of B.

Things got slightly more complex due to the increased use of write caches within disks. With write buffering enabled (sometimes called **immediate reporting**), a disk will inform the OS the write is complete when it simply has been placed in the disk's memory cache, and has not yet reached disk. If the OS then issues a subsequent write, it is not guaranteed to reach the disk after previous writes; thus ordering between writes is not preserved. One solution is to disable write buffering. However, more modern systems take extra precautions and issue explicit **write barriers**; such a barrier, when it completes, guarantees that all writes issued before the barrier will reach disk before any writes issued after the barrier.

All of this machinery requires a great deal of trust in the correct operation of the disk. Unfortunately, recent research shows that some disk manufacturers, in an effort to deliver "higher performing" disks, explicitly ignore write-barrier requests, thus making the disks seemingly run faster but at the risk of incorrect operation [C+13, R+11]. As Kahan said, the fast almost always beats out the slow, even if the fast is wrong.

all five block writes at once, as this would turn five writes into a single sequential write and thus be faster. However, this is unsafe, for the following reason: given such a big write, the disk internally may perform scheduling and complete small pieces of the big write in any order. Thus, the disk internally may (1) write TxB, I[v2], B[v2], and TxE and only later (2) write Db. Unfortunately, if the disk loses power between (1) and (2), this is what ends up on disk:



Why is this a problem? Well, the transaction looks like a valid transaction (it has a begin and an end with matching sequence numbers). Further, the file system can't look at that fourth block and know it is wrong; after all, it is arbitrary user data. Thus, if the system now reboots and runs recovery, it will replay this transaction, and ignorantly copy the contents of the garbage block '??' to the location where Db is supposed to live. This is bad for arbitrary user data in a file; it is much worse if it happens to a critical piece of file system, such as the superblock, which could render the file system unmountable.

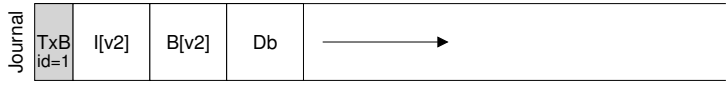
ASIDE: **OPTIMIZING LOG WRITES**

You may have noticed a particular inefficiency of writing to the log. Namely, the file system first has to write out the transaction-begin block and contents of the transaction; only after these writes complete can the file system send the transaction-end block to disk. The performance impact is clear, if you think about how a disk works: usually an extra rotation is incurred (think about why).

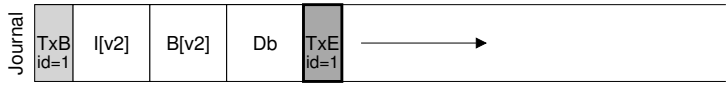
One of our former graduate students, Vijayan Prabhakaran, had a simple idea to fix this problem [P+05]. When writing a transaction to the journal, include a checksum of the contents of the journal in the begin and end blocks. Doing so enables the file system to write the entire transaction at once, without incurring a wait; if, during recovery, the file system sees a mismatch in the computed checksum versus the stored checksum in the transaction, it can conclude that a crash occurred during the write of the transaction and thus discard the file-system update. Thus, with a small tweak in the write protocol and recovery system, a file system can achieve faster common-case performance; on top of that, the system is slightly more reliable, as any reads from the journal are now protected by a checksum.

This simple fix was attractive enough to gain the notice of Linux file system developers, who then incorporated it into the next generation Linux file system, called (you guessed it!) **Linux ext4**. It now ships on millions of machines worldwide, including the Android handheld platform. Thus, every time you write to disk on many Linux-based systems, a little code developed at Wisconsin makes your system a little faster and more reliable.

To avoid this problem, the file system issues the transactional write in two steps. First, it writes all blocks except the TxE block to the journal, issuing these writes all at once. When these writes complete, the journal will look something like this (assuming our append workload again):



When those writes complete, the file system issues the write of the TxE block, thus leaving the journal in this final, safe state:



An important aspect of this process is the atomicity guarantee provided by the disk. It turns out that the disk guarantees that any 512-byte

write will either happen or not (and never be half-written); thus, to make sure the write of TxE is atomic, one should make it a single 512-byte block. Thus, our current protocol to update the file system, with each of its three phases labeled:

1. **Journal write:** Write the contents of the transaction (including TxB, metadata, and data) to the log; wait for these writes to complete.
2. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for write to complete; transaction is said to be **committed**.
3. **Checkpoint:** Write the contents of the update (metadata and data) to their final on-disk locations.

Recovery

Let's now understand how a file system can use the contents of the journal to **recover** from a crash. A crash may happen at any time during this sequence of updates. If the crash happens before the transaction is written safely to the log (i.e., before Step 2 above completes), then our job is easy: the pending update is simply skipped. If the crash happens after the transaction has committed to the log, but before the checkpoint is complete, the file system can **recover** the update as follows. When the system boots, the file system recovery process will scan the log and look for transactions that have committed to the disk; these transactions are thus **replayed** (in order), with the file system again attempting to write out the blocks in the transaction to their final on-disk locations. This form of logging is one of the simplest forms there is, and is called **redo logging**. By recovering the committed transactions in the journal, the file system ensures that the on-disk structures are consistent, and thus can proceed by mounting the file system and readying itself for new requests.

Note that it is fine for a crash to happen at any point during checkpointing, even after some of the updates to the final locations of the blocks have completed. In the worst case, some of these updates are simply performed again during recovery. Because recovery is a rare operation (only taking place after an unexpected system crash), a few redundant writes are nothing to worry about[3].

Batching Log Updates

You might have noticed that the basic protocol could add a lot of extra disk traffic. For example, imagine we create two files in a row, called `file1` and `file2`, in the same directory. To create one file, one has to update a number of on-disk structures, minimally including: the inode bitmap (to allocated a new inode), the newly-created inode of the file, the

[3]Unless you worry about everything, in which case we can't help you. Stop worrying so much, it is unhealthy! But now you're probably worried about over-worrying.

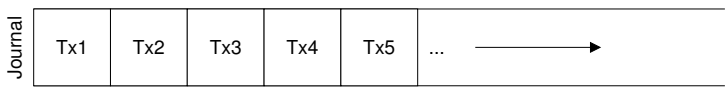
data block of the parent directory containing the new directory entry, as well as the parent directory inode (which now has a new modification time). With journaling, we logically commit all of this information to the journal for each of our two file creations; because the files are in the same directory, and let's assume even have inodes within the same inode block, this means that if we're not careful, we'll end up writing these same blocks over and over.

To remedy this problem, some file systems do not commit each update to disk one at a time (e.g., Linux ext3); rather, one can buffer all updates into a global transaction. In our example above, when the two files are created, the file system just marks the in-memory inode bitmap, inodes of the files, directory data, and directory inode as dirty, and adds them to the list of blocks that form the current transaction. When it is finally time to write these blocks to disk (say, after a timeout of 5 seconds), this single global transaction is committed containing all of the updates described above. Thus, by buffering updates, a file system can avoid excessive write traffic to disk in many cases.

Making The Log Finite

We thus have arrived at a basic protocol for updating file-system on-disk structures. The file system buffers updates in memory for some time; when it is finally time to write to disk, the file system first carefully writes out the details of the transaction to the journal (a.k.a. write-ahead log); after the transaction is complete, the file system checkpoints those blocks to their final locations on disk.

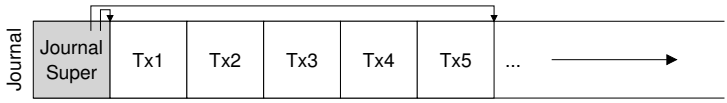
However, the log is of a finite size. If we keep adding transactions to it (as in this figure), it will soon fill. What do you think happens then?



Two problems arise when the log becomes full. The first is simpler, but less critical: the larger the log, the longer recovery will take, as the recovery process must replay all the transactions within the log (in order) to recover. The second is more of an issue: when the log is full (or nearly full), no further transactions can be committed to the disk, thus making the file system "less than useful" (i.e., useless).

To address these problems, journaling file systems treat the log as a circular data structure, re-using it over and over; this is why the journal is sometimes referred to as a **circular log**. To do so, the file system must take action some time after a checkpoint. Specifically, once a transaction has been checkpointed, the file system should free the space it was occupying within the journal, allowing the log space to be reused. There are many ways to achieve this end; for example, you could simply mark the oldest

and newest transactions in the log in a **journal superblock**; all other space is free. Here is a graphical depiction of such a mechanism:



In the journal superblock (not to be confused with the main file system superblock), the journaling system records enough information to know which transactions have not yet been checkpointed, and thus reduces recovery time as well as enables re-use of the log in a circular fashion. And thus we add another step to our basic protocol:

1. **Journal write:** Write the contents of the transaction (containing TxB and the contents of the update) to the log; wait for these writes to complete.
2. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction is now **committed**.
3. **Checkpoint:** Write the contents of the update to their final locations within the file system.
4. **Free:** Some time later, mark the transaction free in the journal by updating the journal superblock.

Thus we have our final data journaling protocol. But there is still a problem: we are writing each data block to the disk *twice*, which is a heavy cost to pay, especially for something as rare as a system crash. Can you figure out a way to retain consistency without writing data twice?

Metadata Journaling

Although recovery is now fast (scanning the journal and replaying a few transactions as opposed to scanning the entire disk), normal operation of the file system is slower than we might desire. In particular, for each write to disk, we are now also writing to the journal first, thus doubling write traffic; this doubling is especially painful during sequential write workloads, which now will proceed at half the peak write bandwidth of the drive. Further, between writes to the journal and writes to the main file system, there is a costly seek, which adds noticeable overhead for some workloads.

Because of the high cost of writing every data block to disk twice, people have tried a few different things in order to speed up performance. For example, the mode of journaling we described above is often called **data journaling** (as in Linux ext3), as it journals all user data (in addition to the metadata of the file system). A simpler (and more common) form of journaling is sometimes called **ordered journaling** (or just **metadata**

journaling), and it is nearly the same, except that user data is *not* written to the journal. Thus, when performing the same update as above, the following information would be written to the journal:

The data block Db, previously written to the log, would instead be written to the file system proper, avoiding the extra write; given that most I/O traffic to the disk is data, not writing data twice substantially reduces the I/O load of journaling. The modification does raise an interesting question, though: when should we write data blocks to disk?

Let's again consider our example append of a file to understand the problem better. The update consists of three blocks: I[v2], B[v2], and Db. The first two are both metadata and will be logged and then checkpointed; the latter will only be written once to the file system. When should we write Db to disk? Does it matter?

As it turns out, the ordering of the data write does matter for metadata-only journaling. For example, what if we write Db to disk *after* the transaction (containing I[v2] and B[v2]) completes? Unfortunately, this approach has a problem: the file system is consistent but I[v2] may end up pointing to garbage data. Specifically, consider the case where I[v2] and B[v2] are written but Db did not make it to disk. The file system will then try to recover. Because Db is *not* in the log, the file system will replay writes to I[v2] and B[v2], and produce a consistent file system (from the perspective of file-system metadata). However, I[v2] will be pointing to garbage data, i.e., at whatever was in the slot where Db was headed.

To ensure this situation does not arise, some file systems (e.g., Linux ext3) write data blocks (of regular files) to the disk *first*, before related metadata is written to disk. Specifically, the protocol is as follows:

1. **Data write:** Write data to final location; wait for completion (the wait is optional; see below for details).
2. **Journal metadata write:** Write the begin block and metadata to the log; wait for writes to complete.
3. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now **committed**.
4. **Checkpoint metadata:** Write the contents of the metadata update to their final locations within the file system.
5. **Free:** Later, mark the transaction free in journal superblock.

By forcing the data write first, a file system can guarantee that a pointer will never point to garbage. Indeed, this rule of "write the pointed to object before the object with the pointer to it" is at the core of crash consistency, and is exploited even further by other crash consistency schemes [GP94] (see below for details).

In most systems, metadata journaling (akin to ordered journaling of ext3) is more popular than full data journaling. For example, Windows NTFS and SGI's XFS both use non-ordered metadata journaling. Linux ext3 gives you the option of choosing either data, ordered, or unordered modes (in unordered mode, data can be written at any time). All of these modes keep metadata consistent; they vary in their semantics for data.

Finally, note that forcing the data write to complete (Step 1) before issuing writes to the journal (Step 2) is not required for correctness, as indicated in the protocol above. Specifically, it would be fine to issue data writes as well as the transaction-begin block and metadata to the journal; the only real requirement is that Steps 1 and 2 complete before the issuing of the journal commit block (Step 3).

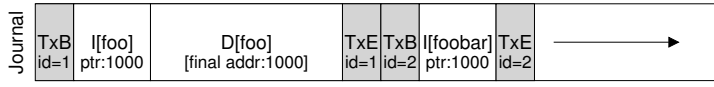
Tricky Case: Block Reuse

There are some interesting corner cases that make journaling more tricky, and thus are worth discussing. A number of them revolve around block reuse; as Stephen Tweedie (one of the main forces behind ext3) said:

> "What's the hideous part of the entire system? ... It's deleting files. Everything to do with delete is hairy. Everything to do with delete... you have nightmares around what happens if blocks get deleted and then reallocated." [T00]

The particular example Tweedie gives is as follows. Suppose you are using some form of metadata journaling (and thus data blocks for files are *not* journaled). Let's say you have a directory called `foo`. The user adds an entry to `foo` (say by creating a file), and thus the contents of `foo` (because directories are considered metadata) are written to the log; assume the location of the `foo` directory data is block 1000. The log thus contains something like this:

At this point, the user deletes everything in the directory as well as the directory itself, freeing up block 1000 for reuse. Finally, the user creates a new file (say `foobar`), which ends up reusing the same block (1000) that used to belong to `foo`. The inode of `foobar` is committed to disk, as is its data; note, however, because metadata journaling is in use, only the inode of `foobar` is committed to the journal; the newly-written data in block 1000 in the file `foobar` is *not* journaled.



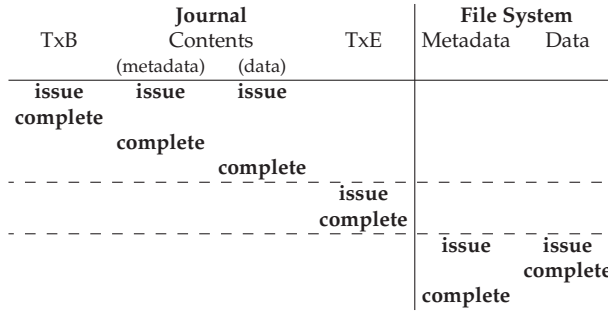


Figure 42.1: **Data Journaling Timeline**

Now assume a crash occurs and all of this information is still in the log. During replay, the recovery process simply replays everything in the log, including the write of directory data in block 1000; the replay thus overwrites the user data of current file `foobar` with old directory contents! Clearly this is not a correct recovery action, and certainly it will be a surprise to the user when reading the file `foobar`.

There are a number of solutions to this problem. One could, for example, never reuse blocks until the delete of said blocks is checkpointed out of the journal. What Linux ext3 does instead is to add a new type of record to the journal, known as a **revoke** record. In the case above, deleting the directory would cause a revoke record to be written to the journal. When replaying the journal, the system first scans for such revoke records; any such revoked data is never replayed, thus avoiding the problem mentioned above.

Wrapping Up Journaling: A Timeline

Before ending our discussion of journaling, we summarize the protocols we have discussed with timelines depicting each of them. Figure 42.1 shows the protocol when journaling data as well as metadata, whereas Figure 42.2 shows the protocol when journaling only metadata.

In each figure, time increases in the downward direction, and each row in the figure shows the logical time that a write can be issued or might complete. For example, in the data journaling protocol (Figure 42.1), the writes of the transaction begin block (TxB) and the contents of the transaction can logically be issued at the same time, and thus can be completed in any order; however, the write to the transaction end block (TxE) must not be issued until said previous writes complete. Similarly, the checkpointing writes to data and metadata blocks cannot begin until the transaction end block has committed. Horizontal dashed lines show where write-ordering requirements must be obeyed.

A similar timeline is shown for the metadata journaling protocol. Note that the data write can logically be issued at the same time as the writes

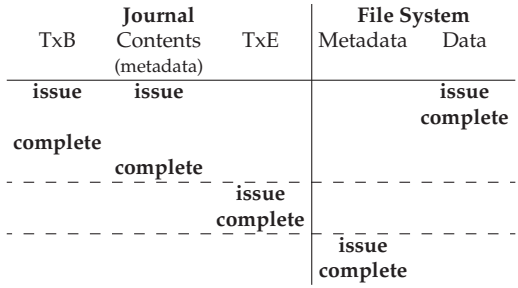


Figure 42.2: **Metadata Journaling Timeline**

to the transaction begin and the contents of the journal; however, it must be issued and complete before the transaction end has been issued.

Finally, note that the time of completion marked for each write in the timelines is arbitrary. In a real system, completion time is determined by the I/O subsystem, which may reorder writes to improve performance. The only guarantees about ordering that we have are those that must be enforced for protocol correctness (and are shown via the horizontal dashed lines in the figures).

42.4 Solution #3: Other Approaches

We've thus far described two options in keeping file system metadata consistent: a lazy approach based on `fsck`, and a more active approach known as journaling. However, these are not the only two approaches. One such approach, known as Soft Updates [GP94], was introduced by Ganger and Patt. This approach carefully orders all writes to the file system to ensure that the on-disk structures are never left in an inconsistent state. For example, by writing a pointed-to data block to disk *before* the inode that points to it, we can ensure that the inode never points to garbage; similar rules can be derived for all the structures of the file system. Implementing Soft Updates can be a challenge, however; whereas the journaling layer described above can be implemented with relatively little knowledge of the exact file system structures, Soft Updates requires intricate knowledge of each file system data structure and thus adds a fair amount of complexity to the system.

Another approach is known as **copy-on-write** (yes, **COW**), and is used in a number of popular file systems, including Sun's ZFS [B07]. This technique never overwrites files or directories in place; rather, it places new updates to previously unused locations on disk. After a number of updates are completed, COW file systems flip the root structure of the file system to include pointers to the newly updated structures. Doing so makes keeping the file system consistent straightforward. We'll be learning more about this technique when we discuss the log-structured file system (LFS) in a future chapter; LFS is an early example of a COW.

Another approach is one we just developed here at Wisconsin. In this technique, entitled **backpointer-based consistency** (or **BBC**), no ordering is enforced between writes. To achieve consistency, an additional **back pointer** is added to every block in the system; for example, each data block has a reference to the inode to which it belongs. When accessing a file, the file system can determine if the file is consistent by checking if the forward pointer (e.g., the address in the inode or direct block) points to a block that refers back to it. If so, everything must have safely reached disk and thus the file is consistent; if not, the file is inconsistent, and an error is returned. By adding back pointers to the file system, a new form of lazy crash consistency can be attained [C+12].

Finally, we also have explored techniques to reduce the number of times a journal protocol has to wait for disk writes to complete. Entitled **optimistic crash consistency** [C+13], this new approach issues as many writes to disk as possible and uses a generalized form of the **transaction checksum** [P+05], as well as a few other techniques, to detect inconsistencies should they arise. For some workloads, these optimistic techniques can improve performance by an order of magnitude. However, to truly function well, a slightly different disk interface is required [C+13].

42.5 Summary

We have introduced the problem of crash consistency, and discussed various approaches to attacking this problem. The older approach of building a file system checker works but is likely too slow to recover on modern systems. Thus, many file systems now use journaling. Journaling reduces recovery time from O(size-of-the-disk-volume) to O(size-of-the-log), thus speeding recovery substantially after a crash and restart. For this reason, many modern file systems use journaling. We have also seen that journaling can come in many different forms; the most commonly used is ordered metadata journaling, which reduces the amount of traffic to the journal while still preserving reasonable consistency guarantees for both file system metadata as well as user data.

References

[B07] "ZFS: The Last Word in File Systems"
Jeff Bonwick and Bill Moore
Available: http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf
ZFS uses copy-on-write and journaling, actually, as in some cases, logging writes to disk will perform better.

[C+12] "Consistency Without Ordering"
Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '12, San Jose, California
A recent paper of ours about a new form of crash consistency based on back pointers. Read it for the exciting details!

[C+13] "Optimistic Crash Consistency"
Vijay Chidambaram, Thanu S. Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
SOSP '13, Nemacolin Woodlands Resort, PA, November 2013
Our work on a more optimistic and higher performance journaling protocol. For workloads that call `fsync()` a lot, performance can be greatly improved.

[GP94] "Metadata Update Performance in File Systems"
Gregory R. Ganger and Yale N. Patt
OSDI '94
A clever paper about using careful ordering of writes as the main way to achieve consistency. Implemented later in BSD-based systems.

[G+08] "SQCK: A Declarative File System Checker"
Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
OSDI '08, San Diego, California
Our own paper on a new and better way to build a file system checker using SQL queries. We also show some problems with the existing checker, finding numerous bugs and odd behaviors, a direct result of the complexity of `fsck`.

[H87] "Reimplementing the Cedar File System Using Logging and Group Commit"
Robert Hagmann
SOSP '87, Austin, Texas, November 1987
The first work (that we know of) that applied write-ahead logging (a.k.a. journaling) to a file system.

[M+13] "ffsck: The Fast File System Checker"
Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '13, San Jose, California, February 2013
A recent paper of ours detailing how to make fsck an order of magnitude faster. Some of the ideas have already been incorporated into the BSD file system checker [MK96] and are deployed today.

[MK96] "Fsck - The UNIX File System Check Program"
Marshall Kirk McKusick and T. J. Kowalski
Revised in 1996
Describes the first comprehensive file-system checking tool, the eponymous `fsck`. Written by some of the same people who brought you FFS.

[MJLF84] "A Fast File System for UNIX"
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry
ACM Transactions on Computing Systems.
August 1984, Volume 2:3
You already know enough about FFS, right? But yeah, it is OK to reference papers like this more than once in a book.

[P+05] "IRON File Systems"
Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
SOSP '05, Brighton, England, October 2005
A paper mostly focused on studying how file systems react to disk failures. Towards the end, we introduce a transaction checksum to speed up logging, which was eventually adopted into Linux ext4.

[PAA05] "Analysis and Evolution of Journaling File Systems"
Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
USENIX '05, Anaheim, California, April 2005
An early paper we wrote analyzing how journaling file systems work.

[R+11] "Coerced Cache Eviction and Discreet-Mode Journaling"
Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
DSN '11, Hong Kong, China, June 2011
Our own paper on the problem of disks that buffer writes in a memory cache instead of forcing them to disk, even when explicitly told not to do that! Our solution to overcome this problem: if you want A to be written to disk before B, first write A, then send a lot of "dummy" writes to disk, hopefully causing A to be forced to disk to make room for them in the cache. A neat if impractical solution.

[T98] "Journaling the Linux ext2fs File System"
Stephen C. Tweedie
The Fourth Annual Linux Expo, May 1998
Tweedie did much of the heavy lifting in adding journaling to the Linux ext2 file system; the result, not surprisingly, is called ext3. Some nice design decisions include the strong focus on backwards compatibility, e.g., you can just add a journaling file to an existing ext2 file system and then mount it as an ext3 file system.

[T00] "EXT3, Journaling Filesystem"
Stephen Tweedie
Talk at the Ottawa Linux Symposium, July 2000
olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html
A transcript of a talk given by Tweedie on ext3.

[T01] "The Linux ext2 File System"
Theodore Ts'o, June, 2001.
Available: http://e2fsprogs.sourceforge.net/ext2.html
A simple Linux file system based on the ideas found in FFS. For a while it was quite heavily used; now it is really just in the kernel as an example of a simple file system.

43

Log-structured File Systems

In the early 90's, a group at Berkeley led by Professor John Ousterhout and graduate student Mendel Rosenblum developed a new file system known as the log-structured file system [RO91]. Their motivation to do so was based on the following observations:

- **Memory sizes were growing**: As memory got bigger, more data could be cached in memory. As more data is cached, disk traffic would increasingly consist of writes, as reads would be serviced in the cache. Thus, file system performance would largely be determined by its performance for writes.
- **There was a large and growing gap between random I/O performance and sequential I/O performance**: Transfer bandwidth increases roughly 50%-100% every year; seek and rotational delay costs decrease much more slowly, maybe at 5%-10% per year [P98]. Thus, if one is able to use disks in a sequential manner, one gets a huge performance advantage, which grows over time.
- **Existing file systems perform poorly on many common workloads**: For example, FFS [MJLF84] would perform a large number of writes to create a new file of size one block: one for a new inode, one to update the inode bitmap, one to the directory data block that the file is in, one to the directory inode to update it, one to the new data block that is apart of the new file, and one to the data bitmap to mark the data block as allocated. Thus, although FFS would place all of these blocks within the same block group, FFS would incur many short seeks and subsequent rotational delays and thus performance would fall far short of peak sequential bandwidth.
- **File systems were not RAID-aware**: For example, RAID-4 and RAID-5 have the **small-write problem** where a logical write to a single block causes 4 physical I/Os to take place. Existing file systems do not try to avoid this worst-case RAID writing behavior.

An ideal file system would thus focus on write performance, and try to make use of the sequential bandwidth of the disk. Further, it would perform well on common workloads that not only write out data but also

update on-disk metadata structures frequently. Finally, it would work well on RAIDs as well as single disks.

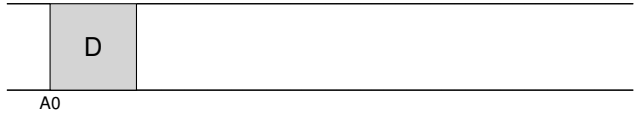
The new type of file system Rosenblum and Ousterhout introduced was called **LFS**, short for the **Log-structured File System**. When writing to disk, LFS first buffers all updates (including metadata!) in an in-memory **segment**; when the segment is full, it is written to disk in one long, sequential transfer to an unused part of the disk. LFS never overwrites existing data, but rather *always* writes segments to free locations. Because segments are large, the disk is used efficiently, and performance of the file system approaches its zenith.

THE CRUX:
HOW TO MAKE ALL WRITES SEQUENTIAL WRITES?

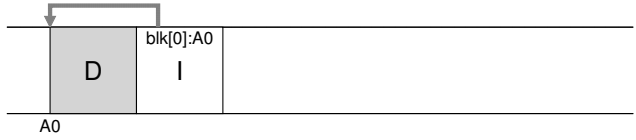
How can a file system turns all writes into sequential writes? For reads, this task is impossible, as the desired block to be read may be anywhere on disk. For writes, however, the file system always has a choice, and it is exactly this choice we hope to exploit.

43.1 Writing To Disk Sequentially

We thus have our first challenge: how do we transform all updates to file-system state into a series of sequential writes to disk? To understand this better, let's use a simple example. Imagine we are writing a data block D to a file. Writing the data block to disk might result in the following on-disk layout, with D written at disk address $A0$:



However, when a user writes a data block, it is not only data that gets written to disk; there is also other **metadata** that needs to be updated. In this case, let's also write the **inode** (I) of the file to disk, and have it point to the data block D. When written to disk, the data block and inode would look something like this (note that the inode looks as big as the data block, which generally isn't the case; in most systems, data blocks are 4 KB in size, whereas an inode is much smaller, around 128 bytes):



TIP: DETAILS MATTER

All interesting systems are comprised of a few general ideas and a number of details. Sometimes, when you are learning about these systems, you think to yourself "Oh, I get the general idea; the rest is just details," and you use this to only half-learn how things really work. Don't do this! Many times, the details are critical. As we'll see with LFS, the general idea is easy to understand, but to really build a working system, you have to think through *all* of the tricky cases.

This basic idea, of simply writing all updates (such as data blocks, inodes, etc.) to the disk sequentially, sits at the heart of LFS. If you understand this, you get the basic idea. But as with all complicated systems, the devil is in the details.

43.2 Writing Sequentially And Effectively

Unfortunately, writing to disk sequentially is not (alone) enough to guarantee efficient writes. For example, imagine if we wrote a single block to address A, at time T. We then wait a little while, and write to the disk at address $A + 1$ (the next block address in sequential order), but at time $T + \delta$. In-between the first and second writes, unfortunately, the disk has rotated; when you issue the second write, it will thus wait for most of a rotation before being committed (specifically, if the rotation takes time $T_{rotation}$, the disk will wait $T_{rotation} - \delta$ before it can commit the second write to the disk surface). And thus you can hopefully see that simply writing to disk in sequential order is not enough to achieve peak performance; rather, you must issue a large number of *contiguous* writes (or one large write) to the drive in order to achieve good write performance.

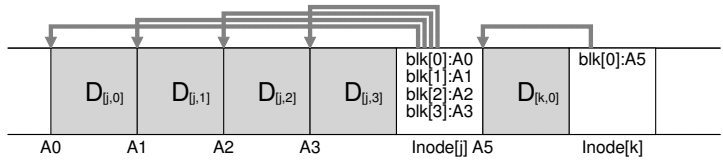
To achieve this end, LFS uses an ancient technique known as **write buffering**[1]. Before writing to the disk, LFS keeps track of updates in memory; when it has received a sufficient number of updates, it writes them to disk all at once, thus ensuring efficient use of the disk.

The large chunk of updates LFS writes at one time is referred to by the name of a **segment**. Although this term is over-used in computer systems, here it just means a large-ish chunk which LFS uses to group writes. Thus, when writing to disk, LFS buffers updates in an in-memory segment, and then writes the segment all at once to the disk. As long as the segment is large enough, these writes will be efficient.

Here is an example, in which LFS buffers two sets of updates into a small segment; actual segments are larger (a few MB). The first update is

[1] Indeed, it is hard to find a good citation for this idea, since it was likely invented by many and very early on in the history of computing. For a study of the benefits of write buffering, see Solworth and Orji [SO90]; to learn about its potential harms, see Mogul [M94].

of four block writes to file j; the second is one block being added to file k. LFS then commits the entire segment of seven blocks to disk at once. The resulting on-disk layout of these blocks is as follows:



43.3 How Much To Buffer?

This raises the following question: how many updates LFS should buffer before writing to disk? The answer, of course, depends on the disk itself, specifically how high the positioning overhead is in comparison to the transfer rate; see the FFS chapter for a similar analysis.

For example, assume that positioning (i.e., rotation and seek overheads) before each write takes roughly $T_{position}$ seconds. Assume further that the disk transfer rate is R_{peak} MB/s. How much should LFS buffer before writing when running on such a disk?

The way to think about this is that every time you write, you pay a fixed overhead of the positioning cost. Thus, how much do you have to write in order to **amortize** that cost? The more you write, the better (obviously), and the closer you get to achieving peak bandwidth.

To obtain a concrete answer, let's assume we are writing out D MB. The time to write out this chunk of data (T_{write}) is the positioning time $T_{position}$ plus the time to transfer D ($\frac{D}{R_{peak}}$), or:

$$T_{write} = T_{position} + \frac{D}{R_{peak}} \tag{43.1}$$

And thus the effective *rate* of writing ($R_{effective}$), which is just the amount of data written divided by the total time to write it, is:

$$R_{effective} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}}. \tag{43.2}$$

What we're interested in is getting the effective rate ($R_{effective}$) close to the peak rate. Specifically, we want the effective rate to be some fraction F of the peak rate, where $0 < F < 1$ (a typical F might be 0.9, or 90% of the peak rate). In mathematical form, this means we want $R_{effective} = F \times R_{peak}$.

At this point, we can solve for D:

$$R_{effective} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak} \tag{43.3}$$

$$D = F \times R_{peak} \times (T_{position} + \frac{D}{R_{peak}}) \tag{43.4}$$

$$D = (F \times R_{peak} \times T_{position}) + (F \times R_{peak} \times \frac{D}{R_{peak}}) \tag{43.5}$$

$$D = \frac{F}{1 - F} \times R_{peak} \times T_{position} \tag{43.6}$$

Let's do an example, with a disk with a positioning time of 10 milliseconds and peak transfer rate of 100 MB/s; assume we want an effective bandwidth of 90% of peak ($F = 0.9$). In this case, $D = \frac{0.9}{0.1} \times 100\ MB/s \times 0.01\ seconds = 9\ MB$. Try some different values to see how much we need to buffer in order to approach peak bandwidth. How much is needed to reach 95% of peak? 99%?

43.4 Problem: Finding Inodes

To understand how we find an inode in LFS, let us briefly review how to find an inode in a typical UNIX file system. In a typical file system such as FFS, or even the old UNIX file system, finding inodes is easy, because they are organized in an array and placed on disk at fixed locations.

For example, the old UNIX file system keeps all inodes at a fixed portion of the disk. Thus, given an inode number and the start address, to find a particular inode, you can calculate its exact disk address simply by multiplying the inode number by the size of an inode, and adding that to the start address of the on-disk array; array-based indexing, given an inode number, is fast and straightforward.

Finding an inode given an inode number in FFS is only slightly more complicated, because FFS splits up the inode table into chunks and places a group of inodes within each cylinder group. Thus, one must know how big each chunk of inodes is and the start addresses of each. After that, the calculations are similar and also easy.

In LFS, life is more difficult. Why? Well, we've managed to scatter the inodes all throughout the disk! Worse, we never overwrite in place, and thus the latest version of an inode (i.e., the one we want) keeps moving.

43.5 Solution Through Indirection: The Inode Map

To remedy this, the designers of LFS introduced a **level of indirection** between inode numbers and the inodes through a data structure called the **inode map (imap)**. The imap is a structure that takes an inode number as input and produces the disk address of the most recent version of the inode. Thus, you can imagine it would often be implemented as a simple *array*, with 4 bytes (a disk pointer) per entry. Any time an inode is written to disk, the imap is updated with its new location.

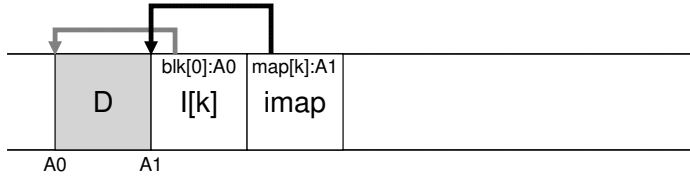
TIP: USE A LEVEL OF INDIRECTION

People often say that the solution to all problems in Computer Science is simply a **level of indirection**. This is clearly not true; it is just the solution to *most* problems. You certainly can think of every virtualization we have studied, e.g., virtual memory, as simply a level of indirection. And certainly the inode map in LFS is a virtualization of inode numbers. Hopefully you can see the great power of indirection in these examples, allowing us to freely move structures around (such as pages in the VM example, or inodes in LFS) without having to change every reference to them. Of course, indirection can have a downside too: **extra overhead.** So next time you have a problem, try solving it with indirection. But make sure to think about the overheads of doing so first.

The imap, unfortunately, needs to be kept persistent (i.e., written to disk); doing so allows LFS to keep track of the locations of inodes across crashes, and thus operate as desired. Thus, a question: where should the imap reside on disk?

It could live on a fixed part of the disk, of course. Unfortunately, as it gets updated frequently, this would then require updates to file structures to be followed by writes to the imap, and hence performance would suffer (i.e., there would be more disk seeks, between each update and the fixed location of the imap).

Instead, LFS places chunks of the inode map right next to where it is writing all of the other new information. Thus, when appending a data block to a file k, LFS actually writes the new data block, its inode, and a piece of the inode map all together onto the disk, as follows:



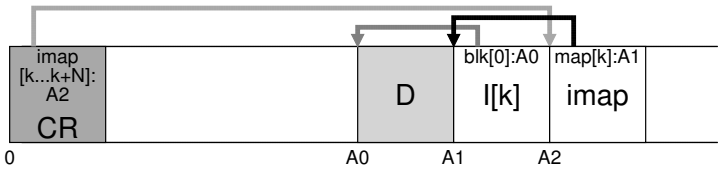
In this picture, the piece of the imap array stored in the block marked *imap* tells LFS that the inode k is at disk address $A1$; this inode, in turn, tells LFS that its data block D is at address $A0$.

43.6 The Checkpoint Region

The clever reader (that's you, right?) might have noticed a problem here. How do we find the inode map, now that pieces of it are also now spread across the disk? In the end, there is no magic: the file system must have *some* fixed and known location on disk to begin a file lookup.

LFS has just such a fixed place on disk for this, known as the **checkpoint region (CR)**. The checkpoint region contains pointers to (i.e., addresses of) the latest pieces of the inode map, and thus the inode map pieces can be found by reading the CR first. Note the checkpoint region is only updated periodically (say every 30 seconds or so), and thus performance is not ill-affected. Thus, the overall structure of the on-disk layout contains a checkpoint region (which points to the latest pieces of the inode map); the inode map pieces each contain addresses of the inodes; the inodes point to files (and directories) just like typical UNIX file systems.

Here is an example of the checkpoint region (note it is all the way at the beginning of the disk, at address 0), and a single imap chunk, inode, and data block. A real file system would of course have a much bigger CR (indeed, it would have two, as we'll come to understand later), many imap chunks, and of course many more inodes, data blocks, etc.



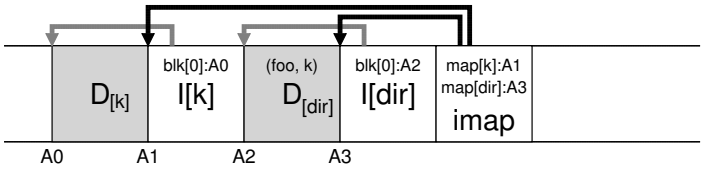
43.7 Reading A File From Disk: A Recap

To make sure you understand how LFS works, let us now walk through what must happen to read a file from disk. Assume we have nothing in memory to begin. The first on-disk data structure we must read is the checkpoint region. The checkpoint region contains pointers (i.e., disk addresses) to the entire inode map, and thus LFS then reads in the entire inode map and caches it in memory. After this point, when given an inode number of a file, LFS simply looks up the inode-number to inode-disk-address mapping in the imap, and reads in the most recent version of the inode. To read a block from the file, at this point, LFS proceeds exactly as a typical UNIX file system, by using direct pointers or indirect pointers or doubly-indirect pointers as need be. In the common case, LFS should perform the same number of I/Os as a typical file system when reading a file from disk; the entire imap is cached and thus the extra work LFS does during a read is to look up the inode's address in the imap.

43.8 What About Directories?

Thus far, we've simplified our discussion a bit by only considering inodes and data blocks. However, to access a file in a file system (such as `/home/remzi/foo`, one of our favorite fake file names), some directories must be accessed too. So how does LFS store directory data?

Fortunately, directory structure is basically identical to classic UNIX file systems, in that a directory is just a collection of (name, inode number) mappings. For example, when creating a file on disk, LFS must both write a new inode, some data, as well as the directory data and its inode that refer to this file. Remember that LFS will do so sequentially on the disk (after buffering the updates for some time). Thus, creating a file `foo` in a directory would lead to the following new structures on disk:



The piece of the inode map contains the information for the location of both the directory file dir as well as the newly-created file f. Thus, when accessing file `foo` (with inode number f), you would first look in the inode map (usually cached in memory) to find the location of the inode of directory dir ($A3$); you then read the directory inode, which gives you the location of the directory data ($A2$); reading this data block gives you the name-to-inode-number mapping of (`foo`, k). You then consult the inode map again to find the location of inode number k ($A1$), and finally read the desired data block at address $A0$.

There is one other serious problem in LFS that the inode map solves, known as the **recursive update problem** [Z+12]. The problem arises in any file system that never updates in place (such as LFS), but rather moves updates to new locations on the disk.

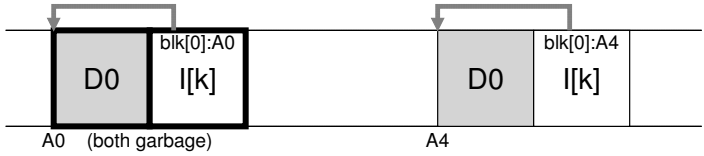
Specifically, whenever an inode is updated, its location on disk changes. If we hadn't been careful, this would have also entailed an update to the directory that points to this file, which then would have mandated a change to the parent of that directory, and so on, all the way up the file system tree.

LFS cleverly avoids this problem with the inode map. Even though the location of an inode may change, the change is never reflected in the directory itself; rather, the imap structure is updated while the directory holds the same name-to-inumber mapping. Thus, through indirection, LFS avoids the recursive update problem.

43.9 A New Problem: Garbage Collection

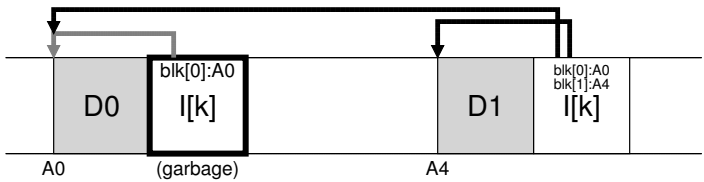
You may have noticed another problem with LFS; it repeatedly writes the latest version of a file (including its inode and data) to new locations on disk. This process, while keeping writes efficient, implies that LFS leaves old versions of file structures scattered throughout the disk. We (rather unceremoniously) call these old versions **garbage**.

For example, let's imagine the case where we have an existing file referred to by inode number k, which points to a single data block $D0$. We now overwrite that block, generating both a new inode and a new data block. The resulting on-disk layout of LFS would look something like this (note we omit the imap and other structures for simplicity; a new chunk of imap would also have to be written to disk to point to the new inode):



In the diagram, you can see that both the inode and data block have two versions on disk, one old (the one on the left) and one current and thus **live** (the one on the right). By the simple act of overwriting a data block, a number of new structures must be persisted by LFS, thus leaving old versions of said blocks on the disk.

As another example, imagine we instead append a block to that original file k. In this case, a new version of the inode is generated, but the old data block is still pointed to by the inode. Thus, it is still live and very much apart of the current file system:



So what should we do with these older versions of inodes, data blocks, and so forth? One could keep those older versions around and allow users to restore old file versions (for example, when they accidentally overwrite or delete a file, it could be quite handy to do so); such a file system is known as a **versioning file system** because it keeps track of the different versions of a file.

However, LFS instead keeps only the latest live version of a file; thus (in the background), LFS must periodically find these old dead versions of file data, inodes, and other structures, and **clean** them; cleaning should thus make blocks on disk free again for use in a subsequent writes. Note that the process of cleaning is a form of **garbage collection**, a technique that arises in programming languages that automatically free unused memory for programs.

Earlier we discussed segments as important as they are the mechanism that enables large writes to disk in LFS. As it turns out, they are also quite integral to effective cleaning. Imagine what would happen if the LFS

cleaner simply went through and freed single data blocks, inodes, etc., during cleaning. The result: a file system with some number of free **holes** mixed between allocated space on disk. Write performance would drop considerably, as LFS would not be able to find a large contiguous region to write to disk sequentially and with high performance.

Instead, the LFS cleaner works on a segment-by-segment basis, thus clearing up large chunks of space for subsequent writing. The basic cleaning process works as follows. Periodically, the LFS cleaner reads in a number of old (partially-used) segments, determines which blocks are live within these segments, and then write out a new set of segments with just the live blocks within them, freeing up the old ones for writing. Specifically, we expect the cleaner to read in M existing segments, **compact** their contents into N new segments (where $N < M$), and then write the N segments to disk in new locations. The old M segments are then freed and can be used by the file system for subsequent writes.

We are now left with two problems, however. The first is mechanism: how can LFS tell which blocks within a segment are live, and which are dead? The second is policy: how often should the cleaner run, and which segments should it pick to clean?

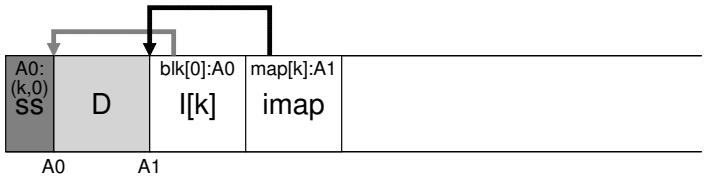
43.10 Determining Block Liveness

We address the mechanism first. Given a data block D within an on-disk segment S, LFS must be able to determine whether D is live. To do so, LFS adds a little extra information to each segment that describes each block. Specifically, LFS includes, for each data block D, its inode number (which file it belongs to) and its offset (which block of the file this is). This information is recorded in a structure at the head of the segment known as the **segment summary block**.

Given this information, it is straightforward to determine whether a block is live or dead. For a block D located on disk at address A, look in the segment summary block and find its inode number N and offset T. Next, look in the imap to find where N lives and read N from disk (perhaps it is already in memory, which is even better). Finally, using the offset T, look in the inode (or some indirect block) to see where the inode thinks the Tth block of this file is on disk. If it points exactly to disk address A, LFS can conclude that the block D is live. If it points anywhere else, LFS can conclude that D is not in use (i.e., it is dead) and thus know that this version is no longer needed. A pseudocode summary of this process is shown here:

```
(N, T) = SegmentSummary[A];
inode  = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

Here is a diagram depicting the mechanism, in which the segment summary block (marked SS) records that the data block at address $A0$ is actually a part of file k at offset 0. By checking the imap for k, you can find the inode, and see that it does indeed point to that location.



There are some shortcuts LFS takes to make the process of determining liveness more efficient. For example, when a file is truncated or deleted, LFS increases its **version number** and records the new version number in the imap. By also recording the version number in the on-disk segment, LFS can short circuit the longer check described above simply by comparing the on-disk version number with a version number in the imap, thus avoiding extra reads.

43.11 A Policy Question: Which Blocks To Clean, And When?

On top of the mechanism described above, LFS must include a set of policies to determine both when to clean and which blocks are worth cleaning. Determining when to clean is easier; either periodically, during idle time, or when you have to because the disk is full.

Determining which blocks to clean is more challenging, and has been the subject of many research papers. In the original LFS paper [RO91], the authors describe an approach which tries to segregate *hot* and *cold* segment. A hot segment is one in which the contents are being frequently over-written; thus, for such a segment, the best policy is to wait a long time before cleaning it, as more and more blocks are getting over-written (in new segments) and thus being freed for use. A cold segment, in contrast, may have a few dead blocks but the rest of its contents are relatively stable. Thus, the authors conclude that one should clean cold segments sooner and hot segments later, and develop a heuristic that does exactly that. However, as with most policies, this is just one approach, and by definition is not "the best" approach; later approaches show how to do better [MR+97].

43.12 Crash Recovery And The Log

One final problem: what happens if the system crashes while LFS is writing to disk? As you may recall in the previous chapter about journaling, crashes during updates are tricky for file systems, and thus some-

thing LFS must consider as well.

During normal operation, LFS buffers writes in a segment, and then (when the segment is full, or when some amount of time has elapsed), writes the segment to disk. LFS organizes these writes in a **log**, i.e., the checkpoint region points to a head and tail segment, and each segment points to the next segment to be written. LFS also periodically updates the checkpoint region. Crashes could clearly happen during either of these operations (write to a segment, write to the CR). So how does LFS handle crashes during writes to these structures?

Let's cover the second case first. To ensure that the CR update happens atomically, LFS actually keeps two CRs, one at either end of the disk, and writes to them alternately. LFS also implements a careful protocol when updating the CR with the latest pointers to the inode map and other information; specifically, it first writes out a header (with timestamp), then the body of the CR, and then finally one last block (also with a timestamp). If the system crashes during a CR update, LFS can detect this by seeing an inconsistent pair of timestamps. LFS will always choose to use the most recent CR that has consistent timestamps, and thus consistent update of the CR is achieved.

Let's now address the first case. Because LFS writes the CR every 30 seconds or so, the last consistent snapshot of the file system may be quite old. Thus, upon reboot, LFS can easily recover by simply reading in the checkpoint region, the imap pieces it points to, and subsequent files and directories; however, the last many seconds of updates would be lost.

To improve upon this, LFS tries to rebuild many of those segments through a technique known as **roll forward** in the database community. The basic idea is to start with the last checkpoint region, find the end of the log (which is included in the CR), and then use that to read through the next segments and see if there are any valid updates within it. If there are, LFS updates the file system accordingly and thus recovers much of the data and metadata written since the last checkpoint. See Rosenblum's award-winning dissertation for details [R92].

43.13 Summary

LFS introduces a new approach to updating the disk. Instead of overwriting files in places, LFS always writes to an unused portion of the disk, and then later reclaims that old space through cleaning. This approach, which in database systems is called **shadow paging** [L77] and in file-system-speak is sometimes called **copy-on-write**, enables highly efficient writing, as LFS can gather all updates into an in-memory segment and then write them out together sequentially.

The downside to this approach is that it generates garbage; old copies of the data are scattered throughout the disk, and if one wants to reclaim such space for subsequent usage, one must clean old segments periodically. Cleaning became the focus of much controversy in LFS, and

TIP: TURN FLAWS INTO VIRTUES

Whenever your system has a fundamental flaw, see if you can turn it around into a feature or something useful. NetApp's WAFL does this with old file contents; by making old versions available, WAFL no longer has to worry about cleaning, and thus provides a cool feature and removes the LFS cleaning problem all in one wonderful twist. Are there other examples of this in systems? Undoubtedly, but you'll have to think of them yourself, because this chapter is over with a capital "O". Over. Done. Kaput. We're out. Peace!

concerns over cleaning costs [SS+95] perhaps limited LFS's initial impact on the field. However, some modern commercial file systems, including NetApp's **WAFL** [HLM94], Sun's **ZFS** [B07], and Linux **btrfs** [M07] adopt a similar copy-on-write approach to writing to disk, and thus the intellectual legacy of LFS lives on in these modern file systems. In particular, WAFL got around cleaning problems by turning them into a feature; by providing old versions of the file system via **snapshots**, users could access old files whenever they deleted current ones accidentally.

References

[B07] "ZFS: The Last Word in File Systems"
Jeff Bonwick and Bill Moore
Available: http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf
Slides on ZFS; unfortunately, there is no great ZFS paper.

[HLM94] "File System Design for an NFS File Server Appliance"
Dave Hitz, James Lau, Michael Malcolm
USENIX Spring '94
WAFL takes many ideas from LFS and RAID and puts it into a high-speed NFS appliance for the multi-billion dollar storage company NetApp.

[L77] "Physical Integrity in a Large Segmented Database"
R. Lorie
ACM Transactions on Databases, 1977, Volume 2:1, pages 91-104
The original idea of shadow paging is presented here.

[M07] "The Btrfs Filesystem"
Chris Mason
September 2007
Available: oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf
A recent copy-on-write Linux file system, slowly gaining in importance and usage.

[MJLF84] "A Fast File System for UNIX"
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry
ACM TOCS, August, 1984, Volume 2, Number 3
The original FFS paper; see the chapter on FFS for more details.

[MR+97] "Improving the Performance of Log-structured File Systems
with Adaptive Methods"
Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello,
Randolph Y. Wang, Thomas E. Anderson
SOSP 1997, pages 238-251, October, Saint Malo, France
A more recent paper detailing better policies for cleaning in LFS.

[M94] "A Better Update Policy"
Jeffrey C. Mogul
USENIX ATC '94, June 1994
In this paper, Mogul finds that read workloads can be harmed by buffering writes for too long and then sending them to the disk in a big burst. Thus, he recommends sending writes more frequently and in smaller batches.

[P98] "Hardware Technology Trends and Database Opportunities"
David A. Patterson
ACM SIGMOD '98 Keynote Address, Presented June 3, 1998, Seattle, Washington
Available: http://www.cs.berkeley.edu/~pattrsn/talks/keynote.html
A great set of slides on technology trends in computer systems. Hopefully, Patterson will create another of these sometime soon.

[RO91] "Design and Implementation of the Log-structured File System"
Mendel Rosenblum and John Ousterhout
SOSP '91, Pacific Grove, CA, October 1991
The original SOSP paper about LFS, which has been cited by hundreds of other papers and inspired many real systems.

[R92] "Design and Implementation of the Log-structured File System"
Mendel Rosenblum
http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-696.pdf
The award-winning dissertation about LFS, with many of the details missing from the paper.

[SS+95] "File system logging versus clustering: a performance comparison"
Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan
USENIX 1995 Technical Conference, New Orleans, Louisiana, 1995
A paper that showed the LFS performance sometimes has problems, particularly for workloads with many calls to `fsync()` (such as database workloads). The paper was controversial at the time.

[SO90] "Write-Only Disk Caches"
Jon A. Solworth, Cyril U. Orji
SIGMOD '90, Atlantic City, New Jersey, May 1990
An early study of write buffering and its benefits. However, buffering for too long can be harmful: see Mogul [M94] for details.

[Z+12] "De-indirection for Flash-based SSDs with Nameless Writes"
Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '13, San Jose, California, February 2013
Our paper on a new way to build flash-based storage devices. Because FTLs (flash-translation layers) are usually built in a log-structured style, some of the same issues arise in flash-based devices that do in LFS. In this case, it is the recursive update problem, which LFS solves neatly with an imap. A similar structure exists in most SSDs.

41

Locality and The Fast File System

When the UNIX operating system was first introduced, the UNIX wizard himself Ken Thompson wrote the first file system. We will call that the "old UNIX file system", and it was really simple. Basically, its data structures looked like this on the disk:

The super block (S) contained information about the entire file system: how big the volume is, how many inodes there are, a pointer to the head of a free list of blocks, and so forth. The inode region of the disk contained all the inodes for the file system. Finally, most of the disk was taken up by data blocks.

The good thing about the old file system was that it was simple, and supported the basic abstractions the file system was trying to deliver: files and the directory hierarchy. This easy-to-use system was a real step forward from the clumsy, record-based storage systems of the past, and the directory hierarchy a true advance over simpler, one-level hierarchies provided by earlier systems.

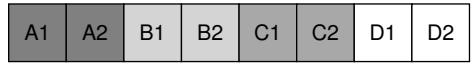
41.1 The Problem: Poor Performance

The problem: performance was terrible. As measured by Kirk McKusick and his colleagues at Berkeley [MJLF84], performance started off bad and got worse over time, to the point where the file system was delivering only 2% of overall disk bandwidth!

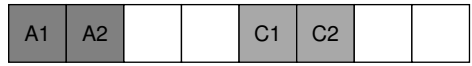
The main issue was that the old UNIX file system treated the disk like it was a random-access memory; data was spread all over the place without regard to the fact that the medium holding the data was a disk, and thus had real and expensive positioning costs. For example, the data blocks of a file were often very far away from its inode, thus inducing an expensive seek whenever one first read the inode and then the data blocks of a file (a pretty common operation).

Worse, the file system would end up getting quite **fragmented**, as the free space was not carefully managed. The free list would end up pointing to a bunch of blocks spread across the disk, and as files got allocated, they would simply take the next free block. The result was that a logically contiguous file would be accessed by going back and forth across the disk, thus reducing performance dramatically.

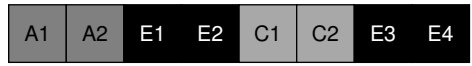
For example, imagine the following data block region, which contains four files (A, B, C, and D), each of size 2 blocks:



If B and D are deleted, the resulting layout is:



As you can see, the free space is fragmented into two chunks of two blocks, instead of one nice contiguous chunk of four. Let's say we now wish to allocate a file E, of size four blocks:



You can see what happens: E gets spread across the disk, and as a result, when accessing E, you don't get peak (sequential) performance from the disk. Rather, you first read E1 and E2, then seek, then read E3 and E4. This fragmentation problem happened all the time in the old UNIX file system, and it hurt performance. (A side note: this problem is exactly what disk defragmentation tools help with; they will reorganize on-disk data to place files contiguously and make free space one or a few contiguous regions, moving data around and then rewriting inodes and such to reflect the changes)

One other problem: the original block size was too small (512 bytes). Thus, transferring data from the disk was inherently inefficient. Smaller blocks were good because they minimized **internal fragmentation** (waste within the block), but bad for transfer as each block might require a positioning overhead to reach it. We can summarize the problem as follows:

THE CRUX:
HOW TO ORGANIZE ON-DISK DATA TO IMPROVE PERFORMANCE

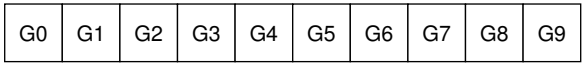
How can we organize file system data structures so as to improve performance? What types of allocation policies do we need on top of those data structures? How do we make the file system "disk aware"?

41.2 FFS: Disk Awareness Is The Solution

A group at Berkeley decided to build a better, faster file system, which they cleverly called the **Fast File System (FFS)**. The idea was to design the file system structures and allocation policies to be "disk aware" and thus improve performance, which is exactly what they did. FFS thus ushered in a new era of file system research; by keeping the same *interface* to the file system (the same APIs, including `open()`, `read()`, `write()`, `close()`, and other file system calls) but changing the internal *implementation*, the authors paved the path for new file system construction, work that continues today. Virtually all modern file systems adhere to the existing interface (and thus preserve compatibility with applications) while changing their internals for performance, reliability, or other reasons.

41.3 Organizing Structure: The Cylinder Group

The first step was to change the on-disk structures. FFS divides the disk into a bunch of groups known as **cylinder groups** (some modern file systems like Linux ext2 and ext3 just call them **block groups**). We can thus imagine a disk with ten cylinder groups:



These groups are the central mechanism that FFS uses to improve performance; by placing two files within the same group, FFS can ensure that accessing one after the other will not result in long seeks across the disk.

Thus, FFS needs to have the ability to allocate files and directories within each of these groups. Each group looks like this:

We now describe the components of a cylinder group. A copy of the **super block** (S) is found in each group for reliability reasons (e.g., if one gets corrupted or scratched, you can still mount and access the file system by using one of the others).

Within each group, we need to track whether the inodes and data blocks of the group are allocated. A per-group **inode bitmap** (ib) and **data bitmap** (db) serve this role for inodes and data blocks in each group. Bitmaps are an excellent way to manage free space in a file system because it is easy to find a large chunk of free space and allocate it to a file, perhaps avoiding some of the fragmentation problems of the free list in the old file system.

Finally, the inode and data block regions are just like in the previous very simple file system. Most of each cylinder group, as usual, is comprised of data blocks.

ASIDE: **FFS FILE CREATION**

As an example, think about what data structures must be updated when a file is created; assume, for this example, that the user creates a new file `/foo/bar.txt` and that the file is one block long (4KB). The file is new, and thus needs a new inode; thus, both the inode bitmap and the newly-allocated inode will be written to disk. The file also has data in it and thus it too must be allocated; the data bitmap and a data block will thus (eventually) be written to disk. Hence, at least four writes to the current cylinder group will take place (recall that these writes may be buffered in memory for a while before the write takes place). But this is not all! In particular, when creating a new file, we must also place the file in the file-system hierarchy; thus, the directory must be updated. Specifically, the parent directory `foo` must be updated to add the entry for `bar.txt`; this update may fit in an existing data block of `foo` or require a new block to be allocated (with associated data bitmap). The inode of `foo` must also be updated, both to reflect the new length of the directory as well as to update time fields (such as last-modified-time). Overall, it is a lot of work just to create a new file! Perhaps next time you do so, you should be more thankful, or at least surprised that it all works so well.

41.4 Policies: How To Allocate Files and Directories

With this group structure in place, FFS now has to decide how to place files and directories and associated metadata on disk to improve performance. The basic mantra is simple: *keep related stuff together* (and its corollary, keep unrelated stuff far apart).

Thus, to obey the mantra, FFS has to decide what is "related" and place it within the same block group; conversely, unrelated items should be placed into different block groups. To achieve this end, FFS makes use of a few simple placement heuristics.

The first is the placement of directories. FFS employs a simple approach: find the cylinder group with a low number of allocated directories (because we want to balance directories across groups) and a high number of free inodes (because we want to subsequently be able to allocate a bunch of files), and put the directory data and inode in that group. Of course, other heuristics could be used here (e.g., taking into account the number of free data blocks).

For files, FFS does two things. First, it makes sure (in the general case) to allocate the data blocks of a file in the same group as its inode, thus preventing long seeks between inode and data (as in the old file system). Second, it places all files that are in the same directory in the cylinder group of the directory they are in. Thus, if a user creates four files, `/dir1/1.txt`, `/dir1/2.txt`, `/dir1/3.txt`, and `/dir99/4.txt`, FFS would try to place the first three near one another (same group) and the fourth far away (in some other group).

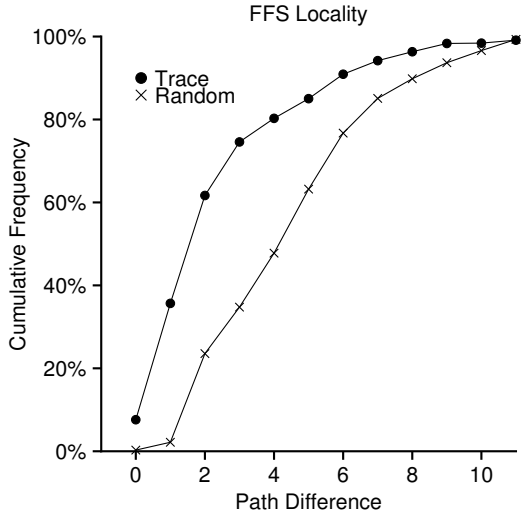


Figure 41.1: **FFS Locality For SEER Traces**

It should be noted that these heuristics are not based on extensive studies of file-system traffic or anything particularly nuanced; rather, they are based on good old-fashioned common sense (isn't that what CS stands for after all?). Files in a directory *are* often accessed together (imagine compiling a bunch of files and then linking them into a single executable). Because they are, FFS will often improve performance, making sure that seeks between related files are short.

41.5 Measuring File Locality

To understand better whether these heuristics make sense, we decided to analyze some traces of file system access and see if indeed there is namespace locality; for some reason, there doesn't seem to be a good study of this topic in the literature.

Specifically, we took the SEER traces [K94] and analyzed how "far away" file accesses were from one another in the directory tree. For example, if file `f` is opened, and then re-opened next in the trace (before any other files are opened), the distance between these two opens in the directory tree is zero (as they are the same file). If a file `f` in directory `dir` (i.e., `dir/f`) is opened, and followed by an open of file `g` in the same directory (i.e., `dir/g`), the distance between the two file accesses is one, as they share the same directory but are not the same file. Our distance metric, in other words, measures how far up the directory tree you have to travel to find the *common ancestor* of two files; the closer they are in the tree, the lower the metric.

Figure 41.1 shows the locality observed in the SEER traces over all workstations in the SEER cluster over the entirety of all traces. The graph plots the difference metric along the x-axis, and shows the cumulative percentage of file opens that were of that difference along the y-axis. Specifically, for the SEER traces (marked "Trace" in the graph), you can see that about 7% of file accesses were to the file that was opened previously, and that nearly 40% of file accesses were to either the same file or to one in the same directory (i.e., a difference of zero or one). Thus, the FFS locality assumption seems to make sense (at least for these traces).

Interestingly, another 25% or so of file accesses were to files that had a distance of two. This type of locality occurs when the user has structured a set of related directories in a multi-level fashion and consistently jumps between them. For example, if a user has a `src` directory and builds object files (`.o` files) into a `obj` directory, and both of these directories are sub-directories of a main `proj` directory, a common access pattern will be `proj/src/foo.c` followed by `proj/obj/foo.o`. The distance between these two accesses is two, as `proj` is the common ancestor. FFS does *not* capture this type of locality in its policies, and thus more seeking will occur between such accesses.

We also show what locality would be for a "Random" trace for the sake of comparison. We generated the random trace by selecting files from within an existing SEER trace in random order, and calculating the distance metric between these randomly-ordered accesses. As you can see, there is less namespace locality in the random traces, as expected. However, because eventually every file shares a common ancestor (e.g., the root), there is some locality eventually, and thus random trace is useful as a comparison point.

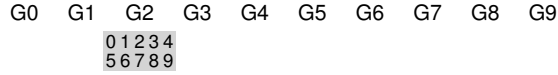
41.6 The Large-File Exception

In FFS, there is one important exception to the general policy of file placement, and it arises for large files. Without a different rule, a large file would entirely fill the block group it is first placed within (and maybe others). Filling a block group in this manner is undesirable, as it prevents subsequent "related" files from being placed within this block group, and thus may hurt file-access locality.

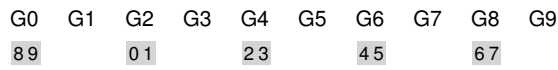
Thus, for large files, FFS does the following. After some number of blocks are allocated into the first block group (e.g., 12 blocks, or the number of direct pointers available within an inode), FFS places the next "large" chunk of the file (e.g., those pointed to by the first indirect block) in another block group (perhaps chosen for its low utilization). Then, the next chunk of the file is placed in yet another different block group, and so on.

Let's look at some pictures to understand this policy better. Without the large-file exception, a single large file would place all of its blocks into one part of the disk. We use a small example of a file with 10 blocks to illustrate the behavior visually.

Here is the depiction of FFS without the large-file exception:



With the large-file exception, we might see something more like this, with the file spread across the disk in chunks:



The astute reader will note that spreading blocks of a file across the disk will hurt performance, particularly in the relatively common case of sequential file access (e.g., when a user or application reads chunks 0 through 9 in order). And you are right! It will. We can help this a little, by choosing our chunk size carefully.

Specifically, if the chunk size is large enough, we will still spend most of our time transferring data from disk and just a relatively little time seeking between chunks of the block. This process of reducing an overhead by doing more work per overhead paid is called **amortization** and is a common technique in computer systems.

Let's do an example: assume that the average positioning time (i.e., seek and rotation) for a disk is 10 ms. Assume further that the disk transfers data at 40 MB/s. If our goal was to spend half our time seeking between chunks and half our time transferring data (and thus achieve 50% of peak disk performance), we would thus need to spend 10 ms transferring data for every 10 ms positioning. So the question becomes: how big does a chunk have to be in order to spend 10 ms in transfer? Easy, just use our old friend, math, in particular the dimensional analysis we spoke of in the chapter on disks:

$$\frac{40\ \cancel{MB}}{\cancel{sec}} \cdot \frac{1024\ KB}{1\ \cancel{MB}} \cdot \frac{1\ \cancel{sec}}{1000\ \cancel{ms}} \cdot 10\ \cancel{ms} = 409.6\ KB \tag{41.1}$$

Basically, what this equation says is this: if you transfer data at 40 MB/s, you need to transfer only 409.6KB every time you seek in order to spend half your time seeking and half your time transferring. Similarly, you can compute the size of the chunk you would need to achieve 90% of peak bandwidth (turns out it is about 3.69MB), or even 99% of peak bandwidth (40.6MB!). As you can see, the closer you want to get to peak, the bigger these chunks get (see Figure 41.2 for a plot of these values).

FFS did not use this type of calculation in order to spread large files across groups, however. Instead, it took a simple approach, based on the structure of the inode itself. The first twelve direct blocks were placed in the same group as the inode; each subsequent indirect block, and all the blocks it pointed to, was placed in a different group. With a block size of 4KB, and 32-bit disk addresses, this strategy implies that every 1024 blocks of the file (4MB) were placed in separate groups, the lone exception being the first 48KB of the file as pointed to by direct pointers.

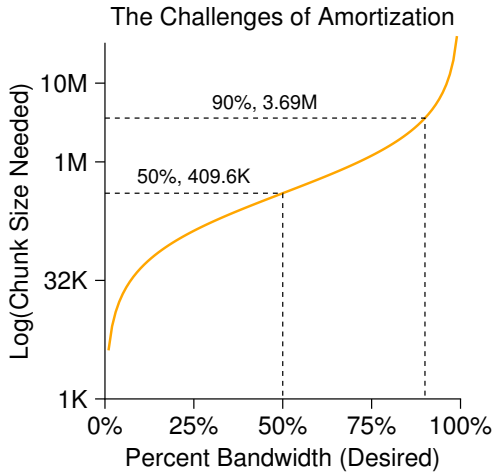


Figure 41.2: **Amortization: How Big Do Chunks Have To Be?**

We should note that the trend in disk drives is that transfer rate improves fairly rapidly, as disk manufacturers are good at cramming more bits into the same surface, but the mechanical aspects of drives related to seeks (disk arm speed and the rate of rotation) improve rather slowly [P98]. The implication is that over time, mechanical costs become relatively more expensive, and thus, to amortize said costs, you have to transfer more data between seeks.

41.7 A Few Other Things About FFS

FFS introduced a few other innovations too. In particular, the designers were extremely worried about accommodating small files; as it turned out, many files were 2KB or so in size back then, and using 4KB blocks, while good for transferring data, was not so good for space efficiency. This **internal fragmentation** could thus lead to roughly half the disk being wasted for a typical file system.

The solution the FFS designers hit upon was simple and solved the problem. They decided to introduce **sub-blocks**, which were 512-byte little blocks that the file system could allocate to files. Thus, if you created a small file (say 1KB in size), it would occupy two sub-blocks and thus not waste an entire 4KB block. As the file grew, the file system will continue allocating 512-byte blocks to it until it acquires a full 4KB of data. At that point, FFS will find a 4KB block, *copy* the sub-blocks into it, and free the sub-blocks for future use.

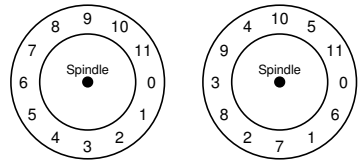


Figure 41.3: **FFS: Standard Versus Parameterized Placement**

You might observe that this process is inefficient, requiring a lot of extra work for the file system (in particular, a lot of extra I/O to perform the copy). And you'd be right again! Thus, FFS generally avoided this pessimal behavior by modifying the `libc` library; the library would buffer writes and then issue them in 4KB chunks to the file system, thus avoiding the sub-block specialization entirely in most cases.

A second neat thing that FFS introduced was a disk layout that was optimized for performance. In those times (before SCSI and other more modern device interfaces), disks were much less sophisticated and required the host CPU to control their operation in a more hands-on way. A problem arose in FFS when a file was placed on consecutive sectors of the disk, as on the left in Figure 41.3.

In particular, the problem arose during sequential reads. FFS would first issue a read to block 0; by the time the read was complete, and FFS issued a read to block 1, it was too late: block 1 had rotated under the head and now the read to block 1 would incur a full rotation.

FFS solved this problem with a different layout, as you can see on the right in Figure 41.3. By skipping over every other block (in the example), FFS has enough time to request the next block before it went past the disk head. In fact, FFS was smart enough to figure out for a particular disk *how many* blocks it should skip in doing layout in order to avoid the extra rotations; this technique was called **parameterization**, as FFS would figure out the specific performance parameters of the disk and use those to decide on the exact staggered layout scheme.

You might be thinking: this scheme isn't so great after all. In fact, you will only get 50% of peak bandwidth with this type of layout, because you have to go around each track twice just to read each block once. Fortunately, modern disks are much smarter: they internally read the entire track in and buffer it in an internal disk cache (often called a **track buffer** for this very reason). Then, on subsequent reads to the track, the disk will just return the desired data from its cache. File systems thus no longer have to worry about these incredibly low-level details. Abstraction and higher-level interfaces can be a good thing, when designed properly.

Some other usability improvements were added as well. FFS was one of the first file systems to allow for **long file names**, thus enabling more expressive names in the file system instead of a the traditional fixed-size approach (e.g., 8 characters). Further, a new concept was introduced

TIP: MAKE THE SYSTEM USABLE
Probably the most basic lesson from FFS is that not only did it introduce the conceptually good idea of disk-aware layout, but it also added a number of features that simply made the system more usable. Long file names, symbolic links, and a rename operation that worked atomically all improved the utility of a system; while hard to write a research paper about (imagine trying to read a 14-pager about "The Symbolic Link: Hard Link's Long Lost Cousin"), such small features made FFS more useful and thus likely increased its chances for adoption. Making a system usable is often as or more important than its deep technical innovations.

called a **symbolic link**. As discussed in a previous chapter, hard links are limited in that they both could not point to directories (for fear of introducing loops in the file system hierarchy) and that they can only point to files within the same volume (i.e., the inode number must still be meaningful). Symbolic links allow the user to create an "alias" to any other file or directory on a system and thus are much more flexible. FFS also introduced an atomic `rename()` operation for renaming files. Usability improvements, beyond the basic technology, also likely gained FFS a stronger user base.

41.8 Summary

The introduction of FFS was a watershed moment in file system history, as it made clear that the problem of file management was one of the most interesting issues within an operating system, and showed how one might begin to deal with that most important of devices, the hard disk. Since that time, hundreds of new file systems have developed, but still today many file systems take cues from FFS (e.g., Linux ext2 and ext3 are obvious intellectual descendants). Certainly all modern systems account for the main lesson of FFS: treat the disk like it's a disk.

References

[MJLF84] "A Fast File System for UNIX"
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry
ACM Transactions on Computing Systems.
August, 1984. Volume 2, Number 3.
pages 181-197.
McKusick was recently honored with the IEEE Reynold B. Johnson award for his contributions to file systems, much of which was based on his work building FFS. In his acceptance speech, he discussed the original FFS software: only 1200 lines of code! Modern versions are a little more complex, e.g., the BSD FFS descendant now is in the 50-thousand lines-of-code range.

[P98] "Hardware Technology Trends and Database Opportunities"
David A. Patterson
Keynote Lecture at the ACM SIGMOD Conference (SIGMOD '98)
June, 1998
A great and simple overview of disk technology trends and how they change over time.

[K94] "The Design of the SEER Predictive Caching System"
G. H. Kuenning
MOBICOMM '94, Santa Cruz, California, December 1994
According to Kuenning, this is the best overview of the SEER project, which led to (among other things) the collection of these traces.

39

Interlude: File and Directories

Thus far we have seen the development of two key operating system abstractions: the process, which is a virtualization of the CPU, and the address space, which is a virtualization of memory. In tandem, these two abstractions allow a program to run as if it is in its own private, isolated world; as if it has its own processor (or processors); as if it has its own memory. This illusion makes programming the system much easier and thus is prevalent today not only on desktops and servers but increasingly on all programmable platforms including mobile phones and the like.

In this section, we add one more critical piece to the virtualization puzzle: **persistent storage**. A persistent-storage device, such as a classic **hard disk drive** or a more modern **solid-state storage device**, stores information permanently (or at least, for a long time). Unlike memory, whose contents are lost when there is a power loss, a persistent-storage device keeps such data intact. Thus, the OS must take extra care with such a device: this is where users keep data that they really care about.

CRUX: HOW TO MANAGE A PERSISTENT DEVICE

How should the OS manage a persistent device? What are the APIs? What are the important aspects of the implementation?

Thus, in the next few chapters, we will explore critical techniques for managing persistent data, focusing on methods to improve performance and reliability. We begin, however, with an overview of the API: the interfaces you'll expect to see when interacting with a UNIX file system.

39.1 Files and Directories

Two key abstractions have developed over time in the virtualization of storage. The first is the **file**. A file is simply a linear array of bytes, each of which you can read or write. Each file has some kind of **low-level**

name, usually a number of some kind; often, the user is not aware of this name (as we will see). For historical reasons, the low-level name of a file is often referred to as its **inode number**. We'll be learning a lot more about inodes in future chapters; for now, just assume that each file has an inode number associated with it.

In most systems, the OS does not know much about the structure of the file (e.g., whether it is a picture, or a text file, or C code); rather, the responsibility of the file system is simply to store such data persistently on disk and make sure that when you request the data again, you get what you put there in the first place. Doing so is not as simple as it seems!

The second abstraction is that of a **directory**. A directory, like a file, also has a low-level name (i.e., an inode number), but its contents are quite specific: it contains a list of (user-readable name, low-level name) pairs. For example, let's say there is a file with the low-level name "10", and it is referred to by the user-readable name of "foo". The directory "foo" resides in thus would have an entry ("foo", "10") that maps the user-readable name to the low-level name. Each entry in a directory refers to either files or other directories. By placing directories within other directories, users are able to build an arbitrary **directory tree** (or **directory hierarchy**), under which all files and directories are stored.

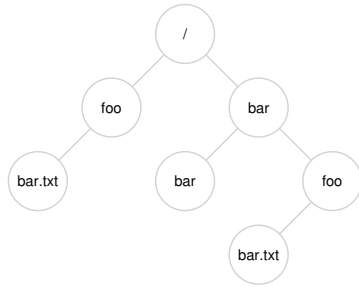


Figure 39.1: **An Example Directory Tree**

The directory hierarchy starts at a **root directory** (in UNIX-based systems, the root directory is simply referred to as `/`) and uses some kind of **separator** to name subsequent **sub-directories** until the desired file or directory is named. For example, if a user created a directory `foo` in the root directory `/`, and then created a file `bar.txt` in the directory `foo`, we could refer to the file by its **absolute pathname**, which in this case would be `/foo/bar.txt`. See Figure 39.1 for a more complex directory tree; valid directories in the example are `/`, `/foo`, `/bar`, `/bar/bar`, `/bar/foo` and valid files are `/foo/bar.txt` and `/bar/foo/bar.txt`. Directories and files can have the same name as long as they are in different locations in the file-system tree (e.g., there are two files named `bar.txt` in the figure, `/foo/bar.txt` and `/bar/foo/bar.txt`).

TIP: THINK CAREFULLY ABOUT NAMING
Naming is an important aspect of computer systems [SK09]. In UNIX systems, virtually everything that you can think of is named through the file system. Beyond just files, devices, pipes, and even processes [K84] can be found in what looks like a plain old file system. This uniformity of naming eases your conceptual model of the system, and makes the system simpler and more modular. Thus, whenever creating a system or interface, think carefully about what names you are using.

You may also notice that the file name in this example often has two parts: `bar` and `txt`, separated by a period. The first part is an arbitrary name, whereas the second part of the file name is usually used to indicate the **type** of the file, e.g., whether it is C code (e.g., `.c`), or an image (e.g., `.jpg`), or a music file (e.g., `.mp3`). However, this is usually just a **convention**: there is usually no enforcement that the data contained in a file named `main.c` is indeed C source code.

Thus, we can see one great thing provided by the file system: a convenient way to **name** all the files we are interested in. Names are important in systems as the first step to accessing any resource is being able to name it. In UNIX systems, the file system thus provides a unified way to access files on disk, USB stick, CD-ROM, many other devices, and in fact many other things, all located under the single directory tree.

39.2 The File System Interface

Let's now discuss the file system interface in more detail. We'll start with the basics of creating, accessing, and deleting files. You may think this is straightforward, but along the way we'll discover the mysterious call that is used to remove files, known as `unlink()`. Hopefully, by the end of this chapter, this mystery won't be so mysterious to you!

39.3 Creating Files

We'll start with the most basic of operations: creating a file. This can be accomplished with the `open` system call; by calling `open()` and passing it the `O_CREAT` flag, a program can create a new file. Here is some example code to create a file called "foo" in the current working directory.

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
```

The routine `open()` takes a number of different flags. In this example, the program creates the file (`O_CREAT`), can only write to that file while opened in this manner (`O_WRONLY`), and, if the file already exists, first truncate it to a size of zero bytes thus removing any existing content (`O_TRUNC`).

ASIDE: THE CREAT() SYSTEM CALL

The older way of creating a file is to call `creat()`, as follows:

```
int fd = creat("foo");
```

You can think of `creat()` as `open()` with the following flags: `O_CREAT | O_WRONLY | O_TRUNC`. Because `open()` can create a file, the usage of `creat()` has somewhat fallen out of favor (indeed, it could just be implemented as a library call to `open()`); however, it does hold a special place in UNIX lore. Specifically, when Ken Thompson was asked what he would do differently if he were redesigning UNIX, he replied: "I'd spell creat with an e."

One important aspect of `open()` is what it returns: a **file descriptor**. A file descriptor is just an integer, private per process, and is used in UNIX systems to access files; thus, once a file is opened, you use the file descriptor to read or write the file, assuming you have permission to do so. In this way, a file descriptor is a **capability** [L84], i.e., an opaque handle that gives you the power to perform certain operations. Another way to think of a file descriptor is as a pointer to an object of type file; once you have such an object, you can call other "methods" to access the file, like `read()` and `write()`. We'll see just how a file descriptor is used below.

39.4 Reading and Writing Files

Once we have some files, of course we might like to read or write them. Let's start by reading an existing file. If we were typing at a command line, we might just use the program `cat` to dump the contents of the file to the screen.

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

In this code snippet, we redirect the output of the program `echo` to the file `foo`, which then contains the word "hello" in it. We then use `cat` to see the contents of the file. But how does the `cat` program access the file `foo`?

To find this out, we'll use an incredibly useful tool to trace the system calls made by a program. On Linux, the tool is called **strace**; other systems have similar tools (see **dtruss** on Mac OS X, or **truss** on some older UNIX variants). What `strace` does is trace every system call made by a program while it runs, and dump the trace to the screen for you to see.

TIP: USE STRACE (AND SIMILAR TOOLS)

The `strace` tool provides an awesome way to see what programs are up to. By running it, you can trace which system calls a program makes, see the arguments and return codes, and generally get a very good idea of what is going on.

The tool also takes some arguments which can be quite useful. For example, `-f` follows any fork'd children too; `-t` reports the time of day at each call; `-e trace=open,close,read,write` only traces calls to those system calls and ignores all others. There are many more powerful flags — read the man pages and find out how to harness this wonderful tool.

Here is an example of using `strace` to figure out what `cat` is doing (some calls removed for readability):

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)       = 3
read(3, "hello\n", 4096)                = 6
write(1, "hello\n", 6)                  = 6
hello
read(3, "", 4096)                       = 0
close(3)                                = 0
...
prompt>
```

The first thing that `cat` does is open the file for reading. A couple of things we should note about this; first, that the file is only opened for reading (not writing), as indicated by the `O_RDONLY` flag; second, that the 64-bit offset be used (`O_LARGEFILE`); third, that the call to `open()` succeeds and returns a file descriptor, which has the value of 3.

Why does the first call to `open()` return 3, not 0 or perhaps 1 as you might expect? As it turns out, each running process already has three files open, standard input (which the process can read to receive input), standard output (which the process can write to in order to dump information to the screen), and standard error (which the process can write error messages to). These are represented by file descriptors 0, 1, and 2, respectively. Thus, when you first open another file (as `cat` does above), it will almost certainly be file descriptor 3.

After the open succeeds, `cat` uses the `read()` system call to repeatedly read some bytes from a file. The first argument to `read()` is the file descriptor, thus telling the file system which file to read; a process can of course have multiple files open at once, and thus the descriptor enables the operating system to know which file a particular read refers to. The second argument points to a buffer where the result of the `read()` will be placed; in the system-call trace above, strace shows the results of the read in this spot ("hello"). The third argument is the size of the buffer, which

in this case is 4 KB. The call to `read()` returns successfully as well, here returning the number of bytes it read (6, which includes 5 for the letters in the word "hello" and one for an end-of-line marker).

At this point, you see another interesting result of the strace: a single call to the `write()` system call, to the file descriptor 1. As we mentioned above, this descriptor is known as the standard output, and thus is used to write the word "hello" to the screen as the program `cat` is meant to do. But does it call `write()` directly? Maybe (if it is highly optimized). But if not, what `cat` might do is call the library routine `printf()`; internally, `printf()` figures out all the formatting details passed to it, and eventually calls write on the standard output to print the results to the screen.

The `cat` program then tries to read more from the file, but since there are no bytes left in the file, the `read()` returns 0 and the program knows that this means it has read the entire file. Thus, the program calls `close()` to indicate that it is done with the file "foo", passing in the corresponding file descriptor. The file is thus closed, and the reading of it thus complete.

Writing a file is accomplished via a similar set of steps. First, a file is opened for writing, then the `write()` system call is called, perhaps repeatedly for larger files, and then `close()`. Use `strace` to trace writes to a file, perhaps of a program you wrote yourself, or by tracing the `dd` utility, e.g., `dd if=foo of=bar`.

39.5 Reading And Writing, But Not Sequentially

Thus far, we've discussed how to read and write files, but all access has been **sequential**; that is, we have either read a file from the beginning to the end, or written a file out from beginning to end.

Sometimes, however, it is useful to be able to read or write to a specific offset within a file; for example, if you build an index over a text document, and use it to look up a specific word, you may end up reading from some **random** offsets within the document. To do so, we will use the `lseek()` system call. Here is the function prototype:

```
off_t lseek(int fildes, off_t offset, int whence);
```

The first argument is familiar (a file descriptor). The second argument is the `offset`, which positions the **file offset** to a particular location within the file. The third argument, called `whence` for historical reasons, determines exactly how the seek is performed. From the man page:

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
the file plus offset bytes.
```

As you can tell from this description, for each file a process opens, the OS tracks a "current" offset, which determines where the next read or

ASIDE: **CALLING LSEEK() DOES NOT PERFORM A DISK SEEK**
The poorly-named system call `lseek()` confuses many a student trying to understand disks and how the file systems atop them work. Do not confuse the two! The `lseek()` call simply changes a variable in OS memory that tracks, for a particular process, at which offset to which its next read or write will start. A disk seek occurs when a read or write issued to the disk is not on the same track as the last read or write, and thus necessitates a head movement. Making this even more confusing is the fact that calling `lseek()` to read or write from/to random parts of a file, and then reading/writing to those random parts, will indeed lead to more disk seeks. Thus, calling `lseek()` can certainly lead to a seek in an upcoming read or write, but absolutely does not cause any disk I/O to occur itself.

write will begin reading from or writing to within the file. Thus, part of the abstraction of an open file is that it has a current offset, which is updated in one of two ways. The first is when a read or write of N bytes takes place, N is added to the current offset; thus each read or write *implicitly* updates the offset. The second is *explicitly* with `lseek`, which changes the offset as specified above.

Note that this call `lseek()` has nothing to do with the **seek** operation of a disk, which moves the disk arm. The call to `lseek()` simply changes the value of a variable within the kernel; when the I/O is performed, depending on where the disk head is, the disk may or may not perform an actual seek to fulfill the request.

39.6 Writing Immediately with `fsync()`

Most times when a program calls `write()`, it is just telling the file system: please write this data to persistent storage, at some point in the future. The file system, for performance reasons, will **buffer** such writes in memory for some time (say 5 seconds, or 30); at that later point in time, the write(s) will actually be issued to the storage device. From the perspective of the calling application, writes seem to complete quickly, and only in rare cases (e.g., the machine crashes after the `write()` call but before the write to disk) will data be lost.

However, some applications require something more than this eventual guarantee. For example, in a database management system (DBMS), development of a correct recovery protocol requires the ability to force writes to disk from time to time.

To support these types of applications, most file systems provide some additional control APIs. In the UNIX world, the interface provided to applications is known as `fsync(int fd)`. When a process calls `fsync()` for a particular file descriptor, the file system responds by forcing all **dirty** (i.e., not yet written) data to disk, for the file referred to by the specified

file descriptor. The `fsync()` routine returns once all of these writes are complete.

Here is a simple example of how to use `fsync()`. The code opens the file `foo`, writes a single chunk of data to it, and then calls `fsync()` to ensure the writes are forced immediately to disk. Once the `fsync()` returns, the application can safely move on, knowing that the data has been persisted (if `fsync()` is correctly implemented, that is).

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

Interestingly, this sequence does not guarantee everything that you might expect; in some cases, you also need to `fsync()` the directory that contains the file `foo`. Adding this step ensures not only that the file itself is on disk, but that the file, if newly created, also is durably a part of the directory. Not surprisingly, this type of detail is often overlooked, leading to many application-level bugs [P+13].

39.7 Renaming Files

Once we have a file, it is sometimes useful to be able to give a file a different name. When typing at the command line, this is accomplished with `mv` command; in this example, the file `foo` is renamed `bar`:

```
prompt> mv foo bar
```

Using `strace`, we can see that `mv` uses the system call `rename(char *old, char *new)`, which takes precisely two arguments: the original name of the file (`old`) and the new name (`new`).

One interesting guarantee provided by the `rename()` call is that it is (usually) implemented as an **atomic** call with respect to system crashes; if the system crashes during the renaming, the file will either be named the old name or the new name, and no odd in-between state can arise. Thus, `rename()` is critical for supporting certain kinds of applications that require an atomic update to file state.

Let's be a little more specific here. Imagine that you are using a file editor (e.g., emacs), and you insert a line into the middle of a file. The file's name, for the example, is `foo.txt`. The way the editor might update the file to guarantee that the new file has the original contents plus the line inserted is as follows (ignoring error-checking for simplicity):

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

What the editor does in this example is simple: write out the new version of the file under a temporary name (`foot.txt.tmp`), force it to disk with `fsync()`, and then, when the application is certain the new file metadata and contents are on the disk, rename the temporary file to the original file's name. This last step atomically swaps the new file into place, while concurrently deleting the old version of the file, and thus an atomic file update is achieved.

39.8 Getting Information About Files

Beyond file access, we expect the file system to keep a fair amount of information about each file it is storing. We generally call such data about files **metadata**. To see the metadata for a certain file, we can use the `stat()` or `fstat()` system calls. These calls take a pathname (or file descriptor) to a file and fill in a `stat` structure as seen here:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

You can see that there is a lot of information kept about each file, including its size (in bytes), its low-level name (i.e., inode number), some ownership information, and some information about when the file was accessed or modified, among other things. To see this information, you can use the command line tool `stat`:

```
prompt> echo hello > file
prompt> stat file
  File: 'file'
  Size: 6 Blocks: 8          IO Block: 4096   regular file
Device: 811h/2065d Inode: 67158084    Links: 1
Access: (0640/-rw-r-----) Uid: (30686/ remzi) Gid: (30686/ remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

As it turns out, each file system usually keeps this type of information in a structure called an **inode**[1]. We'll be learning a lot more about inodes when we talk about file system implementation. For now, you should just think of an inode as a persistent data structure kept by the file system that has information like we see above inside of it.

39.9 Removing Files

At this point, we know how to create files and access them, either sequentially or not. But how do you delete files? If you've used UNIX, you probably think you know: just run the program `rm`. But what system call does `rm` use to remove a file?

Let's use our old friend `strace` again to find out. Here we remove that pesky file "foo":

```
prompt> strace rm foo
...
unlink("foo")                           = 0
...
```

We've removed a bunch of unrelated cruft from the traced output, leaving just a single call to the mysteriously-named system call `unlink()`. As you can see, `unlink()` just takes the name of the file to be removed, and returns zero upon success. But this leads us to a great puzzle: why is this system call named "unlink"? Why not just "remove" or "delete". To understand the answer to this puzzle, we must first understand more than just files, but also directories.

39.10 Making Directories

Beyond files, a set of directory-related system calls enable you to make, read, and delete directories. Note you can never write to a directory directly; because the format of the directory is considered file system metadata, you can only update a directory indirectly by, for example, creating files, directories, or other object types within it. In this way, the file system makes sure that the contents of the directory always are as expected.

To create a directory, a single system call, `mkdir()`, is available. The eponymous `mkdir` program can be used to create such a directory. Let's take a look at what happens when we run the `mkdir` program to make a simple directory called `foo`:

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)                      = 0
...
prompt>
```

[1] Some file systems call these structures similar, but slightly different, names, such as dnodes; the basic idea is similar however.

TIP: BE WARY OF POWERFUL COMMANDS

The program `rm` provides us with a great example of powerful commands, and how sometimes too much power can be a bad thing. For example, to remove a bunch of files at once, you can type something like:

```
prompt> rm *
```

where the `*` will match all files in the current directory. But sometimes you want to also delete the directories too, and in fact all of their contents. You can do this by telling `rm` to recursively descend into each directory, and remove its contents too:

```
prompt> rm -rf *
```

Where you get into trouble with this small string of characters is when you issue the command, accidentally, from the root directory of a file system, thus removing every file and directory from it. Oops!

Thus, remember the double-edged sword of powerful commands; while they give you the ability to do a lot of work with a small number of keystrokes, they also can quickly and readily do a great deal of harm.

When such a directory is created, it is considered "empty", although it does have a bare minimum of contents. Specifically, an empty directory has two entries: one entry that refers to itself, and one entry that refers to its parent. The former is referred to as the "." (dot) directory, and the latter as ".." (dot-dot). You can see these directories by passing a flag (`-a`) to the program `ls`:

```
prompt> ls -a
./  ../
prompt> ls -al
total 8
drwxr-x---  2 remzi remzi    6 Apr 30 16:17 ./
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ../
```

39.11 Reading Directories

Now that we've created a directory, we might wish to read one too. Indeed, that is exactly what the program `ls` does. Let's write our own little tool like `ls` and see how it is done.

Instead of just opening a directory as if it were a file, we instead use a new set of calls. Below is an example program that prints the contents of a directory. The program uses three calls, `opendir()`, `readdir()`, and `closedir()`, to get the job done, and you can see how simple the interface is; we just use a simple loop to read one directory entry at a time, and print out the name and inode number of each file in the directory.

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%d %s\n", (int) d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}
```

The declaration below shows the information available within each directory entry in the `struct dirent` data structure:

```
struct dirent {
    char           d_name[256]; /* filename */
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* offset to the next dirent */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file */
};
```

Because directories are light on information (basically, just mapping the name to the inode number, along with a few other details), a program may want to call `stat()` on each file to get more information on each, such as its length or other detailed information. Indeed, this is exactly what `ls` does when you pass it the `-l` flag; try `strace` on `ls` with and without that flag to see for yourself.

39.12 Deleting Directories

Finally, you can delete a directory with a call to `rmdir()` (which is used by the program of the same name, `rmdir`). Unlike file deletion, however, removing directories is more dangerous, as you could potentially delete a large amount of data with a single command. Thus, `rmdir()` has the requirement that the directory be empty (i.e., only has "." and ".." entries) before it is deleted. If you try to delete a non-empty directory, the call to `rmdir()` simply will fail.

39.13 Hard Links

We now come back to the mystery of why removing a file is performed via `unlink()`, by understanding a new way to make an entry in the file system tree, through a system call known as `link()`. The `link()` system call takes two arguments, an old pathname and a new one; when you "link" a new file name to an old one, you essentially create another way to refer to the same file. The command-line program `ln` is used to do this, as we see in this example:

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

Here we created a file with the word "hello" in it, and called the file `file`[2]. We then create a hard link to that file using the `ln` program. After this, we can examine the file by either opening `file` or `file2`.

The way `link` works is that it simply creates another name in the directory you are creating the link to, and refers it to the *same* inode number (i.e., low-level name) of the original file. The file is not copied in any way; rather, you now just have two human names (`file` and `file2`) that both refer to the same file. We can even see this in the directory itself, by printing out the inode number of each file:

```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

By passing the `-i` flag to `ls`, it prints out the inode number of each file (as well as the file name). And thus you can see what link really has done: just make a new reference to the same exact inode number (67158084 in this example).

By now you might be starting to see why `unlink()` is called `unlink()`. When you create a file, you are really doing *two* things. First, you are making a structure (the inode) that will track virtually all relevant information about the file, including its size, where its blocks are on disk, and so forth. Second, you are *linking* a human-readable name to that file, and putting that link into a directory.

After creating a hard link to a file, to the file system, there is no difference between the original file name (`file`) and the newly created file name (`file2`); indeed, they are both just links to the underlying metadata about the file, which is found in inode number 67158084.

Thus, to remove a file from the file system, we call `unlink()`. In the example above, we could for example remove the file named `file`, and still access the file without difficulty:

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

The reason this works is because when the file system unlinks file, it checks a **reference count** within the inode number. This reference count

[2] Note how creative the authors of this book are. We also used to have a cat named "Cat" (true story). However, she died, and we now have a hamster named "Hammy."

(sometimes called the **link count**) allows the file system to track how many different file names have been linked to this particular inode. When `unlink()` is called, it removes the "link" between the human-readable name (the file that is being deleted) to the given inode number, and decrements the reference count; only when the reference count reaches zero does the file system also free the inode and related data blocks, and thus truly "delete" the file.

You can see the reference count of a file using `stat()` of course. Let's see what it is when we create and delete hard links to a file. In this example, we'll create three links to the same file, and then delete them. Watch the link count!

```
prompt> echo hello > file
prompt> stat file
... Inode: 67158084    Links: 1 ...
prompt> ln file file2
prompt> stat file
... Inode: 67158084    Links: 2 ...
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084    Links: 3 ...
prompt> rm file
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> rm file2
prompt> stat file3
... Inode: 67158084    Links: 1 ...
prompt> rm file3
```

39.14 Symbolic Links

There is one other type of link that is really useful, and it is called a **symbolic link** or sometimes a **soft link**. As it turns out, hard links are somewhat limited: you can't create one to a directory (for fear that you will create a cycle in the directory tree); you can't hard link to files in other disk partitions (because inode numbers are only unique within a particular file system, not across file systems); etc. Thus, a new type of link called the symbolic link was created.

To create such a link, you can use the same program `ln`, but with the `-s` flag. Here is an example:

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

As you can see, creating a soft link looks much the same, and the original file can now be accessed through the file name `file` as well as the symbolic link name `file2`.

However, beyond this surface similarity, symbolic links are actually quite different from hard links. The first difference is that a symbolic link is actually a file itself, of a different type. We've already talked about regular files and directories; symbolic links are a third type the file system knows about. A `stat` on the symlink reveals all:

```
prompt> stat file
 ... regular file ...
prompt> stat file2
 ... symbolic link ...
```

Running `ls` also reveals this fact. If you look closely at the first character of the long-form of the output from `ls`, you can see that the first character in the left-most column is a `-` for regular files, a `d` for directories, and an `l` for soft links. You can also see the size of the symbolic link (4 bytes in this case), as well as what the link points to (the file named `file`).

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May  3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ../
-rw-r-----  1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

The reason that `file2` is 4 bytes is because the way a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file. Because we've linked to a file named `file`, our link file `file2` is small (4 bytes). If we link to a longer pathname, our link file would be bigger:

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi  6 May  3 19:17 alongerfilename
lrwxrwxrwx 1 remzi remzi 15 May  3 19:17 file3 -> alongerfilename
```

Finally, because of the way symbolic links are created, they leave the possibility for what is known as a **dangling reference**:

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

As you can see in this example, quite unlike hard links, removing the original file named `file` causes the link to point to a pathname that no longer exists.

39.15 Making and Mounting a File System

We've now toured the basic interfaces to access files, directories, and certain types of special types of links. But there is one more topic we should discuss: how to assemble a full directory tree from many underlying file systems. This task is accomplished via first making file systems, and then mounting them to make their contents accessible.

To make a file system, most file systems provide a tool, usually referred to as `mkfs` (pronounced "make fs"), that performs exactly this task. The idea is as follows: give the tool, as input, a device (such as a disk partition, e.g., `/dev/sda1`) a file system type (e.g., ext3), and it simply writes an empty file system, starting with a root directory, onto that disk partition. And mkfs said, let there be a file system!

However, once such a file system is created, it needs to be made accessible within the uniform file-system tree. This task is achieved via the `mount` program (which makes the underlying system call `mount()` to do the real work). What mount does, quite simply is take an existing directory as a target **mount point** and essentially paste a new file system onto the directory tree at that point.

An example here might be useful. Imagine we have an unmounted ext3 file system, stored in device partition `/dev/sda1`, that has the following contents: a root directory which contains two sub-directories, `a` and `b`, each of which in turn holds a single file named `foo`. Let's say we wish to mount this file system at the mount point `/home/users`. We would type something like this:

```
prompt> mount -t ext3 /dev/sda1 /home/users
```

If successful, the mount would thus make this new file system available. However, note how the new file system is now accessed. To look at the contents of the root directory, we would use `ls` like this:

```
prompt> ls /home/users/
a b
```

As you can see, the pathname `/home/users/` now refers to the root of the newly-mounted directory. Similarly, we could access files `a` and `b` with the pathnames `/home/users/a` and `/home/users/b`. Finally, the files named `foo` could be accessed via `/home/users/a/foo` and `/home/users/b/foo`. And thus the beauty of mount: instead of having a number of separate file systems, mount unifies all file systems into one tree, making naming uniform and convenient.

To see what is mounted on your system, and at which points, simply run the `mount` program. You'll see something like this:

```
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```

This crazy mix shows that a whole number of different file systems, including ext3 (a standard disk-based file system), the proc file system (a file system for accessing information about current processes), tmpfs (a file system just for temporary files), and AFS (a distributed file system) are all glued together onto this one machine's file-system tree.

39.16 Summary

The file system interface in UNIX systems (and indeed, in any system) is seemingly quite rudimentary, but there is a lot to understand if you wish to master it. Nothing is better, of course, than simply using it (a lot). So please do so! Of course, read more; as always, Stevens [SR05] is the place to begin.

We've toured the basic interfaces, and hopefully understood a little bit about how they work. Even more interesting is how to implement a file system that meets the needs of the API, a topic we will delve into in great detail next.

References

[K84] "Processes as Files"
Tom J. Killian
USENIX, June 1984
The paper that introduced the /proc file system, where each process can be treated as a file within a pseudo file system. A clever idea that you can still see in modern UNIX *systems.*

[L84] "Capability-Based Computer Systems"
Henry M. Levy
Digital Press, 1984
Available: http://homes.cs.washington.edu/ levy/capabook
An excellent overview of early capability-based systems.

[P+13] "Towards Efficient, Portable Application-Level Consistency"
Thanumalayan S. Pillai, Vijay Chidambaram, Joo-Young Hwang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
HotDep '13, November 2013
Our own work that shows how readily applications can make mistakes in committing data to disk; in particular, assumptions about the file system creep into applications and thus make the applications work correctly only if they are running on a specific file system.

[SK09] "Principles of Computer System Design"
Jerome H. Saltzer and M. Frans Kaashoek
Morgan-Kaufmann, 2009
This tour de force of systems is a must-read for anybody interested in the field. It's how they teach systems at MIT. Read it once, and then read it a few more times to let it all soak in.

[SR05] "Advanced Programming in the UNIX Environment"
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
We have probably referenced this book a few hundred thousand times. It is that useful to you, if you care to become an awesome systems programmer.

Homework

In this homework, we'll just familiarize ourselves with how the APIs described in the chapter work. To do so, you'll just write a few different programs, mostly based on various UNIX utilities.

Questions

1. **Stat:** Write your own version of the command line program `stat`, which simply calls the `stat()` system call on a given file or directory. Print out file size, number of blocks allocated, reference (link) count, and so forth. What is the link count of a directory, as the number of entries in the directory changes? Useful interfaces: `stat()`
2. **List Files:** Write a program that lists files in the given directory. When called without any arguments, the program should just print the file names. When invoked with the `-l` flag, the program should print out information about each file, such as the owner, group, permissions, and other information obtained from the `stat()` system call. The program should take one additional argument, which is the directory to read, e.g., `myls -l directory`. If no directory is given, the program should just use the current working directory. Useful interfaces: `stat()`, `opendir()`, `readdir()`, `getcwd()`.
3. **Tail:** Write a program that prints out the last few lines of a file. The program should be efficient, in that it seeks to near the end of the file, reads in a block of data, and then goes backwards until it finds the requested number of lines; at this point, it should print out those lines from beginning to the end of the file. To invoke the program, one should type: `mytail -n file`, where `n` is the number of lines at the end of the file to print. Useful interfaces: `stat()`, `lseek()`, `open()`, `read()`, `close()`.
4. **Recursive Search:** Write a program that prints out the names of each file and directory in the file system tree, starting at a given point in the tree. For example, when run without arguments, the program should start with the current working directory and print its contents, as well as the contents of any sub-directories, etc., until the entire tree, root at the CWD, is printed. If given a single argument (of a directory name), use that as the root of the tree instead. Refine your recursive search with more fun options, similar to the powerful `find` command line tool. Useful interfaces: you figure it out.

37

Hard Disk Drives

The last chapter introduced the general concept of an I/O device and showed you how the OS might interact with such a beast. In this chapter, we dive into more detail about one device in particular: the **hard disk drive**. These drives have been the main form of persistent data storage in computer systems for decades and much of the development of file system technology (coming soon) is predicated on their behavior. Thus, it is worth understanding the details of a disk's operation before building the file system software that manages it. Many of these details are available in excellent papers by Ruemmler and Wilkes [RW92] and Anderson, Dykes, and Riedel [ADR03].

CRUX: HOW TO STORE AND ACCESS DATA ON DISK
How do modern hard-disk drives store data? What is the interface? How is the data actually laid out and accessed? How does disk scheduling improve performance?

37.1 The Interface

Let's start by understanding the interface to a modern disk drive. The basic interface for all modern drives is straightforward. The drive consists of a large number of sectors (512-byte blocks), each of which can be read or written. The sectors are numbered from 0 to $n - 1$ on a disk with n sectors. Thus, we can view the disk as an array of sectors; 0 to $n - 1$ is thus the **address space** of the drive.

Multi-sector operations are possible; indeed, many file systems will read or write 4KB at a time (or more). However, when updating the disk, the only guarantee drive manufactures make is that a single 512-byte write is **atomic** (i.e., it will either complete in its entirety or it won't complete at all); thus, if an untimely power loss occurs, only a portion of a larger write may complete (sometimes called a **torn write**).

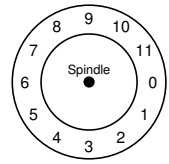


Figure 37.1: **A Disk With Just A Single Track**

There are some assumptions most clients of disk drives make, but that are not specified directly in the interface; Schlosser and Ganger have called this the "unwritten contract" of disk drives [SG04]. Specifically, one can usually assume that accessing two blocks that are near one-another within the drive's address space will be faster than accessing two blocks that are far apart. One can also usually assume that accessing blocks in a contiguous chunk (i.e., a sequential read or write) is the fastest access mode, and usually much faster than any more random access pattern.

37.2 Basic Geometry

Let's start to understand some of the components of a modern disk. We start with a **platter**, a circular hard surface on which data is stored persistently by inducing magnetic changes to it. A disk may have one or more platters; each platter has 2 sides, each of which is called a **surface**. These platters are usually made of some hard material (such as aluminum), and then coated with a thin magnetic layer that enables the drive to persistently store bits even when the drive is powered off.

The platters are all bound together around the **spindle**, which is connected to a motor that spins the platters around (while the drive is powered on) at a constant (fixed) rate. The rate of rotation is often measured in **rotations per minute (RPM)**, and typical modern values are in the 7,200 RPM to 15,000 RPM range. Note that we will often be interested in the time of a single rotation, e.g., a drive that rotates at 10,000 RPM means that a single rotation takes about 6 milliseconds (6 ms).

Data is encoded on each surface in concentric circles of sectors; we call one such concentric circle a **track**. A single surface contains many thousands and thousands of tracks, tightly packed together, with hundreds of tracks fitting into the width of a human hair.

To read and write from the surface, we need a mechanism that allows us to either sense (i.e., read) the magnetic patterns on the disk or to induce a change in (i.e., write) them. This process of reading and writing is accomplished by the **disk head**; there is one such head per surface of the drive. The disk head is attached to a single **disk arm**, which moves across the surface to position the head over the desired track.

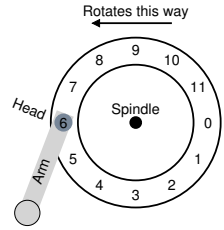


Figure 37.2: **A Single Track Plus A Head**

37.3 A Simple Disk Drive

Let's understand how disks work by building up a model one track at a time. Assume we have a simple disk with a single track (Figure 37.1).

This track has just 12 sectors, each of which is 512 bytes in size (our typical sector size, recall) and addressed therefore by the numbers 0 through 11. The single platter we have here rotates around the spindle, to which a motor is attached. Of course, the track by itself isn't too interesting; we want to be able to read or write those sectors, and thus we need a disk head, attached to a disk arm, as we now see (Figure 37.2).

In the figure, the disk head, attached to the end of the arm, is positioned over sector 6, and the surface is rotating counter-clockwise.

Single-track Latency: The Rotational Delay

To understand how a request would be processed on our simple, one-track disk, imagine we now receive a request to read block 0. How should the disk service this request?

In our simple disk, the disk doesn't have to do much. In particular, it must just wait for the desired sector to rotate under the disk head. This wait happens often enough in modern drives, and is an important enough component of I/O service time, that it has a special name: **rotational delay** (sometimes **rotation delay**, though that sounds weird). In the example, if the full rotational delay is R, the disk has to incur a rotational delay of about $\frac{R}{2}$ to wait for 0 to come under the read/write head (if we start at 6). A worst-case request on this single track would be to sector 5, causing nearly a full rotational delay in order to service such a request.

Multiple Tracks: Seek Time

So far our disk just has a single track, which is not too realistic; modern disks of course have many millions. Let's thus look at ever-so-slightly more realistic disk surface, this one with three tracks (Figure 37.3, left).

In the figure, the head is currently positioned over the innermost track (which contains sectors 24 through 35); the next track over contains the next set of sectors (12 through 23), and the outermost track contains the first sectors (0 through 11).

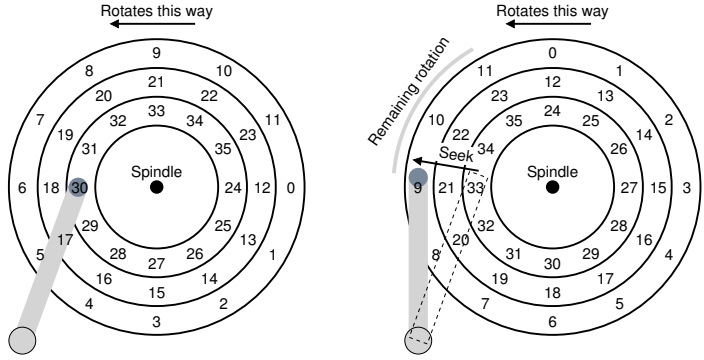


Figure 37.3: **Three Tracks Plus A Head (Right: With Seek)**

To understand how the drive might access a given sector, we now trace what would happen on a request to a distant sector, e.g., a read to sector 11. To service this read, the drive has to first move the disk arm to the correct track (in this case, the outermost one), in a process known as a **seek**. Seeks, along with rotations, are one of the most costly disk operations.

The seek, it should be noted, has many phases: first an *acceleration* phase as the disk arm gets moving; then *coasting* as the arm is moving at full speed, then *deceleration* as the arm slows down; finally *settling* as the head is carefully positioned over the correct track. The **settling time** is often quite significant, e.g., 0.5 to 2 ms, as the drive must be certain to find the right track (imagine if it just got close instead!).

After the seek, the disk arm has positioned the head over the right track. A depiction of the seek is found in Figure 37.3 (right).

As we can see, during the seek, the arm has been moved to the desired track, and the platter of course has rotated, in this case about 3 sectors. Thus, sector 9 is just about to pass under the disk head, and we must only endure a short rotational delay to complete the transfer.

When sector 11 passes under the disk head, the final phase of I/O will take place, known as the **transfer**, where data is either read from or written to the surface. And thus, we have a complete picture of I/O time: first a seek, then waiting for the rotational delay, and finally the transfer.

Some Other Details

Though we won't spend too much time on it, there are some other interesting details about how hard drives operate. Many drives employ some kind of **track skew** to make sure that sequential reads can be properly serviced even when crossing track boundaries. In our simple example disk, this might appear as seen in Figure 37.4.

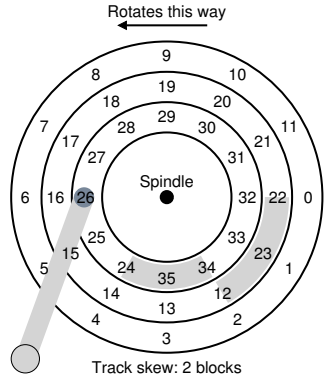


Figure 37.4: **Three Tracks: Track Skew Of 2**

Sectors are often skewed like this because when switching from one track to another, the disk needs time to reposition the head (even to neighboring tracks). Without such skew, the head would be moved to the next track but the desired next block would have already rotated under the head, and thus the drive would have to wait almost the entire rotational delay to access the next block.

Another reality is that outer tracks tend to have more sectors than inner tracks, which is a result of geometry; there is simply more room out there. These tracks are often referred to as **multi-zoned** disk drives, where the disk is organized into multiple zones, and where a zone is consecutive set of tracks on a surface. Each zone has the same number of sectors per track, and outer zones have more sectors than inner zones.

Finally, an important part of any modern disk drive is its **cache**, for historical reasons sometimes called a **track buffer**. This cache is just some small amount of memory (usually around 8 or 16 MB) which the drive can use to hold data read from or written to the disk. For example, when reading a sector from the disk, the drive might decide to read in all of the sectors on that track and cache them in its memory; doing so allows the drive to quickly respond to any subsequent requests to the same track.

On writes, the drive has a choice: should it acknowledge the write has completed when it has put the data in its memory, or after the write has actually been written to disk? The former is called **write back** caching (or sometimes **immediate reporting**), and the latter **write through**. Write back caching sometimes makes the drive appear "faster", but can be dangerous; if the file system or applications require that data be written to disk in a certain order for correctness, write-back caching can lead to problems (read the chapter on file-system journaling for details).

ASIDE: **DIMENSIONAL ANALYSIS**

Remember in Chemistry class, how you solved virtually every problem by simply setting up the units such that they canceled out, and somehow the answers popped out as a result? That chemical magic is known by the highfalutin name of **dimensional analysis** and it turns out it is useful in computer systems analysis too.

Let's do an example to see how dimensional analysis works and why it is useful. In this case, assume you have to figure out how long, in milliseconds, a single rotation of a disk takes. Unfortunately, you are given only the **RPM** of the disk, or **rotations per minute**. Let's assume we're talking about a 10K RPM disk (i.e., it rotates 10,000 times per minute). How do we set up the dimensional analysis so that we get time per rotation in milliseconds?

To do so, we start by putting the desired units on the left; in this case, we wish to obtain the time (in milliseconds) per rotation, so that is exactly what we write down: $\frac{Time\ (ms)}{1\ Rotation}$. We then write down everything we know, making sure to cancel units where possible. First, we obtain $\frac{1\ minute}{10{,}000\ Rotations}$ (keeping rotation on the bottom, as that's where it is on the left), then transform minutes into seconds with $\frac{60\ seconds}{1\ minute}$, and then finally transform seconds in milliseconds with $\frac{1000\ ms}{1\ second}$. The final result is the following (with units nicely canceled):

$$\frac{Time\ (ms)}{1\ Rot.} = \frac{1\ \cancel{minute}}{10{,}000\ Rot.} \cdot \frac{60\ \cancel{seconds}}{1\ \cancel{minute}} \cdot \frac{1000\ ms}{1\ \cancel{second}} = \frac{60{,}000\ ms}{10{,}000\ Rot.} = \frac{6\ ms}{Rotation}$$

As you can see from this example, dimensional analysis makes what seems obvious into a simple and repeatable process. Beyond the RPM calculation above, it comes in handy with I/O analysis regularly. For example, you will often be given the transfer rate of a disk, e.g., 100 MB/second, and then asked: how long does it take to transfer a 512 KB block (in milliseconds)? With dimensional analysis, it's easy:

$$\frac{Time\ (ms)}{1\ Request} = \frac{512\ \cancel{KB}}{1\ Request} \cdot \frac{1\ \cancel{MB}}{1024\ \cancel{KB}} \cdot \frac{1\ \cancel{second}}{100\ \cancel{MB}} \cdot \frac{1000\ ms}{1\ \cancel{second}} = \frac{5\ ms}{Request}$$

37.4 I/O Time: Doing The Math

Now that we have an abstract model of the disk, we can use a little analysis to better understand disk performance. In particular, we can now represent I/O time as the sum of three major components:

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer} \tag{37.1}$$

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

Figure 37.5: **Disk Drive Specs: SCSI Versus SATA**

Note that the rate of I/O ($R_{I/O}$), which is often more easily used for comparison between drives (as we will do below), is easily computed from the time. Simply divide the size of the transfer by the time it took:

$$R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}} \tag{37.2}$$

To get a better feel for I/O time, let us perform the following calculation. Assume there are two workloads we are interested in. The first, known as the **random** workload, issues small (e.g., 4KB) reads to random locations on the disk. Random workloads are common in many important applications, including database management systems. The second, known as the **sequential** workload, simply reads a large number of sectors consecutively from the disk, without jumping around. Sequential access patterns are quite common and thus important as well.

To understand the difference in performance between random and sequential workloads, we need to make a few assumptions about the disk drive first. Let's look at a couple of modern disks from Seagate. The first, known as the Cheetah 15K.5 [S09b], is a high-performance SCSI drive. The second, the Barracuda [S09a], is a drive built for capacity. Details on both are found in Figure 37.5.

As you can see, the drives have quite different characteristics, and in many ways nicely summarize two important components of the disk drive market. The first is the "high performance" drive market, where drives are engineered to spin as fast as possible, deliver low seek times, and transfer data quickly. The second is the "capacity" market, where cost per byte is the most important aspect; thus, the drives are slower but pack as many bits as possible into the space available.

From these numbers, we can start to calculate how well the drives would do under our two workloads outlined above. Let's start by looking at the random workload. Assuming each 4 KB read occurs at a random location on disk, we can calculate how long each such read would take. On the Cheetah:

$$T_{seek} = 4\ ms,\ T_{rotation} = 2\ ms,\ T_{transfer} = 30\ microsecs \tag{37.3}$$

TIP: USE DISKS SEQUENTIALLY

When at all possible, transfer data to and from disks in a sequential manner. If sequential is not possible, at least think about transferring data in large chunks: the bigger, the better. If I/O is done in little random pieces, I/O performance will suffer dramatically. Also, users will suffer. Also, you will suffer, knowing what suffering you have wrought with your careless random I/Os.

The average seek time (4 milliseconds) is just taken as the average time reported by the manufacturer; note that a full seek (from one end of the surface to the other) would likely take two or three times longer. The average rotational delay is calculated from the RPM directly. 15000 RPM is equal to 250 RPS (rotations per second); thus, each rotation takes 4 ms. On average, the disk will encounter a half rotation and thus 2 ms is the average time. Finally, the transfer time is just the size of the transfer over the peak transfer rate; here it is vanishingly small (30 *microseconds*; note that we need 1000 microseconds just to get 1 millisecond!).

Thus, from our equation above, $T_{I/O}$ for the Cheetah roughly equals 6 ms. To compute the rate of I/O, we just divide the size of the transfer by the average time, and thus arrive at $R_{I/O}$ for the Cheetah under the random workload of about 0.66 MB/s. The same calculation for the Barracuda yields a $T_{I/O}$ of about 13.2 ms, more than twice as slow, and thus a rate of about 0.31 MB/s.

Now let's look at the sequential workload. Here we can assume there is a single seek and rotation before a very long transfer. For simplicity, assume the size of the transfer is 100 MB. Thus, $T_{I/O}$ for the Barracuda and Cheetah is about 800 ms and 950 ms, respectively. The rates of I/O are thus very nearly the peak transfer rates of 125 MB/s and 105 MB/s, respectively. Figure 37.6 summarizes these numbers.

The figure shows us a number of important things. First, and most importantly, there is a huge gap in drive performance between random and sequential workloads, almost a factor of 200 or so for the Cheetah and more than a factor 300 difference for the Barracuda. And thus we arrive at the most obvious design tip in the history of computing.

A second, more subtle point: there is a large difference in performance between high-end "performance" drives and low-end "capacity" drives. For this reason (and others), people are often willing to pay top dollar for the former while trying to get the latter as cheaply as possible.

	Cheetah	Barracuda
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$R_{I/O}$ Sequential	125 MB/s	105 MB/s

Figure 37.6: **Disk Drive Performance: SCSI Versus SATA**

ASIDE: **COMPUTING THE "AVERAGE" SEEK**

In many books and papers, you will see average disk-seek time cited as being roughly one-third of the full seek time. Where does this come from?

Turns out it arises from a simple calculation based on average seek *distance*, not time. Imagine the disk as a set of tracks, from 0 to N. The seek distance between any two tracks x and y is thus computed as the absolute value of the difference between them: $|x - y|$.

To compute the average seek distance, all you need to do is to first add up all possible seek distances:

$$\sum_{x=0}^{N}\sum_{y=0}^{N}|x-y|. \tag{37.4}$$

Then, divide this by the number of different possible seeks: N^2. To compute the sum, we'll just use the integral form:

$$\int_{x=0}^{N}\int_{y=0}^{N}|x-y|\,\mathrm{d}y\,\mathrm{d}x. \tag{37.5}$$

To compute the inner integral, let's break out the absolute value:

$$\int_{y=0}^{x}(x-y)\,\mathrm{d}y+\int_{y=x}^{N}(y-x)\,\mathrm{d}y. \tag{37.6}$$

Solving this leads to $(xy-\frac{1}{2}y^2)\big|_0^x+(\frac{1}{2}y^2-xy)\big|_x^N$ which can be simplified to $(x^2-Nx+\frac{1}{2}N^2)$. Now we have to compute the outer integral:

$$\int_{x=0}^{N}(x^2-Nx+\frac{1}{2}N^2)\,\mathrm{d}x, \tag{37.7}$$

which results in:

$$(\frac{1}{3}x^3-\frac{N}{2}x^2+\frac{N^2}{2}x)\Big|_0^N=\frac{N^3}{3}. \tag{37.8}$$

Remember that we still have to divide by the total number of seeks (N^2) to compute the average seek distance: $(\frac{N^3}{3})/(N^2)=\frac{1}{3}N$. Thus the average seek distance on a disk, over all possible seeks, is one-third the full distance. And now when you hear that an average seek is one-third of a full seek, you'll know where it came from.

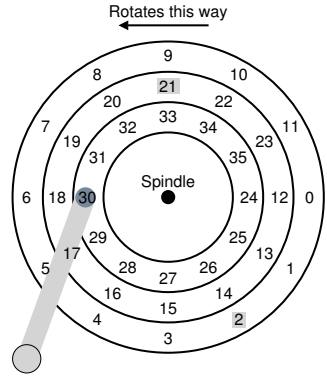


Figure 37.7: **SSTF: Scheduling Requests 21 And 2**

37.5 Disk Scheduling

Because of the high cost of I/O, the OS has historically played a role in deciding the order of I/Os issued to the disk. More specifically, given a set of I/O requests, the **disk scheduler** examines the requests and decides which one to schedule next [SCO90, JW91].

Unlike job scheduling, where the length of each job is usually unknown, with disk scheduling, we can make a good guess at how long a "job" (i.e., disk request) will take. By estimating the seek and possible the rotational delay of a request, the disk scheduler can know how long each request will take, and thus (greedily) pick the one that will take the least time to service first. Thus, the disk scheduler will try to follow the **principle of SJF (shortest job first)** in its operation.

SSTF: Shortest Seek Time First

One early disk scheduling approach is known as **shortest-seek-time-first** (**SSTF**) (also called **shortest-seek-first** or **SSF**). SSTF orders the queue of I/O requests by track, picking requests on the nearest track to complete first. For example, assuming the current position of the head is over the inner track, and we have requests for sectors 21 (middle track) and 2 (outer track), we would then issue the request to 21 first, wait for it to complete, and then issue the request to 2 (Figure 37.7).

SSTF works well in this example, seeking to the middle track first and then the outer track. However, SSTF is not a panacea, for the following reasons. First, the drive geometry is not available to the host OS; rather, it sees an array of blocks. Fortunately, this problem is rather easily fixed. Instead of SSTF, an OS can simply implement **nearest-block-first** (**NBF**), which schedules the request with the nearest block address next.

The second problem is more fundamental: **starvation**. Imagine in our example above if there were a steady stream of requests to the inner track, where the head currently is positioned. Requests to any other tracks would then be ignored completely by a pure SSTF approach. And thus the crux of the problem:

CRUX: HOW TO HANDLE DISK STARVATION
How can we implement SSTF-like scheduling but avoid starvation?

Elevator (a.k.a. SCAN or C-SCAN)

The answer to this query was developed some time ago (see [CKR72] for example), and is relatively straightforward. The algorithm, originally called **SCAN**, simply moves across the disk servicing requests in order across the tracks. Let us call a single pass across the disk a *sweep*. Thus, if a request comes for a block on a track that has already been serviced on this sweep of the disk, it is not handled immediately, but rather queued until the next sweep.

SCAN has a number of variants, all of which do about the same thing. For example, Coffman et al. introduced **F-SCAN**, which freezes the queue to be serviced when it is doing a sweep [CKR72]; this action places requests that come in during the sweep into a queue to be serviced later. Doing so avoids starvation of far-away requests, by delaying the servicing of late-arriving (but nearer by) requests.

C-SCAN is another common variant, short for **Circular SCAN**. Instead of sweeping in one direction across the disk, the algorithm sweeps from outer-to-inner, and then inner-to-outer, etc.

For reasons that should now be obvious, this algorithm (and its variants) is sometimes referred to as the **elevator** algorithm, because it behaves like an elevator which is either going up or down and not just servicing requests to floors based on which floor is closer. Imagine how annoying it would be if you were going down from floor 10 to 1, and somebody got on at 3 and pressed 4, and the elevator went up to 4 because it was "closer" than 1! As you can see, the elevator algorithm, when used in real life, prevents fights from taking place on elevators. In disks, it just prevents starvation.

Unfortunately, SCAN and its cousins do not represent the best scheduling technology. In particular, SCAN (or SSTF even) do not actually adhere as closely to the principle of SJF as they could. In particular, they ignore rotation. And thus, another crux:

CRUX: HOW TO ACCOUNT FOR DISK ROTATION COSTS
How can we implement an algorithm that more closely approximates SJF by taking *both* seek and rotation into account?

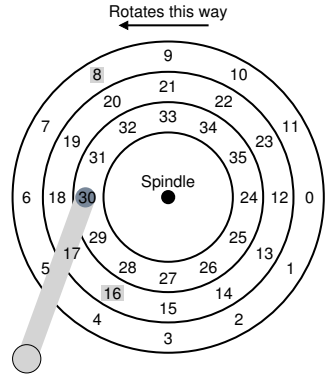


Figure 37.8: **SSTF: Sometimes Not Good Enough**

SPTF: Shortest Positioning Time First

Before discussing **shortest positioning time first** or **SPTF** scheduling (sometimes also called **shortest access time first** or **SATF**), which is the solution to our problem, let us make sure we understand the problem in more detail. Figure 37.8 presents an example.

In the example, the head is currently positioned over sector 30 on the inner track. The scheduler thus has to decide: should it schedule sector 16 (on the middle track) or sector 8 (on the outer track) for its next request. So which should it service next?

The answer, of course, is "it depends". In engineering, it turns out "it depends" is almost always the answer, reflecting that trade-offs are part of the life of the engineer; such maxims are also good in a pinch, e.g., when you don't know an answer to your boss's question, you might want to try this gem. However, it is almost always better to know *why* it depends, which is what we discuss here.

What it depends on here is the relative time of seeking as compared to rotation. If, in our example, seek time is much higher than rotational delay, then SSTF (and variants) are just fine. However, imagine if seek is quite a bit faster than rotation. Then, in our example, it would make more sense to seek *further* to service request 8 on the outer track than it would to perform the shorter seek to the middle track to service 16, which has to rotate all the way around before passing under the disk head.

On modern drives, as we saw above, both seek and rotation are roughly equivalent (depending, of course, on the exact requests), and thus SPTF is useful and improves performance. However, it is even more difficult to implement in an OS, which generally does not have a good idea where track boundaries are or where the disk head currently is (in a rotational sense). Thus, SPTF is usually performed inside a drive, described below.

TIP: IT ALWAYS DEPENDS (LIVNY'S LAW)
Almost any question can be answered with "it depends", as our colleague Miron Livny always says. However, use with caution, as if you answer too many questions this way, people will stop asking you questions altogether. For example, somebody asks: "want to go to lunch?" You reply: "it depends, are *you* coming along?"

Other Scheduling Issues

There are many other issues we do not discuss in this brief description of basic disk operation, scheduling, and related topics. One such issue is this: *where* is disk scheduling performed on modern systems? In older systems, the operating system did all the scheduling; after looking through the set of pending requests, the OS would pick the best one, and issue it to the disk. When that request completed, the next one would be chosen, and so forth. Disks were simpler then, and so was life.

In modern systems, disks can accommodate multiple outstanding requests, and have sophisticated internal schedulers themselves (which can implement SPTF accurately; inside the disk controller, all relevant details are available, including exact head position). Thus, the OS scheduler usually picks what it thinks the best few requests are (say 16) and issues them all to disk; the disk then uses its internal knowledge of head position and detailed track layout information to service said requests in the best possible (SPTF) order.

Another important related task performed by disk schedulers is **I/O merging**. For example, imagine a series of requests to read blocks 33, then 8, then 34, as in Figure 37.8. In this case, the scheduler should **merge** the requests for blocks 33 and 34 into a single two-block request; any reordering that the scheduler does is performed upon the merged requests. Merging is particularly important at the OS level, as it reduces the number of requests sent to the disk and thus lowers overheads.

One final problem that modern schedulers address is this: how long should the system wait before issuing an I/O to disk? One might naively think that the disk, once it has even a single I/O, should immediately issue the request to the drive; this approach is called **work-conserving**, as the disk will never be idle if there are requests to serve. However, research on **anticipatory disk scheduling** has shown that sometimes it is better to wait for a bit [ID01], in what is called a **non-work-conserving** approach. By waiting, a new and "better" request may arrive at the disk, and thus overall efficiency is increased. Of course, deciding when to wait, and for how long, can be tricky; see the research paper for details, or check out the Linux kernel implementation to see how such ideas are transitioned into practice (if you are the ambitious sort).

37.6 Summary

We have presented a summary of how disks work. The summary is actually a detailed functional model; it does not describe the amazing physics, electronics, and material science that goes into actual drive design. For those interested in even more details of that nature, we suggest a different major (or perhaps minor); for those that are happy with this model, good! We can now proceed to using the model to build more interesting systems on top of these incredible devices.

References

[ADR03] "More Than an Interface: SCSI vs. ATA"
Dave Anderson, Jim Dykes, Erik Riedel
FAST '03, 2003
One of the best recent-ish references on how modern disk drives really work; a must read for anyone interested in knowing more.

[CKR72] "Analysis of Scanning Policies for Reducing Disk Seek Times"
E.G. Coffman, L.A. Klimko, B. Ryan
SIAM Journal of Computing, September 1972, Vol 1. No 3.
Some of the early work in the field of disk scheduling.

[ID01] "Anticipatory Scheduling: A Disk-scheduling Framework
To Overcome Deceptive Idleness In Synchronous I/O"
Sitaram Iyer, Peter Druschel
SOSP '01, October 2001
A cool paper showing how waiting can improve disk scheduling: better requests may be on their way!

[JW91] "Disk Scheduling Algorithms Based On Rotational Position"
D. Jacobson, J. Wilkes
Technical Report HPL-CSP-91-7rev1, Hewlett-Packard (February 1991)
A more modern take on disk scheduling. It remains a technical report (and not a published paper) because the authors were scooped by Seltzer et al. [SCO90].

[RW92] "An Introduction to Disk Drive Modeling"
C. Ruemmler, J. Wilkes
IEEE Computer, 27:3, pp. 17-28, March 1994
A terrific introduction to the basics of disk operation. Some pieces are out of date, but most of the basics remain.

[SCO90] "Disk Scheduling Revisited"
Margo Seltzer, Peter Chen, John Ousterhout
USENIX 1990
A paper that talks about how rotation matters too in the world of disk scheduling.

[SG04] "MEMS-based storage devices and standard disk interfaces:
A square peg in a round hole?"
Steven W. Schlosser, Gregory R. Ganger
FAST '04, pp. 87-100, 2004
While the MEMS aspect of this paper hasn't yet made an impact, the discussion of the contract between file systems and disks is wonderful and a lasting contribution.

[S09a] "Barracuda ES.2 data sheet"
`http://www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah_15k_5.pdf` *A data sheet; read at your own risk. Risk of what? Boredom.*

[S09b] "Cheetah 15K.5"
`http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_es.pdf` *See above commentary on data sheets.*

Homework

This homework uses `disk.py` to familiarize you with how a modern hard drive works. It has a lot of different options, and unlike most of the other simulations, has a graphical animator to show you exactly what happens when the disk is in action. See the README for details.

1. Compute the seek, rotation, and transfer times for the following sets of requests: `-a 0`, `-a 6`, `-a 30`, `-a 7,30,8`, and finally `-a 10,11,12,13`.
2. Do the same requests above, but change the seek rate to different values: `-S 2`, `-S 4`, `-S 8`, `-S 10`, `-S 40`, `-S 0.1`. How do the times change?
3. Do the same requests above, but change the rotation rate: `-R 0.1`, `-R 0.5`, `-R 0.01`. How do the times change?
4. You might have noticed that some request streams would be better served with a policy better than FIFO. For example, with the request stream `-a 7,30,8`, what order should the requests be processed in? Now run the shortest seek-time first (SSTF) scheduler (`-p SSTF`) on the same workload; how long should it take (seek, rotation, transfer) for each request to be served?
5. Now do the same thing, but using the shortest access-time first (SATF) scheduler (`-p SATF`). Does it make any difference for the set of requests as specified by `-a 7,30,8`? Find a set of requests where SATF does noticeably better than SSTF; what are the conditions for a noticeable difference to arise?
6. You might have noticed that the request stream `-a 10,11,12,13` wasn't particularly well handled by the disk. Why is that? Can you introduce a track skew to address this problem (`-o skew`, where `skew` is a non-negative integer)? Given the default seek rate, what should the skew be to minimize the total time for this set of requests? What about for different seek rates (e.g., `-S 2`, `-S 4`)? In general, could you write a formula to figure out the skew, given the seek rate and sector layout information?
7. Multi-zone disks pack more sectors into the outer tracks. To configure this disk in such a way, run with the `-z` flag. Specifically, try running some requests against a disk run with `-z 10,20,30` (the numbers specify the angular space occupied by a sector, per track; in this example, the outer track will be packed with a sector every 10 degrees, the middle track every 20 degrees, and the inner track with a sector every 30 degrees). Run some random requests (e.g., `-a -1 -A 5,-1,0`, which specifies that random requests should be used via the `-a -1` flag and that five requests ranging from 0 to the max be generated), and see if you can compute the seek, rotation, and transfer times. Use different random seeds (`-s 1`, `-s 2`, etc.). What is the bandwidth (in sectors per unit time) on the outer, middle, and inner tracks?

8. Scheduling windows determine how many sector requests a disk can examine at once in order to determine which sector to serve next. Generate some random workloads of a lot of requests (e.g., `-A 1000,-1,0`, with different seeds perhaps) and see how long the SATF scheduler takes when the scheduling window is changed from 1 up to the number of requests (e.g., `-w 1` up to `-w 1000`, and some values in between). How big of scheduling window is needed to approach the best possible performance? Make a graph and see. Hint: use the `-c` flag and don't turn on graphics with `-G` to run these more quickly. When the scheduling window is set to 1, does it matter which policy you are using?
9. Avoiding starvation is important in a scheduler. Can you think of a series of requests such that a particular sector is delayed for a very long time given a policy such as SATF? Given that sequence, how does it perform if you use a **bounded SATF** or **BSATF** scheduling approach? In this approach, you specify the scheduling window (e.g., `-w 4`) as well as the BSATF policy (`-p BSATF`); the scheduler then will only move onto the next window of requests when *all* of the requests in the current window have been serviced. Does this solve the starvation problem? How does it perform, as compared to SATF? In general, how should a disk make this trade-off between performance and starvation avoidance?
10. All the scheduling policies we have looked at thus far are **greedy**, in that they simply pick the next best option instead of looking for the optimal schedule over a set of requests. Can you find a set of requests in which this greedy approach is not optimal?

38

Redundant Arrays of Inexpensive Disks (RAIDs)

When we use a disk, we sometimes wish it to be faster; I/O operations are slow and thus can be the bottleneck for the entire system. When we use a disk, we sometimes wish it to be larger; more and more data is being put online and thus our disks are getting fuller and fuller. When we use a disk, we sometimes wish for it to be more reliable; when a disk fails, if our data isn't backed up, all that valuable data is gone.

CRUX: HOW TO MAKE A LARGE, FAST, RELIABLE DISK
How can we make a large, fast, and reliable storage system? What are the key techniques? What are trade-offs between different approaches?

In this chapter, we introduce the **Redundant Array of Inexpensive Disks** better known as **RAID** [P+88], a technique to use multiple disks in concert to build a faster, bigger, and more reliable disk system. The term was introduced in the late 1980s by a group of researchers at U.C. Berkeley (led by Professors David Patterson and Randy Katz and then student Garth Gibson); it was around this time that many different researchers simultaneously arrived upon the basic idea of using multiple disks to build a better storage system [BG88, K86,K88,PB86,SG86].

Externally, a RAID looks like a disk: a group of blocks one can read or write. Internally, the RAID is a complex beast, consisting of multiple disks, memory (both volatile and non-), and one or more processors to manage the system. A hardware RAID is very much like a computer system, specialized for the task of managing a group of disks.

RAIDs offer a number of advantages over a single disk. One advantage is *performance*. Using multiple disks in parallel can greatly speed up I/O times. Another benefit is *capacity*. Large data sets demand large disks. Finally, RAIDs can improve *reliability*; spreading data across multiple disks (without RAID techniques) makes the data vulnerable to the loss of a single disk; with some form of **redundancy**, RAIDs can tolerate the loss of a disk and keep operating as if nothing were wrong.

TIP: TRANSPARENCY ENABLES DEPLOYMENT
When considering how to add new functionality to a system, one should always consider whether such functionality can be added **transparently**, in a way that demands no changes to the rest of the system. Requiring a complete rewrite of the existing software (or radical hardware changes) lessens the chance of impact of an idea. RAID is a perfect example, and certainly its transparency contributed to its success; administrators could install a SCSI-based RAID storage array instead of a SCSI disk, and the rest of the system (host computer, OS, etc.) did not have to change one bit to start using it. By solving this problem of **deployment**, RAID was made more successful from day one.

Amazingly, RAIDs provide these advantages **transparently** to systems that use them, i.e., a RAID just looks like a big disk to the host system. The beauty of transparency, of course, is that it enables one to simply replace a disk with a RAID and not change a single line of software; the operating system and client applications continue to operate without modification. In this manner, transparency greatly improves the **deployability** of RAID, enabling users and administrators to put a RAID to use without worries of software compatibility.

We now discuss some of the important aspects of RAIDs. We begin with the interface, fault model, and then discuss how one can evaluate a RAID design along three important axes: capacity, reliability, and performance. We then discuss a number of other issues that are important to RAID design and implementation.

38.1 Interface And RAID Internals

To a file system above, a RAID looks like a big, (hopefully) fast, and (hopefully) reliable disk. Just as with a single disk, it presents itself as a linear array of blocks, each of which can be read or written by the file system (or other client).

When a file system issues a *logical I/O* request to the RAID, the RAID internally must calculate which disk (or disks) to access in order to complete the request, and then issue one or more *physical I/Os* to do so. The exact nature of these physical I/Os depends on the RAID level, as we will discuss in detail below. However, as a simple example, consider a RAID that keeps two copies of each block (each one on a separate disk); when writing to such a **mirrored** RAID system, the RAID will have to perform two physical I/Os for every one logical I/O it is issued.

A RAID system is often built as a separate hardware box, with a standard connection (e.g., SCSI, or SATA) to a host. Internally, however, RAIDs are fairly complex, consisting of a microcontroller that runs firmware to direct the operation of the RAID, volatile memory such as DRAM to buffer data blocks as they are read and written, and in some cases,

non-volatile memory to buffer writes safely and perhaps even specialized logic to perform parity calculations (useful in some RAID levels, as we will also see below). At a high level, a RAID is very much a specialized computer system: it has a processor, memory, and disks; however, instead of running applications, it runs specialized software designed to operate the RAID.

38.2 Fault Model

To understand RAID and compare different approaches, we must have a fault model in mind. RAIDs are designed to detect and recover from certain kinds of disk faults; thus, knowing exactly which faults to expect is critical in arriving upon a working design.

The first fault model we will assume is quite simple, and has been called the **fail-stop** fault model [S84]. In this model, a disk can be in exactly one of two states: working or failed. With a working disk, all blocks can be read or written. In contrast, when a disk has failed, we assume it is permanently lost.

One critical aspect of the fail-stop model is what it assumes about fault detection. Specifically, when a disk has failed, we assume that this is easily detected. For example, in a RAID array, we would assume that the RAID controller hardware (or software) can immediately observe when a disk has failed.

Thus, for now, we do not have to worry about more complex "silent" failures such as disk corruption. We also do not have to worry about a single block becoming inaccessible upon an otherwise working disk (sometimes called a latent sector error). We will consider these more complex (and unfortunately, more realistic) disk faults later.

38.3 How To Evaluate A RAID

As we will soon see, there are a number of different approaches to building a RAID. Each of these approaches has different characteristics which are worth evaluating, in order to understand their strengths and weaknesses.

Specifically, we will evaluate each RAID design along three axes. The first axis is **capacity**; given a set of N disks, how much useful capacity is available to clients of the RAID? Without redundancy, the answer is obviously N; in contrast, if we have a system that keeps two copies of each block, we will obtain a useful capacity of N/2. Different schemes (e.g., parity-based ones) tend to fall in between.

The second axis of evaluation is **reliability**. How many disk faults can the given design tolerate? In alignment with our fault model, we assume only that an entire disk can fail; in later chapters (i.e., on data integrity), we'll think about how to handle more complex failure modes.

Finally, the third axis is **performance**. Performance is somewhat chal-

lenging to evaluate, because it depends heavily on the workload presented to the disk array. Thus, before evaluating performance, we will first present a set of typical workloads that one should consider.

We now consider three important RAID designs: RAID Level 0 (striping), RAID Level 1 (mirroring), and RAID Levels 4/5 (parity-based redundancy). The naming of each of these designs as a "level" stems from the pioneering work of Patterson, Gibson, and Katz at Berkeley [P+88].

38.4 RAID Level 0: Striping

The first RAID level is actually not a RAID level at all, in that there is no redundancy. However, RAID level 0, or **striping** as it is better known, serves as an excellent upper-bound on performance and capacity and thus is worth understanding.

The simplest form of striping will **stripe** blocks across the disks of the system as follows (assume here a 4-disk array):

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 38.1: **RAID-0: Simple Striping**

From Figure 38.1, you get the basic idea: spread the blocks of the array across the disks in a round-robin fashion. This approach is designed to extract the most parallelism from the array when requests are made for contiguous chunks of the array (as in a large, sequential read, for example). We call the blocks in the same row a **stripe**; thus, blocks 0, 1, 2, and 3 are in the same stripe above.

In the example, we have made the simplifying assumption that only 1 block (each of say size 4KB) is placed on each disk before moving on to the next. However, this arrangement need not be the case. For example, we could arrange the blocks across disks as in Figure 38.2:

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	chunk size:
1	3	5	7	2 blocks
8	10	12	14	
9	11	13	15	

Figure 38.2: **Striping with a Bigger Chunk Size**

In this example, we place two 4KB blocks on each disk before moving on to the next disk. Thus, the **chunk size** of this RAID array is 8KB, and a stripe thus consists of 4 chunks or 32KB of data.

ASIDE: **THE RAID MAPPING PROBLEM**

Before studying the capacity, reliability, and performance characteristics of the RAID, we first present an aside on what we call **the mapping problem**. This problem arises in all RAID arrays; simply put, given a logical block to read or write, how does the RAID know exactly which physical disk and offset to access?

For these simple RAID levels, we do not need much sophistication in order to correctly map logical blocks onto their physical locations. Take the first striping example above (chunk size = 1 block = 4KB). In this case, given a logical block address A, the RAID can easily compute the desired disk and offset with two simple equations:

```
Disk   = A % number_of_disks
Offset = A / number_of_disks
```

Note that these are all integer operations (e.g., 4 / 3 = 1 not 1.33333...).

Let's see how these equations work for a simple example. Imagine in the first RAID above that a request arrives for block 14. Given that there are 4 disks, this would mean that the disk we are interested in is (14 % 4 = 2): disk 2. The exact block is calculated as (14 / 4 = 3): block 3. Thus, block 14 should be found on the fourth block (block 3, starting at 0) of the third disk (disk 2, starting at 0), which is exactly where it is.

You can think about how these equations would be modified to support different chunk sizes. Try it! It's not too hard.

Chunk Sizes

Chunk size mostly affects performance of the array. For example, a small chunk size implies that many files will get striped across many disks, thus increasing the parallelism of reads and writes to a single file; however, the positioning time to access blocks across multiple disks increases, because the positioning time for the entire request is determined by the maximum of the positioning times of the requests across all drives.

A big chunk size, on the other hand, reduces such intra-file parallelism, and thus relies on multiple concurrent requests to achieve high throughput. However, large chunk sizes reduce positioning time; if, for example, a single file fits within a chunk and thus is placed on a single disk, the positioning time incurred while accessing it will just be the positioning time of a single disk.

Thus, determining the "best" chunk size is hard to do, as it requires a great deal of knowledge about the workload presented to the disk system [CL95]. For the rest of this discussion, we will assume that the array uses a chunk size of a single block (4KB). Most arrays use larger chunk sizes (e.g., 64 KB), but for the issues we discuss below, the exact chunk size does not matter; thus we use a single block for the sake of simplicity.

Back To RAID-0 Analysis

Let us now evaluate the capacity, reliability, and performance of striping. From the perspective of capacity, it is perfect: given N disks, striping delivers N disks worth of useful capacity. From the standpoint of reliability, striping is also perfect, but in the bad way: any disk failure will lead to data loss. Finally, performance is excellent: all disks are utilized, often in parallel, to service user I/O requests.

Evaluating RAID Performance

In analyzing RAID performance, one can consider two different performance metrics. The first is *single-request latency*. Understanding the latency of a single I/O request to a RAID is useful as it reveals how much parallelism can exist during a single logical I/O operation. The second is *steady-state throughput* of the RAID, i.e., the total bandwidth of many concurrent requests. Because RAIDs are often used in high-performance environments, the steady-state bandwidth is critical, and thus will be the main focus of our analyses.

To understand throughput in more detail, we need to put forth some workloads of interest. We will assume, for this discussion, that there are two types of workloads: **sequential** and **random**. With a sequential workload, we assume that requests to the array come in large contiguous chunks; for example, a request (or series of requests) that accesses 1 MB of data, starting at block (B) and ending at block (B + 1 MB), would be deemed sequential. Sequential workloads are common in many environments (think of searching through a large file for a keyword), and thus are considered important.

For random workloads, we assume that each request is rather small, and that each request is to a different random location on disk. For example, a random stream of requests may first access 4KB at logical address 10, then at logical address 550,000, then at 20,100, and so forth. Some important workloads, such as transactional workloads on a database management system (DBMS), exhibit this type of access pattern, and thus it is considered an important workload.

Of course, real workloads are not so simple, and often have a mix of sequential and random-seeming components as well as behaviors in-between the two. For simplicity, we just consider these two possibilities.

As you can tell, sequential and random workloads will result in widely different performance characteristics from a disk. With sequential access, a disk operates in its most efficient mode, spending little time seeking and waiting for rotation and most of its time transferring data. With random access, just the opposite is true: most time is spent seeking and waiting for rotation and relatively little time is spent transferring data. To capture this difference in our analysis, we will assume that a disk can transfer data at S MB/s under a sequential workload, and R MB/s when under a random workload. In general, S is much greater than R.

To make sure we understand this difference, let's do a simple exercise. Specifically, let's calculate S and R given the following disk characteristics. Assume a sequential transfer of size 10 MB on average, and a random transfer of 10 KB on average. Also, assume the following disk characteristics:

Average seek time	7 ms
Average rotational delay	3 ms
Transfer rate of disk	50 MB/s

To compute S, we need to first figure out how time is spent in a typical 10 MB transfer. First, we spend 7 ms seeking, and then 3 ms rotating. Finally, transfer begins; 10 MB @ 50 MB/s leads to 1/5th of a second, or 200 ms, spent in transfer. Thus, for each 10 MB request, we spend 210 ms completing the request. To compute S, we just need to divide:

$$S = \frac{Amount\ of\ Data}{Time\ to\ access} = \frac{10\ MB}{210\ ms} = 47.62\ MB/s$$

As we can see, because of the large time spent transferring data, S is very near the peak bandwidth of the disk (the seek and rotational costs have been amortized).

We can compute R similarly. Seek and rotation are the same; we then compute the time spent in transfer, which is 10 KB @ 50 MB/s, or 0.195 ms.

$$R = \frac{Amount\ of\ Data}{Time\ to\ access} = \frac{10\ KB}{10.195\ ms} = 0.981\ MB/s$$

As we can see, R is less than 1 MB/s, and S/R is almost 50.

Back To RAID-0 Analysis, Again

Let's now evaluate the performance of striping. As we said above, it is generally good. From a latency perspective, for example, the latency of a single-block request should be just about identical to that of a single disk; after all, RAID-0 will simply redirect that request to one of its disks.

From the perspective of steady-state throughput, we'd expect to get the full bandwidth of the system. Thus, throughput equals N (the number of disks) multiplied by S (the sequential bandwidth of a single disk). For a large number of random I/Os, we can again use all of the disks, and thus obtain $N \cdot R$ MB/s. As we will see below, these values are both the simplest to calculate and will serve as an upper bound in comparison with other RAID levels.

38.5 RAID Level 1: Mirroring

Our first RAID level beyond striping is known as RAID level 1, or mirroring. With a mirrored system, we simply make more than one copy of each block in the system; each copy should be placed on a separate disk, of course. By doing so, we can tolerate disk failures.

In a typical mirrored system, we will assume that for each logical block, the RAID keeps two physical copies of it. Here is an example:

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Figure 38.3: **Simple RAID-1: Mirroring**

In the example, disk 0 and disk 1 have identical contents, and disk 2 and disk 3 do as well; the data is striped across these mirror pairs. In fact, you may have noticed that there are a number of different ways to place block copies across the disks. The arrangement above is a common one and is sometimes called **RAID-10** or (**RAID 1+0**) because it uses mirrored pairs (RAID-1) and then stripes (RAID-0) on top of them; another common arrangement is **RAID-01** (or **RAID 0+1**), which contains two large striping (RAID-0) arrays, and then mirrors (RAID-1) on top of them. For now, we will just talk about mirroring assuming the above layout.

When reading a block from a mirrored array, the RAID has a choice: it can read either copy. For example, if a read to logical block 5 is issued to the RAID, it is free to read it from either disk 2 or disk 3. When writing a block, though, no such choice exists: the RAID must update *both* copies of the data, in order to preserve reliability. Do note, though, that these writes can take place in parallel; for example, a write to logical block 5 could proceed to disks 2 and 3 at the same time.

RAID-1 Analysis

Let us assess RAID-1. From a capacity standpoint, RAID-1 is expensive; with the mirroring level = 2, we only obtain half of our peak useful capacity. Thus, with N disks, the useful capacity of mirroring is N/2.

From a reliability standpoint, RAID-1 does well. It can tolerate the failure of any one disk. You may also notice RAID-1 can actually do better than this, with a little luck. Imagine, in the figure above, that disk 0 and disk 2 both failed. In such a situation, there is no data loss! More generally, a mirrored system (with mirroring level of 2) can tolerate 1 disk failure for certain, and up to N/2 failures depending on which disks fail. In practice, we generally don't like to leave things like this to chance; thus most people consider mirroring to be good for handling a single failure.

Finally, we analyze performance. From the perspective of the latency of a single read request, we can see it is the same as the latency on a single disk; all the RAID-1 does is direct the read to one of its copies. A write is a little different: it requires two physical writes to complete before it is done. These two writes happen in parallel, and thus the time will be roughly equivalent to the time of a single write; however, because the logical write must wait for both physical writes to complete, it suffers the worst-case seek and rotational delay of the two requests, and thus (on average) will be slightly higher than a write to a single disk.

ASIDE: **THE RAID CONSISTENT-UPDATE PROBLEM**

Before analyzing RAID-1, let us first discuss a problem that arises in any multi-disk RAID system, known as the **consistent-update problem** [DAA05]. The problem occurs on a write to any RAID that has to update multiple disks during a single logical operation. In this case, let us assume we are considering a mirrored disk array.

Imagine the write is issued to the RAID, and then the RAID decides that it must be written to two disks, disk 0 and disk 1. The RAID then issues the write to disk 0, but just before the RAID can issue the request to disk 1, a power loss (or system crash) occurs. In this unfortunate case, let us assume that the request to disk 0 completed (but clearly the request to disk 1 did not, as it was never issued).

The result of this untimely power loss is that the two copies of the block are now **inconsistent**; the copy on disk 0 is the new version, and the copy on disk 1 is the old. What we would like to happen is for the state of both disks to change **atomically**, i.e., either both should end up as the new version or neither.

The general way to solve this problem is to use a **write-ahead log** of some kind to first record what the RAID is about to do (i.e., update two disks with a certain piece of data) before doing it. By taking this approach, we can ensure that in the presence of a crash, the right thing will happen; by running a **recovery** procedure that replays all pending transactions to the RAID, we can ensure that no two mirrored copies (in the RAID-1 case) are out of sync.

One last note: because logging to disk on every write is prohibitively expensive, most RAID hardware includes a small amount of non-volatile RAM (e.g., battery-backed) where it performs this type of logging. Thus, consistent update is provided without the high cost of logging to disk.

To analyze steady-state throughput, let us start with the sequential workload. When writing out to disk sequentially, each logical write must result in two physical writes; for example, when we write logical block 0 (in the figure above), the RAID internally would write it to both disk 0 and disk 1. Thus, we can conclude that the maximum bandwidth obtained during sequential writing to a mirrored array is $(\frac{N}{2} \cdot S)$, or half the peak bandwidth.

Unfortunately, we obtain the exact same performance during a sequential read. One might think that a sequential read could do better, because it only needs to read one copy of the data, not both. However, let's use an example to illustrate why this doesn't help much. Imagine we need to read blocks 0, 1, 2, 3, 4, 5, 6, and 7. Let's say we issue the read of 0 to disk 0, the read of 1 to disk 2, the read of 2 to disk 1, and the read of 3 to disk 3. We continue by issuing reads to 4, 5, 6, and 7 to disks 0, 2, 1, and 3, respectively. One might naively think that because we are utilizing all disks, we are achieving the full bandwidth of the array.

To see that this is not the case, however, consider the requests a single

disk receives (say disk 0). First, it gets a request for block 0; then, it gets a request for block 4 (skipping block 2). In fact, each disk receives a request for every other block. While it is rotating over the skipped block, it is not delivering useful bandwidth to the client. Thus, each disk will only deliver half its peak bandwidth. And thus, the sequential read will only obtain a bandwidth of ($\frac{N}{2} \cdot S$) MB/s.

Random reads are the best case for a mirrored RAID. In this case, we can distribute the reads across all the disks, and thus obtain the full possible bandwidth. Thus, for random reads, RAID-1 delivers $N \cdot R$ MB/s.

Finally, random writes perform as you might expect: $\frac{N}{2} \cdot R$ MB/s. Each logical write must turn into two physical writes, and thus while all the disks will be in use, the client will only perceive this as half the available bandwidth. Even though a write to logical block X turns into two parallel writes to two different physical disks, the bandwidth of many small requests only achieves half of what we saw with striping. As we will soon see, getting half the available bandwidth is actually pretty good!

38.6 RAID Level 4: Saving Space With Parity

We now present a different method of adding redundancy to a disk array known as **parity**. Parity-based approaches attempt to use less capacity and thus overcome the huge space penalty paid by mirrored systems. They do so at a cost, however: performance.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.4: **RAID-4 with Parity**

Here is an example five-disk RAID-4 system (Figure 38.4). For each stripe of data, we have added a single **parity** block that stores the redundant information for that stripe of blocks. For example, parity block P1 has redundant information that it calculated from blocks 4, 5, 6, and 7.

To compute parity, we need to use a mathematical function that enables us to withstand the loss of any one block from our stripe. It turns out the simple function **XOR** does the trick quite nicely. For a given set of bits, the XOR of all of those bits returns a 0 if there are an even number of 1's in the bits, and a 1 if there are an odd number of 1's. For example:

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

In the first row (0,0,1,1), there are two 1's (C2, C3), and thus XOR of all of those values will be 0 (P); similarly, in the second row there is only one 1 (C1), and thus the XOR must be 1 (P). You can remember this in a simple way: that the number of 1s in any row must be an even (not odd) number; that is the **invariant** that the RAID must maintain in order for parity to be correct.

From the example above, you might also be able to guess how parity information can be used to recover from a failure. Imagine the column labeled C2 is lost. To figure out what values must have been in the column, we simply have to read in all the other values in that row (including the XOR'd parity bit) and **reconstruct** the right answer. Specifically, assume the first row's value in column C2 is lost (it is a 1); by reading the other values in that row (0 from C0, 0 from C1, 1 from C3, and 0 from the parity column P), we get the values 0, 0, 1, and 0. Because we know that XOR keeps an even number of 1's in each row, we know what the missing data must be: a 1. And that is how reconstruction works in a XOR-based parity scheme! Note also how we compute the reconstructed value: we just XOR the data bits and the parity bits together, in the same way that we calculated the parity in the first place.

Now you might be wondering: we are talking about XORing all of these bits, and yet above we know that the RAID places 4KB (or larger) blocks on each disk; how do we apply XOR to a bunch of blocks to compute the parity? It turns out this is easy as well. Simply perform a bitwise XOR across each bit of the data blocks; put the result of each bitwise XOR into the corresponding bit slot in the parity block. For example, if we had blocks of size 4 bits (yes, this is still quite a bit smaller than a 4KB block, but you get the picture), they might look something like this:

Block0	Block1	Block2	Block3	Parity
00	10	11	10	11
10	01	00	01	10

As you can see from the figure, the parity is computed for each bit of each block and the result placed in the parity block.

RAID-4 Analysis

Let us now analyze RAID-4. From a capacity standpoint, RAID-4 uses 1 disk for parity information for every group of disks it is protecting. Thus, our useful capacity for a RAID group is (N-1).

Reliability is also quite easy to understand: RAID-4 tolerates 1 disk failure and no more. If more than one disk is lost, there is simply no way to reconstruct the lost data.

Finally, there is performance. This time, let us start by analyzing steady-state throughput. Sequential read performance can utilize all of the disks except for the parity disk, and thus deliver a peak effective bandwidth of $(N-1) \cdot S$ MB/s (an easy case).

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.5: **Full-stripe Writes In RAID-4**

To understand the performance of sequential writes, we must first understand how they are done. When writing a big chunk of data to disk, RAID-4 can perform a simple optimization known as a **full-stripe write**. For example, imagine the case where the blocks 0, 1, 2, and 3 have been sent to the RAID as part of a write request (Figure 38.5).

In this case, the RAID can simply calculate the new value of P0 (by performing an XOR across the blocks 0, 1, 2, and 3) and then write all of the blocks (including the parity block) to the five disks above in parallel (highlighted in gray in the figure). Thus, full-stripe writes are the most efficient way for RAID-4 to write to disk.

Once we understand the full-stripe write, calculating the performance of sequential writes on RAID-4 is easy; the effective bandwidth is also $(N-1)\cdot S$ MB/s. Even though the parity disk is constantly in use during the operation, the client does not gain performance advantage from it.

Now let us analyze the performance of random reads. As you can also see from the figure above, a set of 1-block random reads will be spread across the data disks of the system but not the parity disk. Thus, the effective performance is: $(N-1)\cdot R$ MB/s.

Random writes, which we have saved for last, present the most interesting case for RAID-4. Imagine we wish to overwrite block 1 in the example above. We could just go ahead and overwrite it, but that would leave us with a problem: the parity block P0 would no longer accurately reflect the correct parity value of the stripe; in this example, P0 must also be updated. How can we update it both correctly and efficiently?

It turns out there are two methods. The first, known as **additive parity**, requires us to do the following. To compute the value of the new parity block, read in all of the other data blocks in the stripe in parallel (in the example, blocks 0, 2, and 3) and XOR those with the new block (1). The result is your new parity block. To complete the write, you can then write the new data and new parity to their respective disks, also in parallel.

The problem with this technique is that it scales with the number of disks, and thus in larger RAIDs requires a high number of reads to compute parity. Thus, the **subtractive parity** method.

For example, imagine this string of bits (4 data bits, one parity):

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0

Let's imagine that we wish to overwrite bit C2 with a new value which we will call $C2_{new}$. The subtractive method works in three steps. First, we read in the old data at C2 ($C2_{old}$ = 1) and the old parity (P_{old} = 0).

Then, we compare the old data and the new data; if they are the same (e.g., $C2_{new}$ = $C2_{old}$), then we know the parity bit will also remain the same (i.e., P_{new} = P_{old}). If, however, they are different, then we must flip the old parity bit to the opposite of its current state, that is, if (P_{old} == 1), P_{new} will be set to 0; if (P_{old} == 0), P_{new} will be set to 1. We can express this whole mess neatly with XOR (where $\oplus$ is the XOR operator):

$$P_{new} = (C_{old} \oplus C_{new}) \oplus P_{old} \tag{38.1}$$

Because we are dealing with blocks, not bits, we perform this calculation over all the bits in the block (e.g., 4096 bytes in each block multiplied by 8 bits per byte). Thus, in most cases, the new block will be different than the old block and thus the new parity block will too.

You should now be able to figure out when we would use the additive parity calculation and when we would use the subtractive method. Think about how many disks would need to be in the system so that the additive method performs fewer I/Os than the subtractive method; what is the cross-over point?

For this performance analysis, let us assume we are using the subtractive method. Thus, for each write, the RAID has to perform 4 physical I/Os (two reads and two writes). Now imagine there are lots of writes submitted to the RAID; how many can RAID-4 perform in parallel? To understand, let us again look at the RAID-4 layout (Figure 38.6).

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

Figure 38.6: **Example: Writes To 4, 13, And Respective Parity Blocks**

Now imagine there were 2 small writes submitted to the RAID-4 at about the same time, to blocks 4 and 13 (marked with * in the diagram). The data for those disks is on disks 0 and 1, and thus the read and write to data could happen in parallel, which is good. The problem that arises is with the parity disk; both the requests have to read the related parity blocks for 4 and 13, parity blocks 1 and 3 (marked with +). Hopefully, the issue is now clear: the parity disk is a bottleneck under this type of workload; we sometimes thus call this the **small-write problem** for parity-based RAIDs. Thus, even though the data disks could be accessed in parallel, the parity disk prevents any parallelism from materializing; all writes to the system will be serialized because of the parity disk. Because the parity disk has to perform two I/Os (one read, one write) per logical I/O, we can compute the performance of small random writes in RAID-4 by computing the parity disk's performance on those two I/Os, and thus we achieve (R/2) MB/s. RAID-4 throughput under random small writes is terrible; it does not improve as you add disks to the system.

We conclude by analyzing I/O latency in RAID-4. As you now know, a single read (assuming no failure) is just mapped to a single disk, and thus its latency is equivalent to the latency of a single disk request. The latency of a single write requires two reads and then two writes; the reads can happen in parallel, as can the writes, and thus total latency is about twice that of a single disk (with some differences because we have to wait for both reads to complete and thus get the worst-case positioning time, but then the updates don't incur seek cost and thus may be a better-than-average positioning cost).

38.7 RAID Level 5: Rotating Parity

To address the small-write problem (at least, partially), Patterson, Gibson, and Katz introduced RAID-5. RAID-5 works almost identically to RAID-4, except that it **rotates** the parity block across drives (Figure 38.7).

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Figure 38.7: **RAID-5 With Rotated Parity**

As you can see, the parity block for each stripe is now rotated across the disks, in order to remove the parity-disk bottleneck for RAID-4.

RAID-5 Analysis

Much of the analysis for RAID-5 is identical to RAID-4. For example, the effective capacity and failure tolerance of the two levels are identical. So are sequential read and write performance. The latency of a single request (whether a read or a write) is also the same as RAID-4.

Random read performance is a little better, because we can utilize all of the disks. Finally, random write performance improves noticeably over RAID-4, as it allows for parallelism across requests. Imagine a write to block 1 and a write to block 10; this will turn into requests to disk 1 and disk 4 (for block 1 and its parity) and requests to disk 0 and disk 2 (for block 10 and its parity). Thus, they can proceed in parallel. In fact, we can generally assume that given a large number of random requests, we will be able to keep all the disks about evenly busy. If that is the case, then our total bandwidth for small writes will be $\frac{N}{4} \cdot R$ MB/s. The factor of four loss is due to the fact that each RAID-5 write still generates 4 total I/O operations, which is simply the cost of using parity-based RAID.

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	N	$N/2$	$N-1$	$N-1$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Latency				
Read	T	T	T	T
Write	T	T	$2T$	$2T$

Figure 38.8: **RAID Capacity, Reliability, and Performance**

Because RAID-5 is basically identical to RAID-4 except in the few cases where it is better, it has almost completely replaced RAID-4 in the marketplace. The only place where it has not is in systems that know they will never perform anything other than a large write, thus avoiding the small-write problem altogether [HLM94]; in those cases, RAID-4 is sometimes used as it is slightly simpler to build.

38.8 RAID Comparison: A Summary

We now summarize our simplified comparison of RAID levels in Figure 38.8. Note that we have omitted a number of details to simplify our analysis. For example, when writing in a mirrored system, the average seek time is a little higher than when writing to just a single disk, because the seek time is the max of two seeks (one on each disk). Thus, random write performance to two disks will generally be a little less than random write performance of a single disk. Also, when updating the parity disk in RAID-4/5, the first read of the old parity will likely cause a full seek and rotation, but the second write of the parity will only result in rotation.

However, the comparison in Figure 38.8 does capture the essential differences, and is useful for understanding tradeoffs across RAID levels. For the latency analysis, we simply use T to represent the time that a request to a single disk would take.

To conclude, if you strictly want performance and do not care about reliability, striping is obviously best. If, however, you want random I/O performance and reliability, mirroring is the best; the cost you pay is in lost capacity. If capacity and reliability are your main goals, then RAID-5 is the winner; the cost you pay is in small-write performance. Finally, if you are always doing sequential I/O and want to maximize capacity, RAID-5 also makes the most sense.

38.9 Other Interesting RAID Issues

There are a number of other interesting ideas that one could (and perhaps should) discuss when thinking about RAID. Here are some things we might eventually write about.

For example, there are many other RAID designs, including Levels 2 and 3 from the original taxonomy, and Level 6 to tolerate multiple disk faults [C+04]. There is also what the RAID does when a disk fails; sometimes it has a **hot spare** sitting around to fill in for the failed disk. What happens to performance under failure, and performance during reconstruction of the failed disk? There are also more realistic fault models, to take into account **latent sector errors** or **block corruption** [B+08], and lots of techniques to handle such faults (see the data integrity chapter for details). Finally, you can even build RAID as a software layer: such **software RAID** systems are cheaper but have other problems, including the consistent-update problem [DAA05].

38.10 Summary

We have discussed RAID. RAID transforms a number of independent disks into a large, more capacious, and more reliable single entity; importantly, it does so transparently, and thus hardware and software above is relatively oblivious to the change.

There are many possible RAID levels to choose from, and the exact RAID level to use depends heavily on what is important to the end-user. For example, mirrored RAID is simple, reliable, and generally provides good performance but at a high capacity cost. RAID-5, in contrast, is reliable and better from a capacity standpoint, but performs quite poorly when there are small writes in the workload. Picking a RAID and setting its parameters (chunk size, number of disks, etc.) properly for a particular workload is challenging, and remains more of an art than a science.

References

[B+08] "An Analysis of Data Corruption in the Storage Stack"
Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '08, San Jose, CA, February 2008
Our own work analyzing how often disks actually corrupt your data. Not often, but sometimes! And thus something a reliable storage system must consider.

[BJ88] "Disk Shadowing"
D. Bitton and J. Gray
VLDB 1988
One of the first papers to discuss mirroring, herein called "shadowing".

[CL95] "Striping in a RAID level 5 disk array"
Peter M. Chen, Edward K. Lee
SIGMETRICS 1995
A nice analysis of some of the important parameters in a RAID-5 disk array.

[C+04] "Row-Diagonal Parity for Double Disk Failure Correction"
P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, S. Sankar
FAST '04, February 2004
Though not the first paper on a RAID system with two disks for parity, it is a recent and highly-understandable version of said idea. Read it to learn more.

[DAA05] "Journal-guided Resynchronization for Software RAID"
Timothy E. Denehy, A. Arpaci-Dusseau, R. Arpaci-Dusseau
FAST 2005
Our own work on the consistent-update problem. Here we solve it for Software RAID by integrating the journaling machinery of the file system above with the software RAID beneath it.

[HLM94] "File System Design for an NFS File Server Appliance"
Dave Hitz, James Lau, Michael Malcolm
USENIX Winter 1994, San Francisco, California, 1994
The sparse paper introducing a landmark product in storage, the write-anywhere file layout or WAFL file system that underlies the NetApp file server.

[K86] "Synchronized Disk Interleaving"
M.Y. Kim.
IEEE Transactions on Computers, Volume C-35: 11, November 1986
Some of the earliest work on RAID is found here.

[K88] "Small Disk Arrays - The Emerging Approach to High Performance"
F. Kurzweil.
Presentation at Spring COMPCON '88, March 1, 1988, San Francisco, California
Another early RAID reference.

[P+88] "Redundant Arrays of Inexpensive Disks"
D. Patterson, G. Gibson, R. Katz.
SIGMOD 1988
*This is considered **the** RAID paper, written by famous authors Patterson, Gibson, and Katz. The paper has since won many test-of-time awards and ushered in the RAID era, including the name RAID itself!*

[PB86] "Providing Fault Tolerance in Parallel Secondary Storage Systems"
A. Park and K. Balasubramaniam
Department of Computer Science, Princeton, CS-TR-O57-86, November 1986
Another early work on RAID.

[SG86] "Disk Striping"
K. Salem and H. Garcia-Molina.
IEEE International Conference on Data Engineering, 1986
And yes, another early RAID work. There are a lot of these, which kind of came out of the woodwork when the RAID paper was published in SIGMOD.

[S84] "Byzantine Generals in Action: Implementing Fail-Stop Processors"
F.B. Schneider.
ACM Transactions on Computer Systems, 2(2):145154, May 1984
Finally, a paper that is not about RAID! This paper is actually about how systems fail, and how to make something behave in a fail-stop manner.

Homework

This section introduces `raid.py`, a simple RAID simulator you can use to shore up your knowledge of how RAID systems work. See the README for details.

Questions

1. Use the simulator to perform some basic RAID mapping tests. Run with different levels (0, 1, 4, 5) and see if you can figure out the mappings of a set of requests. For RAID-5, see if you can figure out the difference between left-symmetric and left-asymmetric layouts. Use some different random seeds to generate different problems than above.
2. Do the same as the first problem, but this time vary the chunk size with `-C`. How does chunk size change the mappings?
3. Do the same as above, but use the `-r` flag to reverse the nature of each problem.
4. Now use the reverse flag but increase the size of each request with the `-S` flag. Try specifying sizes of 8k, 12k, and 16k, while varying the RAID level. What happens to the underlying I/O pattern when the size of the request increases? Make sure to try this with the sequential workload too (`-W sequential`); for what request sizes are RAID-4 and RAID-5 much more I/O efficient?
5. Use the timing mode of the simulator (`-t`) to estimate the performance of 100 random reads to the RAID, while varying the RAID levels, using 4 disks.
6. Do the same as above, but increase the number of disks. How does the performance of each RAID level scale as the number of disks increases?
7. Do the same as above, but use all writes (`-w 100`) instead of reads. How does the performance of each RAID level scale now? Can you do a rough estimate of the time it will take to complete the workload of 100 random writes?
8. Run the timing mode one last time, but this time with a sequential workload (`-W sequential`). How does the performance vary with RAID level, and when doing reads versus writes? How about when varying the size of each request? What size should you write to a RAID when using RAID-4 or RAID-5?

36

I/O Devices

Before delving into the main content of this part of the book (on persistence), we first introduce the concept of an **input/output (I/O) device** and show how the operating system might interact with such an entity. I/O is quite critical to computer systems, of course; imagine a program without any input (it produces the same result each time); now imagine a program with no output (what was the purpose of it running?). Clearly, for computer systems to be interesting, both input and output are required. And thus, our general problem:

CRUX: HOW TO INTEGRATE I/O INTO SYSTEMS
How should I/O be integrated into systems? What are the general mechanisms? How can we make them efficient?

36.1 System Architecture

To begin our discussion, let's look at the structure of a typical system (Figure 36.1). The picture shows a single CPU attached to the main memory of the system via some kind of **memory bus** or interconnect. Some devices are connected to the system via a general **I/O bus**, which in many modern systems would be **PCI** (or one of its many derivatives); graphics and some other higher-performance I/O devices might be found here. Finally, even lower down are one or more of what we call a **peripheral bus**, such as **SCSI**, **SATA**, or **USB**. These connect the slowest devices to the system, including **disks**, **mice**, and other similar components.

One question you might ask is: why do we need a hierarchical structure like this? Put simply: physics, and cost. The faster a bus is, the shorter it must be; thus, a high-performance memory bus does not have much room to plug devices and such into it. In addition, engineering a bus for high performance is quite costly. Thus, system designers have adopted this hierarchical approach, where components that demand high performance (such as the graphics card) are nearer the CPU. Lower per-

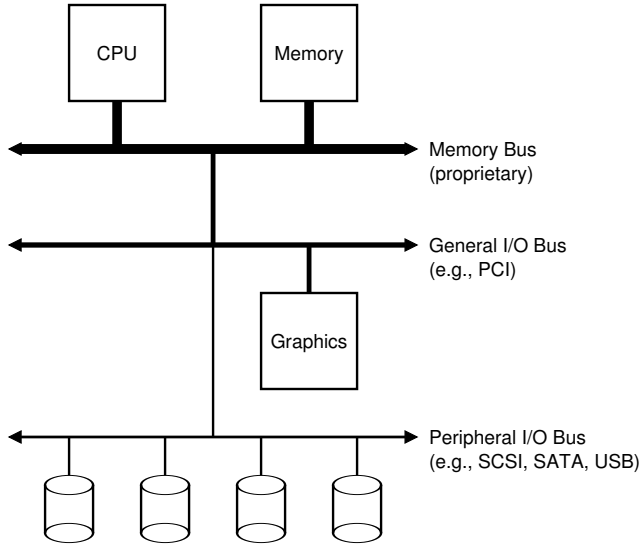


Figure 36.1: **Prototypical System Architecture**

formance components are further away. The benefits of placing disks and other slow devices on a peripheral bus are manifold; in particular, you can place a large number of devices on it.

36.2 A Canonical Device

Let us now look at a canonical device (not a real one), and use this device to drive our understanding of some of the machinery required to make device interaction efficient. From Figure 36.2, we can see that a device has two important components. The first is the hardware **interface** it presents to the rest of the system. Just like a piece of software, hardware must also present some kind of interface that allows the system software to control its operation. Thus, all devices have some specified interface and protocol for typical interaction.

The second part of any device is its **internal structure**. This part of the device is implementation specific and is responsible for implementing the abstraction the device presents to the system. Very simple devices will have one or a few hardware chips to implement their functionality; more complex devices will include a simple CPU, some general purpose memory, and other device-specific chips to get their job done. For example, modern RAID controllers might consist of hundreds of thousands of lines of **firmware** (i.e., software within a hardware device) to implement its functionality.

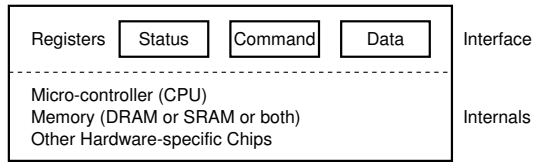


Figure 36.2: **A Canonical Device**

36.3 The Canonical Protocol

In the picture above, the (simplified) device interface is comprised of three registers: a **status** register, which can be read to see the current status of the device; a **command** register, to tell the device to perform a certain task; and a **data** register to pass data to the device, or get data from the device. By reading and writing these registers, the operating system can control device behavior.

Let us now describe a typical interaction that the OS might have with the device in order to get the device to do something on its behalf. The protocol is as follows:

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (Doing so starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

The protocol has four steps. In the first, the OS waits until the device is ready to receive a command by repeatedly reading the status register; we call this **polling** the device (basically, just asking it what is going on). Second, the OS sends some data down to the data register; one can imagine that if this were a disk, for example, that multiple writes would need to take place to transfer a disk block (say 4KB) to the device. When the main CPU is involved with the data movement (as in this example protocol), we refer to it as **programmed I/O (PIO)**. Third, the OS writes a command to the command register; doing so implicitly lets the device know that both the data is present and that it should begin working on the command. Finally, the OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished (it may then get an error code to indicate success or failure).

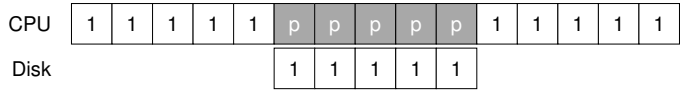
This basic protocol has the positive aspect of being simple and working. However, there are some inefficiencies and inconveniences involved. The first problem you might notice in the protocol is that polling seems inefficient; specifically, it wastes a great deal of CPU time just waiting for the (potentially slow) device to complete its activity, instead of switching to another ready process and thus better utilizing the CPU.

THE CRUX: HOW TO AVOID THE COSTS OF POLLING
How can the OS check device status without frequent polling, and thus lower the CPU overhead required to manage the device?

36.4 Lowering CPU Overhead With Interrupts

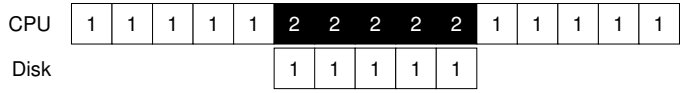
The invention that many engineers came upon years ago to improve this interaction is something we've seen already: the **interrupt**. Instead of polling the device repeatedly, the OS can issue a request, put the calling process to sleep, and context switch to another task. When the device is finally finished with the operation, it will raise a hardware interrupt, causing the CPU to jump into the OS at a pre-determined **interrupt service routine (ISR)** or more simply an **interrupt handler**. The handler is just a piece of operating system code that will finish the request (for example, by reading data and perhaps an error code from the device) and wake the process waiting for the I/O, which can then proceed as desired.

Interrupts thus allow for **overlap** of computation and I/O, which is key for improved utilization. This timeline shows the problem:



In the diagram, Process 1 runs on the CPU for some time (indicated by a repeated `1` on the CPU line), and then issues an I/O request to the disk to read some data. Without interrupts, the system simply spins, polling the status of the device repeatedly until the I/O is complete (indicated by a `p`). The disk services the request and finally Process 1 can run again.

If instead we utilize interrupts and allow for overlap, the OS can do something else while waiting for the disk:



In this example, the OS runs Process 2 on the CPU while the disk services Process 1's request. When the disk request is finished, an interrupt occurs, and the OS wakes up Process 1 and runs it again. Thus, *both* the CPU and the disk are properly utilized during the middle stretch of time.

Note that using interrupts is not *always* the best solution. For example, imagine a device that performs its tasks very quickly: the first poll usually finds the device to be done with task. Using an interrupt in this case will actually *slow down* the system: switching to another process, handling the interrupt, and switching back to the issuing process is expensive. Thus, if a device is fast, it may be best to poll; if it is slow, interrupts, which allow

TIP: INTERRUPTS NOT ALWAYS BETTER THAN PIO
Although interrupts allow for overlap of computation and I/O, they only really make sense for slow devices. Otherwise, the cost of interrupt handling and context switching may outweigh the benefits interrupts provide. There are also cases where a flood of interrupts may overload a system and lead it to livelock [MR96]; in such cases, polling provides more control to the OS in its scheduling and thus is again useful.

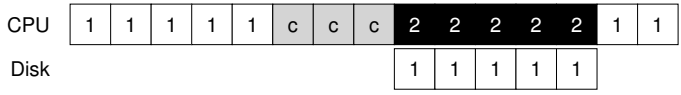
overlap, are best. If the speed of the device is not known, or sometimes fast and sometimes slow, it may be best to use a **hybrid** that polls for a little while and then, if the device is not yet finished, uses interrupts. This **two-phased** approach may achieve the best of both worlds.

Another reason not to use interrupts arises in networks [MR96]. When a huge stream of incoming packets each generate an interrupt, it is possible for the OS to **livelock**, that is, find itself only processing interrupts and never allowing a user-level process to run and actually service the requests. For example, imagine a web server that suddenly experiences a high load due to the "slashdot effect". In this case, it is better to occasionally use polling to better control what is happening in the system and allow the web server to service some requests before going back to the device to check for more packet arrivals.

Another interrupt-based optimization is **coalescing**. In such a setup, a device which needs to raise an interrupt first waits for a bit before delivering the interrupt to the CPU. While waiting, other requests may soon complete, and thus multiple interrupts can be coalesced into a single interrupt delivery, thus lowering the overhead of interrupt processing. Of course, waiting too long will increase the latency of a request, a common trade-off in systems. See Ahmad et al. [A+11] for an excellent summary.

36.5 More Efficient Data Movement With DMA

Unfortunately, there is one other aspect of our canonical protocol that requires our attention. In particular, when using programmed I/O (PIO) to transfer a large chunk of data to a device, the CPU is once again overburdened with a rather trivial task, and thus wastes a lot of time and effort that could better be spent running other processes. This timeline illustrates the problem:

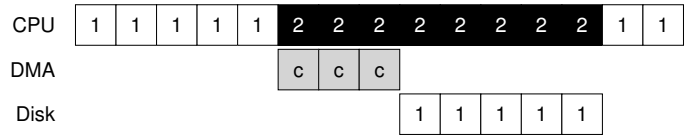


In the timeline, Process 1 is running and then wishes to write some data to the disk. It then initiates the I/O, which must copy the data from memory to the device explicitly, one word at a time (marked `c` in the diagram). When the copy is complete, the I/O begins on the disk and the CPU can finally be used for something else.

THE CRUX: HOW TO LOWER PIO OVERHEADS
With PIO, the CPU spends too much time moving data to and from devices by hand. How can we offload this work and thus allow the CPU to be more effectively utilized?

The solution to this problem is something we refer to as **Direct Memory Access (DMA)**. A DMA engine is essentially a very specific device within a system that can orchestrate transfers between devices and main memory without much CPU intervention.

DMA works as follows. To transfer data to the device, for example, the OS would program the DMA engine by telling it where the data lives in memory, how much data to copy, and which device to send it to. At that point, the OS is done with the transfer and can proceed with other work. When the DMA is complete, the DMA controller raises an interrupt, and the OS thus knows the transfer is complete. The revised timeline:



From the timeline, you can see that the copying of data is now handled by the DMA controller. Because the CPU is free during that time, the OS can do something else, here choosing to run Process 2. Process 2 thus gets to use more CPU before Process 1 runs again.

36.6 Methods Of Device Interaction

Now that we have some sense of the efficiency issues involved with performing I/O, there are a few other problems we need to handle to incorporate devices into modern systems. One problem you may have noticed thus far: we have not really said anything about how the OS actually communicates with the device! Thus, the problem:

THE CRUX: HOW TO COMMUNICATE WITH DEVICES
How should the hardware communicate with a device? Should there be explicit instructions? Or are there other ways to do it?

Over time, two primary methods of device communication have developed. The first, oldest method (used by IBM mainframes for many years) is to have explicit **I/O instructions**. These instructions specify a way for the OS to send data to specific device registers and thus allow the construction of the protocols described above.

For example, on x86, the `in` and `out` instructions can be used to communicate with devices. For example, to send data to a device, the caller specifies a register with the data in it, and a specific *port* which names the device. Executing the instruction leads to the desired behavior.

Such instructions are usually **privileged**. The OS controls devices, and the OS thus is the only entity allowed to directly communicate with them. Imagine if any program could read or write the disk, for example: total chaos (as always), as any user program could use such a loophole to gain complete control over the machine.

The second method to interact with devices is known as **memory-mapped I/O**. With this approach, the hardware makes device registers available as if they were memory locations. To access a particular register, the OS issues a load (to read) or store (to write) the address; the hardware then routes the load/store to the device instead of main memory.

There is not some great advantage to one approach or the other. The memory-mapped approach is nice in that no new instructions are needed to support it, but both approaches are still in use today.

36.7 Fitting Into The OS: The Device Driver

One final problem we will discuss: how to fit devices, each of which have very specific interfaces, into the OS, which we would like to keep as general as possible. For example, consider a file system. We'd like to build a file system that worked on top of SCSI disks, IDE disks, USB keychain drives, and so forth, and we'd like the file system to be relatively oblivious to all of the details of how to issue a read or write request to these difference types of drives. Thus, our problem:

THE CRUX: HOW TO BUILD A DEVICE-NEUTRAL OS
How can we keep most of the OS device-neutral, thus hiding the details of device interactions from major OS subsystems?

The problem is solved through the age-old technique of **abstraction**. At the lowest level, a piece of software in the OS must know in detail how a device works. We call this piece of software a **device driver**, and any specifics of device interaction are encapsulated within.

Let us see how this abstraction might help OS design and implementation by examining the Linux file system software stack. Figure 36.3 is a rough and approximate depiction of the Linux software organization. As you can see from the diagram, a file system (and certainly, an application above) is completely oblivious to the specifics of which disk class it is using; it simply issues block read and write requests to the generic block layer, which routes them to the appropriate device driver, which handles the details of issuing the specific request. Although simplified, the diagram shows how such detail can be hidden from most of the OS.

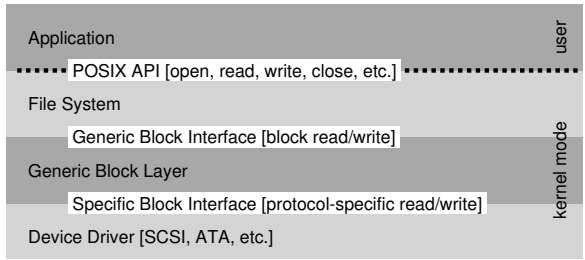


Figure 36.3: **The File System Stack**

Note that such encapsulation can have its downside as well. For example, if there is a device that has many special capabilities, but has to present a generic interface to the rest of the kernel, those special capabilities will go unused. This situation arises, for example, in Linux with SCSI devices, which have very rich error reporting; because other block devices (e.g., ATA/IDE) have much simpler error handling, all that higher levels of software ever receive is a generic `EIO` (generic IO error) error code; any extra detail that SCSI may have provided is thus lost to the file system [G08].

Interestingly, because device drivers are needed for any device you might plug into your system, over time they have come to represent a huge percentage of kernel code. Studies of the Linux kernel reveal that over 70% of OS code is found in device drivers [C01]; for Windows-based systems, it is likely quite high as well. Thus, when people tell you that the OS has millions of lines of code, what they are really saying is that the OS has millions of lines of device-driver code. Of course, for any given installation, most of that code may not be active (i.e., only a few devices are connected to the system at a time). Perhaps more depressingly, as drivers are often written by "amateurs" (instead of full-time kernel developers), they tend to have many more bugs and thus are a primary contributor to kernel crashes [S03].

36.8 Case Study: A Simple IDE Disk Driver

To dig a little deeper here, let's take a quick look at an actual device: an IDE disk drive [L94]. We summarize the protocol as described in this reference [W10]; we'll also peek at the xv6 source code for a simple example of a working IDE driver [CK+08].

An IDE disk presents a simple interface to the system, consisting of four types of register: control, command block, status, and error. These registers are available by reading or writing to specific "I/O addresses" (such as `0x3F6` below) using (on x86) the `in` and `out` I/O instructions.

```
Control Register:
  Address 0x3F6 = 0x80 (0000 1RE0): R=reset, E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6    5      4    3    2    1     0
   BUSY  READY FAULT SEEK  DRQ  CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when Status ERROR==1)
    7      6    5      4    3    2    1    0
   BBK    UNC   MC   IDNF  MCR  ABRT T0NF AMNF

   BBK  = Bad Block
   UNC  = Uncorrectable data error
   MC   = Media Changed
   IDNF = ID mark Not Found
   MCR  = Media Change Requested
   ABRT = Command aborted
   T0NF = Track 0 Not Found
   AMNF = Address Mark Not Found
```

Figure 36.4: **The IDE Interface**

The basic protocol to interact with the device is as follows, assuming it has already been initialized.

- **Wait for drive to be ready.** Read Status Register (0x1F7) until drive is not busy and READY.
- **Write parameters to command registers.** Write the sector count, logical block address (LBA) of the sectors to be accessed, and drive number (master=0x00 or slave=0x10, as IDE permits just two drives) to command registers (0x1F2-0x1F6).
- **Start the I/O.** by issuing read/write to command register. Write READ—WRITE command to command register (0x1F7).
- **Data transfer (for writes):** Wait until drive status is READY and DRQ (drive request for data); write data to data port.
- **Handle interrupts.** In the simplest case, handle an interrupt for each sector transferred; more complex approaches allow batching and thus one final interrupt when the entire transfer is complete.
- **Error handling.** After each operation, read the status register. If the ERROR bit is on, read the error register for details.

Most of this protocol is found in the xv6 IDE driver (Figure 36.5), which (after initialization) works through four primary functions. The first is `ide_rw()`, which queues a request (if there are others pending), or issues it directly to the disk (via `ide_start_request()`); in either

```
static int ide_wait_ready() {
  while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
    ;                               // loop until drive isn't busy
}

static void ide_start_request(struct buf *b) {
  ide_wait_ready();
  outb(0x3f6, 0);                   // generate interrupt
  outb(0x1f2, 1);                   // how many sectors?
  outb(0x1f3, b->sector & 0xff);    // LBA goes here ...
  outb(0x1f4, (b->sector >> 8) & 0xff);   // ... and here
  outb(0x1f5, (b->sector >> 16) & 0xff);  // ... and here!
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);     // this is a WRITE
    outsl(0x1f0, b->data, 512/4);   // transfer data too!
  } else {
    outb(0x1f7, IDE_CMD_READ);      // this is a READ (no data)
  }
}

void ide_rw(struct buf *b) {
  acquire(&ide_lock);
  for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
    ;                               // walk queue
  *pp = b;                          // add request to end
  if (ide_queue == b)               // if q is empty
    ide_start_request(b);           // send req to disk
  while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
    sleep(b, &ide_lock);            // wait for completion
  release(&ide_lock);
}

void ide_intr() {
  struct buf *b;
  acquire(&ide_lock);
  if (!(b->flags & B_DIRTY) && ide_wait_ready() >= 0)
    insl(0x1f0, b->data, 512/4);    // if READ: get data
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);                        // wake waiting process
  if ((ide_queue = b->qnext) != 0) // start next request
    ide_start_request(ide_queue);  // (if one exists)
  release(&ide_lock);
}
```

Figure 36.5: **The xv6 IDE Disk Driver (Simplified)**

case, the routine waits for the request to complete and the calling process is put to sleep. The second is `ide_start_request()`, which is used to send a request (and perhaps data, in the case of a write) to the disk; the `in` and `out` x86 instructions are called to read and write device registers, respectively. The start request routine uses the third function, `ide_wait_ready()`, to ensure the drive is ready before issuing a request to it. Finally, `ide_intr()` is invoked when an interrupt takes place; it reads data from the device (if the request is a read, not a write), wakes the process waiting for the I/O to complete, and (if there are more requests in the I/O queue), launches the next I/O via `ide_start_request()`.

36.9 Historical Notes

Before ending, we include a brief historical note on the origin of some of these fundamental ideas. If you are interested in learning more, read Smotherman's excellent summary [S08].

Interrupts are an ancient idea, existing on the earliest of machines. For example, the UNIVAC in the early 1950's had some form of interrupt vectoring, although it is unclear in exactly which year this feature was available [S08]. Sadly, even in its infancy, we are beginning to lose the origins of computing history.

There is also some debate as to which machine first introduced the idea of DMA. For example, Knuth and others point to the DYSEAC (a "mobile" machine, which at the time meant it could be hauled in a trailer), whereas others think the IBM SAGE may have been the first [S08]. Either way, by the mid 50's, systems with I/O devices that communicated directly with memory and interrupted the CPU when finished existed.

The history here is difficult to trace because the inventions are tied to real, and sometimes obscure, machines. For example, some think that the Lincoln Labs TX-2 machine was first with vectored interrupts [S08], but this is hardly clear.

Because the ideas are relatively obvious — no Einsteinian leap is required to come up with the idea of letting the CPU do something else while a slow I/O is pending — perhaps our focus on "who first?" is misguided. What is certainly clear: as people built these early machines, it became obvious that I/O support was needed. Interrupts, DMA, and related ideas are all direct outcomes of the nature of fast CPUs and slow devices; if you were there at the time, you might have had similar ideas.

36.10 Summary

You should now have a very basic understanding of how an OS interacts with a device. Two techniques, the interrupt and DMA, have been introduced to help with device efficiency, and two approaches to accessing device registers, explicit I/O instructions and memory-mapped I/O, have been described. Finally, the notion of a device driver has been presented, showing how the OS itself can encapsulate low-level details and thus make it easier to build the rest of the OS in a device-neutral fashion.

References

[A+11] "vIC: Interrupt Coalescing for Virtual Machine Storage Device IO"
Irfan Ahmad, Ajay Gulati, Ali Mashtizadeh
USENIX '11
A terrific survey of interrupt coalescing in traditional and virtualized environments.

[C01] "An Empirical Study of Operating System Errors"
Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, Dawson Engler
SOSP '01
One of the first papers to systematically explore how many bugs are in modern operating systems. Among other neat findings, the authors show that device drivers have something like seven times more bugs than mainline kernel code.

[CK+08] "The xv6 Operating System"
Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich
From: http://pdos.csail.mit.edu/6.828/2008/index.html
See `ide.c` for the IDE device driver, with a few more details therein.

[D07] "What Every Programmer Should Know About Memory"
Ulrich Drepper
November, 2007
Available: http://www.akkadia.org/drepper/cpumemory.pdf
A fantastic read about modern memory systems, starting at DRAM and going all the way up to virtualization and cache-optimized algorithms.

[G08] "EIO: Error-handling is Occasionally Correct"
Haryadi Gunawi, Cindy Rubio-Gonzalez, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Ben Liblit
FAST '08, San Jose, CA, February 2008
Our own work on building a tool to find code in Linux file systems that does not handle error return properly. We found hundreds and hundreds of bugs, many of which have now been fixed.

[L94] "AT Attachment Interface for Disk Drives"
Lawrence J. Lamers, X3T10 Technical Editor
Available: ftp://ftp.t10.org/t13/project/d0791r4c-ATA-1.pdf
Reference number: ANSI X3.221 - 1994 *A rather dry document about device interfaces. Read it at your own peril.*

[MR96] "Eliminating Receive Livelock in an Interrupt-driven Kernel"
Jeffrey Mogul and K. K. Ramakrishnan
USENIX '96, San Diego, CA, January 1996
Mogul and colleagues did a great deal of pioneering work on web server network performance. This paper is but one example.

[S08] "Interrupts"
Mark Smotherman, as of July '08
Available: http://people.cs.clemson.edu/~mark/interrupts.html
A treasure trove of information on the history of interrupts, DMA, and related early ideas in computing.

[S03] "Improving the Reliability of Commodity Operating Systems"
Michael M. Swift, Brian N. Bershad, and Henry M. Levy
SOSP '03
Swift's work revived interest in a more microkernel-like approach to operating systems; minimally, it finally gave some good reasons why address-space based protection could be useful in a modern OS.

[W10] "Hard Disk Driver"
Washington State Course Homepage
Available: http://eecs.wsu.edu/~cs460/cs560/HDdriver.html
A nice summary of a simple IDE disk drive's interface and how to build a device driver for it.

Beej's Guide to Network Programming

Using Internet Sockets

Brian “Beej Jorgensen” Hall
beej@beej.us

Version 3.0.15
July 3, 2012

Thanks to everyone who has helped in the past and future with me getting this guide written. Thanks to Ashley for helping me coax the cover design into the best programmer art I could. Thank you to all the people who produce the Free software and packages that I use to make the Guide: GNU, Linux, Slackware, vim, Python, Inkscape, Apache FOP, Firefox, Red Hat, and many others. And finally a big thank-you to the literally thousands of you who have written in with suggestions for improvements and words of encouragement.

I dedicate this guide to some of my biggest heroes and inpirators in the world of computers: Donald Knuth, Bruce Schneier, W. Richard Stevens, and The Woz, my Readership, and the entire Free and Open Source Software Community.

This book is written in XML using the vim editor on a Slackware Linux box loaded with GNU tools. The cover "art" and diagrams are produced with Inkscape. The XML is converted into HTML and XSL-FO by custom Python scripts. The XSL-FO output is then munged by Apache FOP to produce PDF documents, using Liberation fonts. The toolchain is composed of 100% Free and Open Source Software.

Contents

1. Intro

Hey! Socket programming got you down? Is this stuff just a little too difficult to figure out from the **man** pages? You want to do cool Internet programming, but you don't have time to wade through a gob of `structs` trying to figure out if you have to call **`bind()`** before you **`connect()`**, etc., etc.

Well, guess what! I've already done this nasty business, and I'm dying to share the information with everyone! You've come to the right place. This document should give the average competent C programmer the edge s/he needs to get a grip on this networking noise.

And check it out: I've finally caught up with the future (just in the nick of time, too!) and have updated the Guide for IPv6! Enjoy!

1.1. Audience

This document has been written as a tutorial, not a complete reference. It is probably at its best when read by individuals who are just starting out with socket programming and are looking for a foothold. It is certainly not the *complete and total* guide to sockets programming, by any means.

Hopefully, though, it'll be just enough for those man pages to start making sense... `:-)`

1.2. Platform and Compiler

The code contained within this document was compiled on a Linux PC using Gnu's **gcc** compiler. It should, however, build on just about any platform that uses **gcc**. Naturally, this doesn't apply if you're programming for Windows—see the section on Windows programming, below.

1.3. Official Homepage and Books For Sale

This official location of this document is `http://beej.us/guide/bgnet/`. There you will also find example code and translations of the guide into various languages.

To buy nicely bound print copies (some call them "books"), visit `http://beej.us/guide/url/bgbuy`. I'll appreciate the purchase because it helps sustain my document-writing lifestyle!

1.4. Note for Solaris/SunOS Programmers

When compiling for Solaris or SunOS, you need to specify some extra command-line switches for linking in the proper libraries. In order to do this, simply add "`-lnsl -lsocket -lresolv`" to the end of the compile command, like so:

```
$ cc -o server server.c -lnsl -lsocket -lresolv
```

If you still get errors, you could try further adding a "`-lxnet`" to the end of that command line. I don't know what that does, exactly, but some people seem to need it.

Another place that you might find problems is in the call to **`setsockopt()`**. The prototype differs from that on my Linux box, so instead of:

```
int yes=1;
```

enter this:

```
char yes='1';
```

As I don't have a Sun box, I haven't tested any of the above information—it's just what people have told me through email.

1.5. Note for Windows Programmers

At this point in the guide, historically, I've done a bit of bagging on Windows, simply due to the fact that I don't like it very much. But I should really be fair and tell you that Windows has a huge install base and is obviously a perfectly fine operating system.

They say absence makes the heart grow fonder, and in this case, I believe it to be true. (Or maybe it's age.) But what I can say is that after a decade-plus of not using Microsoft OSes for my personal work, I'm much happier! As such, I can sit back and safely say, "Sure, feel free to use Windows!" ...Ok yes, it does make me grit my teeth to say that.

So I still encourage you to try Linux[1], BSD[2], or some flavor of Unix, instead.

But people like what they like, and you Windows folk will be pleased to know that this information is generally applicable to you guys, with a few minor changes, if any.

One cool thing you can do is install Cygwin[3], which is a collection of Unix tools for Windows. I've heard on the grapevine that doing so allows all these programs to compile unmodified.

But some of you might want to do things the Pure Windows Way. That's very gutsy of you, and this is what you have to do: run out and get Unix immediately! No, no—I'm kidding. I'm supposed to be Windows-friendly(er) these days...

This is what you'll have to do (unless you install Cygwin!): first, ignore pretty much all of the system header files I mention in here. All you need to include is:

```
#include <winsock.h>
```

Wait! You also have to make a call to **`WSAStartup()`** before doing anything else with the sockets library. The code to do that looks something like this:

```
#include <winsock.h>

{
    WSADATA wsaData;   // if this doesn't work
    //WSAData wsaData; // then try this instead

    // MAKEWORD(1,1) for Winsock 1.1, MAKEWORD(2,0) for Winsock 2.0:

    if (WSAStartup(MAKEWORD(1,1), &wsaData) != 0) {
        fprintf(stderr, "WSAStartup failed.\n");
        exit(1);
    }
```

You also have to tell your compiler to link in the Winsock library, usually called *`wsock32.lib`* or *`winsock32.lib`*, or *`ws2_32.lib`* for Winsock 2.0. Under VC++, this can be done through the `Project` menu, under `Settings...`. Click the `Link` tab, and look for the box titled “Object/library modules”. Add “wsock32.lib” (or whichever lib is your preference) to that list.

Or so I hear.

Finally, you need to call **`WSACleanup()`** when you're all through with the sockets library. See your online help for details.

Once you do that, the rest of the examples in this tutorial should generally apply, with a few exceptions. For one thing, you can't use **`close()`** to close a socket—you need to use **`closesocket()`**, instead. Also, **`select()`** only works with socket descriptors, not file descriptors (like `0` for `stdin`).

There is also a socket class that you can use, `CSocket`. Check your compilers help pages for more information.

To get more information about Winsock, read the Winsock FAQ[4] and go from there.

Finally, I hear that Windows has no **`fork()`** system call which is, unfortunately, used in some of my examples. Maybe you have to link in a POSIX library or something to get it to work, or you can use **`CreateProcess()`** instead. **`fork()`** takes no arguments, and **`CreateProcess()`** takes about 48 billion arguments. If you're not up to that, the **`CreateThread()`** is a little easier to digest...unfortunately a discussion about multithreading is beyond the scope of this document. I can only talk about so much, you know!

1.6. Email Policy

I'm generally available to help out with email questions so feel free to write in, but I can't guarantee a response. I lead a pretty busy life and there are times when I just can't answer a question you have. When that's the case, I usually just delete the message. It's nothing personal; I just won't ever have the time to give the detailed answer you require.

1. `http://www.linux.com/`
2. `http://www.bsd.org/`
3. `http://www.cygwin.com/`
4. `http://tangentsoft.net/wskfaq/`

As a rule, the more complex the question, the less likely I am to respond. If you can narrow down your question before mailing it and be sure to include any pertinent information (like platform, compiler, error messages you're getting, and anything else you think might help me troubleshoot), you're much more likely to get a response. For more pointers, read ESR's document, How To Ask Questions The Smart Way[5].

If you don't get a response, hack on it some more, try to find the answer, and if it's still elusive, then write me again with the information you've found and hopefully it will be enough for me to help out.

Now that I've badgered you about how to write and not write me, I'd just like to let you know that I *fully* appreciate all the praise the guide has received over the years. It's a real morale boost, and it gladdens me to hear that it is being used for good! `:-)` Thank you!

1.7. Mirroring

You are more than welcome to mirror this site, whether publicly or privately. If you publicly mirror the site and want me to link to it from the main page, drop me a line at `beej@beej.us`.

1.8. Note for Translators

If you want to translate the guide into another language, write me at `beej@beej.us` and I'll link to your translation from the main page. Feel free to add your name and contact info to the translation.

Please note the license restrictions in the Copyright and Distribution section, below.

If you want me to host the translation, just ask. I'll also link to it if you want to host it; either way is fine.

1.9. Copyright and Distribution

5. `http://www.catb.org/~esr/faqs/smart-questions.html`

2. What is a socket?

You hear talk of “sockets” all the time, and perhaps you are wondering just what they are exactly. Well, they're this: a way to speak to other programs using standard Unix file descriptors.

What?

Ok—you may have heard some Unix hacker state, “Jeez, *everything* in Unix is a file!” What that person may have been talking about is the fact that when Unix programs do any sort of I/O, they do it by reading or writing to a file descriptor. A file descriptor is simply an integer associated with an open file. But (and here's the catch), that file can be a network connection, a FIFO, a pipe, a terminal, a real on-the-disk file, or just about anything else. Everything in Unix *is* a file! So when you want to communicate with another program over the Internet you're gonna do it through a file descriptor, you'd better believe it.

“Where do I get this file descriptor for network communication, Mr. Smarty-Pants?” is probably the last question on your mind right now, but I'm going to answer it anyway: You make a call to the **`socket()`** system routine. It returns the socket descriptor, and you communicate through it using the specialized **`send()`** and **`recv()`** (**man send**, **man recv**) socket calls.

“But, hey!” you might be exclaiming right about now. “If it's a file descriptor, why in the name of Neptune can't I just use the normal **`read()`** and **`write()`** calls to communicate through the socket?” The short answer is, “You can!” The longer answer is, “You can, but **`send()`** and **`recv()`** offer much greater control over your data transmission.”

What next? How about this: there are all kinds of sockets. There are DARPA Internet addresses (Internet Sockets), path names on a local node (Unix Sockets), CCITT X.25 addresses (X.25 Sockets that you can safely ignore), and probably many others depending on which Unix flavor you run. This document deals only with the first: Internet Sockets.

2.1. Two Types of Internet Sockets

What's this? There are two types of Internet sockets? Yes. Well, no. I'm lying. There are more, but I didn't want to scare you. I'm only going to talk about two types here. Except for this sentence, where I'm going to tell you that “Raw Sockets” are also very powerful and you should look them up.

All right, already. What are the two types? One is “Stream Sockets”; the other is “Datagram Sockets”, which may hereafter be referred to as “`SOCK_STREAM`” and “`SOCK_DGRAM`”, respectively. Datagram sockets are sometimes called “connectionless sockets”. (Though they can be **`connect()`**'d if you really want. See **`connect()`**, below.)

Stream sockets are reliable two-way connected communication streams. If you output two items into the socket in the order “1, 2”, they will arrive in the order “1, 2” at the opposite end. They will also be error-free. I'm so certain, in fact, they will be error-free, that I'm just going to put my fingers in my ears and chant *la la la la* if anyone tries to claim otherwise.

What uses stream sockets? Well, you may have heard of the **telnet** application, yes? It uses stream sockets. All the characters you type need to arrive in the same order you type them, right? Also, web browsers use the HTTP protocol which uses stream sockets to get pages. Indeed, if you telnet to a web site on port 80, and type “`GET / HTTP/1.0`” and hit RETURN twice, it'll dump the HTML back at you!

How do stream sockets achieve this high level of data transmission quality? They use a protocol called “The Transmission Control Protocol”, otherwise known as “TCP” (see RFC 793[6] for extremely detailed info on TCP.) TCP makes sure your data arrives sequentially and error-free. You may have heard “TCP” before as the better half of “TCP/IP” where “IP” stands for “Internet Protocol” (see RFC 791[7].) IP deals primarily with Internet routing and is not generally responsible for data integrity.

Cool. What about Datagram sockets? Why are they called connectionless? What is the deal, here, anyway? Why are they unreliable? Well, here are some facts: if you send a datagram, it may arrive. It may arrive out of order. If it arrives, the data within the packet will be error-free.

6. `http://tools.ietf.org/html/rfc793`
7. `http://tools.ietf.org/html/rfc791`

Datagram sockets also use IP for routing, but they don't use TCP; they use the "User Datagram Protocol", or "UDP" (see RFC 768[8].)

Why are they connectionless? Well, basically, it's because you don't have to maintain an open connection as you do with stream sockets. You just build a packet, slap an IP header on it with destination information, and send it out. No connection needed. They are generally used either when a TCP stack is unavailable or when a few dropped packets here and there don't mean the end of the Universe. Sample applications: **tftp** (trivial file transfer protocol, a little brother to FTP), **dhcpcd** (a DHCP client), multiplayer games, streaming audio, video conferencing, etc.

"Wait a minute! **tftp** and **dhcpcd** are used to transfer binary applications from one host to another! Data can't be lost if you expect the application to work when it arrives! What kind of dark magic is this?"

Well, my human friend, **tftp** and similar programs have their own protocol on top of UDP. For example, the tftp protocol says that for each packet that gets sent, the recipient has to send back a packet that says, "I got it!" (an "ACK" packet.) If the sender of the original packet gets no reply in, say, five seconds, he'll re-transmit the packet until he finally gets an ACK. This acknowledgment procedure is very important when implementing reliable `SOCK_DGRAM` applications.

For unreliable applications like games, audio, or video, you just ignore the dropped packets, or perhaps try to cleverly compensate for them. (Quake players will know the manifestation this effect by the technical term: *accursed lag*. The word "accursed", in this case, represents any extremely profane utterance.)

Why would you use an unreliable underlying protocol? Two reasons: speed and speed. It's way faster to fire-and-forget than it is to keep track of what has arrived safely and make sure it's in order and all that. If you're sending chat messages, TCP is great; if you're sending 40 positional updates per second of the players in the world, maybe it doesn't matter so much if one or two get dropped, and UDP is a good choice.

2.2. Low level Nonsense and Network Theory

Since I just mentioned layering of protocols, it's time to talk about how networks really work, and to show some examples of how `SOCK_DGRAM` packets are built. Practically, you can probably skip this section. It's good background, however.

Data Encapsulation.

Hey, kids, it's time to learn about *Data Encapsulation*! This is very very important. It's so important that you might just learn about it if you take the networks course here at Chico State `;-)`. Basically, it says this: a packet is born, the packet is wrapped ("encapsulated") in a header (and rarely a footer) by the first protocol (say, the TFTP protocol), then the whole thing (TFTP header included) is encapsulated again by the next protocol (say, UDP), then again by the next (IP), then again by the final protocol on the hardware (physical) layer (say, Ethernet).

When another computer receives the packet, the hardware strips the Ethernet header, the kernel strips the IP and UDP headers, the TFTP program strips the TFTP header, and it finally has the data.

Now I can finally talk about the infamous *Layered Network Model* (aka "ISO/OSI"). This Network Model describes a system of network functionality that has many advantages over other models. For instance, you can write sockets programs that are exactly the same without caring how the data is physically transmitted (serial, thin Ethernet, AUI, whatever) because programs on lower levels deal with it for you. The actual network hardware and topology is transparent to the socket programmer.

Without any further ado, I'll present the layers of the full-blown model. Remember this for network class exams:

- Application

8. `http://tools.ietf.org/html/rfc768`

- Presentation
- Session
- Transport
- Network
- Data Link
- Physical

The Physical Layer is the hardware (serial, Ethernet, etc.). The Application Layer is just about as far from the physical layer as you can imagine—it's the place where users interact with the network.

Now, this model is so general you could probably use it as an automobile repair guide if you really wanted to. A layered model more consistent with Unix might be:

- Application Layer (*telnet, ftp, etc.*)
- Host-to-Host Transport Layer (*TCP, UDP*)
- Internet Layer (*IP and routing*)
- Network Access Layer (*Ethernet, wi-fi, or whatever*)

At this point in time, you can probably see how these layers correspond to the encapsulation of the original data.

See how much work there is in building a simple packet? Jeez! And you have to type in the packet headers yourself using "**cat**"! Just kidding. All you have to do for stream sockets is **`send()`** the data out. All you have to do for datagram sockets is encapsulate the packet in the method of your choosing and **`sendto()`** it out. The kernel builds the Transport Layer and Internet Layer on for you and the hardware does the Network Access Layer. Ah, modern technology.

So ends our brief foray into network theory. Oh yes, I forgot to tell you everything I wanted to say about routing: nothing! That's right, I'm not going to talk about it at all. The router strips the packet to the IP header, consults its routing table, blah blah blah. Check out the IP RFC[9] if you really really care. If you never learn about it, well, you'll live.

9. `http://tools.ietf.org/html/rfc791`

3. IP Addresses, `structs`, and Data Munging

Here's the part of the game where we get to talk code for a change.

But first, let's discuss more non-code! Yay! First I want to talk about IP addresses and ports for just a tad so we have that sorted out. Then we'll talk about how the sockets API stores and manipulates IP addresses and other data.

3.1. IP Addresses, versions 4 and 6

In the good old days back when Ben Kenobi was still called Obi Wan Kenobi, there was a wonderful network routing system called The Internet Protocol Version 4, also called IPv4. It had addresses made up of four bytes (A.K.A. four "octets"), and was commonly written in "dots and numbers" form, like so: `192.0.2.111`.

You've probably seen it around.

In fact, as of this writing, virtually every site on the Internet uses IPv4.

Everyone, including Obi Wan, was happy. Things were great, until some naysayer by the name of Vint Cerf warned everyone that we were about to run out of IPv4 addresses!

(Besides warning everyone of the Coming IPv4 Apocalypse Of Doom And Gloom, Vint Cerf[10] is also well-known for being The Father Of The Internet. So I really am in no position to second-guess his judgment.)

Run out of addresses? How could this be? I mean, there are like billions of IP addresses in a 32-bit IPv4 address. Do we really have billions of computers out there?

Yes.

Also, in the beginning, when there were only a few computers and everyone thought a billion was an impossibly large number, some big organizations were generously allocated millions of IP addresses for their own use. (Such as Xerox, MIT, Ford, HP, IBM, GE, AT&T, and some little company called Apple, to name a few.)

In fact, if it weren't for several stopgap measures, we would have run out a long time ago.

But now we're living in an era where we're talking about every human having an IP address, every computer, every calculator, every phone, every parking meter, and (why not) every puppy dog, as well.

And so, IPv6 was born. Since Vint Cerf is probably immortal (even if his physical form should pass on, heaven forbid, he is probably already existing as some kind of hyper-intelligent ELIZA[11] program out in the depths of the Internet2), no one wants to have to hear him say again "I told you so" if we don't have enough addresses in the next version of the Internet Protocol.

What does this suggest to you?

That we need a *lot* more addresses. That we need not just twice as many addresses, not a billion times as many, not a thousand trillion times as many, but *79 MILLION BILLION TRILLION times as many possible addresses!* That'll show 'em!

You're saying, "Beej, is that true? I have every reason to disbelieve large numbers." Well, the difference between 32 bits and 128 bits might not sound like a lot; it's only 96 more bits, right? But remember, we're talking powers here: 32 bits represents some 4 billion numbers (2^{32}), while 128 bits represents about 340 trillion trillion trillion numbers (for real, 2^{128}). That's like a million IPv4 Internets for *every single star in the Universe*.

Forget this dots-and-numbers look of IPv4, too; now we've got a hexadecimal representation, with each two-byte chunk separated by a colon, like this: `2001:0db8:c9d2:aee5:73e3:934a:a5ae:9551`.

That's not all! Lots of times, you'll have an IP address with lots of zeros in it, and you can compress them between two colons. And you can leave off leading zeros for each byte pair. For instance, each of these pairs of addresses are equivalent:

```
2001:0db8:c9d2:0012:0000:0000:0000:0051
2001:db8:c9d2:12::51
```

10. `http://en.wikipedia.org/wiki/Vinton_Cerf`
11. `http://en.wikipedia.org/wiki/ELIZA`

```
2001:0db8:ab00:0000:0000:0000:0000:0000
2001:db8:ab00::

0000:0000:0000:0000:0000:0000:0000:0001
::1
```

The address `::1` is the *loopback address*. It always means "this machine I'm running on now". In IPv4, the loopback address is 127.0.0.1.

Finally, there's an IPv4-compatibility mode for IPv6 addresses that you might come across. If you want, for example, to represent the IPv4 address 192.0.2.33 as an IPv6 address, you use the following notation: "`::ffff:192.0.2.33`".

We're talking serious fun.

In fact, it's such serious fun, that the Creators of IPv6 have quite cavalierly lopped off trillions and trillions of addresses for reserved use, but we have so many, frankly, who's even counting anymore? There are plenty left over for every man, woman, child, puppy, and parking meter on every planet in the galaxy. And believe me, every planet in the galaxy has parking meters. You know it's true.

3.1.1. Subnets

For organizational reasons, it's sometimes convenient to declare that "this first part of this IP address up through this bit is the *network portion* of the IP address, and the remainder is the *host portion*.

For instance, with IPv4, you might have `192.0.2.12`, and we could say that the first three bytes are the network and the last byte was the host. Or, put another way, we're talking about host `12` on network `192.0.2.0` (see how we zero out the byte that was the host.)

And now for more outdated information! Ready? In the Ancient Times, there were "classes" of subnets, where the first one, two, or three bytes of the address was the network part. If you were lucky enough to have one byte for the network and three for the host, you could have 24 bits-worth of hosts on your network (24 million or so). That was a "Class A" network. On the opposite end was a "Class C", with three bytes of network, and one byte of host (256 hosts, minus a couple that were reserved.)

So as you can see, there were just a few Class As, a huge pile of Class Cs, and some Class Bs in the middle.

The network portion of the IP address is described by something called the *netmask*, which you bitwise-AND with the IP address to get the network number out of it. The netmask usually looks something like `255.255.255.0`. (E.g. with that netmask, if your IP is `192.0.2.12`, then your network is `192.0.2.12` AND `255.255.255.0` which gives `192.0.2.0`.)

Unfortunately, it turned out that this wasn't fine-grained enough for the eventual needs of the Internet; we were running out of Class C networks quite quickly, and we were most definitely out of Class As, so don't even bother to ask. To remedy this, The Powers That Be allowed for the netmask to be an arbitrary number of bits, not just 8, 16, or 24. So you might have a netmask of, say `255.255.255.252`, which is 30 bits of network, and 2 bits of host allowing for four hosts on the network. (Note that the netmask is *ALWAYS* a bunch of 1-bits followed by a bunch of 0-bits.)

But it's a bit unwieldy to use a big string of numbers like `255.192.0.0` as a netmask. First of all, people don't have an intuitive idea of how many bits that is, and secondly, it's really not compact. So the New Style came along, and it's much nicer. You just put a slash after the IP address, and then follow that by the number of network bits in decimal. Like this: `192.0.2.12/30`.

Or, for IPv6, something like this: `2001:db8::/32` or `2001:db8:5413:4028::9db9/64`.

3.1.2. Port Numbers

If you'll kindly remember, I presented you earlier with the Layered Network Model which had the Internet Layer (IP) split off from the Host-to-Host Transport Layer (TCP and UDP). Get up to speed on that before the next paragraph.

Turns out that besides an IP address (used by the IP layer), there is another address that is used by TCP (stream sockets) and, coincidentally, by UDP (datagram sockets). It is the *port number*. It's a 16-bit number that's like the local address for the connection.

Think of the IP address as the street address of a hotel, and the port number as the room number. That's a decent analogy; maybe later I'll come up with one involving the automobile industry.

Say you want to have a computer that handles incoming mail AND web services—how do you differentiate between the two on a computer with a single IP address?

Well, different services on the Internet have different well-known port numbers. You can see them all in the Big IANA Port List[12] or, if you're on a Unix box, in your `/etc/services` file. HTTP (the web) is port 80, telnet is port 23, SMTP is port 25, the game DOOM[13] used port 666, etc. and so on. Ports under 1024 are often considered special, and usually require special OS privileges to use.

And that's about it!

3.2. Byte Order

By Order of the Realm! There shall be two byte orderings, hereafter to be known as Lame and Magnificent!

I joke, but one really is better than the other. `:-)`

There really is no easy way to say this, so I'll just blurt it out: your computer might have been storing bytes in reverse order behind your back. I know! No one wanted to have to tell you.

The thing is, everyone in the Internet world has generally agreed that if you want to represent the two-byte hex number, say `b34f`, you'll store it in two sequential bytes `b3` followed by `4f`. Makes sense, and, as Wilford Brimley[14] would tell you, it's the Right Thing To Do. This number, stored with the big end first, is called *Big-Endian*.

Unfortunately, a few computers scattered here and there throughout the world, namely anything with an Intel or Intel-compatible processor, store the bytes reversed, so `b34f` would be stored in memory as the sequential bytes `4f` followed by `b3`. This storage method is called *Little-Endian*.

But wait, I'm not done with terminology yet! The more-sane *Big-Endian* is also called *Network Byte Order* because that's the order us network types like.

Your computer stores numbers in *Host Byte Order*. If it's an Intel 80x86, Host Byte Order is Little-Endian. If it's a Motorola 68k, Host Byte Order is Big-Endian. If it's a PowerPC, Host Byte Order is... well, it depends!

A lot of times when you're building packets or filling out data structures you'll need to make sure your two- and four-byte numbers are in Network Byte Order. But how can you do this if you don't know the native Host Byte Order?

Good news! You just get to assume the Host Byte Order isn't right, and you always run the value through a function to set it to Network Byte Order. The function will do the magic conversion if it has to, and this way your code is portable to machines of differing endianness.

All righty. There are two types of numbers that you can convert: `short` (two bytes) and `long` (four bytes). These functions work for the `unsigned` variations as well. Say you want to convert a `short` from Host Byte Order to Network Byte Order. Start with “h” for “host”, follow it with “to”, then “n” for “network”, and “s” for “short”: h-to-n-s, or **`htons()`** (read: “Host to Network Short”).

It's almost too easy...

You can use every combination of “n”, “h”, “s”, and “l” you want, not counting the really stupid ones. For example, there is NOT a **`stolh()`** (“Short to Long Host”) function—not at this party, anyway. But there are:

`htons()`	**h**ost **to** **n**etwork **s**hort
`htonl()`	**h**ost **to** **n**etwork **l**ong
`ntohs()`	**n**etwork **to** **h**ost **s**hort
`ntohl()`	**n**etwork **to** **h**ost **l**ong

Basically, you'll want to convert the numbers to Network Byte Order before they go out on the wire, and convert them to Host Byte Order as they come in off the wire.

I don't know of a 64-bit variant, sorry. And if you want to do floating point, check out the section on Serialization, far below.

12. `http://www.iana.org/assignments/port-numbers`
13. `http://en.wikipedia.org/wiki/Doom_(video_game)`
14. `http://en.wikipedia.org/wiki/Wilford_Brimley`

Assume the numbers in this document are in Host Byte Order unless I say otherwise.

3.3. `structs`

Well, we're finally here. It's time to talk about programming. In this section, I'll cover various data types used by the sockets interface, since some of them are a real bear to figure out.

First the easy one: a socket descriptor. A socket descriptor is the following type:

```
int
```

Just a regular `int`.

Things get weird from here, so just read through and bear with me.

My First Struct™—`struct addrinfo`. This structure is a more recent invention, and is used to prep the socket address structures for subsequent use. It's also used in host name lookups, and service name lookups. That'll make more sense later when we get to actual usage, but just know for now that it's one of the first things you'll call when making a connection.

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname

    struct addrinfo *ai_next;      // linked list, next node
};
```

You'll load this struct up a bit, and then call **`getaddrinfo()`**. It'll return a pointer to a new linked list of these structures filled out with all the goodies you need.

You can force it to use IPv4 or IPv6 in the *`ai_family`* field, or leave it as `AF_UNSPEC` to use whatever. This is cool because your code can be IP version-agnostic.

Note that this is a linked list: *`ai_next`* points at the next element—there could be several results for you to choose from. I'd use the first result that worked, but you might have different business needs; I don't know everything, man!

You'll see that the *`ai_addr`* field in the `struct addrinfo` is a pointer to a `struct sockaddr`. This is where we start getting into the nitty-gritty details of what's inside an IP address structure.

You might not usually need to write to these structures; oftentimes, a call to **`getaddrinfo()`** to fill out your `struct addrinfo` for you is all you'll need. You *will*, however, have to peer inside these `structs` to get the values out, so I'm presenting them here.

(Also, all the code written before `struct addrinfo` was invented packed all this stuff by hand, so you'll see a lot of IPv4 code out in the wild that does exactly that. You know, in old versions of this guide and so on.)

Some `structs` are IPv4, some are IPv6, and some are both. I'll make notes of which are what.

Anyway, the `struct sockaddr` holds socket address information for many types of sockets.

```
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};
```

`sa_family` can be a variety of things, but it'll be `AF_INET` (IPv4) or `AF_INET6` (IPv6) for everything we do in this document. *`sa_data`* contains a destination address and port number for the socket. This is rather unwieldy since you don't want to tediously pack the address in the *`sa_data`* by hand.

To deal with `struct sockaddr`, programmers created a parallel structure: `struct sockaddr_in` ("in" for "Internet") to be used with IPv4.

And *this is the important* bit: a pointer to a `struct sockaddr_in` can be cast to a pointer to a `struct sockaddr` and vice-versa. So even though **`connect()`** wants a `struct sockaddr*`, you can still use a `struct sockaddr_in` and cast it at the last minute!

```
// (IPv4 only--see struct sockaddr_in6 for IPv6)
```

```
struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
```

This structure makes it easy to reference elements of the socket address. Note that *`sin_zero`* (which is included to pad the structure to the length of a `struct sockaddr`) should be set to all zeros with the function **`memset()`**. Also, notice that *`sin_family`* corresponds to *`sa_family`* in a `struct sockaddr` and should be set to "`AF_INET`". Finally, the *`sin_port`* must be in *Network Byte Order* (by using **`htons()`**!)

Let's dig deeper! You see the *`sin_addr`* field is a `struct in_addr`. What is that thing? Well, not to be overly dramatic, but it's one of the scariest unions of all time:

```
// (IPv4 only--see struct in6_addr for IPv6)

// Internet address (a structure for historical reasons)
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};
```

Whoa! Well, it *used* to be a union, but now those days seem to be gone. Good riddance. So if you have declared *`ina`* to be of type `struct sockaddr_in`, then *`ina.sin_addr.s_addr`* references the 4-byte IP address (in Network Byte Order). Note that even if your system still uses the God-awful union for `struct in_addr`, you can still reference the 4-byte IP address in exactly the same way as I did above (this due to `#defines`.)

What about IPv6? Similar `structs` exist for it, as well:

```
// (IPv6 only--see struct sockaddr_in and struct in_addr for IPv4)

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char   s6_addr[16];   // IPv6 address
};
```

Note that IPv6 has an IPv6 address and a port number, just like IPv4 has an IPv4 address and a port number.

Also note that I'm not going to talk about the IPv6 flow information or Scope ID fields for the moment... this is just a starter guide. `:-)`

Last but not least, here is another simple structure, `struct sockaddr_storage` that is designed to be large enough to hold both IPv4 and IPv6 structures. (See, for some calls, sometimes you don't know in advance if it's going to fill out your `struct sockaddr` with an IPv4 or IPv6 address. So you pass in this parallel structure, very similar to `struct sockaddr` except larger, and then cast it to the type you need:

```
struct sockaddr_storage {
    sa_family_t  ss_family;     // address family

    // all this is padding, implementation specific, ignore it:
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

What's important is that you can see the address family in the *`ss_family`* field—check this to see if it's `AF_INET` or `AF_INET6` (for IPv4 or IPv6). Then you can cast it to a `struct sockaddr_in` or `struct sockaddr_in6` if you wanna.

3.4. IP Addresses, Part Deux

Fortunately for you, there are a bunch of functions that allow you to manipulate IP addresses. No need to figure them out by hand and stuff them in a `long` with the `<<` operator.

First, let's say you have a `struct sockaddr_in ina`, and you have an IP address "`10.12.110.57`" or "`2001:db8:63b3:1::3490`" that you want to store into it. The function you want to use, **`inet_pton()`**, converts an IP address in numbers-and-dots notation into either a `struct in_addr` or a `struct in6_addr` depending on whether you specify `AF_INET` or `AF_INET6`. ("pton" stands for "presentation to network"—you can call it "printable to network" if that's easier to remember.) The conversion can be made as follows:

```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6

inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr)); // IPv4
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```

(Quick note: the old way of doing things used a function called **`inet_addr()`** or another function called **`inet_aton()`**; these are now obsolete and don't work with IPv6.)

Now, the above code snippet isn't very robust because there is no error checking. See, **`inet_pton()`** returns `-1` on error, or 0 if the address is messed up. So check to make sure the result is greater than 0 before using!

All right, now you can convert string IP addresses to their binary representations. What about the other way around? What if you have a `struct in_addr` and you want to print it in numbers-and-dots notation? (Or a `struct in6_addr` that you want in, uh, "hex-and-colons" notation.) In this case, you'll want to use the function **`inet_ntop()`** ("ntop" means "network to presentation"—you can call it "network to printable" if that's easier to remember), like this:

```
// IPv4:

char ip4[INET_ADDRSTRLEN];  // space to hold the IPv4 string
struct sockaddr_in sa;      // pretend this is loaded with something

inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);

printf("The IPv4 address is: %s\n", ip4);


// IPv6:

char ip6[INET6_ADDRSTRLEN]; // space to hold the IPv6 string
struct sockaddr_in6 sa6;    // pretend this is loaded with something

inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);

printf("The address is: %s\n", ip6);
```

When you call it, you'll pass the address type (IPv4 or IPv6), the address, a pointer to a string to hold the result, and the maximum length of that string. (Two macros conveniently hold the size of the string you'll need to hold the largest IPv4 or IPv6 address: `INET_ADDRSTRLEN` and `INET6_ADDRSTRLEN`.)

(Another quick note to mention once again the old way of doing things: the historical function to do this conversion was called **`inet_ntoa()`**. It's also obsolete and won't work with IPv6.)

Lastly, these functions only work with numeric IP addresses—they won't do any nameserver DNS lookup on a hostname, like "www.example.com". You will use **`getaddrinfo()`** to do that, as you'll see later on.

3.4.1. Private (Or Disconnected) Networks

Lots of places have a firewall that hides the network from the rest of the world for their own protection. And often times, the firewall translates “internal” IP addresses to “external” (that everyone else in the world knows) IP addresses using a process called *Network Address Translation*, or NAT.

Are you getting nervous yet? “Where's he going with all this weird stuff?”

Well, relax and buy yourself a non-alcoholic (or alcoholic) drink, because as a beginner, you don't even have to worry about NAT, since it's done for you transparently. But I wanted to talk about the network behind the firewall in case you started getting confused by the network numbers you were seeing.

For instance, I have a firewall at home. I have two static IPv4 addresses allocated to me by the DSL company, and yet I have seven computers on the network. How is this possible? Two computers can't share the same IP address, or else the data wouldn't know which one to go to!

The answer is: they don't share the same IP addresses. They are on a private network with 24 million IP addresses allocated to it. They are all just for me. Well, all for me as far as anyone else is concerned. Here's what's happening:

If I log into a remote computer, it tells me I'm logged in from 192.0.2.33 which is the public IP address my ISP has provided to me. But if I ask my local computer what it's IP address is, it says 10.0.0.5. Who is translating the IP address from one to the other? That's right, the firewall! It's doing NAT!

10.*x*.*x*.*x* is one of a few reserved networks that are only to be used either on fully disconnected networks, or on networks that are behind firewalls. The details of which private network numbers are available for you to use are outlined in RFC 1918[15], but some common ones you'll see are 10.*x*.*x*.*x* and 192.168.*x*.*x*, where *x* is 0-255, generally. Less common is 172.*y*.*x*.*x*, where *y* goes between 16 and 31.

Networks behind a NATing firewall don't *need* to be on one of these reserved networks, but they commonly are.

(Fun fact! My external IP address isn't really 192.0.2.33. The 192.0.2.*x* network is reserved for make-believe “real” IP addresses to be used in documentation, just like this guide! Wowzers!)

IPv6 has private networks, too, in a sense. They'll start with `fd`*`xx`*`:` (or maybe in the future `fc`*`XX`*`:`), as per RFC 4193[16]. NAT and IPv6 don't generally mix, however (unless you're doing the IPv6 to IPv4 gateway thing which is beyond the scope of this document)—in theory you'll have so many addresses at your disposal that you won't need to use NAT any longer. But if you want to allocate addresses for yourself on a network that won't route outside, this is how to do it.

15. `http://tools.ietf.org/html/rfc1918`
16. `http://tools.ietf.org/html/rfc4193`

4. Jumping from IPv4 to IPv6

But I just want to know what to change in my code to get it going with IPv6! Tell me now!
Ok! Ok!
Almost everything in here is something I've gone over, above, but it's the short version for the impatient. (Of course, there is more than this, but this is what applies to the guide.)

1. First of all, try to use **`getaddrinfo()`** to get all the `struct sockaddr` info, instead of packing the structures by hand. This will keep you IP version-agnostic, and will eliminate many of the subsequent steps.
2. Any place that you find you're hard-coding anything related to the IP version, try to wrap up in a helper function.
3. Change `AF_INET` to `AF_INET6`.
4. Change `PF_INET` to `PF_INET6`.
5. Change `INADDR_ANY` assignments to `in6addr_any` assignments, which are slightly different:

   ```
   struct sockaddr_in sa;
   struct sockaddr_in6 sa6;

   sa.sin_addr.s_addr = INADDR_ANY;  // use my IPv4 address
   sa6.sin6_addr = in6addr_any; // use my IPv6 address
   ```

   Also, the value `IN6ADDR_ANY_INIT` can be used as an initializer when the `struct in6_addr` is declared, like so:

   ```
   struct in6_addr ia6 = IN6ADDR_ANY_INIT;
   ```

6. Instead of `struct sockaddr_in` use `struct sockaddr_in6`, being sure to add "6" to the fields as appropriate (see `struct`s, above). There is no *`sin6_zero`* field.
7. Instead of `struct in_addr` use `struct in6_addr`, being sure to add "6" to the fields as appropriate (see `struct`s, above).
8. Instead of **`inet_aton()`** or **`inet_addr()`**, use **`inet_pton()`**.
9. Instead of **`inet_ntoa()`**, use **`inet_ntop()`**.
10. Instead of **`gethostbyname()`**, use the superior **`getaddrinfo()`**.
11. Instead of **`gethostbyaddr()`**, use the superior **`getnameinfo()`** (although **`gethostbyaddr()`** can still work with IPv6).
12. `INADDR_BROADCAST` no longer works. Use IPv6 multicast instead.

Et voila!

5. System Calls or Bust

This is the section where we get into the system calls (and other library calls) that allow you to access the network functionality of a Unix box, or any box that supports the sockets API for that matter (BSD, Windows, Linux, Mac, what-have-you.) When you call one of these functions, the kernel takes over and does all the work for you automagically.

The place most people get stuck around here is what order to call these things in. In that, the **man** pages are no use, as you've probably discovered. Well, to help with that dreadful situation, I've tried to lay out the system calls in the following sections in *exactly* (approximately) the same order that you'll need to call them in your programs.

That, coupled with a few pieces of sample code here and there, some milk and cookies (which I fear you will have to supply yourself), and some raw guts and courage, and you'll be beaming data around the Internet like the Son of Jon Postel!

(Please note that for brevity, many code snippets below do not include necessary error checking. And they very commonly assume that the result from calls to ***`getaddrinfo()`*** *succeed and return a valid entry in the linked list. Both of these situations are properly addressed in the stand-alone programs, though, so use those as a model.)*

5.1. `getaddrinfo()`—Prepare to launch!

This is a real workhorse of a function with a lot of options, but usage is actually pretty simple. It helps set up the `structs` you need later on.

A tiny bit of history: it used to be that you would use a function called **`gethostbyname()`** to do DNS lookups. Then you'd load that information by hand into a `struct sockaddr_in`, and use that in your calls.

This is no longer necessary, thankfully. (Nor is it desirable, if you want to write code that works for both IPv4 and IPv6!) In these modern times, you now have the function **`getaddrinfo()`** that does all kinds of good stuff for you, including DNS and service name lookups, and fills out the `structs` you need, besides!

Let's take a look!

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *node,     // e.g. "www.example.com" or IP
                const char *service,  // e.g. "http" or port number
                const struct addrinfo *hints,
                struct addrinfo **res);
```

You give this function three input parameters, and it gives you a pointer to a linked-list, *`res`*, of results.

The *`node`* parameter is the host name to connect to, or an IP address.

Next is the parameter *`service`*, which can be a port number, like “80”, or the name of a particular service (found in The IANA Port List[17] or the */etc/services* file on your Unix machine) like “http” or “ftp” or “telnet” or “smtp” or whatever.

Finally, the *`hints`* parameter points to a `struct addrinfo` that you've already filled out with relevant information.

Here's a sample call if you're a server who wants to listen on your host's IP address, port 3490. Note that this doesn't actually do any listening or network setup; it merely sets up structures we'll use later:

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // will point to the results

memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_UNSPEC;     // don't care IPv4 or IPv6
```

17. `http://www.iana.org/assignments/port-numbers`

```
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
    exit(1);
}

// servinfo now points to a linked list of 1 or more struct addrinfos

// ... do everything until you don't need servinfo anymore ....

freeaddrinfo(servinfo); // free the linked-list
```

Notice that I set the *`ai_family`* to `AF_UNSPEC`, thereby saying that I don't care if we use IPv4 or IPv6. You can set it to `AF_INET` or `AF_INET6` if you want one or the other specifically.

Also, you'll see the `AI_PASSIVE` flag in there; this tells **`getaddrinfo()`** to assign the address of my local host to the socket structures. This is nice because then you don't have to hardcode it. (Or you can put a specific address in as the first parameter to **`getaddrinfo()`** where I currently have `NULL`, up there.)

Then we make the call. If there's an error (**`getaddrinfo()`** returns non-zero), we can print it out using the function **`gai_strerror()`**, as you see. If everything works properly, though, *`servinfo`* will point to a linked list of `struct addrinfo`s, each of which contains a `struct sockaddr` of some kind that we can use later! Nifty!

Finally, when we're eventually all done with the linked list that **`getaddrinfo()`** so graciously allocated for us, we can (and should) free it all up with a call to **`freeaddrinfo()`**.

Here's a sample call if you're a client who wants to connect to a particular server, say "www.example.net" port 3490. Again, this doesn't actually connect, but it sets up the structures we'll use later:

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // will point to the results

memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_UNSPEC;     // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets

// get ready to connect
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);

// servinfo now points to a linked list of 1 or more struct addrinfos

// etc.
```

I keep saying that *`servinfo`* is a linked list with all kinds of address information. Let's write a quick demo program to show off this information. This short program[18] will print the IP addresses for whatever host you specify on the command line:

```
/*
** showip.c -- show IP addresses for a host given on the command line
*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>

int main(int argc, char *argv[])
```

18. `http://beej.us/guide/bgnet/examples/showip.c`

```
{
    struct addrinfo hints, *res, *p;
    int status;
    char ipstr[INET6_ADDRSTRLEN];

    if (argc != 2) {
        fprintf(stderr,"usage: showip hostname\n");
        return 1;
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC; // AF_INET or AF_INET6 to force version
    hints.ai_socktype = SOCK_STREAM;

    if ((status = getaddrinfo(argv[1], NULL, &hints, &res)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
        return 2;
    }

    printf("IP addresses for %s:\n\n", argv[1]);

    for(p = res;p != NULL; p = p->ai_next) {
        void *addr;
        char *ipver;

        // get the pointer to the address itself,
        // different fields in IPv4 and IPv6:
        if (p->ai_family == AF_INET) { // IPv4
            struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
            addr = &(ipv4->sin_addr);
            ipver = "IPv4";
        } else { // IPv6
            struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
            addr = &(ipv6->sin6_addr);
            ipver = "IPv6";
        }

        // convert the IP to a string and print it:
        inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
        printf("  %s: %s\n", ipver, ipstr);
    }

    freeaddrinfo(res); // free the linked list

    return 0;
}
```

As you see, the code calls **`getaddrinfo()`** on whatever you pass on the command line, that fills out the linked list pointed to by *`res`*, and then we can iterate over the list and print stuff out or do whatever.

(There's a little bit of ugliness there where we have to dig into the different types of `struct sockaddrs` depending on the IP version. Sorry about that! I'm not sure of a better way around it.)

Sample run! Everyone loves screenshots:

```
$ showip www.example.net
IP addresses for www.example.net:

  IPv4: 192.0.2.88

$ showip ipv6.example.com
IP addresses for ipv6.example.com:

  IPv4: 192.0.2.101
  IPv6: 2001:db8:8c00:22::171
```

Now that we have that under control, we'll use the results we get from **`getaddrinfo()`** to pass to other socket functions and, at long last, get our network connection established! Keep reading!

5.2. socket()—Get the File Descriptor!

I guess I can put it off no longer—I have to talk about the **socket()** system call. Here's the breakdown:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

But what are these arguments? They allow you to say what kind of socket you want (IPv4 or IPv6, stream or datagram, and TCP or UDP).

It used to be people would hardcode these values, and you can absolutely still do that. (*domain* is PF_INET or PF_INET6, *type* is SOCK_STREAM or SOCK_DGRAM, and *protocol* can be set to 0 to choose the proper protocol for the given type. Or you can call **getprotobyname()** to look up the protocol you want, "tcp" or "udp".)

(This PF_INET thing is a close relative of the AF_INET that you can use when initializing the *sin_family* field in your struct sockaddr_in. In fact, they're so closely related that they actually have the same value, and many programmers will call **socket()** and pass AF_INET as the first argument instead of **PF_INET**. Now, get some milk and cookies, because it's times for a story. Once upon a time, a long time ago, it was thought that maybe a address family (what the "AF" in "AF_INET" stands for) might support several protocols that were referred to by their protocol family (what the "PF" in "PF_INET" stands for). That didn't happen. And they all lived happily ever after, The End. So the most correct thing to do is to use AF_INET in your struct sockaddr_in and PF_INET in your call to **socket()**.)

Anyway, enough of that. What you really want to do is use the values from the results of the call to **getaddrinfo()**, and feed them into **socket()** directly like this:

```
int s;
struct addrinfo hints, *res;

// do the lookup
// [pretend we already filled out the "hints" struct]
getaddrinfo("www.example.com", "http", &hints, &res);

// [again, you should do error-checking on getaddrinfo(), and walk
// the "res" linked list looking for valid entries instead of just
// assuming the first one is good (like many of these examples do.)
// See the section on client/server for real examples.]

s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

socket() simply returns to you a *socket descriptor* that you can use in later system calls, or -1 on error. The global variable *errno* is set to the error's value (see the *errno* man page for more details, and a quick note on using *errno* in multithreaded programs.)

Fine, fine, fine, but what good is this socket? The answer is that it's really no good by itself, and you need to read on and make more system calls for it to make any sense.

5.3. bind()—What port am I on?

Once you have a socket, you might have to associate that socket with a port on your local machine. (This is commonly done if you're going to **listen()** for incoming connections on a specific port—multiplayer network games do this when they tell you to "connect to 192.168.5.10 port 3490".) The port number is used by the kernel to match an incoming packet to a certain process's socket descriptor. If you're going to only be doing a **connect()** (because you're the client, not the server), this is probably be unnecessary. Read it anyway, just for kicks.

Here is the synopsis for the **bind()** system call:

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

sockfd is the socket file descriptor returned by **socket()**. *my_addr* is a pointer to a `struct sockaddr` that contains information about your address, namely, port and IP address. *addrlen* is the length in bytes of that address.

Whew. That's a bit to absorb in one chunk. Let's have an example that binds the socket to the host the program is running on, port 3490:

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

By using the `AI_PASSIVE` flag, I'm telling the program to bind to the IP of the host it's running on. If you want to bind to a specific local IP address, drop the `AI_PASSIVE` and put an IP address in for the first argument to **getaddrinfo()**.

bind() also returns `-1` on error and sets *errno* to the error's value.

Lots of old code manually packs the `struct sockaddr_in` before calling **bind()**. Obviously this is IPv4-specific, but there's really nothing stopping you from doing the same thing with IPv6, except that using **getaddrinfo()** is going to be easier, generally. Anyway, the old code looks something like this:

```
// !!! THIS IS THE OLD WAY !!!

int sockfd;
struct sockaddr_in my_addr;

sockfd = socket(PF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT);     // short, network byte order
my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr);
```

In the above code, you could also assign `INADDR_ANY` to the *s_addr* field if you wanted to bind to your local IP address (like the `AI_PASSIVE` flag, above.) The IPv6 version of `INADDR_ANY` is a global variable *in6addr_any* that is assigned into the *sin6_addr* field of your `struct sockaddr_in6`. (There is also a macro `IN6ADDR_ANY_INIT` that you can use in a variable initializer.)

Another thing to watch out for when calling **bind()**: don't go underboard with your port numbers. All ports below 1024 are RESERVED (unless you're the superuser)! You can have any port number above that, right up to 65535 (provided they aren't already being used by another program.)

Sometimes, you might notice, you try to rerun a server and **bind()** fails, claiming "Address already in use." What does that mean? Well, a little bit of a socket that was connected is still hanging around in the kernel, and it's hogging the port. You can either wait for it to clear (a minute or so), or add code to your program allowing it to reuse the port, like this:

```
int yes=1;
//char yes='1'; // Solaris people use this

// lose the pesky "Address already in use" error message
```

```
if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
    perror("setsockopt");
    exit(1);
}
```

One small extra final note about **bind()**: there are times when you won't absolutely have to call it. If you are **connect()**ing to a remote machine and you don't care what your local port is (as is the case with **telnet** where you only care about the remote port), you can simply call **connect()**, it'll check to see if the socket is unbound, and will **bind()** it to an unused local port if necessary.

5.4. connect()—Hey, you!

Let's just pretend for a few minutes that you're a telnet application. Your user commands you (just like in the movie *TRON*) to get a socket file descriptor. You comply and call **socket()**. Next, the user tells you to connect to "10.12.110.57" on port "23" (the standard telnet port.) Yow! What do you do now?

Lucky for you, program, you're now perusing the section on **connect()**—how to connect to a remote host. So read furiously onward! No time to lose!

The **connect()** call is as follows:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

sockfd is our friendly neighborhood socket file descriptor, as returned by the **socket()** call, *serv_addr* is a `struct sockaddr` containing the destination port and IP address, and *addrlen* is the length in bytes of the server address structure.

All of this information can be gleaned from the results of the **getaddrinfo()** call, which rocks.

Is this starting to make more sense? I can't hear you from here, so I'll just have to hope that it is. Let's have an example where we make a socket connection to "www.example.com", port 3490:

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("www.example.com", "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// connect!

connect(sockfd, res->ai_addr, res->ai_addrlen);
```

Again, old-school programs filled out their own `struct sockaddr_in`s to pass to **connect()**. You can do that if you want to. See the similar note in the **bind()** section, above.

Be sure to check the return value from **connect()**—it'll return `-1` on error and set the variable *errno*.

Also, notice that we didn't call **bind()**. Basically, we don't care about our local port number; we only care where we're going (the remote port). The kernel will choose a local port for us, and the site we connect to will automatically get this information from us. No worries.

5.5. listen()—Will somebody please call me?

Ok, time for a change of pace. What if you don't want to connect to a remote host. Say, just for kicks, that you want to wait for incoming connections and handle them in some way. The process is two step: first you **listen()**, then you **accept()** (see below.)

The listen call is fairly simple, but requires a bit of explanation:

```
int listen(int sockfd, int backlog);
```

sockfd is the usual socket file descriptor from the **socket()** system call. *backlog* is the number of connections allowed on the incoming queue. What does that mean? Well, incoming connections are going to wait in this queue until you **accept()** them (see below) and this is the limit on how many can queue up. Most systems silently limit this number to about 20; you can probably get away with setting it to `5` or `10`.

Again, as per usual, **listen()** returns `-1` and sets *errno* on error.

Well, as you can probably imagine, we need to call **bind()** before we call **listen()** so that the server is running on a specific port. (You have to be able to tell your buddies which port to connect to!) So if you're going to be listening for incoming connections, the sequence of system calls you'll make is:

```
getaddrinfo();
socket();
bind();
listen();
/* accept() goes here */
```

I'll just leave that in the place of sample code, since it's fairly self-explanatory. (The code in the **accept()** section, below, is more complete.) The really tricky part of this whole sha-bang is the call to **accept()**.

5.6. accept()—"Thank you for calling port 3490."

Get ready—the **accept()** call is kinda weird! What's going to happen is this: someone far far away will try to **connect()** to your machine on a port that you are **listen()**ing on. Their connection will be queued up waiting to be **accept()**ed. You call **accept()** and you tell it to get the pending connection. It'll return to you a *brand new socket file descriptor* to use for this single connection! That's right, suddenly you have *two socket file descriptors* for the price of one! The original one is still listening for more new connections, and the newly created one is finally ready to **send()** and **recv()**. We're there!

The call is as follows:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

sockfd is the **listen()**ing socket descriptor. Easy enough. *addr* will usually be a pointer to a local `struct sockaddr_storage`. This is where the information about the incoming connection will go (and with it you can determine which host is calling you from which port). *addrlen* is a local integer variable that should be set to `sizeof(struct sockaddr_storage)` before its address is passed to **accept()**. **accept()** will not put more than that many bytes into *addr*. If it puts fewer in, it'll change the value of *addrlen* to reflect that.

Guess what? **accept()** returns `-1` and sets *errno* if an error occurs. Betcha didn't figure that.

Like before, this is a bunch to absorb in one chunk, so here's a sample code fragment for your perusal:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPORT "3490"  // the port users will be connecting to
#define BACKLOG 10     // how many pending connections queue will hold

int main(void)
{
    struct sockaddr_storage their_addr;
    socklen_t addr_size;
    struct addrinfo hints, *res;
    int sockfd, new_fd;

    // !! don't forget your error checking for these calls !!
```

```
    // first, load up address structs with getaddrinfo():

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

    getaddrinfo(NULL, MYPORT, &hints, &res);

    // make a socket, bind it, and listen on it:

    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    bind(sockfd, res->ai_addr, res->ai_addrlen);
    listen(sockfd, BACKLOG);

    // now accept an incoming connection:

    addr_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

    // ready to communicate on socket descriptor new_fd!
    .
    .
    .
```

Again, note that we will use the socket descriptor *new_fd* for all **send()** and **recv()** calls. If you're only getting one single connection ever, you can **close()** the listening *sockfd* in order to prevent more incoming connections on the same port, if you so desire.

5.7. send() and recv()—Talk to me, baby!

These two functions are for communicating over stream sockets or connected datagram sockets. If you want to use regular unconnected datagram sockets, you'll need to see the section on **sendto()** and **recvfrom()**, below.

The **send()** call:

```
int send(int sockfd, const void *msg, int len, int flags);
```

sockfd is the socket descriptor you want to send data to (whether it's the one returned by **socket()** or the one you got with **accept()**.) *msg* is a pointer to the data you want to send, and *len* is the length of that data in bytes. Just set *flags* to `0`. (See the **send()** man page for more information concerning flags.)

Some sample code might be:

```
char *msg = "Beej was here!";
int len, bytes_sent;
.
.
.
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
.
.
.
```

send() returns the number of bytes actually sent out—*this might be less than the number you told it to send!* See, sometimes you tell it to send a whole gob of data and it just can't handle it. It'll fire off as much of the data as it can, and trust you to send the rest later. Remember, if the value returned by **send()** doesn't match the value in *len*, it's up to you to send the rest of the string. The good news is this: if the packet is small (less than 1K or so) it will *probably* manage to send the whole thing all in one go. Again, `-1` is returned on error, and *errno* is set to the error number.

The **recv()** call is similar in many respects:

```
int recv(int sockfd, void *buf, int len, int flags);
```

sockfd is the socket descriptor to read from, *buf* is the buffer to read the information into, *len* is the maximum length of the buffer, and *flags* can again be set to `0`. (See the **recv()** man page for flag information.)

recv() returns the number of bytes actually read into the buffer, or `-1` on error (with *errno* set, accordingly.)

Wait! **recv()** can return `0`. This can mean only one thing: the remote side has closed the connection on you! A return value of `0` is **recv()**'s way of letting you know this has occurred.

There, that was easy, wasn't it? You can now pass data back and forth on stream sockets! Whee! You're a Unix Network Programmer!

5.8. sendto() and recvfrom()—Talk to me, DGRAM-style

"This is all fine and dandy," I hear you saying, "but where does this leave me with unconnected datagram sockets?" No problemo, amigo. We have just the thing.

Since datagram sockets aren't connected to a remote host, guess which piece of information we need to give before we send a packet? That's right! The destination address! Here's the scoop:

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, socklen_t tolen);
```

As you can see, this call is basically the same as the call to **send()** with the addition of two other pieces of information. *to* is a pointer to a `struct sockaddr` (which will probably be another `struct sockaddr_in` or `struct sockaddr_in6` or `struct sockaddr_storage` that you cast at the last minute) which contains the destination IP address and port. *tolen*, an `int` deep-down, can simply be set to `sizeof *to` or `sizeof(struct sockaddr_storage)`.

To get your hands on the destination address structure, you'll probably either get it from **getaddrinfo()**, or from **recvfrom()**, below, or you'll fill it out by hand.

Just like with **send()**, **sendto()** returns the number of bytes actually sent (which, again, might be less than the number of bytes you told it to send!), or `-1` on error.

Equally similar are **recv()** and **recvfrom()**. The synopsis of **recvfrom()** is:

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
             struct sockaddr *from, int *fromlen);
```

Again, this is just like **recv()** with the addition of a couple fields. *from* is a pointer to a local `struct sockaddr_storage` that will be filled with the IP address and port of the originating machine. *fromlen* is a pointer to a local `int` that should be initialized to `sizeof *from` or `sizeof(struct sockaddr_storage)`. When the function returns, *fromlen* will contain the length of the address actually stored in *from*.

recvfrom() returns the number of bytes received, or `-1` on error (with *errno* set accordingly.)

So, here's a question: why do we use `struct sockaddr_storage` as the socket type? Why not `struct sockaddr_in`? Because, you see, we want to not tie ourselves down to IPv4 or IPv6. So we use the generic `struct sockaddr_storage` which we know will be big enough for either.

(So... here's another question: why isn't `struct sockaddr` itself big enough for any address? We even cast the general-purpose `struct sockaddr_storage` to the general-purpose `struct sockaddr`! Seems extraneous and redundant, huh. The answer is, it just isn't big enough, and I'd guess that changing it at this point would be Problematic. So they made a new one.)

Remember, if you **connect()** a datagram socket, you can then simply use **send()** and **recv()** for all your transactions. The socket itself is still a datagram socket and the packets still use UDP, but the socket interface will automatically add the destination and source information for you.

5.9. close() and shutdown()—Get outta my face!

Whew! You've been **send()**ing and **recv()**ing data all day long, and you've had it. You're ready to close the connection on your socket descriptor. This is easy. You can just use the regular Unix file descriptor **close()** function:

```
close(sockfd);
```

This will prevent any more reads and writes to the socket. Anyone attempting to read or write the socket on the remote end will receive an error.

Just in case you want a little more control over how the socket closes, you can use the **shutdown()** function. It allows you to cut off communication in a certain direction, or both ways (just like **close()** does.) Synopsis:

```
int shutdown(int sockfd, int how);
```

sockfd is the socket file descriptor you want to shutdown, and *how* is one of the following:

0	Further receives are disallowed
1	Further sends are disallowed
2	Further sends and receives are disallowed (like **close()**)

shutdown() returns 0 on success, and -1 on error (with *errno* set accordingly.)

If you deign to use **shutdown()** on unconnected datagram sockets, it will simply make the socket unavailable for further **send()** and **recv()** calls (remember that you can use these if you **connect()** your datagram socket.)

It's important to note that **shutdown()** doesn't actually close the file descriptor—it just changes its usability. To free a socket descriptor, you need to use **close()**.

Nothing to it.

(Except to remember that if you're using Windows and Winsock that you should call **closesocket()** instead of **close()**.)

5.10. getpeername()—Who are you?

This function is so easy.

It's so easy, I almost didn't give it its own section. But here it is anyway.

The function **getpeername()** will tell you who is at the other end of a connected stream socket. The synopsis:

```
#include <sys/socket.h>

int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

sockfd is the descriptor of the connected stream socket, *addr* is a pointer to a `struct sockaddr` (or a `struct sockaddr_in`) that will hold the information about the other side of the connection, and *addrlen* is a pointer to an `int`, that should be initialized to `sizeof *addr` or `sizeof(struct sockaddr)`.

The function returns -1 on error and sets *errno* accordingly.

Once you have their address, you can use **inet_ntop()**, **getnameinfo()**, or **gethostbyaddr()** to print or get more information. No, you can't get their login name. (Ok, ok. If the other computer is running an ident daemon, this is possible. This, however, is beyond the scope of this document. Check out RFC 1413[19] for more info.)

5.11. gethostname()—Who am I?

Even easier than **getpeername()** is the function **gethostname()**. It returns the name of the computer that your program is running on. The name can then be used by **gethostbyname()**, below, to determine the IP address of your local machine.

What could be more fun? I could think of a few things, but they don't pertain to socket programming. Anyway, here's the breakdown:

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

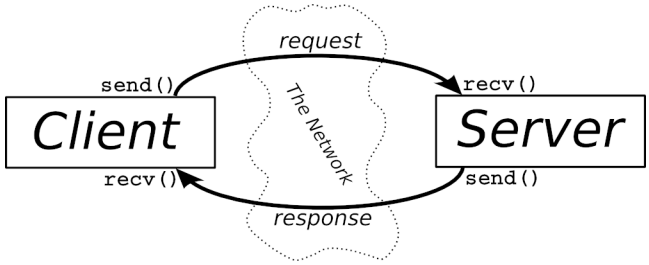
The arguments are simple: *hostname* is a pointer to an array of chars that will contain the hostname upon the function's return, and *size* is the length in bytes of the *hostname* array.

The function returns 0 on successful completion, and -1 on error, setting *errno* as usual.

19. http://tools.ietf.org/html/rfc1413

6. Client-Server Background

It's a client-server world, baby. Just about everything on the network deals with client processes talking to server processes and vice-versa. Take **telnet**, for instance. When you connect to a remote host on port 23 with telnet (the client), a program on that host (called **telnetd**, the server) springs to life. It handles the incoming telnet connection, sets you up with a login prompt, etc.



Client-Server Interaction.

The exchange of information between client and server is summarized in the above diagram.

Note that the client-server pair can speak `SOCK_STREAM`, `SOCK_DGRAM`, or anything else (as long as they're speaking the same thing.) Some good examples of client-server pairs are **telnet**/**telnetd**, **ftp**/**ftpd**, or **Firefox**/**Apache**. Every time you use **ftp**, there's a remote program, **ftpd**, that serves you.

Often, there will only be one server on a machine, and that server will handle multiple clients using **`fork()`**. The basic routine is: server will wait for a connection, **`accept()`** it, and **`fork()`** a child process to handle it. This is what our sample server does in the next section.

6.1. A Simple Stream Server

All this server does is send the string "`Hello, World!\n`" out over a stream connection. All you need to do to test this server is run it in one window, and telnet to it from another with:

```
$ telnet remotehostname 3490
```

where `remotehostname` is the name of the machine you're running it on.

The server code[20]:

```
/*
** server.c -- a stream socket server demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define PORT "3490"  // the port users will be connecting to

#define BACKLOG 10   // how many pending connections queue will hold

void sigchld_handler(int s)
{
    while(waitpid(-1, NULL, WNOHANG) > 0);
}
```

20. `http://beej.us/guide/bgnet/examples/server.c`

```
// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
    int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
    struct addrinfo hints, *servinfo, *p;
    struct sockaddr_storage their_addr; // connector's address information
    socklen_t sin_size;
    struct sigaction sa;
    int yes=1;
    char s[INET6_ADDRSTRLEN];
    int rv;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // use my IP

    if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and bind to the first we can
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("server: socket");
            continue;
        }

        if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
                sizeof(int)) == -1) {
            perror("setsockopt");
            exit(1);
        }

        if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("server: bind");
            continue;
        }

        break;
    }

    if (p == NULL)  {
        fprintf(stderr, "server: failed to bind\n");
        return 2;
    }

    freeaddrinfo(servinfo); // all done with this structure

    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }
```

```
    sa.sa_handler = sigchld_handler; // reap all dead processes
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    printf("server: waiting for connections...\n");

    while(1) {  // main accept() loop
        sin_size = sizeof their_addr;
        new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
        if (new_fd == -1) {
            perror("accept");
            continue;
        }

        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s);
        printf("server: got connection from %s\n", s);

        if (!fork()) { // this is the child process
            close(sockfd); // child doesn't need the listener
            if (send(new_fd, "Hello, world!", 13, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }
        close(new_fd);  // parent doesn't need this
    }

    return 0;
}
```

In case you're curious, I have the code in one big **`main()`** function for (I feel) syntactic clarity. Feel free to split it into smaller functions if it makes you feel better.

(Also, this whole **`sigaction()`** thing might be new to you—that's ok. The code that's there is responsible for reaping zombie processes that appear as the **`fork()`**ed child processes exit. If you make lots of zombies and don't reap them, your system administrator will become agitated.)

You can get the data from this server by using the client listed in the next section.

6.2. A Simple Stream Client

This guy's even easier than the server. All this client does is connect to the host you specify on the command line, port 3490. It gets the string that the server sends.

The client source[21]:

```
/*
** client.c -- a stream socket client demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#include <arpa/inet.h>
```

21. `http://beej.us/guide/bgnet/examples/client.c`

```
#define PORT "3490" // the port client will be connecting to

#define MAXDATASIZE 100 // max number of bytes we can get at once

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct addrinfo hints, *servinfo, *p;
    int rv;
    char s[INET6_ADDRSTRLEN];

    if (argc != 2) {
        fprintf(stderr,"usage: client hostname\n");
        exit(1);
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and connect to the first we can
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("client: socket");
            continue;
        }

        if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("client: connect");
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "client: failed to connect\n");
        return 2;
    }

    inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr),
            s, sizeof s);
    printf("client: connecting to %s\n", s);

    freeaddrinfo(servinfo); // all done with this structure

    if ((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
        perror("recv");
```

```
        exit(1);
    }

    buf[numbytes] = '\0';

    printf("client: received '%s'\n",buf);

    close(sockfd);

    return 0;
}
```

Notice that if you don't run the server before you run the client, **`connect()`** returns "Connection refused". Very useful.

6.3. Datagram Sockets

We've already covered the basics of UDP datagram sockets with our discussion of **`sendto()`** and **`recvfrom()`**, above, so I'll just present a couple of sample programs: *`talker.c`* and *`listener.c`*.

listener sits on a machine waiting for an incoming packet on port 4950. **talker** sends a packet to that port, on the specified machine, that contains whatever the user enters on the command line.

Here is the source for *`listener.c`*[22]:

```
/*
** listener.c -- a datagram sockets "server" demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define MYPORT "4950"   // the port users will be connecting to

#define MAXBUFLEN 100

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;
    int rv;
    int numbytes;
    struct sockaddr_storage their_addr;
    char buf[MAXBUFLEN];
    socklen_t addr_len;
    char s[INET6_ADDRSTRLEN];

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC; // set to AF_INET to force IPv4
```

22. `http://beej.us/guide/bgnet/examples/listener.c`

```
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_flags = AI_PASSIVE; // use my IP

    if ((rv = getaddrinfo(NULL, MYPORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and bind to the first we can
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("listener: socket");
            continue;
        }

        if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("listener: bind");
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "listener: failed to bind socket\n");
        return 2;
    }

    freeaddrinfo(servinfo);

    printf("listener: waiting to recvfrom...\n");

    addr_len = sizeof their_addr;
    if ((numbytes = recvfrom(sockfd, buf, MAXBUFLEN-1 , 0,
        (struct sockaddr *)&their_addr, &addr_len)) == -1) {
        perror("recvfrom");
        exit(1);
    }

    printf("listener: got packet from %s\n",
        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s));
    printf("listener: packet is %d bytes long\n", numbytes);
    buf[numbytes] = '\0';
    printf("listener: packet contains \"%s\"\n", buf);

    close(sockfd);

    return 0;
}
```

Notice that in our call to **`getaddrinfo()`** we're finally using `SOCK_DGRAM`. Also, note that there's no need to **`listen()`** or **`accept()`**. This is one of the perks of using unconnected datagram sockets!

Next comes the source for *`talker.c`*[23]:

```
/*
** talker.c -- a datagram "client" demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

23. `http://beej.us/guide/bgnet/examples/talker.c`

```
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT "4950"   // the port users will be connecting to

int main(int argc, char *argv[])
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;
    int rv;
    int numbytes;

    if (argc != 3) {
        fprintf(stderr,"usage: talker hostname message\n");
        exit(1);
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;

    if ((rv = getaddrinfo(argv[1], SERVERPORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and make a socket
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("talker: socket");
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "talker: failed to bind socket\n");
        return 2;
    }

    if ((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
             p->ai_addr, p->ai_addrlen)) == -1) {
        perror("talker: sendto");
        exit(1);
    }

    freeaddrinfo(servinfo);

    printf("talker: sent %d bytes to %s\n", numbytes, argv[1]);
    close(sockfd);

    return 0;
}
```

And that's all there is to it! Run **listener** on some machine, then run **talker** on another. Watch them communicate! Fun G-rated excitement for the entire nuclear family!

You don't even have to run the server this time! You can run **talker** by itself, and it just happily fires packets off into the ether where they disappear if no one is ready with a **`recvfrom()`** on the other side. Remember: data sent using UDP datagram sockets isn't guaranteed to arrive!

Except for one more tiny detail that I've mentioned many times in the past: connected datagram sockets. I need to talk about this here, since we're in the datagram section of the document. Let's say that **talker** calls **`connect()`** and specifies the **listener**'s address. From that point on, **talker** may only sent to and receive from the address specified by **`connect()`**. For this reason, you don't have to use **`sendto()`** and **`recvfrom()`**; you can simply use **`send()`** and **`recv()`**.

7. Slightly Advanced Techniques

These aren't *really* advanced, but they're getting out of the more basic levels we've already covered. In fact, if you've gotten this far, you should consider yourself fairly accomplished in the basics of Unix network programming! Congratulations!

So here we go into the brave new world of some of the more esoteric things you might want to learn about sockets. Have at it!

7.1. Blocking

Blocking. You've heard about it—now what the heck is it? In a nutshell, “block” is techie jargon for “sleep”. You probably noticed that when you run **listener**, above, it just sits there until a packet arrives. What happened is that it called **`recvfrom()`**, there was no data, and so **`recvfrom()`** is said to “block” (that is, sleep there) until some data arrives.

Lots of functions block. **`accept()`** blocks. All the **`recv()`** functions block. The reason they can do this is because they're allowed to. When you first create the socket descriptor with **`socket()`**, the kernel sets it to blocking. If you don't want a socket to be blocking, you have to make a call to **`fcntl()`**:

```
#include <unistd.h>
#include <fcntl.h>
.
.
.
sockfd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
.
.
.
```

By setting a socket to non-blocking, you can effectively “poll” the socket for information. If you try to read from a non-blocking socket and there's no data there, it's not allowed to block—it will return `-1` and *`errno`* will be set to `EWOULDBLOCK`.

Generally speaking, however, this type of polling is a bad idea. If you put your program in a busy-wait looking for data on the socket, you'll suck up CPU time like it was going out of style. A more elegant solution for checking to see if there's data waiting to be read comes in the following section on **`select()`**.

7.2. `select()`—Synchronous I/O Multiplexing

This function is somewhat strange, but it's very useful. Take the following situation: you are a server and you want to listen for incoming connections as well as keep reading from the connections you already have.

No problem, you say, just an **`accept()`** and a couple of **`recv()`**s. Not so fast, buster! What if you're blocking on an **`accept()`** call? How are you going to **`recv()`** data at the same time? “Use non-blocking sockets!” No way! You don't want to be a CPU hog. What, then?

`select()` gives you the power to monitor several sockets at the same time. It'll tell you which ones are ready for reading, which are ready for writing, and which sockets have raised exceptions, if you really want to know that.

This being said, in modern times **`select()`**, though very portable, is one of the slowest methods for monitoring sockets. One possible alternative is libevent[24], or something similar, that encapsulates all the system-dependent stuff involved with getting socket notifications.

Without any further ado, I'll offer the synopsis of **`select()`**:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds,
```

24. `http://www.monkey.org/~provos/libevent/`

```
           fd_set *exceptfds, struct timeval *timeout);
```

The function monitors "sets" of file descriptors; in particular *readfds*, *writefds*, and *exceptfds*. If you want to see if you can read from standard input and some socket descriptor, *sockfd*, just add the file descriptors `0` and *sockfd* to the set *readfds*. The parameter *numfds* should be set to the values of the highest file descriptor plus one. In this example, it should be set to *sockfd+1*, since it is assuredly higher than standard input (`0`).

When **`select()`** returns, *readfds* will be modified to reflect which of the file descriptors you selected which is ready for reading. You can test them with the macro **`FD_ISSET()`**, below.

Before progressing much further, I'll talk about how to manipulate these sets. Each set is of the type `fd_set`. The following macros operate on this type:

`FD_SET(int fd, fd_set *set);`	Add *fd* to the *set*.
`FD_CLR(int fd, fd_set *set);`	Remove *fd* from the *set*.
`FD_ISSET(int fd, fd_set *set);`	Return true if *fd* is in the *set*.
`FD_ZERO(fd_set *set);`	Clear all entries from the *set*.

Finally, what is this weirded out `struct timeval`? Well, sometimes you don't want to wait forever for someone to send you some data. Maybe every 96 seconds you want to print "Still Going..." to the terminal even though nothing has happened. This time structure allows you to specify a timeout period. If the time is exceeded and **`select()`** still hasn't found any ready file descriptors, it'll return so you can continue processing.

The `struct timeval` has the follow fields:

```
struct timeval {
    int tv_sec;     // seconds
    int tv_usec;    // microseconds
};
```

Just set *tv_sec* to the number of seconds to wait, and set *tv_usec* to the number of microseconds to wait. Yes, that's *micro*seconds, not milliseconds. There are 1,000 microseconds in a millisecond, and 1,000 milliseconds in a second. Thus, there are 1,000,000 microseconds in a second. Why is it "usec"? The "u" is supposed to look like the Greek letter μ (Mu) that we use for "micro". Also, when the function returns, *timeout might* be updated to show the time still remaining. This depends on what flavor of Unix you're running.

Yay! We have a microsecond resolution timer! Well, don't count on it. You'll probably have to wait some part of your standard Unix timeslice no matter how small you set your `struct timeval`.

Other things of interest: If you set the fields in your `struct timeval` to `0`, **`select()`** will timeout immediately, effectively polling all the file descriptors in your sets. If you set the parameter *timeout* to NULL, it will never timeout, and will wait until the first file descriptor is ready. Finally, if you don't care about waiting for a certain set, you can just set it to NULL in the call to **`select()`**.

The following code snippet[25] waits 2.5 seconds for something to appear on standard input:

```
/*
** select.c -- a select() demo
*/

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN 0  // file descriptor for standard input

int main(void)
{
    struct timeval tv;
    fd_set readfds;
```

25. `http://beej.us/guide/bgnet/examples/select.c`

```
    tv.tv_sec = 2;
    tv.tv_usec = 500000;

    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);

    // don't care about writefds and exceptfds:
    select(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");
    else
        printf("Timed out.\n");

    return 0;
}
```

If you're on a line buffered terminal, the key you hit should be RETURN or it will time out anyway.

Now, some of you might think this is a great way to wait for data on a datagram socket—and you are right: it *might* be. Some Unices can use select in this manner, and some can't. You should see what your local man page says on the matter if you want to attempt it.

Some Unices update the time in your `struct timeval` to reflect the amount of time still remaining before a timeout. But others do not. Don't rely on that occurring if you want to be portable. (Use **`gettimeofday()`** if you need to track time elapsed. It's a bummer, I know, but that's the way it is.)

What happens if a socket in the read set closes the connection? Well, in that case, **`select()`** returns with that socket descriptor set as "ready to read". When you actually do **`recv()`** from it, **`recv()`** will return `0`. That's how you know the client has closed the connection.

One more note of interest about **`select()`**: if you have a socket that is **`listen()`**ing, you can check to see if there is a new connection by putting that socket's file descriptor in the *`readfds`* set.

And that, my friends, is a quick overview of the almighty **`select()`** function.

But, by popular demand, here is an in-depth example. Unfortunately, the difference between the dirt-simple example, above, and this one here is significant. But have a look, then read the description that follows it.

This program[26] acts like a simple multi-user chat server. Start it running in one window, then **telnet** to it ("**telnet hostname 9034**") from multiple other windows. When you type something in one **telnet** session, it should appear in all the others.

```
/*
** selectserver.c -- a cheezy multiperson chat server
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define PORT "9034"   // port we're listening on

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }
```

26. `http://beej.us/guide/bgnet/examples/selectserver.c`

```
    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
    fd_set master;    // master file descriptor list
    fd_set read_fds;  // temp file descriptor list for select()
    int fdmax;        // maximum file descriptor number

    int listener;     // listening socket descriptor
    int newfd;        // newly accept()ed socket descriptor
    struct sockaddr_storage remoteaddr; // client address
    socklen_t addrlen;

    char buf[256];    // buffer for client data
    int nbytes;

    char remoteIP[INET6_ADDRSTRLEN];

    int yes=1;        // for setsockopt() SO_REUSEADDR, below
    int i, j, rv;

    struct addrinfo hints, *ai, *p;

    FD_ZERO(&master);    // clear the master and temp sets
    FD_ZERO(&read_fds);

    // get us a socket and bind it
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
        fprintf(stderr, "selectserver: %s\n", gai_strerror(rv));
        exit(1);
    }

    for(p = ai; p != NULL; p = p->ai_next) {
        listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
        if (listener < 0) {
            continue;
        }

        // lose the pesky "address already in use" error message
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

        if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
            close(listener);
            continue;
        }

        break;
    }

    // if we got here, it means we didn't get bound
    if (p == NULL) {
        fprintf(stderr, "selectserver: failed to bind\n");
        exit(2);
    }

    freeaddrinfo(ai); // all done with this

    // listen
    if (listen(listener, 10) == -1) {
        perror("listen");
        exit(3);
    }
```

```
    // add the listener to the master set
    FD_SET(listener, &master);

    // keep track of the biggest file descriptor
    fdmax = listener; // so far, it's this one

    // main loop
    for(;;) {
        read_fds = master; // copy it
        if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(4);
        }

        // run through the existing connections looking for data to read
        for(i = 0; i <= fdmax; i++) {
            if (FD_ISSET(i, &read_fds)) { // we got one!!
                if (i == listener) {
                    // handle new connections
                    addrlen = sizeof remoteaddr;
                    newfd = accept(listener,
                        (struct sockaddr *)&remoteaddr,
                        &addrlen);

                    if (newfd == -1) {
                        perror("accept");
                    } else {
                        FD_SET(newfd, &master); // add to master set
                        if (newfd > fdmax) {    // keep track of the max
                            fdmax = newfd;
                        }
                        printf("selectserver: new connection from %s on "
                            "socket %d\n",
                            inet_ntop(remoteaddr.ss_family,
                                get_in_addr((struct sockaddr*)&remoteaddr),
                                remoteIP, INET6_ADDRSTRLEN),
                            newfd);
                    }
                } else {
                    // handle data from a client
                    if ((nbytes = recv(i, buf, sizeof buf, 0)) <= 0) {
                        // got error or connection closed by client
                        if (nbytes == 0) {
                            // connection closed
                            printf("selectserver: socket %d hung up\n", i);
                        } else {
                            perror("recv");
                        }
                        close(i); // bye!
                        FD_CLR(i, &master); // remove from master set
                    } else {
                        // we got some data from a client
                        for(j = 0; j <= fdmax; j++) {
                            // send to everyone!
                            if (FD_ISSET(j, &master)) {
                                // except the listener and ourselves
                                if (j != listener && j != i) {
                                    if (send(j, buf, nbytes, 0) == -1) {
                                        perror("send");
                                    }
                                }
                            }
                        }
                    }
                } // END handle data from client
            } // END got new incoming connection
```

```
        } // END looping through file descriptors
    } // END for(;;)--and you thought it would never end!

    return 0;
}
```

Notice I have two file descriptor sets in the code: *`master`* and *`read_fds`*. The first, *`master`*, holds all the socket descriptors that are currently connected, as well as the socket descriptor that is listening for new connections.

The reason I have the *`master`* set is that **`select()`** actually *changes* the set you pass into it to reflect which sockets are ready to read. Since I have to keep track of the connections from one call of **`select()`** to the next, I must store these safely away somewhere. At the last minute, I copy the *`master`* into the *`read_fds`*, and then call **`select()`**.

But doesn't this mean that every time I get a new connection, I have to add it to the *`master`* set? Yup! And every time a connection closes, I have to remove it from the *`master`* set? Yes, it does.

Notice I check to see when the *`listener`* socket is ready to read. When it is, it means I have a new connection pending, and I **`accept()`** it and add it to the *`master`* set. Similarly, when a client connection is ready to read, and **`recv()`** returns `0`, I know the client has closed the connection, and I must remove it from the *`master`* set.

If the client **`recv()`** returns non-zero, though, I know some data has been received. So I get it, and then go through the *`master`* list and send that data to all the rest of the connected clients.

And that, my friends, is a less-than-simple overview of the almighty **`select()`** function.

In addition, here is a bonus afterthought: there is another function called **`poll()`** which behaves much the same way **`select()`** does, but with a different system for managing the file descriptor sets. Check it out!

7.3. Handling Partial `send()`s

Remember back in the section about **`send()`**, above, when I said that **`send()`** might not send all the bytes you asked it to? That is, you want it to send 512 bytes, but it returns 412. What happened to the remaining 100 bytes?

Well, they're still in your little buffer waiting to be sent out. Due to circumstances beyond your control, the kernel decided not to send all the data out in one chunk, and now, my friend, it's up to you to get the data out there.

You could write a function like this to do it, too:

```
#include <sys/types.h>
#include <sys/socket.h>

int sendall(int s, char *buf, int *len)
{
    int total = 0;        // how many bytes we've sent
    int bytesleft = *len; // how many we have left to send
    int n;

    while(total < *len) {
        n = send(s, buf+total, bytesleft, 0);
        if (n == -1) { break; }
        total += n;
        bytesleft -= n;
    }

    *len = total; // return number actually sent here

    return n==-1?-1:0; // return -1 on failure, 0 on success
}
```

In this example, *`s`* is the socket you want to send the data to, *`buf`* is the buffer containing the data, and *`len`* is a pointer to an `int` containing the number of bytes in the buffer.

The function returns `-1` on error (and *`errno`* is still set from the call to **`send()`**.) Also, the number of bytes actually sent is returned in *`len`*. This will be the same number of bytes you asked it to send,

unless there was an error. **`sendall()`** will do it's best, huffing and puffing, to send the data out, but if there's an error, it gets back to you right away.

For completeness, here's a sample call to the function:

```
char buf[10] = "Beej!";
int len;

len = strlen(buf);
if (sendall(s, buf, &len) == -1) {
    perror("sendall");
    printf("We only sent %d bytes because of the error!\n", len);
}
```

What happens on the receiver's end when part of a packet arrives? If the packets are variable length, how does the receiver know when one packet ends and another begins? Yes, real-world scenarios are a royal pain in the donkeys. You probably have to *encapsulate* (remember that from the data encapsulation section way back there at the beginning?) Read on for details!

7.4. Serialization—How to Pack Data

It's easy enough to send text data across the network, you're finding, but what happens if you want to send some "binary" data like `int`s or `float`s? It turns out you have a few options.

1. Convert the number into text with a function like **`sprintf()`**, then send the text. The receiver will parse the text back into a number using a function like **`strtol()`**.
2. Just send the data raw, passing a pointer to the data to **`send()`**.
3. Encode the number into a portable binary form. The receiver will decode it.

Sneak preview! Tonight only!

[*Curtain raises*]

Beej says, "I prefer Method Three, above!"

[*THE END*]

(Before I begin this section in earnest, I should tell you that there are libraries out there for doing this, and rolling your own and remaining portable and error-free is quite a challenge. So hunt around and do your homework before deciding to implement this stuff yourself. I include the information here for those curious about how things like this work.)

Actually all the methods, above, have their drawbacks and advantages, but, like I said, in general, I prefer the third method. First, though, let's talk about some of the drawbacks and advantages to the other two.

The first method, encoding the numbers as text before sending, has the advantage that you can easily print and read the data that's coming over the wire. Sometimes a human-readable protocol is excellent to use in a non-bandwidth-intensive situation, such as with Internet Relay Chat (IRC)[27]. However, it has the disadvantage that it is slow to convert, and the results almost always take up more space than the original number!

Method two: passing the raw data. This one is quite easy (but dangerous!): just take a pointer to the data to send, and call send with it.

```
double d = 3490.15926535;

send(s, &d, sizeof d, 0);  /* DANGER--non-portable! */
```

The receiver gets it like this:

```
double d;

recv(s, &d, sizeof d, 0);  /* DANGER--non-portable! */
```

27. `http://en.wikipedia.org/wiki/Internet_Relay_Chat`

Fast, simple—what's not to like? Well, it turns out that not all architectures represent a `double` (or `int` for that matter) with the same bit representation or even the same byte ordering! The code is decidedly non-portable. (Hey—maybe you don't need portability, in which case this is nice and fast.)

When packing integer types, we've already seen how the **`htons()`**-class of functions can help keep things portable by transforming the numbers into Network Byte Order, and how that's the Right Thing to do. Unfortunately, there are no similar functions for `float` types. Is all hope lost?

Fear not! (Were you afraid there for a second? No? Not even a little bit?) There is something we can do: we can pack (or "marshal", or "serialize", or one of a thousand million other names) the data into a known binary format that the receiver can unpack on the remote side.

What do I mean by "known binary format"? Well, we've already seen the **`htons()`** example, right? It changes (or "encodes", if you want to think of it that way) a number from whatever the host format is into Network Byte Order. To reverse (unencode) the number, the receiver calls **`ntohs()`**.

But didn't I just get finished saying there wasn't any such function for other non-integer types? Yes. I did. And since there's no standard way in C to do this, it's a bit of a pickle (that a gratuitous pun there for you Python fans).

The thing to do is to pack the data into a known format and send that over the wire for decoding. For example, to pack `floats`, here's something quick and dirty with plenty of room for improvement:[28]

```
#include <stdint.h>

uint32_t htonf(float f)
{
    uint32_t p;
    uint32_t sign;

    if (f < 0) { sign = 1; f = -f; }
    else { sign = 0; }

    p = ((((uint32_t)f)&0x7fff)<<16) | (sign<<31); // whole part and sign
    p |= (uint32_t)(((f - (int)f) * 65536.0f))&0xffff; // fraction

    return p;
}

float ntohf(uint32_t p)
{
    float f = ((p>>16)&0x7fff); // whole part
    f += (p&0xffff) / 65536.0f; // fraction

    if (((p>>31)&0x1) == 0x1) { f = -f; } // sign bit set

    return f;
}
```

The above code is sort of a naive implementation that stores a `float` in a 32-bit number. The high bit (31) is used to store the sign of the number ("1" means negative), and the next seven bits (30-16) are used to store the whole number portion of the `float`. Finally, the remaining bits (15-0) are used to store the fractional portion of the number.

Usage is fairly straightforward:

```
#include <stdio.h>

int main(void)
{
    float f = 3.1415926, f2;
    uint32_t netf;

    netf = htonf(f);  // convert to "network" form
    f2 = ntohf(netf); // convert back to test

    printf("Original: %f\n", f);        // 3.141593
```

28. `http://beej.us/guide/bgnet/examples/pack.c`

```
    printf(" Network: 0x%08X\n", netf); // 0x0003243F
    printf("Unpacked: %f\n", f2);       // 3.141586

    return 0;
}
```

On the plus side, it's small, simple, and fast. On the minus side, it's not an efficient use of space and the range is severely restricted—try storing a number greater-than 32767 in there and it won't be very happy! You can also see in the above example that the last couple decimal places are not correctly preserved.

What can we do instead? Well, *The* Standard for storing floating point numbers is known as IEEE-754[29]. Most computers use this format internally for doing floating point math, so in those cases, strictly speaking, conversion wouldn't need to be done. But if you want your source code to be portable, that's an assumption you can't necessarily make. (On the other hand, if you want things to be fast, you should optimize this out on platforms that don't need to do it! That's what **`htons()`** and its ilk do.)

Here's some code that encodes floats and doubles into IEEE-754 format[30]. (Mostly—it doesn't encode NaN or Infinity, but it could be modified to do that.)

```
#define pack754_32(f) (pack754((f), 32, 8))
#define pack754_64(f) (pack754((f), 64, 11))
#define unpack754_32(i) (unpack754((i), 32, 8))
#define unpack754_64(i) (unpack754((i), 64, 11))

uint64_t pack754(long double f, unsigned bits, unsigned expbits)
{
    long double fnorm;
    int shift;
    long long sign, exp, significand;
    unsigned significandbits = bits - expbits - 1; // -1 for sign bit

    if (f == 0.0) return 0; // get this special case out of the way

    // check sign and begin normalization
    if (f < 0) { sign = 1; fnorm = -f; }
    else { sign = 0; fnorm = f; }

    // get the normalized form of f and track the exponent
    shift = 0;
    while(fnorm >= 2.0) { fnorm /= 2.0; shift++; }
    while(fnorm < 1.0) { fnorm *= 2.0; shift--; }
    fnorm = fnorm - 1.0;

    // calculate the binary form (non-float) of the significand data
    significand = fnorm * ((1LL<<significandbits) + 0.5f);

    // get the biased exponent
    exp = shift + ((1<<(expbits-1)) - 1); // shift + bias

    // return the final answer
    return (sign<<(bits-1)) | (exp<<(bits-expbits-1)) | significand;
}

long double unpack754(uint64_t i, unsigned bits, unsigned expbits)
{
    long double result;
    long long shift;
    unsigned bias;
    unsigned significandbits = bits - expbits - 1; // -1 for sign bit

    if (i == 0) return 0.0;

    // pull the significand
```

29. http://en.wikipedia.org/wiki/IEEE_754
30. http://beej.us/guide/bgnet/examples/ieee754.c

```
    result = (i&((1LL<<significandbits)-1)); // mask
    result /= (1LL<<significandbits); // convert back to float
    result += 1.0f; // add the one back on

    // deal with the exponent
    bias = (1<<(expbits-1)) - 1;
    shift = ((i>>significandbits)&((1LL<<expbits)-1)) - bias;
    while(shift > 0) { result *= 2.0; shift--; }
    while(shift < 0) { result /= 2.0; shift++; }

    // sign it
    result *= (i>>(bits-1))&1? -1.0: 1.0;

    return result;
}
```

I put some handy macros up there at the top for packing and unpacking 32-bit (probably a `float`) and 64-bit (probably a `double`) numbers, but the **`pack754()`** function could be called directly and told to encode *`bits`*-worth of data (*`expbits`* of which are reserved for the normalized number's exponent.)

Here's sample usage:

```
#include <stdio.h>
#include <stdint.h> // defines uintN_t types
#include <inttypes.h> // defines PRIx macros

int main(void)
{
    float f = 3.1415926, f2;
    double d = 3.14159265358979323, d2;
    uint32_t fi;
    uint64_t di;

    fi = pack754_32(f);
    f2 = unpack754_32(fi);

    di = pack754_64(d);
    d2 = unpack754_64(di);

    printf("float before : %.7f\n", f);
    printf("float encoded: 0x%08" PRIx32 "\n", fi);
    printf("float after  : %.7f\n\n", f2);

    printf("double before : %.20lf\n", d);
    printf("double encoded: 0x%016" PRIx64 "\n", di);
    printf("double after  : %.20lf\n", d2);

    return 0;
}
```

The above code produces this output:

```
float before : 3.1415925
float encoded: 0x40490FDA
float after  : 3.1415925

double before : 3.14159265358979311600
double encoded: 0x400921FB54442D18
double after  : 3.14159265358979311600
```

Another question you might have is how do you pack `structs`? Unfortunately for you, the compiler is free to put padding all over the place in a `struct`, and that means you can't portably send the whole thing over the wire in one chunk. (Aren't you getting sick of hearing “can't do this”, “can't do that”? Sorry! To quote a friend, “Whenever anything goes wrong, I always blame Microsoft.” This one might not be Microsoft's fault, admittedly, but my friend's statement is completely true.)

Back to it: the best way to send the `struct` over the wire is to pack each field independently and then unpack them into the `struct` when they arrive on the other side.

That's a lot of work, is what you're thinking. Yes, it is. One thing you can do is write a helper function to help pack the data for you. It'll be fun! Really!

In the book "The Practice of Programming[31]" by Kernighan and Pike, they implement **`printf()`**-like functions called **`pack()`** and **`unpack()`** that do exactly this. I'd link to them, but apparently those functions aren't online with the rest of the source from the book.

(The Practice of Programming is an excellent read. Zeus saves a kitten every time I recommend it.)

At this point, I'm going to drop a pointer to the BSD-licensed Typed Parameter Language C API[32] which I've never used, but looks completely respectable. Python and Perl programmers will want to check out their language's **`pack()`** and **`unpack()`** functions for accomplishing the same thing. And Java has a big-ol' Serializable interface that can be used in a similar way.

But if you want to write your own packing utility in C, K&P's trick is to use variable argument lists to make **`printf()`**-like functions to build the packets. Here's a version I cooked up[33] on my own based on that which hopefully will be enough to give you an idea of how such a thing can work.

(This code references the **`pack754()`** functions, above. The **`packi*()`** functions operate like the familiar **`htons()`** family, except they pack into a `char` array instead of another integer.)

```
#include <ctype.h>
#include <stdarg.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>

// various bits for floating point types--
// varies for different architectures
typedef float float32_t;
typedef double float64_t;

/*
** packi16() -- store a 16-bit int into a char buffer (like htons())
*/
void packi16(unsigned char *buf, unsigned int i)
{
    *buf++ = i>>8; *buf++ = i;
}

/*
** packi32() -- store a 32-bit int into a char buffer (like htonl())
*/
void packi32(unsigned char *buf, unsigned long i)
{
    *buf++ = i>>24; *buf++ = i>>16;
    *buf++ = i>>8;  *buf++ = i;
}

/*
** unpacki16() -- unpack a 16-bit int from a char buffer (like ntohs())
*/
unsigned int unpacki16(unsigned char *buf)
{
    return (buf[0]<<8) | buf[1];
}

/*
** unpacki32() -- unpack a 32-bit int from a char buffer (like ntohl())
*/
unsigned long unpacki32(unsigned char *buf)
{
    return (buf[0]<<24) | (buf[1]<<16) | (buf[2]<<8) | buf[3];
}
```

31. http://cm.bell-labs.com/cm/cs/tpop/
32. http://tpl.sourceforge.net/
33. http://beej.us/guide/bgnet/examples/pack2.c

```
/*
** pack() -- store data dictated by the format string in the buffer
**
**  h - 16-bit              l - 32-bit
**  c - 8-bit char          f - float, 32-bit
**  s - string (16-bit length is automatically prepended)
*/
int32_t pack(unsigned char *buf, char *format, ...)
{
    va_list ap;
    int16_t h;
    int32_t l;
    int8_t c;
    float32_t f;
    char *s;
    int32_t size = 0, len;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'h': // 16-bit
            size += 2;
            h = (int16_t)va_arg(ap, int); // promoted
            packi16(buf, h);
            buf += 2;
            break;

        case 'l': // 32-bit
            size += 4;
            l = va_arg(ap, int32_t);
            packi32(buf, l);
            buf += 4;
            break;

        case 'c': // 8-bit
            size += 1;
            c = (int8_t)va_arg(ap, int); // promoted
            *buf++ = (c>>0)&0xff;
            break;

        case 'f': // float
            size += 4;
            f = (float32_t)va_arg(ap, double); // promoted
            l = pack754_32(f); // convert to IEEE 754
            packi32(buf, l);
            buf += 4;
            break;

        case 's': // string
            s = va_arg(ap, char*);
            len = strlen(s);
            size += len + 2;
            packi16(buf, len);
            buf += 2;
            memcpy(buf, s, len);
            buf += len;
            break;
        }
    }

    va_end(ap);

    return size;
}

/*
```

```
** unpack() -- unpack data dictated by the format string into the buffer
*/
void unpack(unsigned char *buf, char *format, ...)
{
    va_list ap;
    int16_t *h;
    int32_t *l;
    int32_t pf;
    int8_t *c;
    float32_t *f;
    char *s;
    int32_t len, count, maxstrlen=0;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'h': // 16-bit
            h = va_arg(ap, int16_t*);
            *h = unpacki16(buf);
            buf += 2;
            break;

        case 'l': // 32-bit
            l = va_arg(ap, int32_t*);
            *l = unpacki32(buf);
            buf += 4;
            break;

        case 'c': // 8-bit
            c = va_arg(ap, int8_t*);
            *c = *buf++;
            break;

        case 'f': // float
            f = va_arg(ap, float32_t*);
            pf = unpacki32(buf);
            buf += 4;
            *f = unpack754_32(pf);
            break;

        case 's': // string
            s = va_arg(ap, char*);
            len = unpacki16(buf);
            buf += 2;
            if (maxstrlen > 0 && len > maxstrlen) count = maxstrlen - 1;
            else count = len;
            memcpy(s, buf, count);
            s[count] = '\0';
            buf += len;
            break;

        default:
            if (isdigit(*format)) { // track max str len
                maxstrlen = maxstrlen * 10 + (*format-'0');
            }
        }

        if (!isdigit(*format)) maxstrlen = 0;
    }

    va_end(ap);
}
```

And here is a demonstration program[34] of the above code that packs some data into *buf* and then unpacks it into variables. Note that when calling **unpack()** with a string argument (format specifier "s"), it's wise to put a maximum length count in front of it to prevent a buffer overrun, e.g. "96s". Be wary when unpacking data you get over the network—a malicious user might send badly-constructed packets in an effort to attack your system!

```
#include <stdio.h>

// various bits for floating point types--
// varies for different architectures
typedef float float32_t;
typedef double float64_t;

int main(void)
{
    unsigned char buf[1024];
    int8_t magic;
    int16_t monkeycount;
    int32_t altitude;
    float32_t absurdityfactor;
    char *s = "Great unmitigated Zot!  You've found the Runestaff!";
    char s2[96];
    int16_t packetsize, ps2;

    packetsize = pack(buf, "chhlsf", (int8_t)'B', (int16_t)0, (int16_t)37,
            (int32_t)-5, s, (float32_t)-3490.6677);
    packi16(buf+1, packetsize); // store packet size in packet for kicks

    printf("packet is %" PRId32 " bytes\n", packetsize);

    unpack(buf, "chhl96sf", &magic, &ps2, &monkeycount, &altitude, s2,
        &absurdityfactor);

    printf("'%c' %" PRId32" %" PRId16 " %" PRId32
            " \"%s\" %f\n", magic, ps2, monkeycount,
            altitude, s2, absurdityfactor);

    return 0;
}
```

Whether you roll your own code or use someone else's, it's a good idea to have a general set of data packing routines for the sake of keeping bugs in check, rather than packing each bit by hand each time.

When packing the data, what's a good format to use? Excellent question. Fortunately, RFC 4506[35], the External Data Representation Standard, already defines binary formats for a bunch of different types, like floating point types, integer types, arrays, raw data, etc. I suggest conforming to that if you're going to roll the data yourself. But you're not obligated to. The Packet Police are not right outside your door. At least, I don't *think* they are.

In any case, encoding the data somehow or another before you send it is the right way of doing things!

7.5. Son of Data Encapsulation

What does it really mean to encapsulate data, anyway? In the simplest case, it means you'll stick a header on there with either some identifying information or a packet length, or both.

What should your header look like? Well, it's just some binary data that represents whatever you feel is necessary to complete your project.

Wow. That's vague.

Okay. For instance, let's say you have a multi-user chat program that uses SOCK_STREAMs. When a user types ("says") something, two pieces of information need to be transmitted to the server: what was said and who said it.

34. http://beej.us/guide/bgnet/examples/pack2.c
35. http://tools.ietf.org/html/rfc4506

So far so good? “What's the problem?” you're asking.

The problem is that the messages can be of varying lengths. One person named “tom” might say, “Hi”, and another person named “Benjamin” might say, “Hey guys what is up?”

So you **`send()`** all this stuff to the clients as it comes in. Your outgoing data stream looks like this:

```
t o m H i B e n j a m i n H e y g u y s w h a t i s u p ?
```

And so on. How does the client know when one message starts and another stops? You could, if you wanted, make all messages the same length and just call the **`sendall()`** we implemented, above. But that wastes bandwidth! We don't want to **`send()`** 1024 bytes just so “tom” can say “Hi”.

So we *encapsulate* the data in a tiny header and packet structure. Both the client and server know how to pack and unpack (sometimes referred to as “marshal” and “unmarshal”) this data. Don't look now, but we're starting to define a *protocol* that describes how a client and server communicate!

In this case, let's assume the user name is a fixed length of 8 characters, padded with `'\0'`. And then let's assume the data is variable length, up to a maximum of 128 characters. Let's have a look a sample packet structure that we might use in this situation:

1. `len` (1 byte, unsigned)—The total length of the packet, counting the 8-byte user name and chat data.
2. `name` (8 bytes)—The user's name, NUL-padded if necessary.
3. `chatdata` (*n*-bytes)—The data itself, no more than 128 bytes. The length of the packet should be calculated as the length of this data plus 8 (the length of the name field, above).

Why did I choose the 8-byte and 128-byte limits for the fields? I pulled them out of the air, assuming they'd be long enough. Maybe, though, 8 bytes is too restrictive for your needs, and you can have a 30-byte name field, or whatever. The choice is up to you.

Using the above packet definition, the first packet would consist of the following information (in hex and ASCII):

```
   0A     74 6F 6D 00 00 00 00 00      48 69
(length)  T  o  m    (padding)         H  i
```

And the second is similar:

```
   18     42 65 6E 6A 61 6D 69 6E      48 65 79 20 67 75 79 73 20 77 ...
(length)  B  e  n  j  a  m  i  n       H  e  y     g  u  y  s     w  ...
```

(The length is stored in Network Byte Order, of course. In this case, it's only one byte so it doesn't matter, but generally speaking you'll want all your binary integers to be stored in Network Byte Order in your packets.)

When you're sending this data, you should be safe and use a command similar to **`sendall()`**, above, so you know all the data is sent, even if it takes multiple calls to **`send()`** to get it all out.

Likewise, when you're receiving this data, you need to do a bit of extra work. To be safe, you should assume that you might receive a partial packet (like maybe we receive “`18 42 65 6E 6A`” from Benjamin, above, but that's all we get in this call to **`recv()`**). We need to call **`recv()`** over and over again until the packet is completely received.

But how? Well, we know the number of bytes we need to receive in total for the packet to be complete, since that number is tacked on the front of the packet. We also know the maximum packet size is 1+8+128, or 137 bytes (because that's how we defined the packet.)

There are actually a couple things you can do here. Since you know every packet starts off with a length, you can call **`recv()`** just to get the packet length. Then once you have that, you can call it again specifying exactly the remaining length of the packet (possibly repeatedly to get all the data) until you have the complete packet. The advantage of this method is that you only need a buffer large enough for one packet, while the disadvantage is that you need to call **`recv()`** at least twice to get all the data.

Another option is just to call **`recv()`** and say the amount you're willing to receive is the maximum number of bytes in a packet. Then whatever you get, stick it onto the back of a buffer, and finally check to see if the packet is complete. Of course, you might get some of the next packet, so you'll need to have room for that.

What you can do is declare an array big enough for two packets. This is your work array where you will reconstruct packets as they arrive.

Every time you **`recv()`** data, you'll append it into the work buffer and check to see if the packet is complete. That is, the number of bytes in the buffer is greater than or equal to the length specified in the header (+1, because the length in the header doesn't include the byte for the length itself.) If the number of bytes in the buffer is less than 1, the packet is not complete, obviously. You have to make a special case for this, though, since the first byte is garbage and you can't rely on it for the correct packet length.

Once the packet is complete, you can do with it what you will. Use it, and remove it from your work buffer.

Whew! Are you juggling that in your head yet? Well, here's the second of the one-two punch: you might have read past the end of one packet and onto the next in a single **`recv()`** call. That is, you have a work buffer with one complete packet, and an incomplete part of the next packet! Bloody heck. (But this is why you made your work buffer large enough to hold *two* packets—in case this happened!)

Since you know the length of the first packet from the header, and you've been keeping track of the number of bytes in the work buffer, you can subtract and calculate how many of the bytes in the work buffer belong to the second (incomplete) packet. When you've handled the first one, you can clear it out of the work buffer and move the partial second packet down the to front of the buffer so it's all ready to go for the next **`recv()`**.

(Some of you readers will note that actually moving the partial second packet to the beginning of the work buffer takes time, and the program can be coded to not require this by using a circular buffer. Unfortunately for the rest of you, a discussion on circular buffers is beyond the scope of this article. If you're still curious, grab a data structures book and go from there.)

I never said it was easy. Ok, I did say it was easy. And it is; you just need practice and pretty soon it'll come to you naturally. By Excalibur I swear it!

7.6. Broadcast Packets—Hello, World!

So far, this guide has talked about sending data from one host to one other host. But it is possible, I insist, that you can, with the proper authority, send data to multiple hosts *at the same time*!

With UDP (only UDP, not TCP) and standard IPv4, this is done through a mechanism called *broadcasting*. With IPv6, broadcasting isn't supported, and you have to resort to the often superior technique of *multicasting*, which, sadly I won't be discussing at this time. But enough of the starry-eyed future—we're stuck in the 32-bit present.

But wait! You can't just run off and start broadcasting willy-nilly; You have to set the socket option `SO_BROADCAST` before you can send a broadcast packet out on the network. It's like a one of those little plastic covers they put over the missile launch switch! That's just how much power you hold in your hands!

But seriously, though, there is a danger to using broadcast packets, and that is: every system that receives a broadcast packet must undo all the onion-skin layers of data encapsulation until it finds out what port the data is destined to. And then it hands the data over or discards it. In either case, it's a lot of work for each machine that receives the broadcast packet, and since it is all of them on the local network, that could be a lot of machines doing a lot of unnecessary work. When the game Doom first came out, this was a complaint about its network code.

Now, there is more than one way to skin a cat... wait a minute. Is there really more than one way to skin a cat? What kind of expression is that? Uh, and likewise, there is more than one way to send a broadcast packet. So, to get to the meat and potatoes of the whole thing: how do you specify the destination address for a broadcast message? There are two common ways:

1. Send the data to a specific subnet's broadcast address. This is the subnet's network number with all one-bits set for the host portion of the address. For instance, at home my network is 192.168.1.0, my netmask is 255.255.255.0, so the last byte of the address is my host number (because the first three bytes, according to the netmask, are the network number). So my broadcast address is 192.168.1.255. Under Unix, the **ifconfig** command will actually give you all this data. (If you're curious, the bitwise logic to get your broadcast address is *`network_number`* OR (NOT *`netmask`*).) You can send this type of broadcast packet to remote

networks as well as your local network, but you run the risk of the packet being dropped by the destination's router. (If they didn't drop it, then some random smurf could start flooding their LAN with broadcast traffic.)

2. Send the data to the "global" broadcast address. This is 255.255.255.255, aka `INADDR_BROADCAST`. Many machines will automatically bitwise AND this with your network number to convert it to a network broadcast address, but some won't. It varies. Routers do not forward this type of broadcast packet off your local network, ironically enough.

So what happens if you try to send data on the broadcast address without first setting the `SO_BROADCAST` socket option? Well, let's fire up good old **talker** and **listener** and see what happens.

```
$ talker 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ talker 192.168.1.255 foo
sendto: Permission denied
$ talker 255.255.255.255 foo
sendto: Permission denied
```

Yes, it's not happy at all...because we didn't set the `SO_BROADCAST` socket option. Do that, and now you can **`sendto()`** anywhere you want!

In fact, that's the *only difference* between a UDP application that can broadcast and one that can't. So let's take the old **talker** application and add one section that sets the `SO_BROADCAST` socket option. We'll call this program *`broadcaster.c`*[36]:

```
/*
** broadcaster.c -- a datagram "client" like talker.c, except
**                  this one can broadcast
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT 4950 // the port users will be connecting to

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // connector's address information
    struct hostent *he;
    int numbytes;
    int broadcast = 1;
    //char broadcast = '1'; // if that doesn't work, try this

    if (argc != 3) {
        fprintf(stderr,"usage: broadcaster hostname message\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {  // get the host info
        perror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
```

36. `http://beej.us/guide/bgnet/examples/broadcaster.c`

```
        perror("socket");
        exit(1);
    }

    // this call is what allows broadcast packets to be sent:
    if (setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcast,
        sizeof broadcast) == -1) {
        perror("setsockopt (SO_BROADCAST)");
        exit(1);
    }

    their_addr.sin_family = AF_INET;     // host byte order
    their_addr.sin_port = htons(SERVERPORT); // short, network byte order
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);

    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0,
             (struct sockaddr *)&their_addr, sizeof their_addr)) == -1) {
        perror("sendto");
        exit(1);
    }

    printf("sent %d bytes to %s\n", numbytes,
        inet_ntoa(their_addr.sin_addr));

    close(sockfd);

    return 0;
}
```

What's different between this and a “normal” UDP client/server situation? Nothing! (With the exception of the client being allowed to send broadcast packets in this case.) As such, go ahead and run the old UDP **listener** program in one window, and **broadcaster** in another. You should be now be able to do all those sends that failed, above.

```
$ broadcaster 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ broadcaster 192.168.1.255 foo
sent 3 bytes to 192.168.1.255
$ broadcaster 255.255.255.255 foo
sent 3 bytes to 255.255.255.255
```

And you should see **listener** responding that it got the packets. (If **listener** doesn't respond, it could be because it's bound to an IPv6 address. Try changing the `AF_UNSPEC` in *`listener.c`* to `AF_INET` to force IPv4.)

Well, that's kind of exciting. But now fire up **listener** on another machine next to you on the same network so that you have two copies going, one on each machine, and run **broadcaster** again with your broadcast address... Hey! Both **listener**s get the packet even though you only called **`sendto()`** once! Cool!

If the **listener** gets data you send directly to it, but not data on the broadcast address, it could be that you have a firewall on your local machine that is blocking the packets. (Yes, Pat and Bapper, thank you for realizing before I did that this is why my sample code wasn't working. I told you I'd mention you in the guide, and here you are. So *nyah*.)

Again, be careful with broadcast packets. Since every machine on the LAN will be forced to deal with the packet whether it **`recvfrom()`**s it or not, it can present quite a load to the entire computing network. They are definitely to be used sparingly and appropriately.

8. Common Questions

Where can I get those header files?

If you don't have them on your system already, you probably don't need them. Check the manual for your particular platform. If you're building for Windows, you only need to `#include <winsock.h>`.

What do I do when `bind()` reports "Address already in use"?

You have to use **`setsockopt()`** with the `SO_REUSEADDR` option on the listening socket. Check out the section on **`bind()`** and the section on **`select()`** for an example.

How do I get a list of open sockets on the system?

Use the **netstat**. Check the **man** page for full details, but you should get some good output just typing:

```
$ netstat
```

The only trick is determining which socket is associated with which program. `:-)`

How can I view the routing table?

Run the **route** command (in */sbin* on most Linuxes) or the command **netstat -r**.

How can I run the client and server programs if I only have one computer? Don't I need a network to write network programs?

Fortunately for you, virtually all machines implement a loopback network "device" that sits in the kernel and pretends to be a network card. (This is the interface listed as "`lo`" in the routing table.)

Pretend you're logged into a machine named "`goat`". Run the client in one window and the server in another. Or start the server in the background ("**server &**") and run the client in the same window. The upshot of the loopback device is that you can either **client goat** or **client localhost** (since "`localhost`" is likely defined in your */etc/hosts* file) and you'll have the client talking to the server without a network!

In short, no changes are necessary to any of the code to make it run on a single non-networked machine! Huzzah!

How can I tell if the remote side has closed connection?

You can tell because **`recv()`** will return `0`.

How do I implement a "ping" utility? What is ICMP? Where can I find out more about raw sockets and `SOCK_RAW`?

All your raw sockets questions will be answered in W. Richard Stevens' UNIX Network Programming books. Also, look in the *ping/* subdirectory in Stevens' UNIX Network Programming source code, available online[37].

How do I change or shorten the timeout on a call to `connect()`?

Instead of giving you exactly the same answer that W. Richard Stevens would give you, I'll just refer you to *lib/connect_nonb.c* in the UNIX Network Programming source code[38].

The gist of it is that you make a socket descriptor with **`socket()`**, set it to non-blocking, call **`connect()`**, and if all goes well **`connect()`** will return `-1` immediately and *errno* will be set to `EINPROGRESS`. Then you call **`select()`** with whatever timeout you want, passing the socket descriptor in both the read and write sets. If it doesn't timeout, it means the **`connect()`** call completed. At this point, you'll have to use **`getsockopt()`** with the `SO_ERROR` option to get the return value from the **`connect()`** call, which should be zero if there was no error.

37. `http://www.unpbook.com/src.html`
38. `http://www.unpbook.com/src.html`

Finally, you'll probably want to set the socket back to be blocking again before you start transferring data over it.

Notice that this has the added benefit of allowing your program to do something else while it's connecting, too. You could, for example, set the timeout to something low, like 500 ms, and update an indicator onscreen each timeout, then call **`select()`** again. When you've called **`select()`** and timed-out, say, 20 times, you'll know it's time to give up on the connection.

Like I said, check out Stevens' source for a perfectly excellent example.

How do I build for Windows?

First, delete Windows and install Linux or BSD. `};-)`. No, actually, just see the section on building for Windows in the introduction.

How do I build for Solaris/SunOS? I keep getting linker errors when I try to compile!

The linker errors happen because Sun boxes don't automatically compile in the socket libraries. See the section on building for Solaris/SunOS in the introduction for an example of how to do this.

Why does `select()` keep falling out on a signal?

Signals tend to cause blocked system calls to return `-1` with *`errno`* set to `EINTR`. When you set up a signal handler with **`sigaction()`**, you can set the flag `SA_RESTART`, which is supposed to restart the system call after it was interrupted.

Naturally, this doesn't always work.

My favorite solution to this involves a `goto` statement. You know this irritates your professors to no end, so go for it!

```
select_restart:
if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
    if (errno == EINTR) {
        // some signal just interrupted us, so restart
        goto select_restart;
    }
    // handle the real error here:
    perror("select");
}
```

Sure, you don't *need* to use `goto` in this case; you can use other structures to control it. But I think the `goto` statement is actually cleaner.

How can I implement a timeout on a call to `recv()`?

Use **`select()`**! It allows you to specify a timeout parameter for socket descriptors that you're looking to read from. Or, you could wrap the entire functionality in a single function, like this:

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>

int recvtimeout(int s, char *buf, int len, int timeout)
{
    fd_set fds;
    int n;
    struct timeval tv;

    // set up the file descriptor set
    FD_ZERO(&fds);
    FD_SET(s, &fds);

    // set up the struct timeval for the timeout
    tv.tv_sec = timeout;
    tv.tv_usec = 0;

    // wait until timeout or data received
```

```
    n = select(s+1, &fds, NULL, NULL, &tv);
    if (n == 0) return -2; // timeout!
    if (n == -1) return -1; // error

    // data must be here, so do a normal recv()
    return recv(s, buf, len, 0);
}
.
.
.
// Sample call to recvtimeout():
n = recvtimeout(s, buf, sizeof buf, 10); // 10 second timeout

if (n == -1) {
    // error occurred
    perror("recvtimeout");
}
else if (n == -2) {
    // timeout occurred
} else {
    // got some data in buf
}
.
.
.
```

Notice that **`recvtimeout()`** returns `-2` in case of a timeout. Why not return `0`? Well, if you recall, a return value of `0` on a call to **`recv()`** means that the remote side closed the connection. So that return value is already spoken for, and `-1` means "error", so I chose `-2` as my timeout indicator.

How do I encrypt or compress the data before sending it through the socket?

One easy way to do encryption is to use SSL (secure sockets layer), but that's beyond the scope of this guide. (Check out the OpenSSL project[39] for more info.)

But assuming you want to plug in or implement your own compressor or encryption system, it's just a matter of thinking of your data as running through a sequence of steps between both ends. Each step changes the data in some way.

1. server reads data from file (or wherever)
2. server encrypts/compresses data (you add this part)
3. server **`send()`**s encrypted data

Now the other way around:

1. client **`recv()`**s encrypted data
2. client decrypts/decompresses data (you add this part)
3. client writes data to file (or wherever)

If you're going to compress and encrypt, just remember to compress first. `:-)`

Just as long as the client properly undoes what the server does, the data will be fine in the end no matter how many intermediate steps you add.

So all you need to do to use my code is to find the place between where the data is read and the data is sent (using **`send()`**) over the network, and stick some code in there that does the encryption.

What is this "`PF_INET`" I keep seeing? Is it related to `AF_INET`?

Yes, yes it is. See the section on **`socket()`** for details.

39. `http://www.openssl.org/`

How can I write a server that accepts shell commands from a client and executes them?

For simplicity, lets say the client **`connect()`**s, **`send()`**s, and **`close()`**s the connection (that is, there are no subsequent system calls without the client connecting again.)

The process the client follows is this:

1. **`connect()`** to server
2. **`send("/sbin/ls > /tmp/client.out")`**
3. **`close()`** the connection

Meanwhile, the server is handling the data and executing it:

1. **`accept()`** the connection from the client
2. **`recv(str)`** the command string
3. **`close()`** the connection
4. **`system(str)`** to run the command

Beware! Having the server execute what the client says is like giving remote shell access and people can do things to your account when they connect to the server. For instance, in the above example, what if the client sends "**rm -rf ~**"? It deletes everything in your account, that's what!

So you get wise, and you prevent the client from using any except for a couple utilities that you know are safe, like the **foobar** utility:

```
if (!strncmp(str, "foobar", 6)) {
    sprintf(sysstr, "%s > /tmp/server.out", str);
    system(sysstr);
}
```

But you're still unsafe, unfortunately: what if the client enters "**foobar; rm -rf ~**"? The safest thing to do is to write a little routine that puts an escape ("\") character in front of all non-alphanumeric characters (including spaces, if appropriate) in the arguments for the command.

As you can see, security is a pretty big issue when the server starts executing things the client sends.

I'm sending a slew of data, but when I `recv()`, it only receives 536 bytes or 1460 bytes at a time. But if I run it on my local machine, it receives all the data at the same time. What's going on?

You're hitting the MTU—the maximum size the physical medium can handle. On the local machine, you're using the loopback device which can handle 8K or more no problem. But on Ethernet, which can only handle 1500 bytes with a header, you hit that limit. Over a modem, with 576 MTU (again, with header), you hit the even lower limit.

You have to make sure all the data is being sent, first of all. (See the **`sendall()`** function implementation for details.) Once you're sure of that, then you need to call **`recv()`** in a loop until all your data is read.

Read the section Son of Data Encapsulation for details on receiving complete packets of data using multiple calls to **`recv()`**.

I'm on a Windows box and I don't have the `fork()` system call or any kind of `struct sigaction`. What to do?

If they're anywhere, they'll be in POSIX libraries that may have shipped with your compiler. Since I don't have a Windows box, I really can't tell you the answer, but I seem to remember that Microsoft has a POSIX compatibility layer and that's where **`fork()`** would be. (And maybe even `sigaction`.)

Search the help that came with VC++ for "fork" or "POSIX" and see if it gives you any clues.

If that doesn't work at all, ditch the **`fork()`**/`sigaction` stuff and replace it with the Win32 equivalent: **`CreateProcess()`**. I don't know how to use **`CreateProcess()`**—it takes a bazillion arguments, but it should be covered in the docs that came with VC++.

I'm behind a firewall—how do I let people outside the firewall know my IP address so they can connect to my machine?

Unfortunately, the purpose of a firewall is to prevent people outside the firewall from connecting to machines inside the firewall, so allowing them to do so is basically considered a breach of security.

This isn't to say that all is lost. For one thing, you can still often **`connect()`** through the firewall if it's doing some kind of masquerading or NAT or something like that. Just design your programs so that you're always the one initiating the connection, and you'll be fine.

If that's not satisfactory, you can ask your sysadmins to poke a hole in the firewall so that people can connect to you. The firewall can forward to you either through it's NAT software, or through a proxy or something like that.

Be aware that a hole in the firewall is nothing to be taken lightly. You have to make sure you don't give bad people access to the internal network; if you're a beginner, it's a lot harder to make software secure than you might imagine.

Don't make your sysadmin mad at me. `;-)`

How do I write a packet sniffer? How do I put my Ethernet interface into promiscuous mode?

For those not in the know, when a network card is in "promiscuous mode", it will forward ALL packets to the operating system, not just those that were addressed to this particular machine. (We're talking Ethernet-layer addresses here, not IP addresses--but since ethernet is lower-layer than IP, all IP addresses are effectively forwarded as well. See the section Low Level Nonsense and Network Theory for more info.)

This is the basis for how a packet sniffer works. It puts the interface into promiscuous mode, then the OS gets every single packet that goes by on the wire. You'll have a socket of some type that you can read this data from.

Unfortunately, the answer to the question varies depending on the platform, but if you Google for, for instance, "windows promiscuous ioctl" you'll probably get somewhere. There's what looks like a decent writeup in Linux Journal[40], as well.

How can I set a custom timeout value for a TCP or UDP socket?

It depends on your system. You might search the net for `SO_RCVTIMEO` and `SO_SNDTIMEO` (for use with **`setsockopt()`**) to see if your system supports such functionality.

The Linux man page suggests using **`alarm()`** or **`setitimer()`** as a substitute.

How can I tell which ports are available to use? Is there a list of "official" port numbers?

Usually this isn't an issue. If you're writing, say, a web server, then it's a good idea to use the well-known port 80 for your software. If you're writing just your own specialized server, then choose a port at random (but greater than 1023) and give it a try.

If the port is already in use, you'll get an "Address already in use" error when you try to **`bind()`**. Choose another port. (It's a good idea to allow the user of your software to specify an alternate port either with a config file or a command line switch.)

There is a list of official port numbers[41] maintained by the Internet Assigned Numbers Authority (IANA). Just because something (over 1023) is in that list doesn't mean you can't use the port. For instance, Id Software's DOOM uses the same port as "mdqs", whatever that is. All that matters is that no one else *on the same machine* is using that port when you want to use it.

40. `http://interactive.linuxjournal.com/article/4659`
41. `http://www.iana.org/assignments/port-numbers`

9. Man Pages

In the Unix world, there are a lot of manuals. They have little sections that describe individual functions that you have at your disposal.

Of course, **manual** would be too much of a thing to type. I mean, no one in the Unix world, including myself, likes to type that much. Indeed I could go on and on at great length about how much I prefer to be terse but instead I shall be brief and not bore you with long-winded diatribes about how utterly amazingly brief I prefer to be in virtually all circumstances in their entirety.

[Applause]

Thank you. What I am getting at is that these pages are called “man pages” in the Unix world, and I have included my own personal truncated variant here for your reading enjoyment. The thing is, many of these functions are way more general purpose than I'm letting on, but I'm only going to present the parts that are relevant for Internet Sockets Programming.

But wait! That's not all that's wrong with my man pages:

- They are incomplete and only show the basics from the guide.
- There are many more man pages than this in the real world.
- They are different than the ones on your system.
- The header files might be different for certain functions on your system.
- The function parameters might be different for certain functions on your system.

If you want the real information, check your local Unix man pages by typing **man whatever**, where “whatever” is something that you're incredibly interested in, such as “`accept`”. (I'm sure Microsoft Visual Studio has something similar in their help section. But “man” is better because it is one byte more concise than “help”. Unix wins again!)

So, if these are so flawed, why even include them at all in the Guide? Well, there are a few reasons, but the best are that (a) these versions are geared specifically toward network programming and are easier to digest than the real ones, and (b) these versions contain examples!

Oh! And speaking of the examples, I don't tend to put in all the error checking because it really increases the length of the code. But you should absolutely do error checking pretty much any time you make any of the system calls unless you're totally 100% sure it's not going to fail, and you should probably do it even then!

9.1. accept()

Accept an incoming connection on a listening socket

Prototypes

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Description

Once you've gone through the trouble of getting a SOCK_STREAM socket and setting it up for incoming connections with **listen()**, then you call **accept()** to actually get yourself a new socket descriptor to use for subsequent communication with the newly connected client.

The old socket that you are using for listening is still there, and will be used for further **accept()** calls as they come in.

s	The **listen()**ing socket descriptor.
addr	This is filled in with the address of the site that's connecting to you.
addrlen	This is filled in with the **sizeof()** the structure returned in the *addr* parameter. You can safely ignore it if you assume you're getting a struct sockaddr_in back, which you know you are, because that's the type you passed in for *addr*.

accept() will normally block, and you can use **select()** to peek on the listening socket descriptor ahead of time to see if it's "ready to read". If so, then there's a new connection waiting to be **accept()**ed! Yay! Alternatively, you could set the O_NONBLOCK flag on the listening socket using **fcntl()**, and then it will never block, choosing instead to return -1 with *errno* set to EWOULDBLOCK.

The socket descriptor returned by **accept()** is a bona fide socket descriptor, open and connected to the remote host. You have to **close()** it when you're done with it.

Return Value

accept() returns the newly connected socket descriptor, or -1 on error, with *errno* set appropriately.

Example

```
struct sockaddr_storage their_addr;
socklen_t addr_size;
struct addrinfo hints, *res;
int sockfd, new_fd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, MYPORT, &hints, &res);

// make a socket, bind it, and listen on it:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);
listen(sockfd, BACKLOG);

// now accept an incoming connection:

addr_size = sizeof their_addr;
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);
```

```
// ready to communicate on socket descriptor new_fd!
```

See Also

`socket()`, **`getaddrinfo()`**, **`listen()`**, `struct sockaddr_in`

9.2. bind()

Associate a socket with an IP address and port number

Prototypes

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

Description

When a remote machine wants to connect to your server program, it needs two pieces of information: the IP address and the port number. The **bind()** call allows you to do just that.

First, you call **getaddrinfo()** to load up a **struct sockaddr** with the destination address and port information. Then you call **socket()** to get a socket descriptor, and then you pass the socket and address into **bind()**, and the IP address and port are magically (using actual magic) bound to the socket!

If you don't know your IP address, or you know you only have one IP address on the machine, or you don't care which of the machine's IP addresses is used, you can simply pass the `AI_PASSIVE` flag in the *hints* parameter to **getaddrinfo()**. What this does is fill in the IP address part of the **struct sockaddr** with a special value that tells **bind()** that it should automatically fill in this host's IP address.

What what? What special value is loaded into the **struct sockaddr**'s IP address to cause it to auto-fill the address with the current host? I'll tell you, but keep in mind this is only if you're filling out the `struct sockaddr` by hand; if not, use the results from **getaddrinfo()**, as per above. In IPv4, the *sin_addr.s_addr* field of the `struct sockaddr_in` structure is set to `INADDR_ANY`. In IPv6, the *sin6_addr* field of the `struct sockaddr_in6` structure is assigned into from the global variable *in6addr_any*. Or, if you're declaring a new `struct in6_addr`, you can initialize it to `IN6ADDR_ANY_INIT`.

Lastly, the *addrlen* parameter should be set to `sizeof my_addr`.

Return Value

Returns zero on success, or `-1` on error (and **errno** will be set accordingly.)

Example

```
// modern way of doing things with getaddrinfo()

struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:
// (you should actually walk the "res" linked list and error-check!)

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

```
// example of packing a struct by hand, IPv4

struct sockaddr_in myaddr;
int s;
```

```
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490);

// you can specify an IP address:
inet_pton(AF_INET, "63.161.169.137", &(myaddr.sin_addr));

// or you can let it automatically select one:
myaddr.sin_addr.s_addr = INADDR_ANY;

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&myaddr, sizeof myaddr);
```

See Also

`getaddrinfo()`, **`socket()`**, `struct sockaddr_in`, `struct in_addr`

9.3. connect()

Connect a socket to a server

Prototypes

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *serv_addr,
            socklen_t addrlen);
```

Description

Once you've built a socket descriptor with the **socket()** call, you can **connect()** that socket to a remote server using the well-named **connect()** system call. All you need to do is pass it the socket descriptor and the address of the server you're interested in getting to know better. (Oh, and the length of the address, which is commonly passed to functions like this.)

Usually this information comes along as the result of a call to **getaddrinfo()**, but you can fill out your own `struct sockaddr` if you want to.

If you haven't yet called **bind()** on the socket descriptor, it is automatically bound to your IP address and a random local port. This is usually just fine with you if you're not a server, since you really don't care what your local port is; you only care what the remote port is so you can put it in the *serv_addr* parameter. You *can* call **bind()** if you really want your client socket to be on a specific IP address and port, but this is pretty rare.

Once the socket is **connect()**ed, you're free to **send()** and **recv()** data on it to your heart's content.

Special note: if you **connect()** a `SOCK_DGRAM` UDP socket to a remote host, you can use **send()** and **recv()** as well as **sendto()** and **recvfrom()**. If you want.

Return Value

Returns zero on success, or `-1` on error (and **errno** will be set accordingly.)

Example

```
// connect to www.example.com port 80 (http)

struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;

// we could put "80" instead on "http" on the next line:
getaddrinfo("www.example.com", "http", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// connect it to the address and port we passed in to getaddrinfo():

connect(sockfd, res->ai_addr, res->ai_addrlen);
```

See Also

socket(), **bind()**

9.4. close()

Close a socket descriptor

Prototypes

```
#include <unistd.h>

int close(int s);
```

Description

After you've finished using the socket for whatever demented scheme you have concocted and you don't want to **send()** or **recv()** or, indeed, do *anything else* at all with the socket, you can **close()** it, and it'll be freed up, never to be used again.

The remote side can tell if this happens one of two ways. One: if the remote side calls **recv()**, it will return `0`. Two: if the remote side calls **send()**, it'll receive a signal `SIGPIPE` and send() will return `-1` and *errno* will be set to `EPIPE`.

Windows users: the function you need to use is called **closesocket()**, not **close()**. If you try to use **close()** on a socket descriptor, it's possible Windows will get angry... And you wouldn't like it when it's angry.

Return Value

Returns zero on success, or `-1` on error (and **errno** will be set accordingly.)

Example

```
s = socket(PF_INET, SOCK_DGRAM, 0);
.
.
.
// a whole lotta stuff...*BRRRONNNN!*
.
.
.
close(s);  // not much to it, really.
```

See Also

socket(), **shutdown()**

9.5. getaddrinfo(), freeaddrinfo(), gai_strerror()

Get information about a host name and/or service and load up a `struct sockaddr` with the result.

Prototypes

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *nodename, const char *servname,
                const struct addrinfo *hints, struct addrinfo **res);

void freeaddrinfo(struct addrinfo *ai);

const char *gai_strerror(int ecode);

struct addrinfo {
  int     ai_flags;          // AI_PASSIVE, AI_CANONNAME, ...
  int     ai_family;         // AF_xxx
  int     ai_socktype;       // SOCK_xxx
  int     ai_protocol;       // 0 (auto) or IPPROTO_TCP, IPPROTO_UDP

  socklen_t  ai_addrlen;     // length of ai_addr
  char   *ai_canonname;      // canonical name for nodename
  struct sockaddr  *ai_addr; // binary address
  struct addrinfo  *ai_next; // next structure in linked list
};
```

Description

`getaddrinfo()` is an excellent function that will return information on a particular host name (such as its IP address) and load up a `struct sockaddr` for you, taking care of the gritty details (like if it's IPv4 or IPv6.) It replaces the old functions **`gethostbyname()`** and **`getservbyname()`**.The description, below, contains a lot of information that might be a little daunting, but actual usage is pretty simple. It might be worth it to check out the examples first.

The host name that you're interested in goes in the *`nodename`* parameter. The address can be either a host name, like “www.example.com”, or an IPv4 or IPv6 address (passed as a string). This parameter can also be `NULL` if you're using the `AI_PASSIVE` flag (see below.)

The *`servname`* parameter is basically the port number. It can be a port number (passed as a string, like “80”), or it can be a service name, like “http” or “tftp” or “smtp” or “pop”, etc. Well-known service names can be found in the IANA Port List[42] or in your *`/etc/services`* file.

Lastly, for input parameters, we have *`hints`*. This is really where you get to define what the **`getaddinfo()`** function is going to do. Zero the whole structure before use with **`memset()`**. Let's take a look at the fields you need to set up before use.

The *`ai_flags`* can be set to a variety of things, but here are a couple important ones. (Multiple flags can be specified by bitwise-ORing them together with the **`|`** operator.) Check your man page for the complete list of flags.

`AI_CANONNAME` causes the *`ai_canonname`* of the result to the filled out with the host's canonical (real) name. `AI_PASSIVE` causes the result's IP address to be filled out with `INADDR_ANY` (IPv4)or *`in6addr_any`* (IPv6); this causes a subsequent call to **`bind()`** to auto-fill the IP address of the `struct sockaddr` with the address of the current host. That's excellent for setting up a server when you don't want to hardcode the address.

If you do use the `AI_PASSIVE`, flag, then you can pass `NULL` in the *`nodename`* (since **`bind()`** will fill it in for you later.)

42. `http://www.iana.org/assignments/port-numbers`

Continuing on with the input paramters, you'll likely want to set *`ai_family`* to `AF_UNSPEC` which tells **`getaddrinfo()`** to look for both IPv4 and IPv6 addresses. You can also restrict yourself to one or the other with `AF_INET` or `AF_INET6`.

Next, the *`socktype`* field should be set to `SOCK_STREAM` or `SOCK_DGRAM`, depending on which type of socket you want.

Finally, just leave *`ai_protocol`* at `0` to automatically choose your protocol type.

Now, after you get all that stuff in there, you can *finally* make the call to **`getaddrinfo()`**!

Of course, this is where the fun begins. The *`res`* will now point to a linked list of `struct addrinfo`s, and you can go through this list to get all the addresses that match what you passed in with the hints.

Now, it's possible to get some addresses that don't work for one reason or another, so what the Linux man page does is loops through the list doing a call to **`socket()`** and **`connect()`** (or **`bind()`** if you're setting up a server with the `AI_PASSIVE` flag) until it succeeds.

Finally, when you're done with the linked list, you need to call **`freeaddrinfo()`** to free up the memory (or it will be leaked, and Some People will get upset.)

Return Value

Returns zero on success, or nonzero on error. If it returns nonzero, you can use the function **`gai_strerror()`** to get a printable version of the error code in the return value.

Example

```
// code for a client connecting to a server
// namely a stream socket to www.example.com on port 80 (http)
// either IPv4 or IPv6

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6
hints.ai_socktype = SOCK_STREAM;

if ((rv = getaddrinfo("www.example.com", "http", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    exit(1);
}

// loop through all the results and connect to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("socket");
        continue;
    }

    if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("connect");
        continue;
    }

    break; // if we get here, we must have connected successfully
}

if (p == NULL) {
    // looped off the end of the list with no connection
    fprintf(stderr, "failed to connect\n");
    exit(2);
}

freeaddrinfo(servinfo); // all done with this structure
```

```
// code for a server waiting for connections
// namely a stream socket on port 3490, on this host's IP
// either IPv4 or IPv6.

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // use my IP address

if ((rv = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    exit(1);
}

// loop through all the results and bind to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("socket");
        continue;
    }

    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("bind");
        continue;
    }

    break; // if we get here, we must have connected successfully
}

if (p == NULL) {
    // looped off the end of the list with no successful bind
    fprintf(stderr, "failed to bind socket\n");
    exit(2);
}

freeaddrinfo(servinfo); // all done with this structure
```

See Also

`gethostbyname()`, **`getnameinfo()`**

9.6. gethostname()

Returns the name of the system

Prototypes

```
#include <sys/unistd.h>

int gethostname(char *name, size_t len);
```

Description

Your system has a name. They all do. This is a slightly more Unixy thing than the rest of the networky stuff we've been talking about, but it still has its uses.

For instance, you can get your host name, and then call **gethostbyname()** to find out your IP address.

The parameter *name* should point to a buffer that will hold the host name, and *len* is the size of that buffer in bytes. **gethostname()** won't overwrite the end of the buffer (it might return an error, or it might just stop writing), and it will NUL-terminate the string if there's room for it in the buffer.

Return Value

Returns zero on success, or -1 on error (and **errno** will be set accordingly.)

Example

```
char hostname[128];

gethostname(hostname, sizeof hostname);
printf("My hostname: %s\n", hostname);
```

See Also

gethostbyname()

9.7. gethostbyname(), gethostbyaddr()

Get an IP address for a hostname, or vice-versa

Prototypes

```
#include <sys/socket.h>
#include <netdb.h>

struct hostent *gethostbyname(const char *name); // DEPRECATED!
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Description

PLEASE NOTE: these two functions are superseded by ***`getaddrinfo()`*** *and* ***`getnameinfo()`****!* In particular, **`gethostbyname()`** doesn't work well with IPv6.

These functions map back and forth between host names and IP addresses. For instance, if you have "www.example.com", you can use **`gethostbyname()`** to get its IP address and store it in a `struct in_addr`.

Conversely, if you have a `struct in_addr` or a `struct in6_addr`, you can use **`gethostbyaddr()`** to get the hostname back. **`gethostbyaddr()`** *is* IPv6 compatible, but you should use the newer shinier **`getnameinfo()`** instead.

(If you have a string containing an IP address in dots-and-numbers format that you want to look up the hostname of, you'd be better off using **`getaddrinfo()`** with the `AI_CANONNAME` flag.)

`gethostbyname()` takes a string like "www.yahoo.com", and returns a `struct hostent` which contains tons of information, including the IP address. (Other information is the official host name, a list of aliases, the address type, the length of the addresses, and the list of addresses—it's a general-purpose structure that's pretty easy to use for our specific purposes once you see how.)

`gethostbyaddr()` takes a `struct in_addr` or `struct in6_addr` and brings you up a corresponding host name (if there is one), so it's sort of the reverse of **`gethostbyname()`**. As for parameters, even though *`addr`* is a `char*`, you actually want to pass in a pointer to a `struct in_addr`. *`len`* should be `sizeof(struct in_addr)`, and *`type`* should be `AF_INET`.

So what is this `struct hostent` that gets returned? It has a number of fields that contain information about the host in question.

`char *h_name`	The real canonical host name.
`char **h_aliases`	A list of aliases that can be accessed with arrays—the last element is `NULL`
`int h_addrtype`	The result's address type, which really should be `AF_INET` for our purposes.
`int length`	The length of the addresses in bytes, which is 4 for IP (version 4) addresses.
`char **h_addr_list`	A list of IP addresses for this host. Although this is a `char**`, it's really an array of `struct in_addr*`s in disguise. The last array element is `NULL`.
`h_addr`	A commonly defined alias for *`h_addr_list[0]`*. If you just want any old IP address for this host (yeah, they can have more than one) just use this field.

Return Value

Returns a pointer to a resultant `struct hostent` or success, or `NULL` on error.

Instead of the normal **`perror()`** and all that stuff you'd normally use for error reporting, these functions have parallel results in the variable *`h_errno`*, which can be printed using the functions **`herror()`** or **`hstrerror()`**. These work just like the classic *`errno`*, **`perror()`**, and **`strerror()`** functions you're used to.

Example

```
// THIS IS A DEPRECATED METHOD OF GETTING HOST NAMES
// use getaddrinfo() instead!

#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
    int i;
    struct hostent *he;
    struct in_addr **addr_list;

    if (argc != 2) {
        fprintf(stderr,"usage: ghbn hostname\n");
        return 1;
    }

    if ((he = gethostbyname(argv[1])) == NULL) {  // get the host info
        herror("gethostbyname");
        return 2;
    }

    // print information about this host:
    printf("Official name is: %s\n", he->h_name);
    printf("    IP addresses: ");
    addr_list = (struct in_addr **)he->h_addr_list;
    for(i = 0; addr_list[i] != NULL; i++) {
        printf("%s ", inet_ntoa(*addr_list[i]));
    }
    printf("\n");

    return 0;
}
```

```
// THIS HAS BEEN SUPERCEDED
// use getnameinfo() instead!

struct hostent *he;
struct in_addr ipv4addr;
struct in6_addr ipv6addr;

inet_pton(AF_INET, "192.0.2.34", &ipv4addr);
he = gethostbyaddr(&ipv4addr, sizeof ipv4addr, AF_INET);
printf("Host name: %s\n", he->h_name);

inet_pton(AF_INET6, "2001:db8:63b3:1::beef", &ipv6addr);
he = gethostbyaddr(&ipv6addr, sizeof ipv6addr, AF_INET6);
printf("Host name: %s\n", he->h_name);
```

See Also

`getaddrinfo()`, **`getnameinfo()`**, **`gethostname()`**, *`errno`*, **`perror()`**, **`strerror()`**, `struct in_addr`

9.8. `getnameinfo()`

Look up the host name and service name information for a given `struct sockaddr`.

Prototypes

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, size_t hostlen,
                char *serv, size_t servlen, int flags);
```

Description

This function is the opposite of **`getaddrinfo()`**, that is, this function takes an already loaded `struct sockaddr` and does a name and service name lookup on it. It replaces the old **`gethostbyaddr()`** and **`getservbyport()`** functions.

You have to pass in a pointer to a `struct sockaddr` (which in actuality is probably a `struct sockaddr_in` or `struct sockaddr_in6` that you've cast) in the *`sa`* parameter, and the length of that `struct` in the *`salen`*.

The resultant host name and service name will be written to the area pointed to by the *`host`* and *`serv`* parameters. Of course, you have to specify the max lengths of these buffers in *`hostlen`* and *`servlen`*.

Finally, there are several flags you can pass, but here a a couple good ones. `NI_NOFQDN` will cause the *`host`* to only contain the host name, not the whole domain name. `NI_NAMEREQD` will cause the function to fail if the name cannot be found with a DNS lookup (if you don't specify this flag and the name can't be found, **`getnameinfo()`** will put a string version of the IP address in *`host`* instead.)

As always, check your local man pages for the full scoop.

Return Value

Returns zero on success, or non-zero on error. If the return value is non-zero, it can be passed to **`gai_strerror()`** to get a human-readable string. See **`getaddrinfo`** for more information.

Example

```
struct sockaddr_in6 sa; // could be IPv4 if you want
char host[1024];
char service[20];

// pretend sa is full of good information about the host and port...

getnameinfo(&sa, sizeof sa, host, sizeof host, service, sizeof service, 0);

printf("   host: %s\n", host);    // e.g. "www.example.com"
printf("service: %s\n", service); // e.g. "http"
```

See Also

`getaddrinfo()`, **`gethostbyaddr()`**

9.9. `getpeername()`

Return address info about the remote side of the connection

Prototypes

```
#include <sys/socket.h>

int getpeername(int s, struct sockaddr *addr, socklen_t *len);
```

Description

Once you have either **`accept()`**ed a remote connection, or **`connect()`**ed to a server, you now have what is known as a *peer*. Your peer is simply the computer you're connected to, identified by an IP address and a port. So...

`getpeername()` simply returns a `struct sockaddr_in` filled with information about the machine you're connected to.

Why is it called a "name"? Well, there are a lot of different kinds of sockets, not just Internet Sockets like we're using in this guide, and so "name" was a nice generic term that covered all cases. In our case, though, the peer's "name" is it's IP address and port.

Although the function returns the size of the resultant address in *`len`*, you must preload *`len`* with the size of *`addr`*.

Return Value

Returns zero on success, or `-1` on error (and **`errno`** will be set accordingly.)

Example

```
// assume s is a connected socket

socklen_t len;
struct sockaddr_storage addr;
char ipstr[INET6_ADDRSTRLEN];
int port;

len = sizeof addr;
getpeername(s, (struct sockaddr*)&addr, &len);

// deal with both IPv4 and IPv6:
if (addr.ss_family == AF_INET) {
    struct sockaddr_in *s = (struct sockaddr_in *)&addr;
    port = ntohs(s->sin_port);
    inet_ntop(AF_INET, &s->sin_addr, ipstr, sizeof ipstr);
} else { // AF_INET6
    struct sockaddr_in6 *s = (struct sockaddr_in6 *)&addr;
    port = ntohs(s->sin6_port);
    inet_ntop(AF_INET6, &s->sin6_addr, ipstr, sizeof ipstr);
}

printf("Peer IP address: %s\n", ipstr);
printf("Peer port      : %d\n", port);
```

See Also

`gethostname()`, **`gethostbyname()`**, **`gethostbyaddr()`**

9.10. *errno*

Holds the error code for the last system call

Prototypes

```
#include <errno.h>

int errno;
```

Description

This is the variable that holds error information for a lot of system calls. If you'll recall, things like **`socket()`** and **`listen()`** return `-1` on error, and they set the exact value of *`errno`* to let you know specifically which error occurred.

The header file *`errno.h`* lists a bunch of constant symbolic names for errors, such as `EADDRINUSE`, `EPIPE`, `ECONNREFUSED`, etc. Your local man pages will tell you what codes can be returned as an error, and you can use these at run time to handle different errors in different ways.

Or, more commonly, you can call **`perror()`** or **`strerror()`** to get a human-readable version of the error.

One thing to note, for you multithreading enthusiasts, is that on most systems *`errno`* is defined in a threadsafe manner. (That is, it's not actually a global variable, but it behaves just like a global variable would in a single-threaded environment.)

Return Value

The value of the variable is the latest error to have transpired, which might be the code for "success" if the last action succeeded.

Example

```
s = socket(PF_INET, SOCK_STREAM, 0);
if (s == -1) {
    perror("socket"); // or use strerror()
}

tryagain:
if (select(n, &readfds, NULL, NULL) == -1) {
    // an error has occurred!!

    // if we were only interrupted, just restart the select() call:
    if (errno == EINTR) goto tryagain;  // AAAA!  goto!!!

    // otherwise it's a more serious error:
    perror("select");
    exit(1);
}
```

See Also

`perror()`, **`strerror()`**

9.11. `fcntl()`

Control socket descriptors

Prototypes

```
#include <sys/unistd.h>
#include <sys/fcntl.h>

int fcntl(int s, int cmd, long arg);
```

Description

This function is typically used to do file locking and other file-oriented stuff, but it also has a couple socket-related functions that you might see or use from time to time.

Parameter *s* is the socket descriptor you wish to operate on, *cmd* should be set to `F_SETFL`, and *arg* can be one of the following commands. (Like I said, there's more to **`fcntl()`** than I'm letting on here, but I'm trying to stay socket-oriented.)

`O_NONBLOCK`	Set the socket to be non-blocking. See the section on blocking for more details.
`O_ASYNC`	Set the socket to do asynchronous I/O. When data is ready to be **`recv()`**'d on the socket, the signal `SIGIO` will be raised. This is rare to see, and beyond the scope of the guide. And I think it's only available on certain systems.

Return Value

Returns zero on success, or `-1` on error (and **`errno`** will be set accordingly.)

Different uses of the **`fcntl()`** system call actually have different return values, but I haven't covered them here because they're not socket-related. See your local **`fcntl()`** man page for more information.

Example

```
int s = socket(PF_INET, SOCK_STREAM, 0);

fcntl(s, F_SETFL, O_NONBLOCK);  // set to non-blocking
fcntl(s, F_SETFL, O_ASYNC);     // set to asynchronous I/O
```

See Also

Blocking, **`send()`**

9.12. htons(), htonl(), ntohs(), ntohl()

Convert multi-byte integer types from host byte order to network byte order

Prototypes

```
#include <netinet/in.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Description

Just to make you really unhappy, different computers use different byte orderings internally for their multibyte integers (i.e. any integer that's larger than a `char`.) The upshot of this is that if you **`send()`** a two-byte `short int` from an Intel box to a Mac (before they became Intel boxes, too, I mean), what one computer thinks is the number `1`, the other will think is the number `256`, and vice-versa.

The way to get around this problem is for everyone to put aside their differences and agree that Motorola and IBM had it right, and Intel did it the weird way, and so we all convert our byte orderings to "big-endian" before sending them out. Since Intel is a "little-endian" machine, it's far more politically correct to call our preferred byte ordering "Network Byte Order". So these functions convert from your native byte order to network byte order and back again.

(This means on Intel these functions swap all the bytes around, and on PowerPC they do nothing because the bytes are already in Network Byte Order. But you should always use them in your code anyway, since someone might want to build it on an Intel machine and still have things work properly.)

Note that the types involved are 32-bit (4 byte, probably `int`) and 16-bit (2 byte, very likely `short`) numbers. 64-bit machines might have a **`htonll()`** for 64-bit `int`s, but I've not seen it. You'll just have to write your own.

Anyway, the way these functions work is that you first decide if you're converting *from* host (your machine's) byte order or from network byte order. If "host", the the first letter of the function you're going to call is "h". Otherwise it's "n" for "network". The middle of the function name is always "to" because you're converting from one "to" another, and the penultimate letter shows what you're converting *to*. The last letter is the size of the data, "s" for short, or "l" for long. Thus:

`htons()`	**h**ost **to** **n**etwork **s**hort
`htonl()`	**h**ost **to** **n**etwork **l**ong
`ntohs()`	**n**etwork **to** **h**ost **s**hort
`ntohl()`	**n**etwork **to** **h**ost **l**ong

Return Value

Each function returns the converted value.

Example

```
uint32_t some_long = 10;
uint16_t some_short = 20;

uint32_t network_byte_order;

// convert and send
network_byte_order = htonl(some_long);
send(s, &network_byte_order, sizeof(uint32_t), 0);

some_short == ntohs(htons(some_short)); // this expression is true
```

9.13. `inet_ntoa()`, `inet_aton()`, `inet_addr`

Convert IP addresses from a dots-and-number string to a `struct in_addr` and back

Prototypes

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

// ALL THESE ARE DEPRECATED!  Use inet_pton()  or inet_ntop() instead!!

char *inet_ntoa(struct in_addr in);
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
```

Description

These functions are deprecated because they don't handle IPv6! Use ***`inet_ntop()`*** *or* ***`inet_pton()`*** *instead! They are included here because they can still be found in the wild.*

All of these functions convert from a `struct in_addr` (part of your `struct sockaddr_in`, most likely) to a string in dots-and-numbers format (e.g. “192.168.5.10”) and vice-versa. If you have an IP address passed on the command line or something, this is the easiest way to get a `struct in_addr` to **`connect()`** to, or whatever. If you need more power, try some of the DNS functions like **`gethostbyname()`** or attempt a *coup d'État* in your local country.

The function **`inet_ntoa()`** converts a network address in a `struct in_addr` to a dots-and-numbers format string. The “n” in “ntoa” stands for network, and the “a” stands for ASCII for historical reasons (so it's “Network To ASCII”—the “toa” suffix has an analogous friend in the C library called **`atoi()`** which converts an ASCII string to an integer.)

The function **`inet_aton()`** is the opposite, converting from a dots-and-numbers string into a `in_addr_t` (which is the type of the field `s_addr` in your `struct in_addr`.)

Finally, the function **`inet_addr()`** is an older function that does basically the same thing as **`inet_aton()`**. It's theoretically deprecated, but you'll see it a lot and the police won't come get you if you use it.

Return Value

`inet_aton()` returns non-zero if the address is a valid one, and it returns zero if the address is invalid.

`inet_ntoa()` returns the dots-and-numbers string in a static buffer that is overwritten with each call to the function.

`inet_addr()` returns the address as an `in_addr_t`, or `-1` if there's an error. (That is the same result as if you tried to convert the string “255.255.255.255”, which is a valid IP address. This is why **`inet_aton()`** is better.)

Example

```
struct sockaddr_in antelope;
char *some_addr;

inet_aton("10.0.0.1", &antelope.sin_addr); // store IP in antelope

some_addr = inet_ntoa(antelope.sin_addr); // return the IP
printf("%s\n", some_addr); // prints "10.0.0.1"

// and this call is the same as the inet_aton() call, above:
antelope.sin_addr.s_addr = inet_addr("10.0.0.1");
```

See Also

`inet_ntop()`, **`inet_pton()`**, **`gethostbyname()`**, **`gethostbyaddr()`**

9.14. `inet_ntop()`, `inet_pton()`

Convert IP addresses to human-readable form and back.

Prototypes

```
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t size);

int inet_pton(int af, const char *src, void *dst);
```

Description

These functions are for dealing with human-readable IP addresses and converting them to their binary representation for use with various functions and system calls. The “n” stands for “network”, and “p” for “presentation”. Or “text presentation”. But you can think of it as “printable”. “ntop” is “network to printable”. See?

Sometimes you don't want to look at a pile of binary numbers when looking at an IP address. You want it in a nice printable form, like `192.0.2.180`, or `2001:db8:8714:3a90::12`. In that case, **`inet_ntop()`** is for you.

`inet_ntop()` takes the address family in the *`af`* parameter (either `AF_INET` or `AF_INET6`). The *`src`* parameter should be a pointer to either a `struct in_addr` or `struct in6_addr` containing the address you wish to convert to a string. Finally *`dst`* and *`size`* are the pointer to the destination string and the maximum length of that string.

What should the maximum length of the *`dst`* string be? What is the maximum length for IPv4 and IPv6 addresses? Fortunately there are a couple of macros to help you out. The maximum lengths are: `INET_ADDRSTRLEN` and `INET6_ADDRSTRLEN`.

Other times, you might have a string containing an IP address in readable form, and you want to pack it into a `struct sockaddr_in` or a `struct sockaddr_in6`. In that case, the opposite funcion **`inet_pton()`** is what you're after.

`inet_pton()` also takes an address family (either `AF_INET` or `AF_INET6`) in the *`af`* parameter. The *`src`* parameter is a pointer to a string containing the IP address in printable form. Lastly the *`dst`* parameter points to where the result should be stored, which is probably a `struct in_addr` or `struct in6_addr`.

These functions don't do DNS lookups—you'll need **`getaddinfo()`** for that.

Return Value

`inet_ntop()` returns the *`dst`* parameter on success, or `NULL` on failure (and *`errno`* is set).

`inet_pton()` returns `1` on success. It returns `-1` if there was an error (*`errno`* is set), or `0` if the input isn't a valid IP address.

Example

```
// IPv4 demo of inet_ntop() and inet_pton()

struct sockaddr_in sa;
char str[INET_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET, "192.0.2.33", &(sa.sin_addr));

// now get it back and print it
inet_ntop(AF_INET, &(sa.sin_addr), str, INET_ADDRSTRLEN);

printf("%s\n", str); // prints "192.0.2.33"
```

```
// IPv6 demo of inet_ntop() and inet_pton()
// (basically the same except with a bunch of 6s thrown around)

struct sockaddr_in6 sa;
```

```
char str[INET6_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &(sa.sin6_addr));

// now get it back and print it
inet_ntop(AF_INET6, &(sa.sin6_addr), str, INET6_ADDRSTRLEN);

printf("%s\n", str); // prints "2001:db8:8714:3a90::12"
```

```
// Helper function you can use:

//Convert a struct sockaddr address to a string, IPv4 and IPv6:

char *get_ip_str(const struct sockaddr *sa, char *s, size_t maxlen)
{
    switch(sa->sa_family) {
        case AF_INET:
            inet_ntop(AF_INET, &(((struct sockaddr_in *)sa)->sin_addr),
                    s, maxlen);
            break;

        case AF_INET6:
            inet_ntop(AF_INET6, &(((struct sockaddr_in6 *)sa)->sin6_addr),
                    s, maxlen);
            break;

        default:
            strncpy(s, "Unknown AF", maxlen);
            return NULL;
    }

    return s;
}
```

See Also

`getaddrinfo()`

9.15. `listen()`

Tell a socket to listen for incoming connections

Prototypes

```
#include <sys/socket.h>

int listen(int s, int backlog);
```

Description

You can take your socket descriptor (made with the **`socket()`** system call) and tell it to listen for incoming connections. This is what differentiates the servers from the clients, guys.

The *`backlog`* parameter can mean a couple different things depending on the system you on, but loosely it is how many pending connections you can have before the kernel starts rejecting new ones. So as the new connections come in, you should be quick to **`accept()`** them so that the backlog doesn't fill. Try setting it to 10 or so, and if your clients start getting "Connection refused" under heavy load, set it higher.

Before calling **`listen()`**, your server should call **`bind()`** to attach itself to a specific port number. That port number (on the server's IP address) will be the one that clients connect to.

Return Value

Returns zero on success, or `-1` on error (and **`errno`** will be set accordingly.)

Example

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);

listen(sockfd, 10); // set s up to be a server (listening) socket

// then have an accept() loop down here somewhere
```

See Also

`accept()`, **`bind()`**, **`socket()`**

9.16. `perror()`, `strerror()`

Print an error as a human-readable string

Prototypes

```
#include <stdio.h>
#include <string.h>   // for strerror()

void perror(const char *s);
char *strerror(int errnum);
```

Description

Since so many functions return `-1` on error and set the value of the variable *`errno`* to be some number, it would sure be nice if you could easily print that in a form that made sense to you.

Mercifully, **`perror()`** does that. If you want more description to be printed before the error, you can point the parameter *`s`* to it (or you can leave *`s`* as `NULL` and nothing additional will be printed.)

In a nutshell, this function takes *`errno`* values, like `ECONNRESET`, and prints them nicely, like "Connection reset by peer."

The function **`strerror()`** is very similar to **`perror()`**, except it returns a pointer to the error message string for a given value (you usually pass in the variable *`errno`*.)

Return Value

`strerror()` returns a pointer to the error message string.

Example

```
int s;

s = socket(PF_INET, SOCK_STREAM, 0);

if (s == -1) { // some error has occurred
    // prints "socket error: " + the error message:
    perror("socket error");
}

// similarly:
if (listen(s, 10) == -1) {
    // this prints "an error: " + the error message from errno:
    printf("an error: %s\n", strerror(errno));
}
```

See Also

`errno`

9.17. poll()

Test for events on multiple sockets simultaneously

Prototypes

```
#include <sys/poll.h>

int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

Description

This function is very similar to **select()** in that they both watch sets of file descriptors for events, such as incoming data ready to **recv()**, socket ready to **send()** data to, out-of-band data ready to **recv()**, errors, etc.

The basic idea is that you pass an array of *nfds* `struct pollfd`s in *ufds*, along with a timeout in milliseconds (1000 milliseconds in a second.) The *timeout* can be negative if you want to wait forever. If no event happens on any of the socket descriptors by the timeout, **poll()** will return.

Each element in the array of `struct pollfd`s represents one socket descriptor, and contains the following fields:

```
struct pollfd {
    int fd;         // the socket descriptor
    short events;   // bitmap of events we're interested in
    short revents;  // when poll() returns, bitmap of events that occurred
};
```

Before calling **poll()**, load *fd* with the socket descriptor (if you set *fd* to a negative number, this `struct pollfd` is ignored and its *revents* field is set to zero) and then construct the *events* field by bitwise-ORing the following macros:

`POLLIN`	Alert me when data is ready to **recv()** on this socket.
`POLLOUT`	Alert me when I can **send()** data to this socket without blocking.
`POLLPRI`	Alert me when out-of-band data is ready to **recv()** on this socket.

Once the **poll()** call returns, the *revents* field will be constructed as a bitwise-OR of the above fields, telling you which descriptors actually have had that event occur. Additionally, these other fields might be present:

`POLLERR`	An error has occurred on this socket.
`POLLHUP`	The remote side of the connection hung up.
`POLLNVAL`	Something was wrong with the socket descriptor *fd*—maybe it's uninitialized?

Return Value

Returns the number of elements in the *ufds* array that have had event occur on them; this can be zero if the timeout occurred. Also returns `-1` on error (and **errno** will be set accordingly.)

Example

```
int s1, s2;
int rv;
char buf1[256], buf2[256];
struct pollfd ufds[2];

s1 = socket(PF_INET, SOCK_STREAM, 0);
s2 = socket(PF_INET, SOCK_STREAM, 0);

// pretend we've connected both to a server at this point
//connect(s1, ...)...
//connect(s2, ...)...
```

```
// set up the array of file descriptors.
//
// in this example, we want to know when there's normal or out-of-band
// data ready to be recv()'d...

ufds[0].fd = s1;
ufds[0].events = POLLIN | POLLPRI; // check for normal or out-of-band

ufds[1] = s2;
ufds[1].events = POLLIN; // check for just normal data

// wait for events on the sockets, 3.5 second timeout
rv = poll(ufds, 2, 3500);

if (rv == -1) {
    perror("poll"); // error occurred in poll()
} else if (rv == 0) {
    printf("Timeout occurred!  No data after 3.5 seconds.\n");
} else {
    // check for events on s1:
    if (ufds[0].revents & POLLIN) {
        recv(s1, buf1, sizeof buf1, 0); // receive normal data
    }
    if (ufds[0].revents & POLLPRI) {
        recv(s1, buf1, sizeof buf1, MSG_OOB); // out-of-band data
    }

    // check for events on s2:
    if (ufds[1].revents & POLLIN) {
        recv(s1, buf2, sizeof buf2, 0);
    }
}
```

See Also

`select()`

9.18. recv(), recvfrom()

Receive data on a socket

Prototypes

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

Description

Once you have a socket up and connected, you can read incoming data from the remote side using the **recv()** (for TCP SOCK_STREAM sockets) and **recvfrom()** (for UDP SOCK_DGRAM sockets).

Both functions take the socket descriptor *s*, a pointer to the buffer *buf*, the size (in bytes) of the buffer *len*, and a set of *flags* that control how the functions work.

Additionally, the **recvfrom()** takes a struct sockaddr*, *from* that will tell you where the data came from, and will fill in *fromlen* with the size of struct sockaddr. (You must also initialize *fromlen* to be the size of *from* or struct sockaddr.)

So what wondrous flags can you pass into this function? Here are some of them, but you should check your local man pages for more information and what is actually supported on your system. You bitwise-or these together, or just set *flags* to 0 if you want it to be a regular vanilla **recv()**.

MSG_OOB	Receive Out of Band data. This is how to get data that has been sent to you with the MSG_OOB flag in **send()**. As the receiving side, you will have had signal SIGURG raised telling you there is urgent data. In your handler for that signal, you could call **recv()** with this MSG_OOB flag.
MSG_PEEK	If you want to call **recv()** "just for pretend", you can call it with this flag. This will tell you what's waiting in the buffer for when you call **recv()** "for real" (i.e. *without* the MSG_PEEK flag. It's like a sneak preview into the next **recv()** call.
MSG_WAITALL	Tell **recv()** to not return until all the data you specified in the *len* parameter. It will ignore your wishes in extreme circumstances, however, like if a signal interrupts the call or if some error occurs or if the remote side closes the connection, etc. Don't be mad with it.

When you call **recv()**, it will block until there is some data to read. If you want to not block, set the socket to non-blocking or check with **select()** or **poll()** to see if there is incoming data before calling **recv()** or **recvfrom()**.

Return Value

Returns the number of bytes actually received (which might be less than you requested in the *len* parameter), or -1 on error (and **errno** will be set accordingly.)

If the remote side has closed the connection, **recv()** will return 0. This is the normal method for determining if the remote side has closed the connection. Normality is good, rebel!

Example

```
// stream sockets and recv()

struct addrinfo hints, *res;
int sockfd;
char buf[512];
int byte_count;

// get host info, make socket, and connect it
memset(&hints, 0, sizeof hints);
```

```
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
getaddrinfo("www.example.com", "3490", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
connect(sockfd, res->ai_addr, res->ai_addrlen);

// all right!  now that we're connected, we can receive some data!
byte_count = recv(sockfd, buf, sizeof buf, 0);
printf("recv()'d %d bytes of data in buf\n", byte_count);
```

```
// datagram sockets and recvfrom()

struct addrinfo hints, *res;
int sockfd;
int byte_count;
socklen_t fromlen;
struct sockaddr_storage addr;
char buf[512];
char ipstr[INET6_ADDRSTRLEN];

// get host info, make socket, bind it to port 4950
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_PASSIVE;
getaddrinfo(NULL, "4950", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);

// no need to accept(), just recvfrom():

fromlen = sizeof addr;
byte_count = recvfrom(sockfd, buf, sizeof buf, 0, &addr, &fromlen);

printf("recv()'d %d bytes of data in buf\n", byte_count);
printf("from IP address %s\n",
    inet_ntop(addr.ss_family,
        addr.ss_family == AF_INET?
            ((struct sockadd_in *)&addr)->sin_addr:
            ((struct sockadd_in6 *)&addr)->sin6_addr,
        ipstr, sizeof ipstr);
```

See Also

`send()`, **`sendto()`**, **`select()`**, **`poll()`**, Blocking

9.19. `select()`

Check if sockets descriptors are ready to read/write

Prototypes

```
#include <sys/select.h>

int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);

FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Description

The **`select()`** function gives you a way to simultaneously check multiple sockets to see if they have data waiting to be **`recv()`**d, or if you can **`send()`** data to them without blocking, or if some exception has occurred.

You populate your sets of socket descriptors using the macros, like **`FD_SET()`**, above. Once you have the set, you pass it into the function as one of the following parameters: *`readfds`* if you want to know when any of the sockets in the set is ready to **`recv()`** data, *`writefds`* if any of the sockets is ready to **`send()`** data to, and/or *`exceptfds`* if you need to know when an exception (error) occurs on any of the sockets. Any or all of these parameters can be `NULL` if you're not interested in those types of events. After **`select()`** returns, the values in the sets will be changed to show which are ready for reading or writing, and which have exceptions.

The first parameter, *`n`* is the highest-numbered socket descriptor (they're just `int`s, remember?) plus one.

Lastly, the `struct timeval`, *`timeout`*, at the end—this lets you tell **`select()`** how long to check these sets for. It'll return after the timeout, or when an event occurs, whichever is first. The `struct timeval` has two fields: *`tv_sec`* is the number of seconds, to which is added *`tv_usec`*, the number of microseconds (1,000,000 microseconds in a second.)

The helper macros do the following:

`FD_SET(int fd, fd_set *set);`	Add *`fd`* to the *`set`*.
`FD_CLR(int fd, fd_set *set);`	Remove *`fd`* from the *`set`*.
`FD_ISSET(int fd, fd_set *set);`	Return true if *`fd`* is in the *`set`*.
`FD_ZERO(fd_set *set);`	Clear all entries from the *`set`*.

Return Value

Returns the number of descriptors in the set on success, `0` if the timeout was reached, or `-1` on error (and **`errno`** will be set accordingly.) Also, the sets are modified to show which sockets are ready.

Example

```
int s1, s2, n;
fd_set readfds;
struct timeval tv;
char buf1[256], buf2[256];

// pretend we've connected both to a server at this point
//s1 = socket(...);
//s2 = socket(...);
//connect(s1, ...)...
//connect(s2, ...)...

// clear the set ahead of time
FD_ZERO(&readfds);
```

```
// add our descriptors to the set
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);

// since we got s2 second, it's the "greater", so we use that for
// the n param in select()
n = s2 + 1;

// wait until either socket has data ready to be recv()d (timeout 10.5 secs)
tv.tv_sec = 10;
tv.tv_usec = 500000;
rv = select(n, &readfds, NULL, NULL, &tv);

if (rv == -1) {
    perror("select"); // error occurred in select()
} else if (rv == 0) {
    printf("Timeout occurred!  No data after 10.5 seconds.\n");
} else {
    // one or both of the descriptors have data
    if (FD_ISSET(s1, &readfds)) {
        recv(s1, buf1, sizeof buf1, 0);
    }
    if (FD_ISSET(s2, &readfds)) {
        recv(s1, buf2, sizeof buf2, 0);
    }
}
```

See Also

poll()

9.20. `setsockopt()`, `getsockopt()`

Set various options for a socket

Prototypes

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int s, int level, int optname, void *optval,
               socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval,
               socklen_t optlen);
```

Description

Sockets are fairly configurable beasts. In fact, they are so configurable, I'm not even going to cover it all here. It's probably system-dependent anyway. But I will talk about the basics.

Obviously, these functions get and set certain options on a socket. On a Linux box, all the socket information is in the man page for socket in section 7. (Type: "**man 7 socket**" to get all these goodies.)

As for parameters, *`s`* is the socket you're talking about, level should be set to `SOL_SOCKET`. Then you set the *`optname`* to the name you're interested in. Again, see your man page for all the options, but here are some of the most fun ones:

`SO_BINDTODEVICE`	Bind this socket to a symbolic device name like `eth0` instead of using **`bind()`** to bind it to an IP address. Type the command **ifconfig** under Unix to see the device names.
`SO_REUSEADDR`	Allows other sockets to **`bind()`** to this port, unless there is an active listening socket bound to the port already. This enables you to get around those "Address already in use" error messages when you try to restart your server after a crash.
`SO_BROADCAST`	Allows UDP datagram (`SOCK_DGRAM`) sockets to send and receive packets sent to and from the broadcast address. Does nothing —*NOTHING!!*—to TCP stream sockets! Hahaha!

As for the parameter *`optval`*, it's usually a pointer to an `int` indicating the value in question. For booleans, zero is false, and non-zero is true. And that's an absolute fact, unless it's different on your system. If there is no parameter to be passed, *`optval`* can be `NULL`.

The final parameter, *`optlen`*, is filled out for you by **`getsockopt()`** and you have to specify it for **`setsockopt()`**, where it will probably be `sizeof(int)`.

Warning: on some systems (notably Sun and Windows), the option can be a `char` instead of an `int`, and is set to, for example, a character value of `'1'` instead of an `int` value of `1`. Again, check your own man pages for more info with "**man setsockopt**" and "**man 7 socket**"!

Return Value

Returns zero on success, or `-1` on error (and **`errno`** will be set accordingly.)

Example

```
int optval;
int optlen;
char *optval2;

// set SO_REUSEADDR on a socket to true (1):
optval = 1;
setsockopt(s1, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval);

// bind a socket to a device name (might not work on all systems):
optval2 = "eth1"; // 4 bytes long, so 4, below:
setsockopt(s2, SOL_SOCKET, SO_BINDTODEVICE, optval2, 4);
```

```
// see if the SO_BROADCAST flag is set:
getsockopt(s3, SOL_SOCKET, SO_BROADCAST, &optval, &optlen);
if (optval != 0) {
    print("SO_BROADCAST enabled on s3!\n");
}
```

See Also

`fcntl()`

9.21. send(), sendto()

Send data out over a socket

Prototypes

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t sendto(int s, const void *buf, size_t len,
               int flags, const struct sockaddr *to,
               socklen_t tolen);
```

Description

These functions send data to a socket. Generally speaking, **send()** is used for TCP SOCK_STREAM connected sockets, and **sendto()** is used for UDP SOCK_DGRAM unconnected datagram sockets. With the unconnected sockets, you must specify the destination of a packet each time you send one, and that's why the last parameters of **sendto()** define where the packet is going.

With both **send()** and **sendto()**, the parameter *s* is the socket, *buf* is a pointer to the data you want to send, *len* is the number of bytes you want to send, and *flags* allows you to specify more information about how the data is to be sent. Set *flags* to zero if you want it to be "normal" data. Here are some of the commonly used flags, but check your local **send()** man pages for more details:

MSG_OOB	Send as "out of band" data. TCP supports this, and it's a way to tell the receiving system that this data has a higher priority than the normal data. The receiver will receive the signal SIGURG and it can then receive this data without first receiving all the rest of the normal data in the queue.
MSG_DONTROUTE	Don't send this data over a router, just keep it local.
MSG_DONTWAIT	If **send()** would block because outbound traffic is clogged, have it return EAGAIN. This is like a "enable non-blocking just for this send." See the section on blocking for more details.
MSG_NOSIGNAL	If you **send()** to a remote host which is no longer **recv()**ing, you'll typically get the signal SIGPIPE. Adding this flag prevents that signal from being raised.

Return Value

Returns the number of bytes actually sent, or -1 on error (and **errno** will be set accordingly.) Note that the number of bytes actually sent might be less than the number you asked it to send! See the section on handling partial **send()**s for a helper function to get around this.

Also, if the socket has been closed by either side, the process calling **send()** will get the signal SIGPIPE. (Unless **send()** was called with the MSG_NOSIGNAL flag.)

Example

```
int spatula_count = 3490;
char *secret_message = "The Cheese is in The Toaster";

int stream_socket, dgram_socket;
struct sockaddr_in dest;
int temp;

// first with TCP stream sockets:

// assume sockets are made and connected
//stream_socket = socket(...
//connect(stream_socket, ...
```

```
// convert to network byte order
temp = htonl(spatula_count);
// send data normally:
send(stream_socket, &temp, sizeof temp, 0);

// send secret message out of band:
send(stream_socket, secret_message, strlen(secret_message)+1, MSG_OOB);

// now with UDP datagram sockets:
//getaddrinfo(...
//dest = ...  // assume "dest" holds the address of the destination
//dgram_socket = socket(...

// send secret message normally:
sendto(dgram_socket, secret_message, strlen(secret_message)+1, 0,
       (struct sockaddr*)&dest, sizeof dest);
```

See Also

recv(), **recvfrom()**

9.22. shutdown()

Stop further sends and receives on a socket

Prototypes

```
#include <sys/socket.h>

int shutdown(int s, int how);
```

Description

That's it! I've had it! No more **send()**s are allowed on this socket, but I still want to **recv()** data on it! Or vice-versa! How can I do this?

When you **close()** a socket descriptor, it closes both sides of the socket for reading and writing, and frees the socket descriptor. If you just want to close one side or the other, you can use this **shutdown()** call.

As for parameters, *s* is obviously the socket you want to perform this action on, and what action that is can be specified with the *how* parameter. How can be `SHUT_RD` to prevent further **recv()**s, `SHUT_WR` to prohibit further **send()**s, or `SHUT_RDWR` to do both.

Note that **shutdown()** doesn't free up the socket descriptor, so you still have to eventually **close()** the socket even if it has been fully shut down.

This is a rarely used system call.

Return Value

Returns zero on success, or `-1` on error (and **errno** will be set accordingly.)

Example

```
int s = socket(PF_INET, SOCK_STREAM, 0);

// ...do some send()s and stuff in here...

// and now that we're done, don't allow any more sends()s:
shutdown(s, SHUT_WR);
```

See Also

close()

9.23. socket()

Allocate a socket descriptor

Prototypes

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Description

Returns a new socket descriptor that you can use to do sockety things with. This is generally the first call in the whopping process of writing a socket program, and you can use the result for subsequent calls to **`listen()`**, `bind()`, `accept()`, or a variety of other functions.

In usual usage, you get the values for these parameters from a call to **`getaddrinfo()`**, as shown in the example below. But you can fill them in by hand if you really want to.

`domain`	*`domain`* describes what kind of socket you're interested in. This can, believe me, be a wide variety of things, but since this is a socket guide, it's going to be `PF_INET` for IPv4, and `PF_INET6` for IPv6.
`type`	Also, the *`type`* parameter can be a number of things, but you'll probably be setting it to either `SOCK_STREAM` for reliable TCP sockets (**`send()`**, **`recv()`**) or `SOCK_DGRAM` for unreliable fast UDP sockets (**`sendto()`**, **`recvfrom()`**.) (Another interesting socket type is `SOCK_RAW` which can be used to construct packets by hand. It's pretty cool.)
`protocol`	Finally, the *`protocol`* parameter tells which protocol to use with a certain socket type. Like I've already said, for instance, `SOCK_STREAM` uses TCP. Fortunately for you, when using `SOCK_STREAM` or `SOCK_DGRAM`, you can just set the protocol to 0, and it'll use the proper protocol automatically. Otherwise, you can use **`getprotobyname()`** to look up the proper protocol number.

Return Value

The new socket descriptor to be used in subsequent calls, or `-1` on error (and **`errno`** will be set accordingly.)

Example

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;     // AF_INET, AF_INET6, or AF_UNSPEC
hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM or SOCK_DGRAM

getaddrinfo("www.example.com", "3490", &hints, &res);

// make a socket using the information gleaned from getaddrinfo():
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

See Also

`accept()`, **`bind()`**, **`getaddrinfo()`**, **`listen()`**

9.24. `struct sockaddr` and pals

Structures for handling internet addresses

Prototypes

```
include <netinet/in.h>

// All pointers to socket address structures are often cast to pointers
// to this type before use in various functions and system calls:

struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};


// IPv4 AF_INET sockets:

struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};

struct in_addr {
    unsigned long s_addr;          // load with inet_pton()
};


// IPv6 AF_INET6 sockets:

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char   s6_addr[16];   // load with inet_pton()
};


// General socket address holding structure, big enough to hold either
// struct sockaddr_in or struct sockaddr_in6 data:

struct sockaddr_storage {
    sa_family_t  ss_family;     // address family

    // all this is padding, implementation specific, ignore it:
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

Description

These are the basic structures for all syscalls and functions that deal with internet addresses. Often you'll use **`getaddinfo()`** to fill these structures out, and then will read them when you have to.

In memory, the `struct sockaddr_in` and `struct sockaddr_in6` share the same beginning structure as `struct sockaddr`, and you can freely cast the pointer of one type to the other without any harm, except the possible end of the universe.

Just kidding on that end-of-the-universe thing...if the universe does end when you cast a `struct sockaddr_in*` to a `struct sockaddr*`, I promise you it's pure coincidence and you shouldn't even worry about it.

So, with that in mind, remember that whenever a function says it takes a `struct sockaddr*` you can cast your `struct sockaddr_in*`, `struct sockaddr_in6*`, or `struct sockadd_storage*` to that type with ease and safety.

`struct sockaddr_in` is the structure used with IPv4 addresses (e.g. "192.0.2.10"). It holds an address family (`AF_INET`), a port in *`sin_port`*, and an IPv4 address in *`sin_addr`*.

There's also this `sin_zero` field in `struct sockaddr_in` which some people claim must be set to zero. Other people don't claim anything about it (the Linux documentation doesn't even mention it at all), and setting it to zero doesn't seem to be actually necessary. So, if you feel like it, set it to zero using **`memset()`**.

Now, that `struct in_addr` is a weird beast on different systems. Sometimes it's a crazy `union` with all kinds of `#define`s and other nonsense. But what you should do is only use the *`s_addr`* field in this structure, because many systems only implement that one.

`struct sockadd_in6` and `struct in6_addr` are very similar, except they're used for IPv6.

`struct sockaddr_storage` is a struct you can pass to **`accept()`** or **`recvfrom()`** when you're trying to write IP version-agnostic code and you don't know if the new address is going to be IPv4 or IPv6. The `struct sockaddr_storage` structure is large enough to hold both types, unlike the original small `struct sockaddr`.

Example

```
// IPv4:

struct sockaddr_in ip4addr;
int s;

ip4addr.sin_family = AF_INET;
ip4addr.sin_port = htons(3490);
inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip4addr, sizeof ip4addr);
```

```
// IPv6:

struct sockaddr_in6 ip6addr;
int s;

ip6addr.sin6_family = AF_INET6;
ip6addr.sin6_port = htons(4950);
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr);

s = socket(PF_INET6, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip6addr, sizeof ip6addr);
```

See Also

`accept()`, **`bind()`**, **`connect()`**, **`inet_aton()`**, **`inet_ntoa()`**

10. More References

You've come this far, and now you're screaming for more! Where else can you go to learn more about all this stuff?

10.1. Books

For old-school actual hold-it-in-your-hand pulp paper books, try some of the following excellent books. I used to be an affiliate with a very popular internet bookseller, but their new customer tracking system is incompatible with a print document. As such, I get no more kickbacks. If you feel compassion for my plight, paypal a donation to `beej@beej.us`. `:-)`

Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Prentice Hall. ISBNs for volumes 1-2: 0131411551[43], 0130810819[44].

Internetworking with TCP/IP, volumes I-III by Douglas E. Comer and David L. Stevens. Published by Prentice Hall. ISBNs for volumes I, II, and III: 0131876716[45], 0130319961[46], 0130320714[47].

TCP/IP Illustrated, volumes 1-3 by W. Richard Stevens and Gary R. Wright. Published by Addison Wesley. ISBNs for volumes 1, 2, and 3 (and a 3-volume set): 0201633469[48], 020163354X[49], 0201634953[50], (0201776316[51]).

TCP/IP Network Administration by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 0596002971[52].

Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN 0201433079[53].

10.2. Web References

On the web:

BSD Sockets: A Quick And Dirty Primer[54] (Unix system programming info, too!)

The Unix Socket FAQ[55]

Intro to TCP/IP[56]

TCP/IP FAQ[57]

The Winsock FAQ[58]

And here are some relevant Wikipedia pages:

Berkeley Sockets[59]

Internet Protocol (IP)[60]

43. `http://beej.us/guide/url/unixnet1`
44. `http://beej.us/guide/url/unixnet2`
45. `http://beej.us/guide/url/intertcp1`
46. `http://beej.us/guide/url/intertcp2`
47. `http://beej.us/guide/url/intertcp3`
48. `http://beej.us/guide/url/tcpi1`
49. `http://beej.us/guide/url/tcpi2`
50. `http://beej.us/guide/url/tcpi3`
51. `http://beej.us/guide/url/tcpi123`
52. `http://beej.us/guide/url/tcpna`
53. `http://beej.us/guide/url/advunix`
54. `http://www.frostbytes.com/~jimf/papers/sockets/sockets.html`
55. `http://www.developerweb.net/forum/forumdisplay.php?f=70`
56. `http://pclt.cis.yale.edu/pclt/COMM/TCPIP.HTM`
57. `http://www.faqs.org/faqs/internet/tcp-ip/tcp-ip-faq/part1/`
58. `http://tangentsoft.net/wskfaq/`
59. `http://en.wikipedia.org/wiki/Berkeley_sockets`
60. `http://en.wikipedia.org/wiki/Internet_Protocol`

Transmission Control Protocol (TCP)[61]

User Datagram Protocol (UDP)[62]

Client-Server[63]

Serialization[64] (packing and unpacking data)

10.3. RFCs

RFCs[65]—the real dirt! These are documents that describe assigned numbers, programming APIs, and protocols that are used on the Internet. I've included links to a few of them here for your enjoyment, so grab a bucket of popcorn and put on your thinking cap:

RFC 1[66]—The First RFC; this gives you an idea of what the "Internet" was like just as it was coming to life, and an insight into how it was being designed from the ground up. (This RFC is completely obsolete, obviously!)

RFC 768[67]—The User Datagram Protocol (UDP)

RFC 791[68]—The Internet Protocol (IP)

RFC 793[69]—The Transmission Control Protocol (TCP)

RFC 854[70]—The Telnet Protocol

RFC 959[71]—File Transfer Protocol (FTP)

RFC 1350[72]—The Trivial File Transfer Protocol (TFTP)

RFC 1459[73]—Internet Relay Chat Protocol (IRC)

RFC 1918[74]—Address Allocation for Private Internets

RFC 2131[75]—Dynamic Host Configuration Protocol (DHCP)

RFC 2616[76]—Hypertext Transfer Protocol (HTTP)

RFC 2821[77]—Simple Mail Transfer Protocol (SMTP)

RFC 3330[78]—Special-Use IPv4 Addresses

RFC 3493[79]—Basic Socket Interface Extensions for IPv6

RFC 3542[80]—Advanced Sockets Application Program Interface (API) for IPv6

RFC 3849[81]—IPv6 Address Prefix Reserved for Documentation

RFC 3920[82]—Extensible Messaging and Presence Protocol (XMPP)

61. http://en.wikipedia.org/wiki/Transmission_Control_Protocol
62. http://en.wikipedia.org/wiki/User_Datagram_Protocol
63. http://en.wikipedia.org/wiki/Client-server
64. http://en.wikipedia.org/wiki/Serialization
65. http://www.rfc-editor.org/
66. http://tools.ietf.org/html/rfc1
67. http://tools.ietf.org/html/rfc768
68. http://tools.ietf.org/html/rfc791
69. http://tools.ietf.org/html/rfc793
70. http://tools.ietf.org/html/rfc854
71. http://tools.ietf.org/html/rfc959
72. http://tools.ietf.org/html/rfc1350
73. http://tools.ietf.org/html/rfc1459
74. http://tools.ietf.org/html/rfc1918
75. http://tools.ietf.org/html/rfc2131
76. http://tools.ietf.org/html/rfc2616
77. http://tools.ietf.org/html/rfc2821
78. http://tools.ietf.org/html/rfc3330
79. http://tools.ietf.org/html/rfc3493
80. http://tools.ietf.org/html/rfc3542
81. http://tools.ietf.org/html/rfc3849
82. http://tools.ietf.org/html/rfc3920

RFC 3977[83]—Network News Transfer Protocol (NNTP)

RFC 4193[84]—Unique Local IPv6 Unicast Addresses

RFC 4506[85]—External Data Representation Standard (XDR)

The IETF has a nice online tool for searching and browsing RFCs[86].

83. http://tools.ietf.org/html/rfc3977
84. http://tools.ietf.org/html/rfc4193
85. http://tools.ietf.org/html/rfc4506
86. http://tools.ietf.org/rfc/

Index

18

Paging: Introduction

It is sometimes said that the operating system takes one of two approaches when solving most any space-management problem. The first approach is to chop things up into *variable-sized* pieces, as we saw with **segmentation** in virtual memory. Unfortunately, this solution has inherent difficulties. In particular, when dividing a space into different-size chunks, the space itself can become **fragmented**, and thus allocation becomes more challenging over time.

Thus, it may be worth considering the second approach: to chop up space into *fixed-sized* pieces. In virtual memory, we call this idea **paging**, and it goes back to an early and important system, the Atlas [KE+62, L78]. Instead of splitting up a process's address space into some number of variable-sized logical segments (e.g., code, heap, stack), we divide it into fixed-sized units, each of which we call a **page**. Correspondingly, we view physical memory as an array of fixed-sized slots called **page frames**; each of these frames can contain a single virtual-memory page. Our challenge:

THE CRUX:
HOW TO VIRTUALIZE MEMORY WITH PAGES

How can we virtualize memory with pages, so as to avoid the problems of segmentation? What are the basic techniques? How do we make those techniques work well, with minimal space and time overheads?

18.1 A Simple Example And Overview

To help make this approach more clear, let's illustrate it with a simple example. Figure 18.1 (page 2) presents an example of a tiny address space, only 64 bytes total in size, with four 16-byte pages (virtual pages 0, 1, 2, and 3). Real address spaces are much bigger, of course, commonly 32 bits and thus 4-GB of address space, or even 64 bits[1]; in the book, we'll often use tiny examples to make them easier to digest.

[1] A 64-bit address space is hard to imagine, it is so amazingly large. An analogy might help: if you think of a 32-bit address space as the size of a tennis court, a 64-bit address space is about the size of Europe(!).

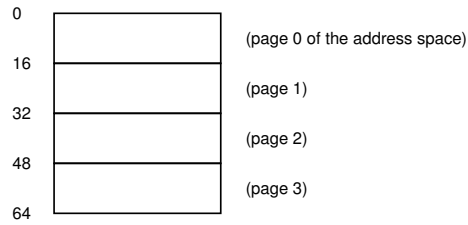


Figure 18.1: **A Simple 64-byte Address Space**

Physical memory, as shown in Figure 18.2, also consists of a number of fixed-sized slots, in this case eight page frames (making for a 128-byte physical memory, also ridiculously small). As you can see in the diagram, the pages of the virtual address space have been placed at different locations throughout physical memory; the diagram also shows the OS using some of physical memory for itself.

Paging, as we will see, has a number of advantages over our previous approaches. Probably the most important improvement will be *flexibility*: with a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space; we won't, for example, make assumptions about the direction the heap and stack grow and how they are used.

Another advantage is the *simplicity* of free-space management that paging affords. For example, when the OS wishes to place our tiny 64-byte address space into our eight-page physical memory, it simply finds four free pages; perhaps the OS keeps a **free list** of all free pages for this, and just grabs the first four free pages off of this list. In the example, the OS

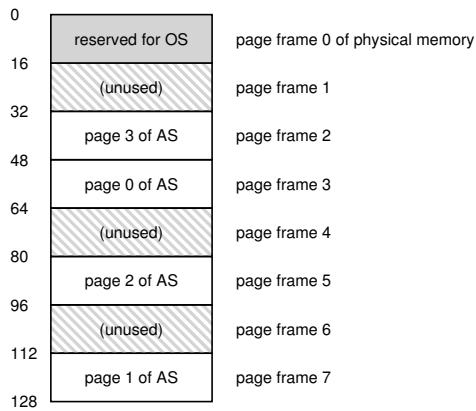


Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

has placed virtual page 0 of the address space (AS) in physical frame 3, virtual page 1 of the AS in physical frame 7, page 2 in frame 5, and page 3 in frame 2. Page frames 1, 4, and 6 are currently free.

To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a *per-process* data structure known as a **page table**. The major role of the page table is to store **address translations** for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides. For our simple example (Figure 18.2, page 2), the page table would thus have the following four entries: (Virtual Page 0 $\rightarrow$ Physical Frame 3), (VP 1 $\rightarrow$ PF 7), (VP 2 $\rightarrow$ PF 5), and (VP 3 $\rightarrow$ PF 2).

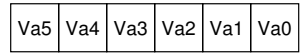
It is important to remember that this page table is a *per-process* data structure (most page table structures we discuss are per-process structures; an exception we'll touch on is the **inverted page table**). If another process were to run in our example above, the OS would have to manage a different page table for it, as its virtual pages obviously map to *different* physical pages (modulo any sharing going on).

Now, we know enough to perform an address-translation example. Let's imagine the process with that tiny address space (64 bytes) is performing a memory access:

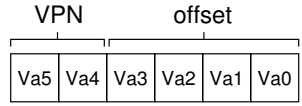
```
movl <virtual address>, %eax
```

Specifically, let's pay attention to the explicit load of the data from address `<virtual address>` into the register `eax` (and thus ignore the instruction fetch that must have happened prior).

To **translate** this virtual address that the process generated, we have to first split it into two components: the **virtual page number (VPN)**, and the **offset** within the page. For this example, because the virtual address space of the process is 64 bytes, we need 6 bits total for our virtual address ($2^6 = 64$). Thus, our virtual address can be conceptualized as follows:



In this diagram, Va5 is the highest-order bit of the virtual address, and Va0 the lowest-order bit. Because we know the page size (16 bytes), we can further divide the virtual address as follows:

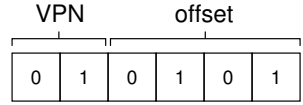


The page size is 16 bytes in a 64-byte address space; thus we need to be able to select 4 pages, and the top 2 bits of the address do just that. Thus, we have a 2-bit virtual page number (VPN). The remaining bits tell us which byte of the page we are interested in, 4 bits in this case; we call this the offset.

When a process generates a virtual address, the OS and hardware must combine to translate it into a meaningful physical address. For example, let us assume the load above was to virtual address 21:

```
movl 21, %eax
```

Turning "21" into binary form, we get "010101", and thus we can examine this virtual address and see how it breaks down into a virtual page number (VPN) and offset:



Thus, the virtual address "21" is on the 5th ("0101"th) byte of virtual page "01" (or 1). With our virtual page number, we can now index our page table and find which physical frame virtual page 1 resides within. In the page table above the physical frame number (PFN) (also sometimes called the **physical page number** or **PPN**) is 7 (binary 111). Thus, we can translate this virtual address by replacing the VPN with the PFN and then issue the load to physical memory (Figure 18.3).

Note the offset stays the same (i.e., it is not translated), because the offset just tells us which byte *within* the page we want. Our final physical address is 1110101 (117 in decimal), and is exactly where we want our load to fetch data from (Figure 18.2, page 2).

With this basic overview in mind, we can now ask (and hopefully, answer) a few basic questions you may have about paging. For example, where are these page tables stored? What are the typical contents of the page table, and how big are the tables? Does paging make the system (too) slow? These and other beguiling questions are answered, at least in part, in the text below. Read on!

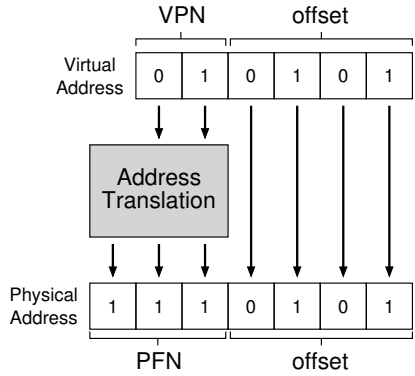


Figure 18.3: **The Address Translation Process**

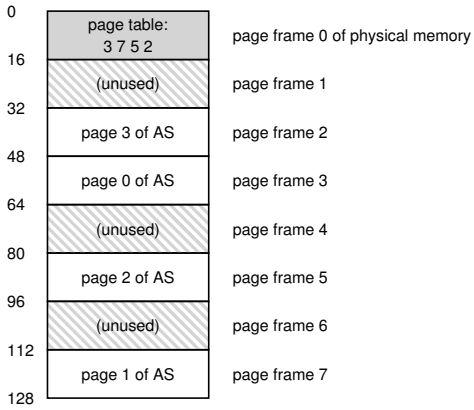


Figure 18.4: **Example: Page Table in Kernel Physical Memory**

18.2 Where Are Page Tables Stored?

Page tables can get terribly large, much bigger than the small segment table or base/bounds pair we have discussed previously. For example, imagine a typical 32-bit address space, with 4KB pages. This virtual address splits into a 20-bit VPN and 12-bit offset (recall that 10 bits would be needed for a 1KB page size, and just add two more to get to 4KB).

A 20-bit VPN implies that there are 2^{20} translations that the OS would have to manage for each process (that's roughly a million); assuming we need 4 bytes per **page table entry (PTE)** to hold the physical translation plus any other useful stuff, we get an immense 4MB of memory needed for each page table! That is pretty large. Now imagine there are 100 processes running: this means the OS would need 400MB of memory just for all those address translations! Even in the modern era, where machines have gigabytes of memory, it seems a little crazy to use a large chunk of it just for translations, no? And we won't even think about how big such a page table would be for a 64-bit address space; that would be too gruesome and perhaps scare you off entirely.

Because page tables are so big, we don't keep any special on-chip hardware in the MMU to store the page table of the currently-running process. Instead, we store the page table for each process in *memory* somewhere. Let's assume for now that the page tables live in physical memory that the OS manages; later we'll see that much of OS memory itself can be virtualized, and thus page tables can be stored in OS virtual memory (and even swapped to disk), but that is too confusing right now, so we'll ignore it. In Figure 18.4 is a picture a page table in OS memory; see the tiny set of translations in there?

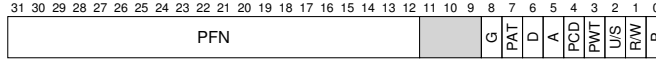


Figure 18.5: **An x86 Page Table Entry (PTE)**

18.3 What's Actually In The Page Table?

Let's talk a little about page table organization. The page table is just a data structure that is used to map virtual addresses (or really, virtual page numbers) to physical addresses (physical frame numbers). Thus, any data structure could work. The simplest form is called a **linear page table**, which is just an array. The OS *indexes* the array by the virtual page number (VPN), and looks up the page-table entry (PTE) at that index in order to find the desired physical frame number (PFN). For now, we will assume this simple linear structure; in later chapters, we will make use of more advanced data structures to help solve some problems with paging.

As for the contents of each PTE, we have a number of different bits in there worth understanding at some level. A **valid bit** is common to indicate whether the particular translation is valid; for example, when a program starts running, it will have code and heap at one end of its address space, and the stack at the other. All the unused space in-between will be marked **invalid**, and if the process tries to access such memory, it will generate a trap to the OS which will likely terminate the process. Thus, the valid bit is crucial for supporting a sparse address space; by simply marking all the unused pages in the address space invalid, we remove the need to allocate physical frames for those pages and thus save a great deal of memory.

We also might have **protection bits**, indicating whether the page could be read from, written to, or executed from. Again, accessing a page in a way not allowed by these bits will generate a trap to the OS.

There are a couple of other bits that are important but we won't talk about much for now. A **present bit** indicates whether this page is in physical memory or on disk (i.e., it has been **swapped out**). We will understand this machinery further when we study how to **swap** parts of the address space to disk to support address spaces that are larger than physical memory; swapping allows the OS to free up physical memory by moving rarely-used pages to disk. A **dirty bit** is also common, indicating whether the page has been modified since it was brought into memory.

A **reference bit** (a.k.a. **accessed bit**) is sometimes used to track whether a page has been accessed, and is useful in determining which pages are popular and thus should be kept in memory; such knowledge is critical during **page replacement**, a topic we will study in great detail in subsequent chapters.

Figure 18.5 shows an example page table entry from the x86 architecture [I09]. It contains a present bit (P); a read/write bit (R/W) which determines if writes are allowed to this page; a user/supervisor bit (U/S)

which determines if user-mode processes can access the page; a few bits (PWT, PCD, PAT, and G) that determine how hardware caching works for these pages; an accessed bit (A) and a dirty bit (D); and finally, the page frame number (PFN) itself.

Read the Intel Architecture Manuals [I09] for more details on x86 paging support. Be forewarned, however; reading manuals such as these, while quite informative (and certainly necessary for those who write code to use such page tables in the OS), can be challenging at first. A little patience, and a lot of desire, is required.

18.4 Paging: Also Too Slow

With page tables in memory, we already know that they might be too big. As it turns out, they can slow things down too. For example, take our simple instruction:

```
movl 21, %eax
```

Again, let's just examine the explicit reference to address 21 and not worry about the instruction fetch. In this example, we'll assume the hardware performs the translation for us. To fetch the desired data, the system must first **translate** the virtual address (21) into the correct physical address (117). Thus, before fetching the data from address 117, the system must first fetch the proper page table entry from the process's page table, perform the translation, and then load the data from physical memory.

To do so, the hardware must know where the page table is for the currently-running process. Let's assume for now that a single **page-table base register** contains the physical address of the starting location of the page table. To find the location of the desired PTE, the hardware will thus perform the following functions:

```
VPN     = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

In our example, `VPN_MASK` would be set to 0x30 (hex 30, or binary 110000) which picks out the VPN bits from the full virtual address; `SHIFT` is set to 4 (the number of bits in the offset), such that we move the VPN bits down to form the correct integer virtual page number. For example, with virtual address 21 (010101), and masking turns this value into 010000; the shift turns it into 01, or virtual page 1, as desired. We then use this value as an index into the array of PTEs pointed to by the page table base register.

Once this physical address is known, the hardware can fetch the PTE from memory, extract the PFN, and concatenate it with the offset from the virtual address to form the desired physical address. Specifically, you can think of the PFN being left-shifted by `SHIFT`, and then logically OR'd with the offset to form the final address as follows:

```
offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

```
1  // Extract the VPN from the virtual address
2  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4  // Form the address of the page-table entry (PTE)
5  PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7  // Fetch the PTE
8  PTE = AccessMemory(PTEAddr)
9
10 // Check if process can access the page
11 if (PTE.Valid == False)
12     RaiseException(SEGMENTATION_FAULT)
13 else if (CanAccess(PTE.ProtectBits) == False)
14     RaiseException(PROTECTION_FAULT)
15 else
16     // Access is OK: form physical address and fetch it
17     offset   = VirtualAddress & OFFSET_MASK
18     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19     Register = AccessMemory(PhysAddr)
```

Figure 18.6: **Accessing Memory With Paging**

Finally, the hardware can fetch the desired data from memory and put it into register `eax`. The program has now succeeded at loading a value from memory!

To summarize, we now describe the initial protocol for what happens on each memory reference. Figure 18.6 shows the basic approach. For every memory reference (whether an instruction fetch or an explicit load or store), paging requires us to perform one extra memory reference in order to first fetch the translation from the page table. That is a lot of work! Extra memory references are costly, and in this case will likely slow down the process by a factor of two or more.

And now you can hopefully see that there are *two* real problems that we must solve. Without careful design of both hardware and software, page tables will cause the system to run too slowly, as well as take up too much memory. While seemingly a great solution for our memory virtualization needs, these two crucial problems must first be overcome.

18.5 A Memory Trace

Before closing, we now trace through a simple memory access example to demonstrate all of the resulting memory accesses that occur when using paging. The code snippet (in C, in a file called `array.c`) that we are interested in is as follows:

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

We compile `array.c` and run it with the following commands:

ASIDE: **DATA STRUCTURE — THE PAGE TABLE**
One of the most important data structures in the memory management subsystem of a modern OS is the **page table**. In general, a page table stores **virtual-to-physical address translations**, thus letting the system know where each page of an address space actually resides in physical memory. Because each address space requires such translations, in general there is one page table per process in the system. The exact structure of the page table is either determined by the hardware (older systems) or can be more flexibly managed by the OS (modern systems).

```
prompt> gcc -o array array.c -Wall -O
prompt> ./array
```

Of course, to truly understand what memory accesses this code snippet (which simply initializes an array) will make, we'll have to know (or assume) a few more things. First, we'll have to **disassemble** the resulting binary (using `objdump` on Linux, or `otool` on a Mac) to see what assembly instructions are used to initialize the array in a loop. Here is the resulting assembly code:

```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne  0x1024
```

The code, if you know a little **x86**, is actually quite easy to understand[2]. The first instruction moves the value zero (shown as `$0x0`) into the virtual memory address of the location of the array; this address is computed by taking the contents of `%edi` and adding `%eax` multiplied by four to it. Thus, `%edi` holds the base address of the array, whereas `%eax` holds the array index (`i`); we multiply by four because the array is an array of integers, each of size four bytes.

The second instruction increments the array index held in `%eax`, and the third instruction compares the contents of that register to the hex value `0x03e8`, or decimal 1000. If the comparison shows that two values are not yet equal (which is what the `jne` instruction tests), the fourth instruction jumps back to the top of the loop.

To understand which memory accesses this instruction sequence makes (at both the virtual and physical levels), we'll have to assume something about where in virtual memory the code snippet and array are found, as well as the contents and location of the page table.

For this example, we assume a virtual address space of size 64KB (unrealistically small). We also assume a page size of 1KB.

[2]We are cheating a little bit here, assuming each instruction is four bytes in size for simplicity; in actuality, x86 instructions are variable-sized.

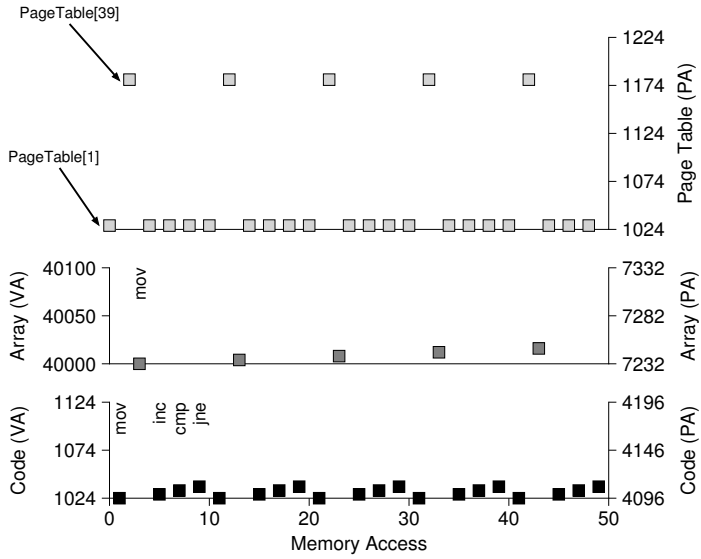


Figure 18.7: **A Virtual (And Physical) Memory Trace**

All we need to know now are the contents of the page table, and its location in physical memory. Let's assume we have a linear (array-based) page table and that it is located at physical address 1KB (1024).

As for its contents, there are just a few virtual pages we need to worry about having mapped for this example. First, there is the virtual page the code lives on. Because the page size is 1KB, virtual address 1024 resides on the second page of the virtual address space (VPN=1, as VPN=0 is the first page). Let's assume this virtual page maps to physical frame 4 (VPN 1 $\rightarrow$ PFN 4).

Next, there is the array itself. Its size is 4000 bytes (1000 integers), and we assume that it resides at virtual addresses 40000 through 44000 (not including the last byte). The virtual pages for this decimal range are VPN=39 ... VPN=42. Thus, we need mappings for these pages. Let's assume these virtual-to-physical mappings for the example: (VPN 39 $\rightarrow$ PFN 7), (VPN 40 $\rightarrow$ PFN 8), (VPN 41 $\rightarrow$ PFN 9), (VPN 42 $\rightarrow$ PFN 10).

We are now ready to trace the memory references of the program. When it runs, each instruction fetch will generate two memory references: one to the page table to find the physical frame that the instruction resides within, and one to the instruction itself to fetch it to the CPU for processing. In addition, there is one explicit memory reference in the form of the `mov` instruction; this adds another page table access first (to translate the array virtual address to the correct physical one) and then the array access itself.

The entire process, for the first five loop iterations, is depicted in Figure 18.7 (page 10). The bottom most graph shows the instruction memory references on the y-axis in black (with virtual addresses on the left, and the actual physical addresses on the right); the middle graph shows array accesses in dark gray (again with virtual on left and physical on right); finally, the topmost graph shows page table memory accesses in light gray (just physical, as the page table in this example resides in physical memory). The x-axis, for the entire trace, shows memory accesses across the first five iterations of the loop; there are 10 memory accesses per loop, which includes four instruction fetches, one explicit update of memory, and five page table accesses to translate those four fetches and one explicit update.

See if you can make sense of the patterns that show up in this visualization. In particular, what will change as the loop continues to run beyond these first five iterations? Which new memory locations will be accessed? Can you figure it out?

This has just been the simplest of examples (only a few lines of C code), and yet you might already be able to sense the complexity of understanding the actual memory behavior of real applications. Don't worry: it definitely gets worse, because the mechanisms we are about to introduce only complicate this already complex machinery. Sorry[3]!

18.6 Summary

We have introduced the concept of **paging** as a solution to our challenge of virtualizing memory. Paging has many advantages over previous approaches (such as segmentation). First, it does not lead to external fragmentation, as paging (by design) divides memory into fixed-sized units. Second, it is quite flexible, enabling the sparse use of virtual address spaces.

However, implementing paging support without care will lead to a slower machine (with many extra memory accesses to access the page table) as well as memory waste (with memory filled with page tables instead of useful application data). We'll thus have to think a little harder to come up with a paging system that not only works, but works well. The next two chapters, fortunately, will show us how to do so.

[3] We're not really sorry. But, we are sorry about not being sorry, if that makes sense.

References

[KE+62] "One-level Storage System"
T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner
IRE Trans. EC-11, 2 (1962), pp. 223-235
(Reprinted in Bell and Newell, "Computer Structures: Readings and Examples" McGraw-Hill, New York, 1971).
The Atlas pioneered the idea of dividing memory into fixed-sized pages and in many senses was an early form of the memory-management ideas we see in modern computer systems.

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"
Intel, 2009
Available: http://www.intel.com/products/processor/manuals
In particular, pay attention to "Volume 3A: System Programming Guide Part 1" and "Volume 3B: System Programming Guide Part 2"

[L78] "The Manchester Mark I and atlas: a historical perspective"
S. H. Lavington
Communications of the ACM archive
Volume 21, Issue 1 (January 1978), pp. 4-12
Special issue on computer architecture
This paper is a great retrospective of some of the history of the development of some important computer systems. As we sometimes forget in the US, many of these new ideas came from overseas.

Homework

In this homework, you will use a simple program, which is known as `paging-linear-translate.py`, to see if you understand how simple virtual-to-physical address translation works with linear page tables. See the README for details.

Questions

- Before doing any translations, let's use the simulator to study how linear page tables change size given different parameters. Compute the size of linear page tables as different parameters change. Some suggested inputs are below; by using the `-v flag`, you can see how many page-table entries are filled.
  First, to understand how linear page table size changes as the address space grows:

  ```
  paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
  paging-linear-translate.py -P 1k -a 2m -p 512m -v -n 0
  paging-linear-translate.py -P 1k -a 4m -p 512m -v -n 0
  ```

  Then, to understand how linear page table size changes as page size grows:

  ```
  paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
  paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0
  paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0
  ```

  Before running any of these, try to think about the expected trends. How should page-table size change as the address space grows? As the page size grows? Why shouldn't we just use really big pages in general?
- Now let's do some translations. Start with some small examples, and change the number of pages that are allocated to the address space with the `-u flag`. For example:

  ```
  paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0
  paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
  paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
  paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
  paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
  ```

  What happens as you increase the percentage of pages that are allocated in each address space?
- Now let's try some different random seeds, and some different (and sometimes quite crazy) address-space parameters, for variety:

  ```
  paging-linear-translate.py -P 8  -a 32   -p 1024 -v -s 1
  paging-linear-translate.py -P 8k -a 32k  -p 1m   -v -s 2
  paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3
  ```

  Which of these parameter combinations are unrealistic? Why?

- Use the program to try out some other problems. Can you find the limits of where the program doesn't work anymore? For example, what happens if the address-space size is *bigger* than physical memory?

13

The Abstraction: Address Spaces

In the early days, building computer systems was easy. Why, you ask? Because users didn't expect much. It is those darned users with their expectations of "ease of use", "high performance", "reliability", etc., that really have led to all these headaches. Next time you meet one of those computer users, thank them for all the problems they have caused.

13.1 Early Systems

From the perspective of memory, early machines didn't provide much of an abstraction to users. Basically, the physical memory of the machine looked something like what you see in Figure 13.1.

The OS was a set of routines (a library, really) that sat in memory (starting at physical address 0 in this example), and there would be one running program (a process) that currently sat in physical memory (starting at physical address 64k in this example) and used the rest of memory. There were few illusions here, and the user didn't expect much from the OS. Life was sure easy for OS developers in those days, wasn't it?

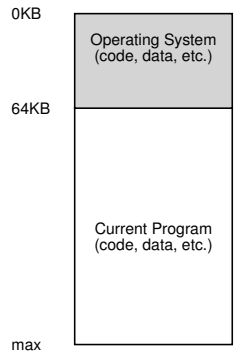


Figure 13.1: **Operating Systems: The Early Days**

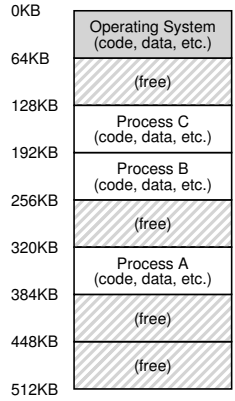


Figure 13.2: **Three Processes: Sharing Memory**

13.2 Multiprogramming and Time Sharing

After a time, because machines were expensive, people began to share machines more effectively. Thus the era of **multiprogramming** was born [DV66], in which multiple processes were ready to run at a given time, and the OS would switch between them, for example when one decided to perform an I/O. Doing so increased the effective **utilization** of the CPU. Such increases in **efficiency** were particularly important in those days where each machine cost hundreds of thousands or even millions of dollars (and you thought your Mac was expensive!).

Soon enough, however, people began demanding more of machines, and the era of **time sharing** was born [S59, L60, M62, M83]. Specifically, many realized the limitations of batch computing, particularly on programmers themselves [CV65], who were tired of long (and hence ineffective) program-debug cycles. The notion of **interactivity** became important, as many users might be concurrently using a machine, each waiting for (or hoping for) a timely response from their currently-executing tasks.

One way to implement time sharing would be to run one process for a short while, giving it full access to all memory (Figure 13.1, page 1), then stop it, save all of its state to some kind of disk (including all of physical memory), load some other process's state, run it for a while, and thus implement some kind of crude sharing of the machine [M+63].

Unfortunately, this approach has a big problem: it is way too slow, particularly as memory grows. While saving and restoring register-level state (the PC, general-purpose registers, etc.) is relatively fast, saving the entire contents of memory to disk is brutally non-performant. Thus, what we'd rather do is leave processes in memory while switching between them, allowing the OS to implement time sharing efficiently (Figure 13.2).

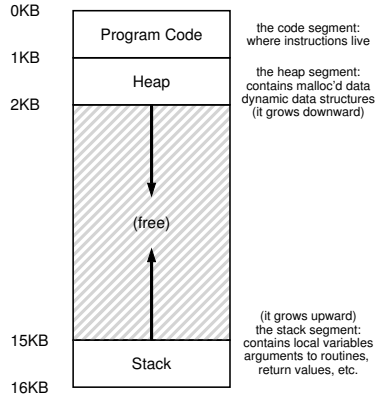


Figure 13.3: **An Example Address Space**

In the diagram, there are three processes (A, B, and C) and each of them have a small part of the 512KB physical memory carved out for them. Assuming a single CPU, the OS chooses to run one of the processes (say A), while the others (B and C) sit in the ready queue waiting to run.

As time sharing became more popular, you can probably guess that new demands were placed on the operating system. In particular, allowing multiple programs to reside concurrently in memory makes **protection** an important issue; you don't want a process to be able to read, or worse, write some other process's memory.

13.3 The Address Space

However, we have to keep those pesky users in mind, and doing so requires the OS to create an **easy to use** abstraction of physical memory. We call this abstraction the **address space**, and it is the running program's view of memory in the system. Understanding this fundamental OS abstraction of memory is key to understanding how memory is virtualized.

The address space of a process contains all of the memory state of the running program. For example, the **code** of the program (the instructions) have to live in memory somewhere, and thus they are in the address space. The program, while it is running, uses a **stack** to keep track of where it is in the function call chain as well as to allocate local variables and pass parameters and return values to and from routines. Finally, the **heap** is used for dynamically-allocated, user-managed memory, such as that you might receive from a call to `malloc()` in C or `new` in an object-oriented language such as C++ or Java. Of course, there are other things in there too (e.g., statically-initialized variables), but for now let us just assume those three components: code, stack, and heap.

In the example in Figure 13.3 (page 3), we have a tiny address space (only 16KB)[1]. The program code lives at the top of the address space (starting at 0 in this example, and is packed into the first 1K of the address space). Code is static (and thus easy to place in memory), so we can place it at the top of the address space and know that it won't need any more space as the program runs.

Next, we have the two regions of the address space that may grow (and shrink) while the program runs. Those are the heap (at the top) and the stack (at the bottom). We place them like this because each wishes to be able to grow, and by putting them at opposite ends of the address space, we can allow such growth: they just have to grow in opposite directions. The heap thus starts just after the code (at 1KB) and grows downward (say when a user requests more memory via `malloc()`); the stack starts at 16KB and grows upward (say when a user makes a procedure call). However, this placement of stack and heap is just a convention; you could arrange the address space in a different way if you'd like (as we'll see later, when multiple **threads** co-exist in an address space, no nice way to divide the address space like this works anymore, alas).

Of course, when we describe the address space, what we are describing is the **abstraction** that the OS is providing to the running program. The program really isn't in memory at physical addresses 0 through 16KB; rather it is loaded at some arbitrary physical address(es). Examine processes A, B, and C in Figure 13.2; there you can see how each process is loaded into memory at a different address. And hence the problem:

THE CRUX: HOW TO VIRTUALIZE MEMORY

How can the OS build this abstraction of a private, potentially large address space for multiple running processes (all sharing memory) on top of a single, physical memory?

When the OS does this, we say the OS is **virtualizing memory**, because the running program thinks it is loaded into memory at a particular address (say 0) and has a potentially very large address space (say 32-bits or 64-bits); the reality is quite different.

When, for example, process A in Figure 13.2 tries to perform a load at address 0 (which we will call a **virtual address**), somehow the OS, in tandem with some hardware support, will have to make sure the load doesn't actually go to physical address 0 but rather to physical address 320KB (where A is loaded into memory). This is the key to virtualization of memory, which underlies every modern computer system in the world.

[1] We will often use small examples like this because (a) it is a pain to represent a 32-bit address space and (b) the math is harder. We like simple math.

TIP: THE PRINCIPLE OF ISOLATION
Isolation is a key principle in building reliable systems. If two entities are properly isolated from one another, this implies that one can fail without affecting the other. Operating systems strive to isolate processes from each other and in this way prevent one from harming the other. By using memory isolation, the OS further ensures that running programs cannot affect the operation of the underlying OS. Some modern OS's take isolation even further, by walling off pieces of the OS from other pieces of the OS. Such **microkernels** [BH70, R+89, S+03] thus may provide greater reliability than typical monolithic kernel designs.

13.4 Goals

Thus we arrive at the job of the OS in this set of notes: to virtualize memory. The OS will not only virtualize memory, though; it will do so with style. To make sure the OS does so, we need some goals to guide us. We have seen these goals before (think of the Introduction), and we'll see them again, but they are certainly worth repeating.

One major goal of a virtual memory (VM) system is **transparency**[2]. The OS should implement virtual memory in a way that is invisible to the running program. Thus, the program shouldn't be aware of the fact that memory is virtualized; rather, the program behaves as if it has its own private physical memory. Behind the scenes, the OS (and hardware) does all the work to multiplex memory among many different jobs, and hence implements the illusion.

Another goal of VM is **efficiency**. The OS should strive to make the virtualization as **efficient** as possible, both in terms of time (i.e., not making programs run much more slowly) and space (i.e., not using too much memory for structures needed to support virtualization). In implementing time-efficient virtualization, the OS will have to rely on hardware support, including hardware features such as TLBs (which we will learn about in due course).

Finally, a third VM goal is **protection**. The OS should make sure to **protect** processes from one another as well as the OS itself from processes. When one process performs a load, a store, or an instruction fetch, it should not be able to access or affect in any way the memory contents of any other process or the OS itself (that is, anything *outside* its address space). Protection thus enables us to deliver the property of **isolation** among processes; each process should be running in its own isolated cocoon, safe from the ravages of other faulty or even malicious processes.

[2]This usage of transparency is sometimes confusing; some students think that "being transparent" means keeping everything out in the open, i.e., what government should be like. Here, it means the opposite: that the illusion provided by the OS should not be visible to applications. Thus, in common usage, a transparent system is one that is hard to notice, not one that responds to requests as stipulated by the Freedom of Information Act.

ASIDE: **EVERY ADDRESS YOU SEE IS VIRTUAL**

Ever write a C program that prints out a pointer? The value you see (some large number, often printed in hexadecimal), is a **virtual address**. Ever wonder where the code of your program is found? You can print that out too, and yes, if you can print it, it also is a virtual address. In fact, any address you can see as a programmer of a user-level program is a virtual address. It's only the OS, through its tricky techniques of virtualizing memory, that knows where in the physical memory of the machine these instructions and data values lie. So never forget: if you print out an address in a program, it's a virtual one, an illusion of how things are laid out in memory; only the OS (and the hardware) knows the real truth.

Here's a little program that prints out the locations of the `main()` routine (where code lives), the value of a heap-allocated value returned from `malloc()`, and the location of an integer on the stack:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char *argv[]) {
4     printf("location of code  : %p\n", (void *) main);
5     printf("location of heap  : %p\n", (void *) malloc(1));
6     int x = 3;
7     printf("location of stack : %p\n", (void *) &x);
8     return x;
9 }
```

When run on a 64-bit Mac OS X machine, we get the following output:

```
location of code  : 0x1095afe50
location of heap  : 0x1096008c0
location of stack : 0x7fff691aea64
```

From this, you can see that code comes first in the address space, then the heap, and the stack is all the way at the other end of this large virtual space. All of these addresses are virtual, and will be translated by the OS and hardware in order to fetch values from their true physical locations.

In the next chapters, we'll focus our exploration on the basic **mechanisms** needed to virtualize memory, including hardware and operating systems support. We'll also investigate some of the more relevant **policies** that you'll encounter in operating systems, including how to manage free space and which pages to kick out of memory when you run low on space. In doing so, we'll build up your understanding of how a modern virtual memory system really works[3].

[3] Or, we'll convince you to drop the course. But hold on; if you make it through VM, you'll likely make it all the way!

13.5 Summary

We have seen the introduction of a major OS subsystem: virtual memory. The VM system is responsible for providing the illusion of a large, sparse, private address space to programs, which hold all of their instructions and data therein. The OS, with some serious hardware help, will take each of these virtual memory references, and turn them into physical addresses, which can be presented to the physical memory in order to fetch the desired information. The OS will do this for many processes at once, making sure to protect programs from one another, as well as protect the OS. The entire approach requires a great deal of mechanism (lots of low-level machinery) as well as some critical policies to work; we'll start from the bottom up, describing the critical mechanisms first. And thus we proceed!

References

[BH70] "The Nucleus of a Multiprogramming System"
Per Brinch Hansen
Communications of the ACM, 13:4, April 1970
The first paper to suggest that the OS, or kernel, should be a minimal and flexible substrate for building customized operating systems; this theme is revisited throughout OS research history.

[CV65] "Introduction and Overview of the Multics System"
F. J. Corbato and V. A. Vyssotsky
Fall Joint Computer Conference, 1965
A great early Multics paper. Here is the great quote about time sharing: "The impetus for time-sharing first arose from professional programmers because of their constant frustration in debugging programs at batch processing installations. Thus, the original goal was to time-share computers to allow simultaneous access by several persons while giving to each of them the illusion of having the whole machine at his disposal."

[DV66] "Programming Semantics for Multiprogrammed Computations"
Jack B. Dennis and Earl C. Van Horn
Communications of the ACM, Volume 9, Number 3, March 1966
An early paper (but not the first) on multiprogramming.

[L60] "Man-Computer Symbiosis"
J. C. R. Licklider
IRE Transactions on Human Factors in Electronics, HFE-1:1, March 1960
A funky paper about how computers and people are going to enter into a symbiotic age; clearly well ahead of its time but a fascinating read nonetheless.

[M62] "Time-Sharing Computer Systems"
J. McCarthy
Management and the Computer of the Future, MIT Press, Cambridge, Mass, 1962
Probably McCarthy's earliest recorded paper on time sharing. However, in another paper [M83], he claims to have been thinking of the idea since 1957. McCarthy left the systems area and went on to become a giant in Artificial Intelligence at Stanford, including the creation of the LISP programming language. See McCarthy's home page for more info: `http://www-formal.stanford.edu/jmc/`

[M+63] "A Time-Sharing Debugging System for a Small Computer"
J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider
AFIPS '63 (Spring), May, 1963, New York, USA
A great early example of a system that swapped program memory to the "drum" when the program wasn't running, and then back into "core" memory when it was about to be run.

[M83] "Reminiscences on the History of Time Sharing"
John McCarthy
Winter or Spring of 1983
Available: http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html
A terrific historical note on where the idea of time-sharing might have come from, including some doubts towards those who cite Strachey's work [S59] as the pioneering work in this area.

[R+89] "Mach: A System Software kernel"
Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, Michael Jones
COMPCON 89, February 1989
Although not the first project on microkernels per se, the Mach project at CMU was well-known and influential; it still lives today deep in the bowels of Mac OS X.

[S59] "Time Sharing in Large Fast Computers"
C. Strachey
Proceedings of the International Conference on Information Processing, UNESCO, June 1959
One of the earliest references on time sharing.

[S+03] "Improving the Reliability of Commodity Operating Systems"
Michael M. Swift, Brian N. Bershad, Henry M. Levy
SOSP 2003
The first paper to show how microkernel-like thinking can improve operating system reliability.

15

Mechanism: Address Translation

In developing the virtualization of the CPU, we focused on a general mechanism known as **limited direct execution** (or **LDE**). The idea behind LDE is simple: for the most part, let the program run directly on the hardware; however, at certain key points in time (such as when a process issues a system call, or a timer interrupt occurs), arrange so that the OS gets involved and makes sure the "right" thing happens. Thus, the OS, with a little hardware support, tries its best to get out of the way of the running program, to deliver an *efficient* virtualization; however, by **interposing** at those critical points in time, the OS ensures that it maintains *control* over the hardware. Efficiency and control together are two of the main goals of any modern operating system.

In virtualizing memory, we will pursue a similar strategy, attaining both efficiency and control while providing the desired virtualization. Efficiency dictates that we make use of hardware support, which at first will be quite rudimentary (e.g., just a few registers) but will grow to be fairly complex (e.g., TLBs, page-table support, and so forth, as you will see). Control implies that the OS ensures that no application is allowed to access any memory but its own; thus, to protect applications from one another, and the OS from applications, we will need help from the hardware here too. Finally, we will need a little more from the VM system, in terms of *flexibility*; specifically, we'd like for programs to be able to use their address spaces in whatever way they would like, thus making the system easier to program. And thus we arrive at the refined crux:

THE CRUX:
HOW TO EFFICIENTLY AND FLEXIBLY VIRTUALIZE MEMORY

How can we build an efficient virtualization of memory? How do we provide the flexibility needed by applications? How do we maintain control over which memory locations an application can access, and thus ensure that application memory accesses are properly restricted? How do we do all of this efficiently?

The generic technique we will use, which you can consider an addition to our general approach of limited direct execution, is something that is referred to as **hardware-based address translation**, or just **address translation** for short. With address translation, the hardware transforms each memory access (e.g., an instruction fetch, load, or store), changing the **virtual** address provided by the instruction to a **physical** address where the desired information is actually located. Thus, on each and every memory reference, an address translation is performed by the hardware to redirect application memory references to their actual locations in memory.

Of course, the hardware alone cannot virtualize memory, as it just provides the low-level mechanism for doing so efficiently. The OS must get involved at key points to set up the hardware so that the correct translations take place; it must thus **manage memory**, keeping track of which locations are free and which are in use, and judiciously intervening to maintain control over how memory is used.

Once again the goal of all of this work is to create a beautiful **illusion**: that the program has its own private memory, where its own code and data reside. Behind that virtual reality lies the ugly physical truth: that many programs are actually sharing memory at the same time, as the CPU (or CPUs) switches between running one program and the next. Through virtualization, the OS (with the hardware's help) turns the ugly machine reality into something that is a useful, powerful, and easy to use abstraction.

15.1 Assumptions

Our first attempts at virtualizing memory will be very simple, almost laughably so. Go ahead, laugh all you want; pretty soon it will be the OS laughing at you, when you try to understand the ins and outs of TLBs, multi-level page tables, and other technical wonders. Don't like the idea of the OS laughing at you? Well, you may be out of luck then; that's just how the OS rolls.

Specifically, we will assume for now that the user's address space must be placed *contiguously* in physical memory. We will also assume, for simplicity, that the size of the address space is not too big; specifically, that it is *less than the size of physical memory*. Finally, we will also assume that each address space is exactly the *same size*. Don't worry if these assumptions sound unrealistic; we will relax them as we go, thus achieving a realistic virtualization of memory.

15.2 An Example

To understand better what we need to do to implement address translation, and why we need such a mechanism, let's look at a simple example. Imagine there is a process whose address space is as indicated in Figure 15.1. What we are going to examine here is a short code sequence

TIP: INTERPOSITION IS POWERFUL

Interposition is a generic and powerful technique that is often used to great effect in computer systems. In virtualizing memory, the hardware will interpose on each memory access, and translate each virtual address issued by the process to a physical address where the desired information is actually stored. However, the general technique of interposition is much more broadly applicable; indeed, almost any well-defined interface can be interposed upon, to add new functionality or improve some other aspect of the system. One of the usual benefits of such an approach is **transparency**; the interposition often is done without changing the client of the interface, thus requiring no changes to said client.

that loads a value from memory, increments it by three, and then stores the value back into memory. You can imagine the C-language representation of this code might look like this:

```
void func() {
    int x;
    x = x + 3; // this is the line of code we are interested in
```

The compiler turns this line of code into assembly, which might look something like this (in x86 assembly). Use `objdump` on Linux or `otool` on Mac OS X to disassemble it:

```
128: movl 0x0(%ebx), %eax     ;load 0+ebx into eax
132: addl $0x03, %eax         ;add 3 to eax register
135: movl %eax, 0x0(%ebx)     ;store eax back to mem
```

This code snippet is relatively straightforward; it presumes that the address of `x` has been placed in the register `ebx`, and then loads the value at that address into the general-purpose register `eax` using the `movl` instruction (for "longword" move). The next instruction adds 3 to `eax`, and the final instruction stores the value in `eax` back into memory at that same location.

In Figure 15.1 (page 4), you can see how both the code and data are laid out in the process's address space; the three-instruction code sequence is located at address 128 (in the code section near the top), and the value of the variable `x` at address 15 KB (in the stack near the bottom). In the figure, the initial value of `x` is 3000, as shown in its location on the stack.

When these instructions run, from the perspective of the process, the following memory accesses take place.

- Fetch instruction at address 128
- Execute this instruction (load from address 15 KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

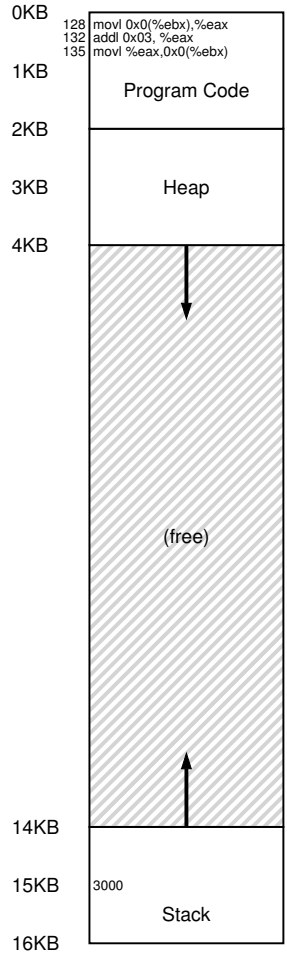


Figure 15.1: **A Process And Its Address Space**

From the program's perspective, its **address space** starts at address 0 and grows to a maximum of 16 KB; all memory references it generates should be within these bounds. However, to virtualize memory, the OS wants to place the process somewhere else in physical memory, not necessarily at address 0. Thus, we have the problem: how can we **relocate** this process in memory in a way that is **transparent** to the process? How can we provide the illusion of a virtual address space starting at 0, when in reality the address space is located at some other physical address?

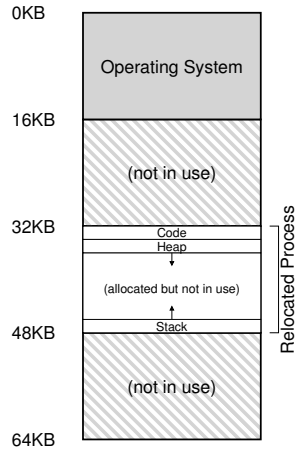


Figure 15.2: **Physical Memory with a Single Relocated Process**

An example of what physical memory might look like once this process's address space has been placed in memory is found in Figure 15.2. In the figure, you can see the OS using the first slot of physical memory for itself, and that it has relocated the process from the example above into the slot starting at physical memory address 32 KB. The other two slots are free (16 KB-32 KB and 48 KB-64 KB).

15.3 Dynamic (Hardware-based) Relocation

To gain some understanding of hardware-based address translation, we'll first discuss its first incarnation. Introduced in the first time-sharing machines of the late 1950's is a simple idea referred to as **base and bounds**; the technique is also referred to as **dynamic relocation**; we'll use both terms interchangeably [SS74].

Specifically, we'll need two hardware registers within each CPU: one is called the **base** register, and the other the **bounds** (sometimes called a **limit** register). This base-and-bounds pair is going to allow us to place the address space anywhere we'd like in physical memory, and do so while ensuring that the process can only access its own address space.

In this setup, each program is written and compiled as if it is loaded at address zero. However, when a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value. In the example above, the OS decides to load the process at physical address 32 KB and thus sets the base register to this value.

Interesting things start to happen when the process is running. Now, when any memory reference is generated by the process, it is **translated** by the processor in the following manner:

```
physical address = virtual address + base
```

ASIDE: **SOFTWARE-BASED RELOCATION**

In the early days, before hardware support arose, some systems performed a crude form of relocation purely via software methods. The basic technique is referred to as **static relocation**, in which a piece of software known as the **loader** takes an executable that is about to be run and rewrites its addresses to the desired offset in physical memory.

For example, if an instruction was a load from address 1000 into a register (e.g., `movl 1000, %eax`), and the address space of the program was loaded starting at address 3000 (and not 0, as the program thinks), the loader would rewrite the instruction to offset each address by 3000 (e.g., `movl 4000, %eax`). In this way, a simple static relocation of the process's address space is achieved.

However, static relocation has numerous problems. First and most importantly, it does not provide protection, as processes can generate bad addresses and thus illegally access other process's or even OS memory; in general, hardware support is likely needed for true protection [WL+93]. Another negative is that once placed, it is difficult to later relocate an address space to another location [M65].

Each memory reference generated by the process is a **virtual address**; the hardware in turn adds the contents of the base register to this address and the result is a **physical address** that can be issued to the memory system.

To understand this better, let's trace through what happens when a single instruction is executed. Specifically, let's look at one instruction from our earlier sequence:

```
128: movl 0x0(%ebx), %eax
```

The program counter (PC) is set to 128; when the hardware needs to fetch this instruction, it first adds the value to the base register value of 32 KB (32768) to get a physical address of 32896; the hardware then fetches the instruction from that physical address. Next, the processor begins executing the instruction. At some point, the process then issues the load from virtual address 15 KB, which the processor takes and again adds to the base register (32 KB), getting the final physical address of 47 KB and thus the desired contents.

Transforming a virtual address into a physical address is exactly the technique we refer to as **address translation**; that is, the hardware takes a virtual address the process thinks it is referencing and transforms it into a physical address which is where the data actually resides. Because this relocation of the address happens at runtime, and because we can move address spaces even after the process has started running, the technique is often referred to as **dynamic relocation** [M65].

TIP: HARDWARE-BASED DYNAMIC RELOCATION
With dynamic relocation, a little hardware goes a long way. Namely, a **base** register is used to transform virtual addresses (generated by the program) into physical addresses. A **bounds** (or **limit**) register ensures that such addresses are within the confines of the address space. Together they provide a simple and efficient virtualization of memory.

Now you might be asking: what happened to that bounds (limit) register? After all, isn't this the base *and* bounds approach? Indeed, it is. As you might have guessed, the bounds register is there to help with protection. Specifically, the processor will first check that the memory reference is *within bounds* to make sure it is legal; in the simple example above, the bounds register would always be set to 16 KB. If a process generates a virtual address that is greater than the bounds, or one that is negative, the CPU will raise an exception, and the process will likely be terminated. The point of the bounds is thus to make sure that all addresses generated by the process are legal and within the "bounds" of the process.

We should note that the base and bounds registers are hardware structures kept on the chip (one pair per CPU). Sometimes people call the part of the processor that helps with address translation the **memory management unit (MMU)**; as we develop more sophisticated memory-management techniques, we will be adding more circuitry to the MMU.

A small aside about bound registers, which can be defined in one of two ways. In one way (as above), it holds the *size* of the address space, and thus the hardware checks the virtual address against it first before adding the base. In the second way, it holds the *physical address* of the end of the address space, and thus the hardware first adds the base and then makes sure the address is within bounds. Both methods are logically equivalent; for simplicity, we'll usually assume the former method.

Example Translations

To understand address translation via base-and-bounds in more detail, let's take a look at an example. Imagine a process with an address space of size 4 KB (yes, unrealistically small) has been loaded at physical address 16 KB. Here are the results of a number of address translations:

Virtual Address		Physical Address
0	→	16 KB
1 KB	→	17 KB
3000	→	19384
4400	→	*Fault (out of bounds)*

As you can see from the example, it is easy for you to simply add the base address to the virtual address (which can rightly be viewed as an *offset* into the address space) to get the resulting physical address. Only if the virtual address is "too big" or negative will the result be a fault, causing an exception to be raised.

ASIDE: **DATA STRUCTURE — THE FREE LIST**
The OS must track which parts of free memory are not in use, so as to be able to allocate memory to processes. Many different data structures can of course be used for such a task; the simplest (which we will assume here) is a **free list**, which simply is a list of the ranges of the physical memory which are not currently in use.

15.4 Hardware Support: A Summary

Let us now summarize the support we need from the hardware (also see Figure 15.3, page 9). First, as discussed in the chapter on CPU virtualization, we require two different CPU modes. The OS runs in **privileged mode** (or **kernel mode**), where it has access to the entire machine; applications run in **user mode**, where they are limited in what they can do. A single bit, perhaps stored in some kind of **processor status word**, indicates which mode the CPU is currently running in; upon certain special occasions (e.g., a system call or some other kind of exception or interrupt), the CPU switches modes.

The hardware must also provide the **base and bounds registers** themselves; each CPU thus has an additional pair of registers, part of the **memory management unit** (**MMU**) of the CPU. When a user program is running, the hardware will translate each address, by adding the base value to the virtual address generated by the user program. The hardware must also be able to check whether the address is valid, which is accomplished by using the bounds register and some circuitry within the CPU.

The hardware should provide special instructions to modify the base and bounds registers, allowing the OS to change them when different processes run. These instructions are **privileged**; only in kernel (or privileged) mode can the registers be modified. Imagine the havoc a user process could wreak[1] if it could arbitrarily change the base register while running. Imagine it! And then quickly flush such dark thoughts from your mind, as they are the ghastly stuff of which nightmares are made.

Finally, the CPU must be able to generate **exceptions** in situations where a user program tries to access memory illegally (with an address that is "out of bounds"); in this case, the CPU should stop executing the user program and arrange for the OS "out-of-bounds" **exception handler** to run. The OS handler can then figure out how to react, in this case likely terminating the process. Similarly, if a user program tries to change the values of the (privileged) base and bounds registers, the CPU should raise an exception and run the "tried to execute a privileged operation while in user mode" handler. The CPU also must provide a method to inform it of the location of these handlers; a few more privileged instructions are thus needed.

[1] Is there anything other than "havoc" that can be "wreaked"?

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Figure 15.3: **Dynamic Relocation: Hardware Requirements**

15.5 Operating System Issues

Just as the hardware provides new features to support dynamic relocation, the OS now has new issues it must handle; the combination of hardware support and OS management leads to the implementation of a simple virtual memory. Specifically, there are a few critical junctures where the OS must get involved to implement our base-and-bounds version of virtual memory.

First, the OS must take action when a process is created, finding space for its address space in memory. Fortunately, given our assumptions that each address space is (a) smaller than the size of physical memory and (b) the same size, this is quite easy for the OS; it can simply view physical memory as an array of slots, and track whether each one is free or in use. When a new process is created, the OS will have to search a data structure (often called a **free list**) to find room for the new address space and then mark it used. With variable-sized address spaces, life is more complicated, but we will leave that concern for future chapters.

Let's look at an example. In Figure 15.2 (page 5), you can see the OS using the first slot of physical memory for itself, and that it has relocated the process from the example above into the slot starting at physical memory address 32 KB. The other two slots are free (16 KB-32 KB and 48 KB-64 KB); thus, the **free list** should consist of these two entries.

Second, the OS must do some work when a process is terminated (i.e., when it exits gracefully, or is forcefully killed because it misbehaved), reclaiming all of its memory for use in other processes or the OS. Upon termination of a process, the OS thus puts its memory back on the free list, and cleans up any associated data structures as need be.

Third, the OS must also perform a few additional steps when a context switch occurs. There is only one base and bounds register pair on each CPU, after all, and their values differ for each running program, as each program is loaded at a different physical address in memory. Thus, the OS must *save and restore* the base-and-bounds pair when it switches be-

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes;* *Reclaim memory from terminated processes;* *Generally manage memory via **free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise;* *likely action is to terminate offending process*

Figure 15.4: **Dynamic Relocation: Operating System Responsibilities**

tween processes. Specifically, when the OS decides to stop running a process, it must save the values of the base and bounds registers to memory, in some per-process structure such as the **process structure** or **process control block** (PCB). Similarly, when the OS resumes a running process (or runs it the first time), it must set the values of the base and bounds on the CPU to the correct values for this process.

We should note that when a process is stopped (i.e., not running), it is possible for the OS to move an address space from one location in memory to another rather easily. To move a process's address space, the OS first deschedules the process; then, the OS copies the address space from the current location to the new location; finally, the OS updates the saved base register (in the process structure) to point to the new location. When the process is resumed, its (new) base register is restored, and it begins running again, oblivious that its instructions and data are now in a completely new spot in memory.

Fourth, the OS must provide **exception handlers**, or functions to be called, as discussed above; the OS installs these handlers at boot time (via privileged instructions). For example, if a process tries to access memory outside its bounds, the CPU will raise an exception; the OS must be prepared to take action when such an exception arises. The common reaction of the OS will be one of hostility: it will likely terminate the offending process. The OS should be highly protective of the machine it is running, and thus it does not take kindly to a process trying to access memory or execute instructions that it shouldn't. Bye bye, misbehaving process; it's been nice knowing you.

Figure 15.5 (page 11) illustrates much of the hardware/OS interaction in a timeline. The figure shows what the OS does at boot time to ready the machine for use, and then what happens when a process (Process A) starts running; note how its memory translations are handled by the hardware with no OS intervention. At some point, a timer interrupt occurs, and the OS switches to Process B, which executes a "bad load" (to an illegal memory address); at that point, the OS must get involved, terminating the process and cleaning up by freeing B's memory and removing it's entry from the process table. As you can see from the diagram, we are still following the basic approach of **limited direct execution**. In most cases, the OS just sets up the hardware appropriately and lets the process run directly on the CPU; Only when the process misbehaves does the OS have to become involved.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of...	
	system call handler	
	timer handler	
	illegal mem-access handler	
	illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A:		
allocate entry in process table		
allocate memory for process		
set base/bounds registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
	Translate virtual address	
	and perform fetch	
		Execute instruction
	If explicit load/store:	
	Ensure address is in-bounds;	
	Translate virtual address	
	and perform load/store	
		...
	Timer interrupt	
	move to **kernel mode**	
	Jump to interrupt handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
(including base/bounds)		
restore regs(B) from proc-struct(B)		
(including base/bounds)		
return-from-trap (into B)		
	restore registers of B	
	move to **user mode**	
	jump to B's PC	
		Process B runs
		Execute bad load
	Load is out-of-bounds;	
	move to **kernel mode**	
	jump to trap handler	
Handle the trap		
Decide to terminate process B		
de-allocate B's memory		
free B's entry in process table		

Figure 15.5: **Limited Direct Execution Protocol (Dynamic Relocation)**

15.6 Summary

In this chapter, we have extended the concept of limited direct execution with a specific mechanism used in virtual memory, known as **address translation**. With address translation, the OS can control each and every memory access from a process, ensuring the accesses stay within the bounds of the address space. Key to the efficiency of this technique is hardware support, which performs the translation quickly for each access, turning virtual addresses (the process's view of memory) into physical ones (the actual view). All of this is performed in a way that is *transparent* to the process that has been relocated; the process has no idea its memory references are being translated, making for a wonderful illusion.

We have also seen one particular form of virtualization, known as base and bounds or dynamic relocation. Base-and-bounds virtualization is quite *efficient*, as only a little more hardware logic is required to add a base register to the virtual address and check that the address generated by the process is in bounds. Base-and-bounds also offers *protection*; the OS and hardware combine to ensure no process can generate memory references outside its own address space. Protection is certainly one of the most important goals of the OS; without it, the OS could not control the machine (if processes were free to overwrite memory, they could easily do nasty things like overwrite the trap table and take over the system).

Unfortunately, this simple technique of dynamic relocation does have its inefficiencies. For example, as you can see in Figure 15.2 (page 5), the relocated process is using physical memory from 32 KB to 48 KB; however, because the process stack and heap are not too big, all of the space between the two is simply *wasted*. This type of waste is usually called **internal fragmentation**, as the space *inside* the allocated unit is not all used (i.e., is fragmented) and thus wasted. In our current approach, although there might be enough physical memory for more processes, we are currently restricted to placing an address space in a fixed-sized slot and thus internal fragmentation can arise[2]. Thus, we are going to need more sophisticated machinery, to try to better utilize physical memory and avoid internal fragmentation. Our first attempt will be a slight generalization of base and bounds known as **segmentation**, which we will discuss next.

[2] A different solution might instead place a fixed-sized stack within the address space, just below the code region, and a growing heap below that. However, this limits flexibility by making recursion and deeply-nested function calls challenging, and thus is something we hope to avoid.

References

[M65] "On Dynamic Program Relocation"
W.C. McGee
IBM Systems Journal
Volume 4, Number 3, 1965, pages 184–199
This paper is a nice summary of early work on dynamic relocation, as well as some basics on static relocation.

[P90] "Relocating loader for MS-DOS .EXE executable files"
Kenneth D. A. Pillay
Microprocessors & Microsystems archive
Volume 14, Issue 7 (September 1990)
An example of a relocating loader for MS-DOS. Not the first one, but just a relatively modern example of how such a system works.

[SS74] "The Protection of Information in Computer Systems"
J. Saltzer and M. Schroeder
CACM, July 1974
From this paper: "The concepts of base-and-bound register and hardware-interpreted descriptors appeared, apparently independently, between 1957 and 1959 on three projects with diverse goals. At M.I.T., McCarthy suggested the base-and-bound idea as part of the memory protection system necessary to make time-sharing feasible. IBM independently developed the base-and-bound register as a mechanism to permit reliable multiprogramming of the Stretch (7030) computer system. At Burroughs, R. Barton suggested that hardware-interpreted descriptors would provide direct support for the naming scope rules of higher level languages in the B5000 computer system." We found this quote on Mark Smotherman's cool history pages [S04]; see them for more information.

[S04] "System Call Support"
Mark Smotherman, May 2004
http://people.cs.clemson.edu/~mark/syscall.html
A neat history of system call support. Smotherman has also collected some early history on items like interrupts and other fun aspects of computing history. See his web pages for more details.

[WL+93] "Efficient Software-based Fault Isolation"
Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham
SOSP '93
A terrific paper about how you can use compiler support to bound memory references from a program, without hardware support. The paper sparked renewed interest in software techniques for isolation of memory references.

Homework

The program `relocation.py` allows you to see how address translations are performed in a system with base and bounds registers. See the README for details.

Questions

- Run with seeds 1, 2, and 3, and compute whether each virtual address generated by the process is in or out of bounds. If in bounds, compute the translation.
- Run with these flags: `-s 0 -n 10`. What value do you have set `-l` (the bounds register) to in order to ensure that all the generated virtual addresses are within bounds?
- Run with these flags: `-s 1 -n 10 -l 100`. What is the maximum value that bounds can be set to, such that the address space still fits into physical memory in its entirety?
- Run some of the same problems above, but with larger address spaces (`-a`) and physical memories (`-p`).
- What fraction of randomly-generated virtual addresses are valid, as a function of the value of the bounds register? Make a graph from running with different random seeds, with limit values ranging from 0 up to the maximum size of the address space.

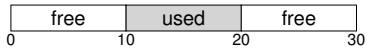
17

Free-Space Management

In this chapter, we take a small detour from our discussion of virtualizing memory to discuss a fundamental aspect of any memory management system, whether it be a malloc library (managing pages of a process's heap) or the OS itself (managing portions of the address space of a process). Specifically, we will discuss the issues surrounding **free-space management**.

Let us make the problem more specific. Managing free space can certainly be easy, as we will see when we discuss the concept of **paging**. It is easy when the space you are managing is divided into fixed-sized units; in such a case, you just keep a list of these fixed-sized units; when a client requests one of them, return the first entry.

Where free-space management becomes more difficult (and interesting) is when the free space you are managing consists of variable-sized units; this arises in a user-level memory-allocation library (as in `malloc()` and `free()`) and in an OS managing physical memory when using **segmentation** to implement virtual memory. In either case, the problem that exists is known as **external fragmentation**: the free space gets chopped into little pieces of different sizes and is thus fragmented; subsequent requests may fail because there is no single contiguous space that can satisfy the request, even though the total amount of free space exceeds the size of the request.



The figure shows an example of this problem. In this case, the total free space available is 20 bytes; unfortunately, it is fragmented into two chunks of size 10 each. As a result, a request for 15 bytes will fail even though there are 20 bytes free. And thus we arrive at the problem addressed in this chapter.

CRUX: HOW TO MANAGE FREE SPACE
How should free space be managed, when satisfying variable-sized requests? What strategies can be used to minimize fragmentation? What are the time and space overheads of alternate approaches?

17.1 Assumptions

Most of this discussion will focus on the great history of allocators found in user-level memory-allocation libraries. We draw on Wilson's excellent survey [W+95] but encourage interested readers to go to the source document itself for more details[1].

We assume a basic interface such as that provided by `malloc()` and `free()`. Specifically, `void *malloc(size_t size)` takes a single parameter, `size`, which is the number of bytes requested by the application; it hands back a pointer (of no particular type, or a **void pointer** in C lingo) to a region of that size (or greater). The complementary routine `void free(void *ptr)` takes a pointer and frees the corresponding chunk. Note the implication of the interface: the user, when freeing the space, does not inform the library of its size; thus, the library must be able to figure out how big a chunk of memory is when handed just a pointer to it. We'll discuss how to do this a bit later on in the chapter.

The space that this library manages is known historically as the `heap`, and the generic data structure used to manage free space in the heap is some kind of **free list**. This structure contains references to all of the free chunks of space in the managed region of memory. Of course, this data structure need not be a list *per se*, but just some kind of data structure to track free space.

We further assume that primarily we are concerned with **external fragmentation**, as described above. Allocators could of course also have the problem of **internal fragmentation**; if an allocator hands out chunks of memory bigger than that requested, any unasked for (and thus unused) space in such a chunk is considered *internal* fragmentation (because the waste occurs inside the allocated unit) and is another example of space waste. However, for the sake of simplicity, and because it is the more interesting of the two types of fragmentation, we'll mostly focus on external fragmentation.

We'll also assume that once memory is handed out to a client, it cannot be relocated to another location in memory. For example, if a program calls `malloc()` and is given a pointer to some space within the heap, that memory region is essentially "owned" by the program (and cannot be moved by the library) until the program returns it via a corresponding call to `free()`. Thus, no **compaction** of free space is possible, which

[1] It is nearly 80 pages long; thus, you really have to be interested!

would be useful to combat fragmentation[2]. Compaction could, however, be used in the OS to deal with fragmentation when implementing **segmentation** (as discussed in said chapter on segmentation).

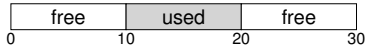
Finally, we'll assume that the allocator manages a contiguous region of bytes. In some cases, an allocator could ask for that region to grow; for example, a user-level memory-allocation library might call into the kernel to grow the heap (via a system call such as `sbrk`) when it runs out of space. However, for simplicity, we'll just assume that the region is a single fixed size throughout its life.

17.2 Low-level Mechanisms

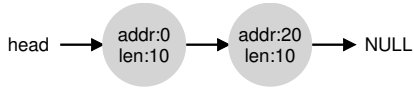
Before delving into some policy details, we'll first cover some common mechanisms used in most allocators. First, we'll discuss the basics of splitting and coalescing, common techniques in most any allocator. Second, we'll show how one can track the size of allocated regions quickly and with relative ease. Finally, we'll discuss how to build a simple list inside the free space to keep track of what is free and what isn't.

Splitting and Coalescing

A free list contains a set of elements that describe the free space still remaining in the heap. Thus, assume the following 30-byte heap:



The free list for this heap would have two elements on it. One entry describes the first 10-byte free segment (bytes 0-9), and one entry describes the other free segment (bytes 20-29):



As described above, a request for anything greater than 10 bytes will fail (returning NULL); there just isn't a single contiguous chunk of memory of that size available. A request for exactly that size (10 bytes) could be satisfied easily by either of the free chunks. But what happens if the request is for something *smaller* than 10 bytes?

Assume we have a request for just a single byte of memory. In this case, the allocator will perform an action known as **splitting**: it will find

[2]Once you hand a pointer to a chunk of memory to a C program, it is generally difficult to determine all references (pointers) to that region, which may be stored in other variables or even in registers at a given point in execution. This may not be the case in more strongly-typed, garbage-collected languages, which would thus enable compaction as a technique to combat fragmentation.

a free chunk of memory that can satisfy the request and split it into two. The first chunk it will return to the caller; the second chunk will remain on the list. Thus, in our example above, if a request for 1 byte were made, and the allocator decided to use the second of the two elements on the list to satisfy the request, the call to malloc() would return 20 (the address of the 1-byte allocated region) and the list would end up looking like this:

In the picture, you can see the list basically stays intact; the only change is that the free region now starts at 21 instead of 20, and the length of that free region is now just 9[3]. Thus, the split is commonly used in allocators when requests are smaller than the size of any particular free chunk.

A corollary mechanism found in many allocators is known as **coalescing** of free space. Take our example from above once more (free 10 bytes, used 10 bytes, and another free 10 bytes).

Given this (tiny) heap, what happens when an application calls free(10), thus returning the space in the middle of the heap? If we simply add this free space back into our list without too much thinking, we might end up with a list that looks like this:

Note the problem: while the entire heap is now free, it is seemingly divided into three chunks of 10 bytes each. Thus, if a user requests 20 bytes, a simple list traversal will not find such a free chunk, and return failure.

What allocators do in order to avoid this problem is coalesce free space when a chunk of memory is freed. The idea is simple: when returning a free chunk in memory, look carefully at the addresses of the chunk you are returning as well as the nearby chunks of free space; if the newly-freed space sits right next to one (or two, as in this example) existing free chunks, merge them into a single larger free chunk. Thus, with coalescing, our final list should look like this:

Indeed, this is what the heap list looked like at first, before any allocations were made. With coalescing, an allocator can better ensure that large free extents are available for the application.

[3] This discussion assumes that there are no headers, an unrealistic but simplifying assumption we make for now.

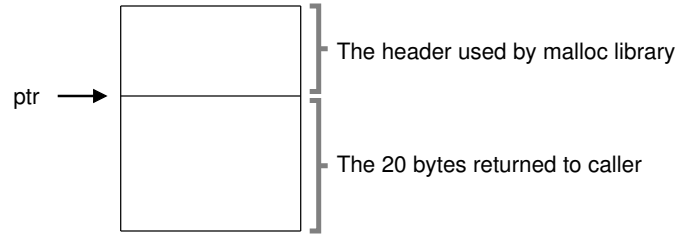


Figure 17.1: **An Allocated Region Plus Header**

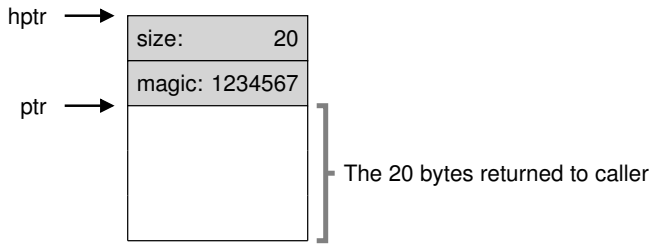


Figure 17.2: **Specific Contents Of The Header**

Tracking The Size Of Allocated Regions

You might have noticed that the interface to `free(void *ptr)` does not take a size parameter; thus it is assumed that given a pointer, the malloc library can quickly determine the size of the region of memory being freed and thus incorporate the space back into the free list.

To accomplish this task, most allocators store a little bit of extra information in a **header** block which is kept in memory, usually just before the handed-out chunk of memory. Let's look at an example again (Figure 17.1). In this example, we are examining an allocated block of size 20 bytes, pointed to by `ptr`; imagine the user called `malloc()` and stored the results in `ptr`, e.g., `ptr = malloc(20);`.

The header minimally contains the size of the allocated region (in this case, 20); it may also contain additional pointers to speed up deallocation, a magic number to provide additional integrity checking, and other information. Let's assume a simple header which contains the size of the region and a magic number, like this:

```
typedef struct __header_t {
    int size;
    int magic;
} header_t;
```

The example above would look like what you see in Figure 17.2. When

the user calls `free(ptr)`, the library then uses simple pointer arithmetic to figure out where the header begins:

```
void free(void *ptr) {
    header_t *hptr = (void *)ptr - sizeof(header_t);
}
```

After obtaining such a pointer to the header, the library can easily determine whether the magic number matches the expected value as a sanity check (`assert(hptr->magic == 1234567)`) and calculate the total size of the newly-freed region via simple math (i.e., adding the size of the header to size of the region). Note the small but critical detail in the last sentence: the size of the free region is the size of the header plus the size of the space allocated to the user. Thus, when a user requests N bytes of memory, the library does not search for a free chunk of size N; rather, it searches for a free chunk of size N plus the size of the header.

Embedding A Free List

Thus far we have treated our simple free list as a conceptual entity; it is just a list describing the free chunks of memory in the heap. But how do we build such a list inside the free space itself?

In a more typical list, when allocating a new node, you would just call `malloc()` when you need space for the node. Unfortunately, within the memory-allocation library, you can't do this! Instead, you need to build the list *inside* the free space itself. Don't worry if this sounds a little weird; it is, but not so weird that you can't do it!

Assume we have a 4096-byte chunk of memory to manage (i.e., the heap is 4KB). To manage this as a free list, we first have to initialize said list; initially, the list should have one entry, of size 4096 (minus the header size). Here is the description of a node of the list:

```
typedef struct __node_t {
    int              size;
    struct __node_t *next;
} node_t;
```

Now let's look at some code that initializes the heap and puts the first element of the free list inside that space. We are assuming that the heap is built within some free space acquired via a call to the system call `mmap()`; this is not the only way to build such a heap but serves us well in this example. Here is the code:

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size   = 4096 - sizeof(node_t);
head->next   = NULL;
```

After running this code, the status of the list is that it has a single entry, of size 4088. Yes, this is a tiny heap, but it serves as a fine example for us

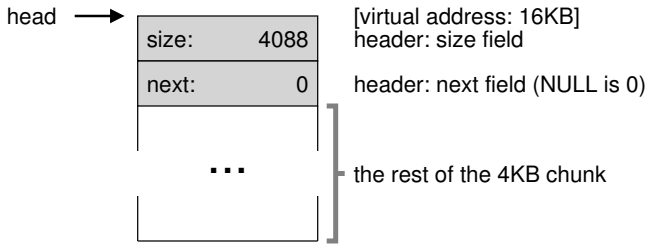


Figure 17.3: **A Heap With One Free Chunk**

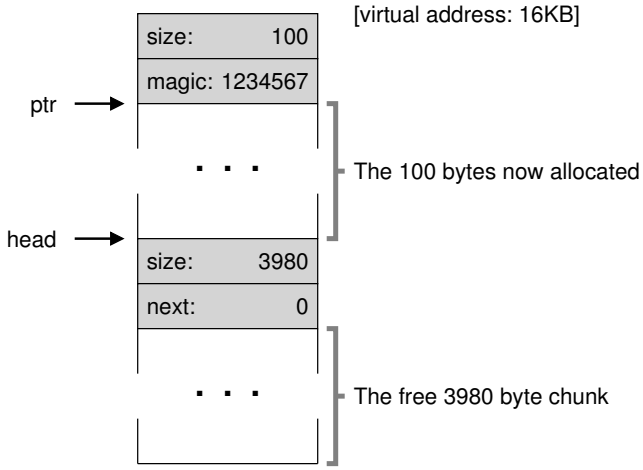


Figure 17.4: **A Heap: After One Allocation**

here. The `head` pointer contains the beginning address of this range; let's assume it is 16KB (though any virtual address would be fine). Visually, the heap thus looks like what you see in Figure 17.3.

Now, let's imagine that a chunk of memory is requested, say of size 100 bytes. To service this request, the library will first find a chunk that is large enough to accommodate the request; because there is only one free chunk (size: 4088), this chunk will be chosen. Then, the chunk will be **split** into two: one chunk big enough to service the request (and header, as described above), and the remaining free chunk. Assuming an 8-byte header (an integer size and an integer magic number), the space in the heap now looks like what you see in Figure 17.4.

Thus, upon the request for 100 bytes, the library allocated 108 bytes out of the existing one free chunk, returns a pointer (marked `ptr` in the figure above) to it, stashes the header information immediately before the

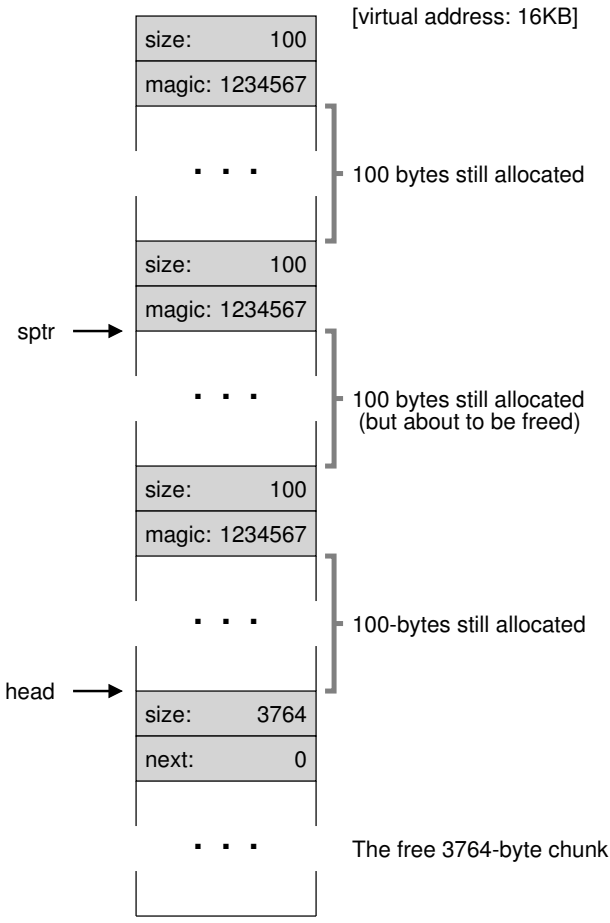


Figure 17.5: **Free Space With Three Chunks Allocated**

allocated space for later use upon `free()`, and shrinks the one free node in the list to 3980 bytes (4088 minus 108).

Now let's look at the heap when there are three allocated regions, each of 100 bytes (or 108 including the header). A visualization of this heap is shown in Figure 17.5.

As you can see therein, the first 324 bytes of the heap are now allocated, and thus we see three headers in that space as well as three 100-byte regions being used by the calling program. The free list remains uninteresting: just a single node (pointed to by `head`), but now only 3764 bytes in size after the three splits. But what happens when the calling program returns some memory via `free()`?

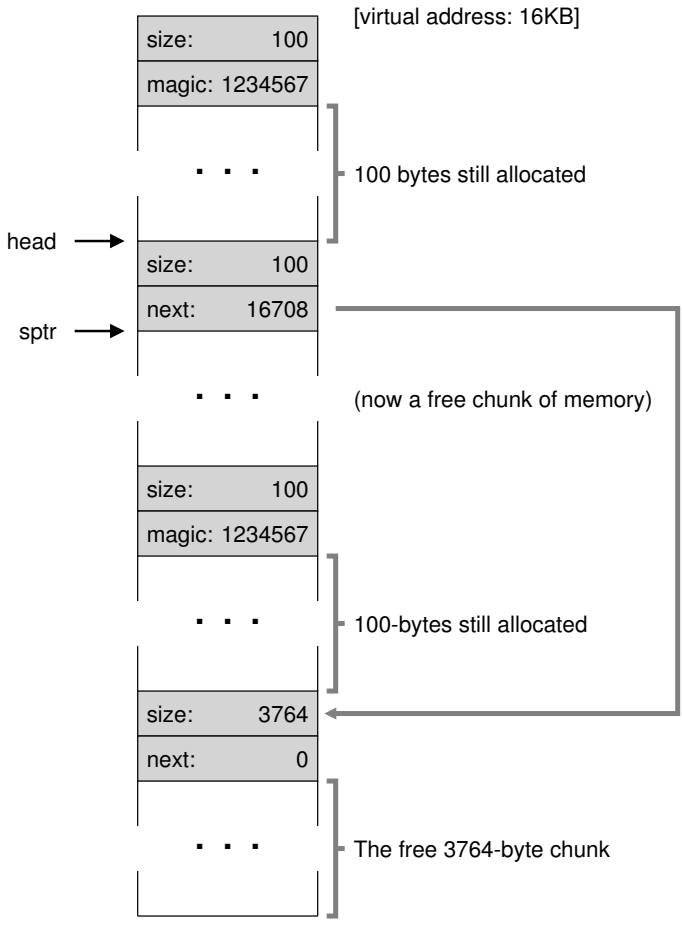


Figure 17.6: **Free Space With Two Chunks Allocated**

In this example, the application returns the middle chunk of allocated memory, by calling `free(16500)` (the value 16500 is arrived upon by adding the start of the memory region, 16384, to the 108 of the previous chunk and the 8 bytes of the header for this chunk). This value is shown in the previous diagram by the pointer `sptr`.

The library immediately figures out the size of the free region, and then adds the free chunk back onto the free list. Assuming we insert at the head of the free list, the space now looks like this (Figure 17.6).

And now we have a list that starts with a small free chunk (100 bytes, pointed to by the head of the list) and a large free chunk (3764 bytes).

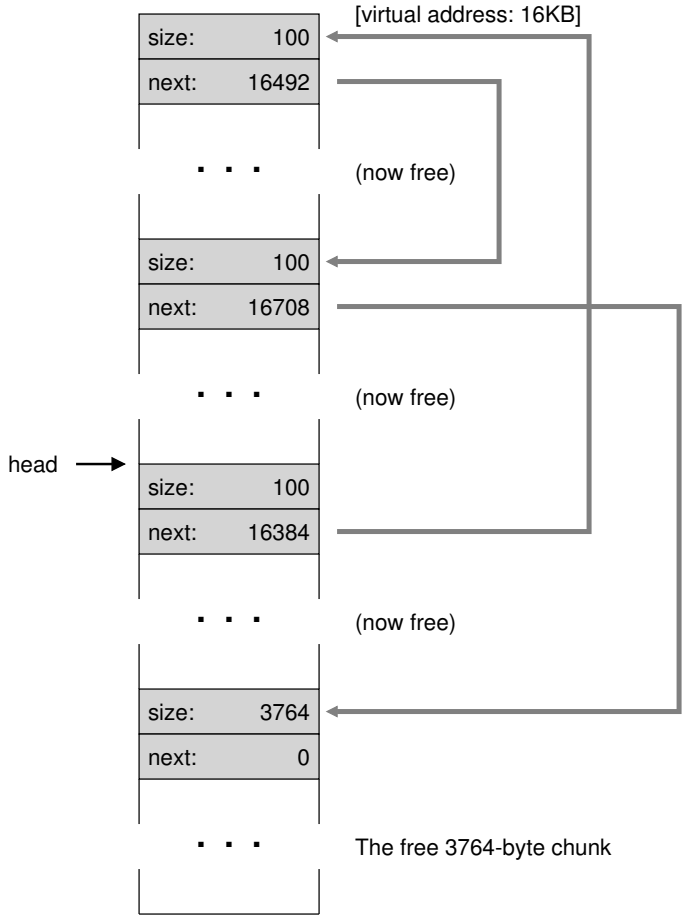


Figure 17.7: **A Non-Coalesced Free List**

Our list finally has more than one element on it! And yes, the free space is fragmented, an unfortunate but common occurrence.

One last example: let's assume now that the last two in-use chunks are freed. Without coalescing, you might end up with a free list that is highly fragmented (see Figure 17.7).

As you can see from the figure, we now have a big mess! Why? Simple, we forgot to **coalesce** the list. Although all of the memory is free, it is chopped up into pieces, thus appearing as a fragmented memory despite not being one. The solution is simple: go through the list and **merge** neighboring chunks; when finished, the heap will be whole again.

Growing The Heap

We should discuss one last mechanism found within many allocation libraries. Specifically, what should you do if the heap runs out of space? The simplest approach is just to fail. In some cases this is the only option, and thus returning NULL is an honorable approach. Don't feel bad! You tried, and though you failed, you fought the good fight.

Most traditional allocators start with a small-sized heap and then request more memory from the OS when they run out. Typically, this means they make some kind of system call (e.g., `sbrk` in most UNIX systems) to grow the heap, and then allocate the new chunks from there. To service the `sbrk` request, the OS finds free physical pages, maps them into the address space of the requesting process, and then returns the value of the end of the new heap; at that point, a larger heap is available, and the request can be successfully serviced.

17.3 Basic Strategies

Now that we have some machinery under our belt, let's go over some basic strategies for managing free space. These approaches are mostly based on pretty simple policies that you could think up yourself; try it before reading and see if you come up with all of the alternatives (or maybe some new ones!).

The ideal allocator is both fast and minimizes fragmentation. Unfortunately, because the stream of allocation and free requests can be arbitrary (after all, they are determined by the programmer), any particular strategy can do quite badly given the wrong set of inputs. Thus, we will not describe a "best" approach, but rather talk about some basics and discuss their pros and cons.

Best Fit

The **best fit** strategy is quite simple: first, search through the free list and find chunks of free memory that are as big or bigger than the requested size. Then, return the one that is the smallest in that group of candidates; this is the so called best-fit chunk (it could be called smallest fit too). One pass through the free list is enough to find the correct block to return.

The intuition behind best fit is simple: by returning a block that is close to what the user asks, best fit tries to reduce wasted space. However, there is a cost; naive implementations pay a heavy performance penalty when performing an exhaustive search for the correct free block.

Worst Fit

The **worst fit** approach is the opposite of best fit; find the largest chunk and return the requested amount; keep the remaining (large) chunk on the free list. Worst fit tries to thus leave big chunks free instead of lots of

small chunks that can arise from a best-fit approach. Once again, however, a full search of free space is required, and thus this approach can be costly. Worse, most studies show that it performs badly, leading to excess fragmentation while still having high overheads.

First Fit

The **first fit** method simply finds the first block that is big enough and returns the requested amount to the user. As before, the remaining free space is kept free for subsequent requests.

First fit has the advantage of speed — no exhaustive search of all the free spaces are necessary — but sometimes pollutes the beginning of the free list with small objects. Thus, how the allocator manages the free list's order becomes an issue. One approach is to use **address-based ordering**; by keeping the list ordered by the address of the free space, coalescing becomes easier, and fragmentation tends to be reduced.

Next Fit

Instead of always beginning the first-fit search at the beginning of the list, the **next fit** algorithm keeps an extra pointer to the location within the list where one was looking last. The idea is to spread the searches for free space throughout the list more uniformly, thus avoiding splintering of the beginning of the list. The performance of such an approach is quite similar to first fit, as an exhaustive search is once again avoided.

Examples

Here are a few examples of the above strategies. Envision a free list with three elements on it, of sizes 10, 30, and 20 (we'll ignore headers and other details here, instead just focusing on how strategies operate):

Assume an allocation request of size 15. A best-fit approach would search the entire list and find that 20 was the best fit, as it is the smallest free space that can accommodate the request. The resulting free list:

As happens in this example, and often happens with a best-fit approach, a small free chunk is now left over. A worst-fit approach is similar but instead finds the largest chunk, in this example 30. The resulting list:

The first-fit strategy, in this example, does the same thing as worst-fit, also finding the first free block that can satisfy the request. The difference is in the search cost; both best-fit and worst-fit look through the entire list; first-fit only examines free chunks until it finds one that fits, thus reducing search cost.

These examples just scratch the surface of allocation policies. More detailed analysis with real workloads and more complex allocator behaviors (e.g., coalescing) are required for a deeper understanding. Perhaps something for a homework section, you say?

17.4 Other Approaches

Beyond the basic approaches described above, there have been a host of suggested techniques and algorithms to improve memory allocation in some way. We list a few of them here for your consideration (i.e., to make you think about a little more than just best-fit allocation).

Segregated Lists

One interesting approach that has been around for some time is the use of **segregated lists**. The basic idea is simple: if a particular application has one (or a few) popular-sized request that it makes, keep a separate list just to manage objects of that size; all other requests are forwarded to a more general memory allocator.

The benefits of such an approach are obvious. By having a chunk of memory dedicated for one particular size of requests, fragmentation is much less of a concern; moreover, allocation and free requests can be served quite quickly when they are of the right size, as no complicated search of a list is required.

Just like any good idea, this approach introduces new complications into a system as well. For example, how much memory should one dedicate to the pool of memory that serves specialized requests of a given size, as opposed to the general pool? One particular allocator, the **slab allocator** by uber-engineer Jeff Bonwick (which was designed for use in the Solaris kernel), handles this issue in a rather nice way [B94].

Specifically, when the kernel boots up, it allocates a number of **object caches** for kernel objects that are likely to be requested frequently (such as locks, file-system inodes, etc.); the object caches thus are each segregated free lists of a given size and serve memory allocation and free requests quickly. When a given cache is running low on free space, it requests some **slabs** of memory from a more general memory allocator (the total amount requested being a multiple of the page size and the object in question). Conversely, when the reference counts of the objects within a given slab all go to zero, the general allocator can reclaim them from the specialized allocator, which is often done when the VM system needs more memory.

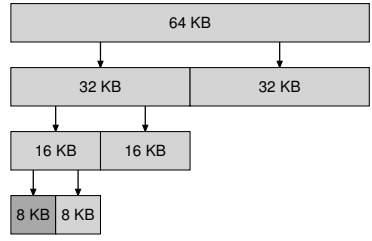
ASIDE: **GREAT ENGINEERS ARE REALLY GREAT**
Engineers like Jeff Bonwick (who not only wrote the slab allocator mentioned herein but also was the lead of an amazing file system, ZFS) are the heart of Silicon Valley. Behind almost any great product or technology is a human (or small group of humans) who are way above average in their talents, abilities, and dedication. As Mark Zuckerberg (of Facebook) says: "Someone who is exceptional in their role is not just a little better than someone who is pretty good. They are 100 times better." This is why, still today, one or two people can start a company that changes the face of the world forever (think Google, Apple, or Facebook). Work hard and you might become such a "100x" person as well. Failing that, work *with* such a person; you'll learn more in day than most learn in a month. Failing that, feel sad.

The slab allocator also goes beyond most segregated list approaches by keeping free objects on the lists in a pre-initialized state. Bonwick shows that initialization and destruction of data structures is costly [B94]; by keeping freed objects in a particular list in their initialized state, the slab allocator thus avoids frequent initialization and destruction cycles per object and thus lowers overheads noticeably.

Buddy Allocation

Because coalescing is critical for an allocator, some approaches have been designed around making coalescing simple. One good example is found in the **binary buddy allocator** [K65].

In such a system, free memory is first conceptually thought of as one big space of size 2^N. When a request for memory is made, the search for free space recursively divides free space by two until a block that is big enough to accommodate the request is found (and a further split into two would result in a space that is too small). At this point, the requested block is returned to the user. Here is an example of a 64KB free space getting divided in the search for a 7KB block:



In the example, the leftmost 8KB block is allocated (as indicated by the darker shade of gray) and returned to the user; note that this scheme can suffer from **internal fragmentation**, as you are only allowed to give out power-of-two-sized blocks.

The beauty of buddy allocation is found in what happens when that block is freed. When returning the 8KB block to the free list, the allocator checks whether the "buddy" 8KB is free; if so, it coalesces the two blocks into a 16KB block. The allocator then checks if the buddy of the 16KB block is still free; if so, it coalesces those two blocks. This recursive coalescing process continues up the tree, either restoring the entire free space or stopping when a buddy is found to be in use.

The reason buddy allocation works so well is that it is simple to determine the buddy of a particular block. How, you ask? Think about the addresses of the blocks in the free space above. If you think carefully enough, you'll see that the address of each buddy pair only differs by a single bit; which bit is determined by the level in the buddy tree. And thus you have a basic idea of how binary buddy allocation schemes work. For more detail, as always, see the Wilson survey [W+95].

Other Ideas

One major problem with many of the approaches described above is their lack of **scaling**. Specifically, searching lists can be quite slow. Thus, advanced allocators use more complex data structures to address these costs, trading simplicity for performance. Examples include balanced binary trees, splay trees, or partially-ordered trees [W+95].

Given that modern systems often have multiple processors and run multi-threaded workloads (something you'll learn about in great detail in the section of the book on Concurrency), it is not surprising that a lot of effort has been spent making allocators work well on multiprocessor-based systems. Two wonderful examples are found in Berger et al. [B+00] and Evans [E06]; check them out for the details.

These are but two of the thousands of ideas people have had over time about memory allocators; read on your own if you are curious. Failing that, read about how the glibc allocator works [S15], to give you a sense of what the real world is like.

17.5 Summary

In this chapter, we've discussed the most rudimentary forms of memory allocators. Such allocators exist everywhere, linked into every C program you write, as well as in the underlying OS which is managing memory for its own data structures. As with many systems, there are many trade-offs to be made in building such a system, and the more you know about the exact workload presented to an allocator, the more you could do to tune it to work better for that workload. Making a fast, space-efficient, scalable allocator that works well for a broad range of workloads remains an on-going challenge in modern computer systems.

References

[B+00] "Hoard: A Scalable Memory Allocator for Multithreaded Applications"
Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson
ASPLOS-IX, November 2000
Berger and company's excellent allocator for multiprocessor systems. Beyond just being a fun paper, also used in practice!

[B94] "The Slab Allocator: An Object-Caching Kernel Memory Allocator"
Jeff Bonwick
USENIX '94
A cool paper about how to build an allocator for an operating system kernel, and a great example of how to specialize for particular common object sizes.

[E06] "A Scalable Concurrent malloc(3) Implementation for FreeBSD"
Jason Evans
http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf
April 2006
A detailed look at how to build a real modern allocator for use in multiprocessors. The "jemalloc" allocator is in widespread use today, within FreeBSD, NetBSD, Mozilla Firefox, and within Facebook.

[K65] "A Fast Storage Allocator"
Kenneth C. Knowlton
Communications of the ACM, Volume 8, Number 10, October 1965
The common reference for buddy allocation. Random strange fact: Knuth gives credit for the idea not to Knowlton but to Harry Markowitz, a Nobel-prize winning economist. Another strange fact: Knuth communicates all of his emails via a secretary; he doesn't send email himself, rather he tells his secretary what email to send and then the secretary does the work of emailing. Last Knuth fact: he created TeX, the tool used to typeset this book. It is an amazing piece of software[4].

[S15] "Understanding glibc malloc"
Sploitfun
February, 2015
https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
A deep dive into how glibc malloc works. Amazingly detailed and a very cool read.

[W+95] "Dynamic Storage Allocation: A Survey and Critical Review"
Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles
International Workshop on Memory Management
Kinross, Scotland, September 1995
An excellent and far-reaching survey of many facets of memory allocation. Far too much detail to go into in this tiny chapter!

[4] Actually we use LaTeX, which is based on Lamport's additions to TeX, but close enough.

Homework

The program, `malloc.py`, lets you explore the behavior of a simple free-space allocator as described in the chapter. See the README for details of its basic operation.

Questions

1. First run with the flags `-n 10 -H 0 -p BEST -s 0` to generate a few random allocations and frees. Can you predict what alloc()/free() will return? Can you guess the state of the free list after each request? What do you notice about the free list over time?
2. How are the results different when using a WORST fit policy to search the free list (`-p WORST`)? What changes?
3. What about when using FIRST fit (`-p FIRST`)? What speeds up when you use first fit?
4. For the above questions, how the list is kept ordered can affect the time it takes to find a free location for some of the policies. Use the different free list orderings (`-l ADDRSORT`, `-l SIZESORT+`, `-l SIZESORT-`) to see how the policies and the list orderings interact.
5. Coalescing of a free list can be quite important. Increase the number of random allocations (say to `-n 1000`). What happens to larger allocation requests over time? Run with and without coalescing (i.e., without and with the `-C` flag). What differences in outcome do you see? How big is the free list over time in each case? Does the ordering of the list matter in this case?
6. What happens when you change the percent allocated fraction `-P` to higher than 50? What happens to allocations as it nears 100? What about as it nears 0?
7. What kind of specific requests can you make to generate a highly-fragmented free space? Use the `-A` flag to create fragmented free lists, and see how different policies and options change the organization of the free list.

14

Interlude: Memory API

In this interlude, we discuss the memory allocation interfaces in UNIX systems. The interfaces provided are quite simple, and hence the chapter is short and to the point[1]. The main problem we address is this:

CRUX: HOW TO ALLOCATE AND MANAGE MEMORY
In UNIX/C programs, understanding how to allocate and manage memory is critical in building robust and reliable software. What interfaces are commonly used? What mistakes should be avoided?

14.1 Types of Memory

In running a C program, there are two types of memory that are allocated. The first is called **stack** memory, and allocations and deallocations of it are managed *implicitly* by the compiler for you, the programmer; for this reason it is sometimes called **automatic** memory.

Declaring memory on the stack in C is easy. For example, let's say you need some space in a function `func()` for an integer, called `x`. To declare such a piece of memory, you just do something like this:

```
void func() {
    int x; // declares an integer on the stack
    ...
}
```

The compiler does the rest, making sure to make space on the stack when you call into `func()`. When your return from the function, the compiler deallocates the memory for you; thus, if you want some information to live beyond the call invocation, you had better not leave that information on the stack.

It is this need for long-lived memory that gets us to the second type of memory, called **heap** memory, where all allocations and deallocations

[1] Indeed, we hope all chapters are! But this one is shorter and pointier, we think.

are *explicitly* handled by you, the programmer. A heavy responsibility, no doubt! And certainly the cause of many bugs. But if you are careful and pay attention, you will use such interfaces correctly and without too much trouble. Here is an example of how one might allocate a pointer to an integer on the heap:

```
void func() {
    int *x = (int *) malloc(sizeof(int));
    ...
}
```

A couple of notes about this small code snippet. First, you might notice that both stack and heap allocation occur on this line: first the compiler knows to make room for a pointer to an integer when it sees your declaration of said pointer (`int *x`); subsequently, when the program calls `malloc()`, it requests space for an integer on the heap; the routine returns the address of such an integer (upon success, or `NULL` on failure), which is then stored on the stack for use by the program.

Because of its explicit nature, and because of its more varied usage, heap memory presents more challenges to both users and systems. Thus, it is the focus of the remainder of our discussion.

14.2 The `malloc()` Call

The `malloc()` call is quite simple: you pass it a size asking for some room on the heap, and it either succeeds and gives you back a pointer to the newly-allocated space, or fails and returns `NULL`[2].

The manual page shows what you need to do to use malloc; type `man malloc` at the command line and you will see:

```
#include <stdlib.h>
...
void *malloc(size_t size);
```

From this information, you can see that all you need to do is include the header file `stdlib.h` to use malloc. In fact, you don't really need to even do this, as the C library, which all C programs link with by default, has the code for `malloc()` inside of it; adding the header just lets the compiler check whether you are calling `malloc()` correctly (e.g., passing the right number of arguments to it, of the right type).

The single parameter `malloc()` takes is of type `size_t` which simply describes how many bytes you need. However, most programmers do not type in a number here directly (such as 10); indeed, it would be considered poor form to do so. Instead, various routines and macros are utilized. For example, to allocate space for a double-precision floating point value, you simply do this:

```
double *d = (double *) malloc(sizeof(double));
```

[2] Note that `NULL` in C isn't really anything special at all, just a macro for the value zero.

TIP: WHEN IN DOUBT, TRY IT OUT

If you aren't sure how some routine or operator you are using behaves, there is no substitute for simply trying it out and making sure it behaves as you expect. While reading the manual pages or other documentation is useful, how it works in practice is what matters. Write some code and test it! That is no doubt the best way to make sure your code behaves as you desire. Indeed, that is what we did to double-check the things we were saying about `sizeof()` were actually true!

Wow, that's lot of `double`-ing! This invocation of `malloc()` uses the `sizeof()` operator to request the right amount of space; in C, this is generally thought of as a *compile-time* operator, meaning that the actual size is known at *compile time* and thus a number (in this case, 8, for a double) is substituted as the argument to `malloc()`. For this reason, `sizeof()` is correctly thought of as an operator and not a function call (a function call would take place at run time).

You can also pass in the name of a variable (and not just a type) to `sizeof()`, but in some cases you may not get the desired results, so be careful. For example, let's look at the following code snippet:

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

In the first line, we've declared space for an array of 10 integers, which is fine and dandy. However, when we use `sizeof()` in the next line, it returns a small value, such as 4 (on 32-bit machines) or 8 (on 64-bit machines). The reason is that in this case, `sizeof()` thinks we are simply asking how big a *pointer* to an integer is, not how much memory we have dynamically allocated. However, sometimes `sizeof()` does work as you might expect:

```
int x[10];
printf("%d\n", sizeof(x));
```

In this case, there is enough static information for the compiler to know that 40 bytes have been allocated.

Another place to be careful is with strings. When declaring space for a string, use the following idiom: `malloc(strlen(s) + 1)`, which gets the length of the string using the function `strlen()`, and adds 1 to it in order to make room for the end-of-string character. Using `sizeof()` may lead to trouble here.

You might also notice that `malloc()` returns a pointer to type `void`. Doing so is just the way in C to pass back an address and let the programmer decide what to do with it. The programmer further helps out by using what is called a **cast**; in our example above, the programmer casts the return type of `malloc()` to a pointer to a `double`. Casting doesn't really accomplish anything, other than tell the compiler and other

programmers who might be reading your code: "yeah, I know what I'm doing." By casting the result of `malloc()`, the programmer is just giving some reassurance; the cast is not needed for the correctness.

14.3 The `free()` Call

As it turns out, allocating memory is the easy part of the equation; knowing when, how, and even if to free memory is the hard part. To free heap memory that is no longer in use, programmers simply call `free()`:

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

The routine takes one argument, a pointer that was returned by `malloc()`. Thus, you might notice, the size of the allocated region is not passed in by the user, and must be tracked by the memory-allocation library itself.

14.4 Common Errors

There are a number of common errors that arise in the use of `malloc()` and `free()`. Here are some we've seen over and over again in teaching the undergraduate operating systems course. All of these examples compile and run with nary a peep from the compiler; while compiling a C program is necessary to build a correct C program, it is far from sufficient, as you will learn (often in the hard way).

Correct memory management has been such a problem, in fact, that many newer languages have support for **automatic memory management**. In such languages, while you call something akin to `malloc()` to allocate memory (usually **new** or something similar to allocate a new object), you never have to call something to free space; rather, a **garbage collector** runs and figures out what memory you no longer have references to and frees it for you.

Forgetting To Allocate Memory

Many routines expect memory to be allocated before you call them. For example, the routine `strcpy(dst, src)` copies a string from a source pointer to a destination pointer. However, if you are not careful, you might do this:

```
char *src = "hello";
char *dst;        // oops! unallocated
strcpy(dst, src); // segfault and die
```

When you run this code, it will likely lead to a **segmentation fault**[3], which is a fancy term for **YOU DID SOMETHING WRONG WITH MEMORY YOU FOOLISH PROGRAMMER AND I AM ANGRY.**

[3] Although it sounds arcane, you will soon learn why such an illegal memory access is called a segmentation fault; if that isn't incentive to read on, what is?

TIP: IT COMPILED OR IT RAN $\neq$ IT IS CORRECT
Just because a program compiled(!) or even ran once or many times correctly does not mean the program is correct. Many events may have conspired to get you to a point where you believe it works, but then something changes and it stops. A common student reaction is to say (or yell) "But it worked before!" and then blame the compiler, operating system, hardware, or even (dare we say it) the professor. But the problem is usually right where you think it would be, in your code. Get to work and debug it before you blame those other components.

In this case, the proper code might instead look like this:

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // work properly
```

Alternately, you could use `strdup()` and make your life even easier. Read the `strdup` man page for more information.

Not Allocating Enough Memory

A related error is not allocating enough memory, sometimes called a **buffer overflow**. In the example above, a common error is to make *almost* enough room for the destination buffer.

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // work properly
```

Oddly enough, depending on how malloc is implemented and many other details, this program will often run seemingly correctly. In some cases, when the string copy executes, it writes one byte too far past the end of the allocated space, but in some cases this is harmless, perhaps overwriting a variable that isn't used anymore. In some cases, these overflows can be incredibly harmful, and in fact are the source of many security vulnerabilities in systems [W06]. In other cases, the malloc library allocated a little extra space anyhow, and thus your program actually doesn't scribble on some other variable's value and works quite fine. In even other cases, the program will indeed fault and crash. And thus we learn another valuable lesson: even though it ran correctly once, doesn't mean it's correct.

Forgetting to Initialize Allocated Memory

With this error, you call `malloc()` properly, but forget to fill in some values into your newly-allocated data type. Don't do this! If you do forget, your program will eventually encounter an **uninitialized read**, where it

reads from the heap some data of unknown value. Who knows what might be in there? If you're lucky, some value such that the program still works (e.g., zero). If you're not lucky, something random and harmful.

Forgetting To Free Memory

Another common error is known as a **memory leak**, and it occurs when you forget to free memory. In long-running applications or systems (such as the OS itself), this is a huge problem, as slowly leaking memory eventually leads one to run out of memory, at which point a restart is required. Thus, in general, when you are done with a chunk of memory, you should make sure to free it. Note that using a garbage-collected language doesn't help here: if you still have a reference to some chunk of memory, no garbage collector will ever free it, and thus memory leaks remain a problem even in more modern languages.

In some cases, it may seem like not calling `free()` is reasonable. For example, your program is short-lived, and will soon exit; in this case, when the process dies, the OS will clean up all of its allocated pages and thus no memory leak will take place per se. While this certainly "works" (see the aside on page 7), it is probably a bad habit to develop, so be wary of choosing such a strategy. In the long run, one of your goals as a programmer is to develop good habits; one of those habits is understanding how you are managing memory, and (in languages like C), freeing the blocks you have allocated. Even if you can get away with not doing so, it is probably good to get in the habit of freeing each and every byte you explicitly allocate.

Freeing Memory Before You Are Done With It

Sometimes a program will free memory before it is finished using it; such a mistake is called a **dangling pointer**, and it, as you can guess, is also a bad thing. The subsequent use can crash the program, or overwrite valid memory (e.g., you called `free()`, but then called `malloc()` again to allocate something else, which then recycles the errantly-freed memory).

Freeing Memory Repeatedly

Programs also sometimes free memory more than once; this is known as the **double free**. The result of doing so is undefined. As you can imagine, the memory-allocation library might get confused and do all sorts of weird things; crashes are a common outcome.

Calling `free()` Incorrectly

One last problem we discuss is the call of `free()` incorrectly. After all, `free()` expects you only to pass to it one of the pointers you received from `malloc()` earlier. When you pass in some other value, bad things can (and do) happen. Thus, such **invalid frees** are dangerous and of course should also be avoided.

ASIDE: **WHY NO MEMORY IS LEAKED ONCE YOUR PROCESS EXITS**

When you write a short-lived program, you might allocate some space using `malloc()`. The program runs and is about to complete: is there need to call `free()` a bunch of times just before exiting? While it seems wrong not to, no memory will be "lost" in any real sense. The reason is simple: there are really two levels of memory management in the system.

The first is level of memory management is performed by the OS, which hands out memory to processes when they run, and takes them back when processes exit (or otherwise die). The second level of management is *within* each process, for example within the heap when you call `malloc()` and `free()`. Even if you fail to call `free()` (and thus leak memory in the heap), the operating system will reclaim *all* the memory of the process (including those pages for code, stack, and, as relevant here, heap) when the program is finished running. No matter what the state of your heap in your address space, the OS takes back all of those pages when the process dies, thus ensuring that no memory is lost despite the fact that you didn't free it.

Thus, for short-lived programs, leaking memory often does not cause any operational problems (though it may be considered poor form). When you write a long-running server (such as a web server or database management system, which never exit), leaked memory is a much bigger issue, and will eventually lead to a crash when the application runs out of memory. And of course, leaking memory is an even larger issue inside one particular program: the operating system itself. Showing us once again: those who write the kernel code have the toughest job of all...

Summary

As you can see, there are lots of ways to abuse memory. Because of frequent errors with memory, a whole ecosphere of tools have developed to help find such problems in your code. Check out both **purify** [HJ92] and **valgrind** [SN05]; both are excellent at helping you locate the source of your memory-related problems. Once you become accustomed to using these powerful tools, you will wonder how you survived without them.

14.5 Underlying OS Support

You might have noticed that we haven't been talking about system calls when discussing `malloc()` and `free()`. The reason for this is simple: they are not system calls, but rather library calls. Thus the malloc library manages space within your virtual address space, but itself is built on top of some system calls which call into the OS to ask for more memory or release some back to the system.

One such system call is called `brk`, which is used to change the location of the program's **break**: the location of the end of the heap. It takes one argument (the address of the new break), and thus either increases or decreases the size of the heap based on whether the new break is larger or smaller than the current break. An additional call `sbrk` is passed an increment but otherwise serves a similar purpose.

Note that you should never directly call either `brk` or `sbrk`. They are used by the memory-allocation library; if you try to use them, you will likely make something go (horribly) wrong. Stick to `malloc()` and `free()` instead.

Finally, you can also obtain memory from the operating system via the `mmap()` call. By passing in the correct arguments, `mmap()` can create an **anonymous** memory region within your program — a region which is not associated with any particular file but rather with **swap space**, something we'll discuss in detail later on in virtual memory. This memory can then also be treated like a heap and managed as such. Read the manual page of `mmap()` for more details.

14.6 Other Calls

There are a few other calls that the memory-allocation library supports. For example, `calloc()` allocates memory and also zeroes it before returning; this prevents some errors where you assume that memory is zeroed and forget to initialize it yourself (see the paragraph on "uninitialized reads" above). The routine `realloc()` can also be useful, when you've allocated space for something (say, an array), and then need to add something to it: `realloc()` makes a new larger region of memory, copies the old region into it, and returns the pointer to the new region.

14.7 Summary

We have introduced some of the APIs dealing with memory allocation. As always, we have just covered the basics; more details are available elsewhere. Read the C book [KR88] and Stevens [SR05] (Chapter 7) for more information. For a cool modern paper on how to detect and correct many of these problems automatically, see Novark et al. [N+07]; this paper also contains a nice summary of common problems and some neat ideas on how to find and fix them.

References

[HJ92] Purify: Fast Detection of Memory Leaks and Access Errors
R. Hastings and B. Joyce
USENIX Winter '92
The paper behind the cool Purify tool, now a commercial product.

[KR88] "The C Programming Language"
Brian Kernighan and Dennis Ritchie
Prentice-Hall 1988
The C book, by the developers of C. Read it once, do some programming, then read it again, and then keep it near your desk or wherever you program.

[N+07] "Exterminator: Automatically Correcting Memory Errors with High Probability"
Gene Novark, Emery D. Berger, and Benjamin G. Zorn
PLDI 2007
A cool paper on finding and correcting memory errors automatically, and a great overview of many common errors in C and C++ programs.

[SN05] "Using Valgrind to Detect Undefined Value Errors with Bit-precision"
J. Seward and N. Nethercote
USENIX '05
How to use valgrind to find certain types of errors.

[SR05] "Advanced Programming in the UNIX Environment"
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
We've said it before, we'll say it again: read this book many times and use it as a reference whenever you are in doubt. The authors are always surprised at how each time they read something in this book, they learn something new, even after many years of C programming.

[W06] "Survey on Buffer Overflow Attacks and Countermeasures"
Tim Werthman
Available: www.nds.rub.de/lehre/seminar/SS06/Werthmann_BufferOverflow.pdf
A nice survey of buffer overflows and some of the security problems they cause. Refers to many of the famous exploits.

Homework (Code)

In this homework, you will gain some familiarity with memory allocation. First, you'll write some buggy programs (fun!). Then, you' use some tools to help you find the bugs you inserted. Then, you will realize how awesome these tools are and use them in the future, thus making yourself more happy and productive.

The first tool you'll use is `gdb`, the debugger. There is a lot to learn about this debugger; here we'll only scratch the surface.

The second tool you'll use is `valgrind` [SN05]. This tool helps find memory leaks and other insidious memory problems in your program. If it's not installed on your system, go to the website and do so:

```
http://valgrind.org/downloads/current.html
```

Questions

- First, write a simple program called `null.c` that creates a pointer to an integer, sets it to `NULL`, and then tries to dereference it. Compile this into an executable called `null`. What happens when you run this program?
- Next, compile this program with symbol information included (with the `-g` flag). Doing so lets puts more information into the executable, enabling the debugger access more useful information about variable names and the like. Run the program under the debugger by typing `gdb null` and then, once `gdb` is running, typing `run`. What does gdb show you?
- Finally, use the `valgrind` tool on this program. We'll use the `memcheck` tool that is a part of `valgrind` to analyze what happens. Run this by typing in the following: `valgrind --leak-check=yes null`. What happens when you run this? Can you interpret the output from the tool?
- Write a simple program that allocates memory using `malloc()` but forgets to free it before exiting. What happens when this program runs? Can you use `gdb` to find any problems with it? How about `valgrind` (again with the `--leak-check=yes` flag)?
- Write a program that creates an array of integers called `data` of size 100 using `malloc`; then, set `data[100]` to zero. What happens when you run this program? What happens when you run this program using `valgrind`? Is the program correct?
- Create a program that allocates an array of integers (as above), frees them, and then tries to print the value of one of the elements of the array. Does the program run? What happens when you use `valgrind` on it?
- Now pass a funny value to free (e.g., a pointer in the middle of the array you allocated above). What happens? Do you need tools to find this type of problem?

- Try out some of the other interfaces to memory allocation. For example, create a simple vector-like data structure and related routines that use `realloc()` to manage the vector. Use an array to store the vectors elements; when a user adds an entry to the vector, use `realloc()` to allocate more space for it. How well does such a vector perform? How does it compare to a linked list? Use `valgrind` to help you find bugs.
- Spend more time and read about using `gdb` and `valgrind`. Knowing your tools is critical; spend the time and learn how to become an expert debugger in the UNIX and C environment.