CS 0449: Introduction to

SYSTEMS SOFTWARE

Jonathan Misurda
Computer Science Department
University of Pittsburgh
jmisurda@cs.pitt.edu
http://www.cs.pitt.edu/∼jmisurda

Version 3, revision 0

Last modified: August 12, 2014 at 8:54 P.M.

This reference is dedicated to the students of CS 0449, Fall 2007 (2081). Their patience in dealing with a changing course and feedback on the first version of this text was greatly appreciated.

Contents

List of Figures

List of Code Listings

Preface

At some point late last decade, the core curriculum of many major cs programs, including the one here at the University of Pittsburgh, switched to teaching Java, C♯, or python. While there is no mistaking the prevalence or importance of a modern, Object-Oriented, garbage-collected programming language, there is also plenty of room for an "old-fashioned" do-it-yourself language like C. It is still the language of large programs and Operating Systems, and learning it opens the door to doing work with real-life systems like GNU/Linux.

Armed with a C compiler, we can produce executable programs, but how were they made? What does the file contain? What actually happens to make the program run? To be able to answer such questions about a program and its interactions seems to be fundamental to the issue of defining a system. Biologists have long known the benefit of studying life in its natural environment, and Computer Scientists should do no different. If we follow such a model and study a program's "life" as we run it on the computer, we will begin to learn about each of the parts that work together. Only then can we truly appreciate the abstractions that the Operating System, the hardware, and high-level programming languages provide to us.

This material picks up where high-level programming languages stop, by looking at interactions with memory, libraries, system calls, threading, and networking. This course, however, avoids discussing the implementations of the abstractions the system provides, leaving those topics for more advanced, specialized courses later in the curriculum.

I started out writing this text not as a book, but as a study guide for the second exam in *CS 0449: Introduction to Systems Software*. I did not have a specific textbook except for the C programming portion of the course, and felt that the students could use something to help them tie in the lectures, the course slides, and the projects. So I sat down one Friday afternoon and, through the next four days, wrote a 60-page "pamphlet." I've since decided to stick with my effort for the next term, as part of a four-book curriculum that includes two other freely available texts on Linux Programming and Linux Device Drivers (links available in the Bibliography Section).

Over time, I hope to continue to add new topics and refine the material presented here. I appreciate any feedback that you, the reader, might have. Any accepted significant improvement is usually worth some extra credit to my students. All correspondence can be directed to the email address found on the front cover.

Acknowledgments

No book, not even one so quickly written, is produced in a vacuum. I would like to thank all of my students for their patience while we all came to terms with my vision for the course. I especially want to thank Nathaniel Buck, Stacey Crotti, Jeff Kaminski, and Gerald Maloney for taking the time to provide detailed feedback. I'd also like to thank my parents and friends for contributing in ways from just basic support to the artistic. This text would not be the same without all of your help.

Notes on the Second Edition

In this edition, I have attempted to improve the integration of the material into the modern Java curriculum. Several sections (some old, some new) are marked with the ☕ icon and show up in the PDF outline in *italics*. These indicate that the material relates to the Java language or the Java Virtual Machine.

—Jonathan Misurda
May 1, 2008

1 | Pointers

As part of implementing a sorting algorithm, we often need to exchange the values of two items in an array. Good programming practice suggests that when we have some commonly-reused code we should wrap it in a function:

```
void swap(int a, int b) {
        int t = a;
        a = b;
        b = t;
}
```

Then we can call it from our program with `swap(a,b)`. However, if we initially set `x=3; y=5;` and run the above swap function, the values of `x` and `y` remain unchanged. This should be unsurprising to us because we know that when we pass parameters to a function they are passed "by value" (also known as passing "by copy").

This means that `a` and `b` contain the same values as `x` and `y` at the moment that the call to `swap()` occurs because there is an implicit assignment of the form `a=x; b=y;` at the call site. From that point on in the function `swap()`, any changes to `a` and `b` have no effect on the original `x` and `y`. Thus, our swap code works fine inside the scope of `swap()` but once the scope is destroyed when the function returns, the changes are not reflected in the calling function.

Java Content

We then wonder if there is another way to write our `swap()` so that it succeeds. Our first inclination might be to attempt to return multiple values. In C, like in Java, this is not directly possible. A function may only return one thing. However, we could wrap our values in a structure or array and return the one aggregate object, but then we have to do the work of extracting the values from the object, which is just as much work as doing the swap in the first place. We soon realize that we cannot write a swap function that exchanges the values of its arguments in any reasonable fashion. If we are programming in Java with the primitive data types, this is where our attempts must stop. However, in C we have a way to rewrite the function to allow it to actually work.

In fact, this should not be surprising because we are essentially asking a function to return multiple pieces of data, and we have already seen one function that can do that: `scanf()`. If we pass `scanf()` a format string like `"%d %d"` it will set two argument variables to the integers that the user inputs. In essence, it is doing what we just said could not be done: It is modifying the values of its parameters. How is that accomplished? The answer is by using pointers.

1.1 Basic Pointers

A **pointer** is a variable that holds an address. An **address** is simply the index into memory that a particular value lives at. Memory (specifically RAM) is treated as an array of bytes, and just like our regular arrays, each element has a numerical index. We haven't needed pointers until now because we have given our variables names and used those names to refer to these locations. We don't even know the actual addresses because the compiler and the system automate the layout and management of many of our variables. However, unlike in Java, it is possible in C to ask for the location of a particular variable in memory.

Referring to a location via a name or an address is not something unique to C, or even to computers in general. For example, you can refer to the room in which I work at Pitt as "Jon's Office," which is a name, or as "6203 Sennott Square" which is an address. Both are ways of referring to the same location and this duplication is known as *aliasing*.

1.1.1 Fundamental Operations

We can declare a pointer variable using a special syntax. Pointers in C have a type to them just like our variables did, but in the context of pointers, this type indicates that we are pointing to a memory location that stores a particular data type. For instance, if we want to declare a pointer that will hold the address in memory of where an integer lives, we would declare it as:

```
int *p;
```

where the asterisk indicates that `p` is a pointer. We need to be careful with declaring multiple variables on one line because its behavior in regards to pointers is surprising. If we have the declaration:

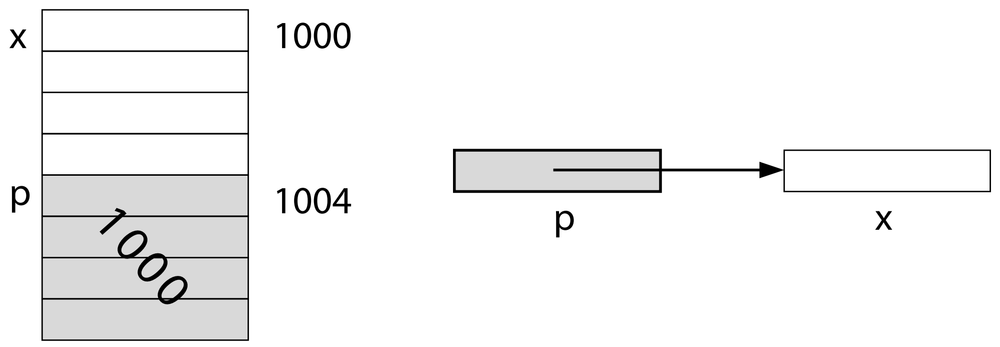


Figure 1.1: Two diagrammatic representations of a pointer pointing to a variable.

```
int *p, q;
```

or even:

```
int* p, q;
```

we get an *integer pointer* named p and an *integer* named q. No matter where you place the asterisk, it binds to the next variable. To avoid confusion, it is best to declare every variable on its own separate line.

To set the value of a pointer, we need to be able to get an address from an existing variable name. In C, we can use the address-of operator, which is the unary ampersand (&) to take a variable name and return its address:

```
int x;
int *p;

p = &x;
```

This code listing declares an integer x that lives someplace in memory and an integer pointer p that also lives somewhere in memory (since pointers are variables too). The assignment sets the pointer p to "point to" the variable x by taking its address and storing it in p.

Figure 1.1 shows two ways of picturing this relationship. On the left, we have a possible layout of RAM, where x lives at address 1000 and p lives at address 1004. After the assignment, p contains the value 1000, the address of x. On the right, much more abstractly, is shown the "points-to" relationship.

Now that we have the pointer `p` we can use it as another name for `x`. But in order to accomplish that, we need to be able to traverse the link of the points-to relationship. When we follow the arrow and want to talk about the location a pointer points-to rather than the pointer itself, we are doing a **dereference** operation. The dereference operator in C is also the asterisk (`*`). Notice that although we used the asterisk in the pointer definition to declare a variable as a pointer, the dereference operator is different.

When we place the asterisk to the left of a pointer variable or expression that yields a pointer, we chase the pointer link and are now referring to the location it points to. That means that the following two statements are equivalent:

```
x = 4;            *p = 4;
```

Note that it is usually a mistake to assign a pointer variable a value that is not computed from taking the address of something or from a function that returns a pointer. This general rule should remind us that `p = 4;` would not be appropriate because we do not normally know in advance where memory for objects will be reserved.

1.2 Passing Pointers to Functions

With the basic operations of address-of and dereference, we can begin to use pointers to do new and useful tasks. If we make a slight modification to our previous `swap()` function, we can get it to work:

```c
void swap(int *a, int *b) {
        int t = *a;
        *a = *b;
        *b = t;
}
```

We also need to change the way we invoke it. If we have our same variables, `x` and `y`, we would call the function as `swap(&x, &y)`. After `swap()` returns, we now find that `x` is 5 and `y` is 3. In other words, the swap worked.

To better understand what happened here, we can trace the code constructing a picture like before. Figure 1.2 shows the steps. When the `swap()` function is called, there are four variables in memory: `x` and `y` which contain 3 and 5, respectively, and `a` and `b` which get set to the addresses of `x` and `y`. Next, a temporary variable `t` is created and initialized to the value of what `a` points to, i.e., the value of `x`. We then

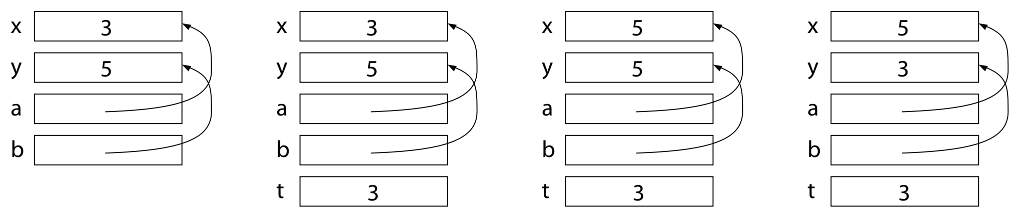


Figure 1.2: The values of variables traced in the `swap()` function.

copy the value of what `b` points to (namely the value of `y`) into the location that `a` points to. Finally, we copy into the location pointed to by `b` our temporary variable.

Our swap function is an example of one of the two ways that pointers as function parameters are used in C. In the case of `swap()`, `scanf()`, and `fread()`, among many others, the parameters that are passed as pointers are actually acting as additional *return values* from the functions.

The other reason (which is not mutually exclusive from the previous reason) that pointers are used as parameters is for time and space efficiency in passing large objects (such as arrays, structs, or arrays of structs). Since we have already established that C is pass-by-value, if we pass a large object to a function, that object would have to be duplicated and that might take a long time. Instead, we can pass a pointer (which is really just integer-sized), thus taking no noticeable time to copy. This is why `fwrite()` takes a pointer parameter even though it does not change the object in memory.

1.3 Pointers, Arrays, and Strings

With our knowledge of pointers, we now can understand why we had to do certain things such as prefix variable names with an ampersand for `scanf()`. However, you may recall there passing a string was the exception to that rule. To understand why this is the case, we need to explain a fundamental identity in C:

The name of an array is a (read-only) pointer to it's beginning.

Imagine we declare an array: `int a[4];`. If we wish to refer to a particular integer in the array, we can subscript the array and write something along the lines of `a[i]`, which represents the i^{th} item. An alternative way to view it is that `a[i]` is i integers away from the start of the array. This is valid because we know that all elements of an array are laid out consecutively in memory. Thus, in essence, if we knew where the entire array started, we could easily add an offset to that address and find a particular element in the array.

To express it mathematically, if we have an array with address base, then the i^{th} element lives at base $+ i \times$ sizeof(type) where type is the type of the elements in the array. This simple calculation is so convenient that C decides to enable it by making the name of the array be a pointer to the beginning of the array in memory. This means that if we have an array element indexed as `a[3]`, it lives at the address $a + 3 \times$ `sizeof`$($`int`$)$.

However, when we convert this formula to C code, there is a slight adjustment we must make. The appropriate way to rewrite `a[3]` using pointers and offsets is `*(a+3)`. Where did the `sizeof(int)` term go? The answer lies in how C does **pointer arithmetic**.

1.3.1 Pointer Arithmetic

Pointers are allowed three mathematical operations to be performed on them. Addition and subtraction of an integer offset is allowed in order to support the aforementioned address/offset calculations. The third operation is that two pointers of the same type are allowed to be subtracted from each other in order to determine an offset between the pointers.

In the case of addition or subtraction of an integer offset, C recognizes that if you start with a pointer to a particular type, you will still want a pointer to that same type when you do your arithmetic. If the expression `a+1` added one byte to the pointer, we'd now be pointing into the middle of the integer in memory. What C wants is for `a` to point to the next int, so it automatically scales the offset by the size of the type the pointer points to.

Because addition and subtraction are supported, C also allows the pre- and post-increment and -decrement operators to be applied to pointers. These are still equivalent to the +1 and -1 operations as on integers.

This means that a particularly sadistic person could write the following function:

```
void f(char *a, char *b) {
        while(*a++ = *b++) ;
}
```

This function is one we have already discussed in the course. It is `strcpy()`. The way that it works is that the post-increment allows us to walk one character at a time through both the source and destination arrays. The dereferences turn the pointers into character values. The assignment does the copy of a letter, and yields the right-hand side as the result of the expression. The right-hand side is then evaluated in a boolean context to determine if the loop stops or continues. Since every character is non-zero, the loop continues. The loop terminates when the nul-terminator character is copied, because its value is zero and thus false. The loop needs no body, the entirety of the work is done as side-effects of the loop condition.

1.4 Terms and Definitions

The following terms were introduced or defined in this chapter:

Address The index into memory where a particular value lives.

Dereference To follow the link of a pointer to the location it points to. In C, the dereference operator is *.

Pointer A variable that holds an address.

Pointer Arithmetic Adding or subtracting an offset to a pointer value. In C, this offset is automatically scaled by the type the pointer points to.

2 | Variables: Scope & Lifetime

In a programming language, **scope** is a term that refers to the region in a program where a symbol is legal and meaningful. A **symbol** is a name that represents a constant, literal, or variable. Scope serves a dual purpose. It allows for a symbol to be restricted to some logical division of the program, and it allows for duplicate names to exist in non-overlapping regions.

Restricting the portion of a program where a symbol is legal allows for a simple form of encapsulation, also known as "data-hiding." By restricting a symbol's access to a programmer-defined, limited region of the program, a programmer can be sure that minimal code affects the value of a particular variable. Allowing duplicate names is important when there are many programmers working together. Imagine a language where every variable was global. For many people to work together, they would have to avoid using the same variable names as anyone else. This would lead to some cumbersome naming convention and not encourage using the clearest name for a variable.

While scope is a compile-time property of code, when the program is actually executed, variables are created and destroyed. The time from which a particular memory location is allocated until it is deallocated is referred to as that variable's **lifetime**.

The precise rules governing scope are programming language dependent, and a language construct rather than something enforced by the organization of the computer. If we have a variable that is restricted to a particular function (usually called a **local variable**), there is nothing preventing that variable from being created at program start and not destroyed until program termination.

What determines the lifetime of a variable is whether it is **static** or **dynamic** data. Static means non-moving, just as *static electricity* is an electrical charge separated from an opposite charge, and not moving towards it (as shown in Figure 2.1). In computing, static refers to when a program is compiled or anytime before it is run. If

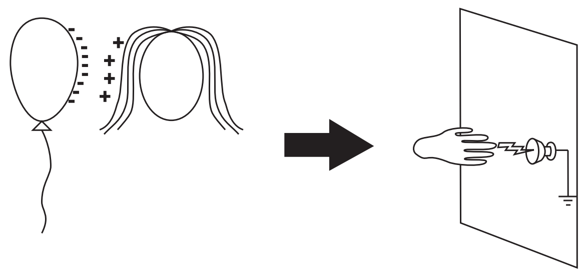


Figure 2.1: Static electricity is a built-up charge that is separated from an opposite charge. When you touch a grounded object, the charges suddenly flow to cancel out, giving you a shock.

the compiler can compute exactly how much space will be needed for variables, the variables are static data. Dynamic data is that which is allocated while the program runs, since its size may depend on input or random values.

At first glance, local variables would seem to be static data, since the compiler can determine exactly how much space is necessary for them. However, while the compiler may be able to compute the memory needs of a function, there is not always a way to determine how many times that function may be called if, for instance, it is recursive. Since each invocation of the function needs its own copy of the local variables, allocation of their storage must be dealt with at run-time.

Static data can be allocated when the program begins and freed when the program exits. Dynamic data, on the other hand, will need special facilities to handle the fluctuations in the demand for memory while the program runs.

2.1 Scope and Lifetime in C

Scopes in C are defined by files and blocks. Remember that a block is a region of code enclosed in curly braces: { and }. In C, local variables are legal in the scope they are declared in, as well as all scopes that are nested inside of that scope. Figure 2.2 illustrates three nested scopes in a C program: the file, the function, and a loop within the function. A variable declared in the file is accessible anywhere, but a variable declared inside the loop can only be used inside that loop.

While two variables with the same name may not occupy the same scope, there is nothing preventing a nested scope from naming a variable the same as one in an enclosing scope. When this occurs, the variable of the outer (larger) scope is

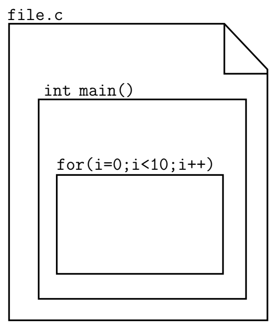


Figure 2.2: The nested scopes in a C program.

hidden, and the innermost variable is said to be **shadowing** the outer one.

Listing 2.1 shows an example of a shadowed variable. The new block redeclares the `shadowed` variable, and when this program is run, the value "6" is displayed on the screen. In general, using shadowed variables is not good practice and can lead to a great deal of confusion to someone reading the code.

2.1.1 Global Variables

A **global variable** is a variable that is accessible to all functions and which retains its value throughout the entire execution of the program. In C, any variable declared outside of a function could be considered global. However, C treats a file as a scope, so "global" variables are actually limited to the file they are declared in.

What makes these file-scoped variables special is that they can be imported into the scope of other files by the `extern` keyword. If File A contains a global declaration like `int x`, File B can also refer to that variable by redeclaring it, but with the `extern` qualifier: `extern int x`.

2.1.2 Automatic Variables

Variable declarations that occur inside functions are implicitly declared `auto`, meaning an automatic variable. Automatic variables are variables that are created and destroyed automatically by code generated from the compiler. In general, the lifetime of automatic variables is the lifetime of the block they are defined in. However, it may be more convenient for all automatic variables to be created on function entry and destroyed at function return. This is implementation specific and has

```
#include <stdio.h>

int main() {
        int shadowed;
        shadowed = 4;
        {
                int shadowed;
                shadowed = 6;
                printf("%d\n", shadowed);
        }
        return 0;
}
```

Listing 2.1: Variables in inner scopes can shadow variables of the same name in enclosing scopes.

no bearing on the correctness of a program since the scope is narrower than the lifetime.

The only problem that could arise with automatic variables comes as an abuse of pointers. Consider the code of Listing 2.2. Here function `f()` creates and returns a pointer to an automatic variable, which function `main()` captures. However, when `main()` goes to use the memory location referenced by the pointer `p`, that variable is "dead" and can no longer safely be used. The `gcc` compiler is kind enough to issue a warning if we do this:

```
(14) thot $ gcc escape.c
escape.c: In function `f':
escape.c:5: warning: function returns address of local variable
```

However, the code actually compiles and a program is produced. Proving again that C is not picky about what you do, no matter if you mean it or not.

2.1.3 Register variables

The old C compilers were not always very intelligent when it came to machine code generation. Thus, a programmer was allowed to give a hint to the compiler to indicate that a certain variable was particularly important. Such heavily accessed variables should be stored in architectural registers rather than in memory. Declar-

```
int *f() {
        int x;
        x = 5;
        return &x;
}

int main() {
        int *p;
        p = f();
        *p = 4;
        return 0;
}
```

Listing 2.2: A pointer error caused by automatic destruction of variables.

ing a variable as `register` could have a significant improvement on the performance of frequently executed code such as in loops.

Modern compilers support the `register` keyword, but most simply ignore it, due to the fact that the compiler will examine the code and make intelligent decisions about what data to place in registers or in memory. This is done for every variable without having to specify anything more than a compiler option, if even that.

2.1.4 Static Variables

Static is a much-overloaded term in Computer Science. Earlier it was defined as pertaining to when a program is compiled. In Java, it was used to declare a method or field that existed independently of any instances of a class. In C, `static` is a keyword with two new meanings, depending on whether it is applied to a local variable, or to a global variable or function.

Static Local Variables

So far, the distinction between scope and lifetime seems somewhat unnecessary. Local variables have a lifetime approximately that of their scope, and global variables need to live for the whole execution of a program. However, if we think about all possible combinations of scope and lifetime, there are two we have not addressed. The first, a global scope but a local lifetime, is a recipe for disaster. This is basically what occurs when returning a pointer to an automatic variable, and even the com-

```
char *asctime(const struct tm *timeptr) {
    static char wday_name[7][3] = {
        "Sun","Mon","Tue","Wed","Thu","Fri","Sat"
    };
    static char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    static char result[26];

    sprintf(result,
         "%.3s␣%.3s%3d␣%.2d:%.2d:%.2d␣%d\n",
         wday_name[timeptr->tm_wday],
         mon_name[timeptr->tm_mon],
         timeptr->tm_mday, timeptr->tm_hour,
         timeptr->tm_min, timeptr->tm_sec,
         1900 + timeptr->tm_year);

     return result;
}
```

Listing 2.3: Unlike automatic variables, pointers to static locals can be used as return values.

piler warned that was a bad idea. The other combination is a variable with a local scope but a global lifetime. This combination would imply that the variable could be used only within the block it was declared in but would retain its value between function invocations. Such a variable type might be useful as a way to eliminate the need for a global variable to fulfill this role. Anytime a global variable can be eliminated is usually a good thing.

By declaring a local variable `static`, the variable now will keep its value between function invocations, unlike an automatic variable. Interestingly, an implication of this is that static local variables can safely have pointers to them as return values. Listing 2.3 shows the `man` page for the `asctime()` function, which builds the string in a static local variable. The advantage to this is that the function can handle the allocation but does not require the caller to use `free()` as if `malloc()` had been used.

```c
#include <string.h>
#include <stdio.h>

int main() {
    char str[] = "The␣quick␣brown";
    char *tok;

    tok = strtok(str, "␣");

    while(tok != NULL) {
                printf("token:␣%s\n", tok);
                tok = strtok(NULL, "␣");
    }

    return 0;
}
```

Listing 2.4: The `strtok()` function can tokenize a string based on a set of delimiter characters.

The quintessential example of static local variables is the standard library function `strtok()`, whose behavior is somewhat atypical. Listing 2.4 shows an example of splitting a string based on the space character. The first time `strtok()` is called, the string to tokenize and the list of delimiters are passed. However, the second and subsequent times the function is invoked, the string parameter should be `NULL`. Additionally, `strtok()` is destructive to the string that is being tokenized. In order to understand this, imagine that the following string is passed to `strtok()` with space as the delimiter:

t	h	e		q	u	i	c	k		b	r	o	w	n	\0

On the first call to `strtok()`, the return value should point to a string that contains the word "the." On the second call, where `NULL` is passed, the return value should point to "quick." If we now examine the original string in a debugger, we would see the following:

t	h	e	\0	q	u	i	c	k	\0	b	r	o	w	n	\0

The delimiter characters have all been replaced by the null terminator! That explains why we cannot pass the original string on the second call, since even `strtok()` will stop processing when it encounters the null, thinking that the string is over. So now we have "lost" the remainder of the string. The `strtok()` function, however, remembers it for us in a static local variable. Passing `NULL` tells the function to use the saved pointer from the last call and to pick up tokenizing the string from the point it left off last.

Static Global Variables

The `static` keyword, when applied to file-scope variables, takes on an entirely different meaning. Coming from an object-oriented language such as Java, the natural comparison is to consider the `static` keyword as equivalent to the `private` keyword. In C, `static` restricts use of a file-scoped variable to only the file it was declared in. No other file may import it into that file's scope through the `extern` keyword. The variable is hidden and may not be shared.

As a note, this is also true if the `static` keyword is used to prefix a function. The function then may not be called by code from any other files in the program. In this manner, variables and functions can be kept isolated from each other, one of the original points of scoping.

2.1.5 Volatile Variables

In most cases, the compiler will decide when to put a value into a register and when to keep it in memory. In certain cases, the memory location that data is stored in is special. It could be a shared variable that multiple threads or processes (see Chapter 9) are using. It could be a special memory location that the Operating System maps to a piece of hardware. In these cases, the value of the variable in the register may not match the value that is in memory. Declaring a variable as `volatile` tells the C compiler to always reload the variable's value from memory before using it. In a way, this could be thought of as the opposite of the `register` keyword.

The `volatile` keyword is somewhat rare to see in normal desktop applications, but it is useful in systems software when interacting with hardware devices. Though you may not encounter it much, it is nonetheless important to remember for that one time you might need it (or more if you develop low-level programs).

2.2 Summary Table

The following table summarizes the scope and lifetimes that C provides:

	Scope	**Lifetime**
Automatic	The block it is defined in	The life of the function
Global	The entire file plus any files that import it using `extern`	The life of the program
Static Global	The entire file, but may not be imported into any other file using `#1`extern	The life of the program
Static Local	The block it is defined in	The life of the program

2.3 Terms and Definitions

The following terms were introduced or defined in this chapter:

Dynamic While a program is being run.

Lifetime The time from which a particular memory location is allocated until it is deallocated.

Local variable A variable with a scope limited to a function or smaller region.

Scope The region in a program where a symbol is legal and meaningful.

Shadowing A variable in an inner scope with the same name as one in an enclosing scope. The innermost declaration is the variable that is used; it shadows the outermost declaration.

Static While a program is being compiled.

Symbol A name that represents a constant, literal, or variable.

3 Compiling & Linking: From Code to Executable

In this chapter, we begin to look at the computer as a **system**: A group of programs written by many different people, all trying to work together to accomplish useful tasks. No single component exposes the innermost workings of a system quite as well as the compiler. We begin this chapter by briefly describing the process of compilation, linking, and the makeup of executable files.

Certainly programming languages other than C can be compiled and executed, but C's original purpose was for writing Operating Systems. Thus we will discuss the C compiler as the prototypical compiler. Modern Linux systems use the `gcc` compiler, created by Richard Stallman as part of his free GNU system. This text will assume the `gcc` compiler, and it has been used to test the code listings found throughout the book.

3.1 The Stages of Compilation

Figure 3.1 shows the typical stages that a C program goes through while being compiled under the `gcc` compiler on Unix/Linux. Starting with one or more C code source files, the files are first sent to `cpp`, the C **Preprocessor**. The preprocessor is responsible for doing textual replacement on the input files by following all of the commands that begin with a `#`. After the preprocessor does its job, the fully expanded C code is passed to `cc1`, the **compiler**, which is responsible for syntax checking and code generation. If there are multiple sources of code in your program (which if you are using any of the standard library code, there are), `ld`, the **linker**, needs to resolve how to find this code. Some code will be copied into the executable and some will be left out until the program is loaded. At this point, however, there is a recognizable executable file (assuming there were no errors).

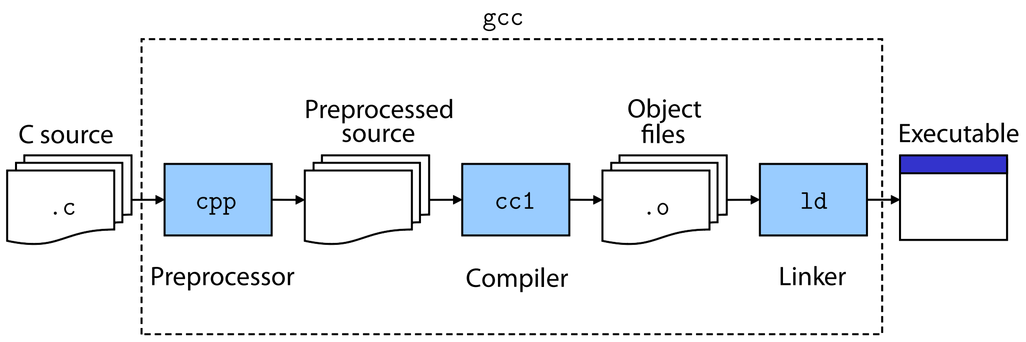


Figure 3.1: The phases of the `gcc` compiler for C programs.

3.1.1 The Preprocessor

The two most common preprocessor directives are `#include` and `#define`. When the preprocessor encounters a `#include` command, it seeks out the file whose name comes after the include command and inserts its contents directly into the C code file. The preprocessor knows where to look for these files by the delimiters used around the filename. Here are two examples:

```
#include <stdio.h>
#include "myheader.h"
```

If $<$ and $>$ are used, the preprocessor looks on the include path[1] where the standard header files for the system are found. A **header file** contains function and data type definitions that are found outside of a particular file. If " and " are used instead, the local directory is searched for the named file to include.

The directive `#define` creates a macro. A **macro** is a simple or parameterized symbol that is expanded to some other text. One common use of a macro is to define a constant. For example, we might define the value of π in the following way:

```
#define PI 3.1415926535
```

Now we may use the symbol `PI` whenever we want the value of π:

```
double degrees_to_radians(double degrees) {
        return degrees * (PI / 180.0);
}
```

1 Under Linux this is usually `/usr/include`.

We can actually parameterize our macros to allow for more generic substitutions. For instance, if we frequently wanted to know which of two numbers is larger, we could create a macro called `MAX` that does this for us:

```
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
```

Notice that we do not need to put any type names in our definition. This is because the preprocessor has no understanding of code or types; it is just following simple substitution rules: Whatever is given as `a` and `b` will be inserted into the code. For instance:

```
c = MAX(3,4);
```

will become:

```
c = (((3) > (4)) ? (3) : (4));
```

However,

```
c = MAX("bob","fred");
```

will become:

```
c = ((("bob") > ("fred")) ? ("bob") : ("fred"));
```

which is not legal C syntax. The preprocessor will do anything you tell it to, but there is no guarantee that what it produces is appropriate C.

3.1.2 The Compiler

The compiler is a complex and oftentimes daunting piece of software. There is an entire course on it alone. (And another one at the graduate level!) Thankfully, what we need to understand about the compiler is fairly simple. The compiler takes our source code, checks to make sure it is legal syntax, and then generates machine code. This machine code may be **optimized** — meaning that the compiler had some rule by which it transformed your code into something it believed would run faster or take up less memory.

The output of the compiler is an **object file**, which is denoted by the `.o` extension when using `gcc`.[2] There is one object file produced per source code file. An object file contains machine code, but any function call to code that is in a different source file is left unresolved. The address the code should jump to is unknown at this stage.

2 Visual Studio under Windows produces object files with the extension `.obj`

3.1.3 The Linker

Code in an executable can come from one of three places:

1. The actual source code
2. Libraries
3. Automatically generated code from the linker

The job of the linker is to assemble the code from these three places and create the final program file.

The source code is, of course, the code the programmer has written. The **libraries** are collections of code that accomplish common tasks provided by a compiler writer, system designer, or other third party.[3] C programs nearly always refer to code provided by the **C Standard Library**. The C Standard Library contains helpful functions to deal with input and output, files, strings, math, and other common programming tasks. For instance, we have made much use of the function `printf()` in our programs. To gain access to this function in our code, two independent steps must be done.

The first step is to inform the compiler that there is a function named `printf()` that takes a format string followed by a variable number of arguments. The compiler needs to know this information to do type checking (to ensure the return value and parameters match their specifications) and to generate appropriate code (which will be elaborated upon in Chapter 6). This information is provided by a function prototype declaration that resides in `<stdio.h>`.

The second step is to tell the linker how to find this code. The simplest way to assemble all three sources of code into a program is to literally put it all into one big file. This is referred to as **static linking**. It is not the only option, however. Remember that static is a term that is often used to describe the time a program is compiled. Its opposite is dynamic — while the program is running. It comes then as no surprise that we also have the option to **dynamically link** libraries, so that the code is not present in the executable program but is inserted later while the program loads (link loading) or executes (dynamic loading).

Each of these techniques has the same goal: put the code necessary for our program to run into RAM. Most common computer architectures follow the **von Neumann Architecture** where both code and data must be loaded into a main memory

3 You can always write your own libraries as well!

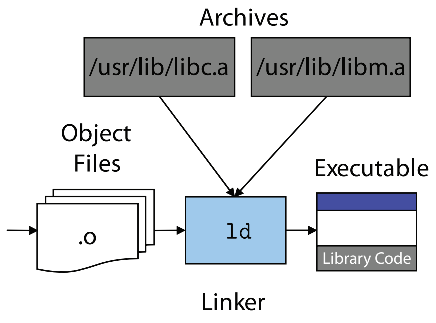


Figure 3.2: In static linking, the linker inserts the code from library archives directly into the executable file.

before instructions can be fetched and executed. Static linking puts the code into the executable so that when the Operating System loads the program, the code is trivially there. Dynamic linking defers the loading of library code until runtime. We will now discuss the issues and trade-offs for each of these three mechanisms.

Static Linking

Static linking occurs during compilation time. Figure 3.2 gives an overview of the linker's function during static linking. When the linker is invoked as part of the compilation process, code is taken from the libraries and inserted into the executable file directly. The code for the libraries in Unix/Linux comes from archive (`.a`) files. The linker reads in these files and then copies the appropriate code into the executable file, updating the targets of the appropriate `call` instructions.

The advantages of static linking are primarily simplicity and speed. Because all targets of calls are known at compile time, efficient machine code can be generated and executed. Additionally, understanding and debugging the resultant program is straightforward. A statically-linked program contains all of the code that is necessary for it to run. There are no dependencies on external files and installation can be as simple as copying the executable to a new machine.

There are two major disadvantages to static linking, however. The first is an issue of storage. If you examine a typical Unix/Linux machine, you will find hundreds of programs that all make use of the `printf()` function. If every one of these programs had the code for `printf()` statically linked into its executable, we would have megabytes of disk space being used to store just the code for `printf()`.

The second major disadvantage is exposed by examining such programs under

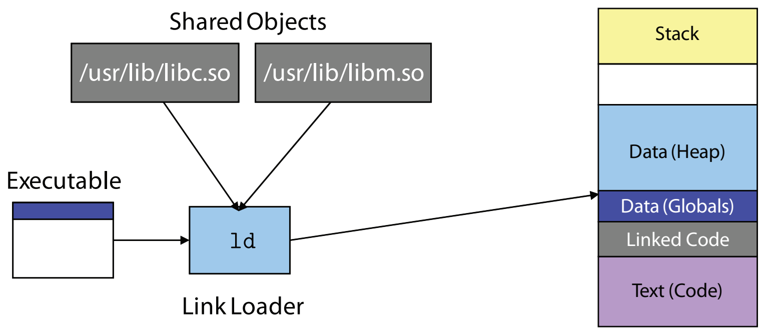


Figure 3.3: Dynamic linking resolves calls to library functions at load time.

the light of Software Engineering, where modularity — the ability to break a program up into simple, reusable pieces — is emphasized. Imagine that a bug in `printf()` is subsequently discovered on our system that is entirely statically linked. To fix our bug, we will have to find every program that uses `printf()` and then recompile them to include the fixed library. For programs whose source code we do not have, we would have to wait until the vendor releases an update, if ever.

Dynamic Linking

A better approach for code that will be shared between multiple programs is to use dynamic linking. Figure 3.3 shows the process. Notice that the executable file has already been produced, and that we are about to load and execute the program. In dynamic linking, the linker is invoked twice: once at compile time and once every time the program is loaded into memory. For this reason, the linker is sometimes referred to as a **link loader**.

When the linker is invoked as part of the compiler (`ld` as a part of `gcc`, for instance) the linker knows that the program will eventually be loaded, and any library calls will be resolved. The linker then inserts some extra information into the executable file to help the linker at load time. When the linker runs again, it takes this information, makes sure the dynamically linked library is loaded into memory, and updates all the appropriate addresses to point to the proper locations.

On Unix/Linux, dynamically linked libraries are called **shared objects** and have `.so` as their extension. Windows calls them Dynamically Linked Libraries (appropriately) with the extension `.DLL`.

Dynamic linking allows for a system to have a single copy of a library stored on disk and for the library code to be copied into the process's address space at load time.

The term for the remove of duplicate information is **deduplication**. Deduplication is a common technique that is necessary for systems to work with large amounts of data. Consider an email provider like Google's GMail. If a spammer managed to get a list of one million GMail addresses, they could send a 1MB file to each of them. That would require GMail to naïvely store 1TB of data for this one email message. That obviously wouldn't be possible. Instead, GMail deduplicates email messages and attachments, storing them just once. Each user has a reference to that object that they see. The disadvantage is that GMail cannot delete the file when any individual user clicks delete, but must rather wait for everyone who received the email to delete it before reclaiming that space.

Fixing a bug in a library only requires the library to be updated. Since it is an independent file, the one-and-only copy of the library can be replaced with the updated version and all future references to the library at load time will be resolved with the updated version.

While dynamic linking solves the storage and update problems, it introduces some issues of its own. The first issue is that the extra work to resolve the addresses to their actual location is done at load time and thus slows the program's execution. The solution to this is to let the linking at load time be "lazy."

In the "lazy" approach to dynamic linking, a `call` instruction jumps to a large table in memory that contains the actual address of any function that was dynamically linked. However, this incurs extra penalties in the cost of doing the `call` instruction the first time. To make the second execution of the `call` faster, the linker may rewrite the call to jump directly to the proper location, a technique known as **back-patching**.

A second issue that arises with dynamic linking concerns versioning. A programmer may wish to use a new function included with the most recent version of a library. If the programmer distributes the program without including this library, the end user may have an older version of that library without the important function. To get around this, the developer may wish to distribute the shared libraries with the application. But if every application does this, we are not much better off than with static linking.[4] Unix/Linux solves this with a file naming system that includes the version number, allowing a programmer to specify a particular version if it is needed. Windows does not have an elegant solution to this problem, which is

4 The developer can still fix bugs in either the main application or the library without having to distribute both files. However, this is somewhat minor compared to the advantage of having just one copy of a library on a system.

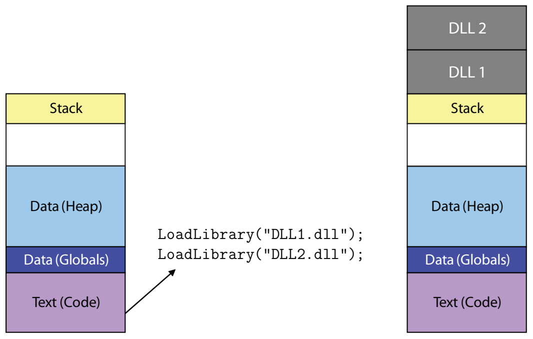


Figure 3.4: Dynamically loaded libraries are loaded programmatically and on-demand.

affectionately referred to as "DLL Hell."[5]

A third issue is related to security. A malicious programmer could name their own library the same as one the program expects to find on the system already. As long as it exports the same functions and data, they could write the library to do whatever they please. This seems to be a fatal flaw with dynamic linking. However, the problem is not related directly to linking but rather the security of the file system where the libraries are stored. Thus, the solution to this problem is to restrict the permissions of who can alter the location where shared libraries are stored on disk.

Dynamic Loading

Dynamic loading is a subset of dynamic linking where the linking occurs completely on demand, usually by a program's explicit request to the Operating System to load a library. Figure 3.4 shows an example of a Windows program making a request to load two DLLs programmatically. In Windows, a programmer can make a call to `LoadLibrary()` to ask for a particular library to be loaded by passing the name as a string parameter. Under Unix/Linux there is an analogous call, `dlopen()`.

One challenge for the Operating System in supporting dynamic loading is where to place the newly loaded code in memory. To handle this and traditional load-time linking, a portion of a process's address space (see Chapter 5) will typically be reserved for libraries.

Care has to be taken by the programmer to handle the error condition that arises

5 Recent versions of Windows protect core DLLs from being overwritten by older versions, but this does not solve 100% of all problem cases.

if the load fails. It is possible that a library has been deleted or not installed, and the code must be robust to not simply crash. This is one significant issue with dynamic loading, since with compile- or load-time linking the program will not compile or run without all of its dependencies available. A user will not be happy if they lose their work because in the middle of doing something the program terminates, saying that a necessary library was not found.

For this reason, core functionality of the program is probably best included using static or dynamic linking. Dynamic loading is often used for loading *plugins* such as support for additional audio or video formats in a media player or special effect plugins for an image editor. If the plugins are not present, the user may be inconvenienced, but they will likely still be able to use the program to do basic tasks.

3.2 Executable File Formats

After the linker runs as part of the compilation process, if there were no errors an executable file is created. The system has a format by which it expects the code and data of a program to be laid out on disk, which we call an **executable file format**.

Each system has its own file format, but the major ones that have been used are outlined here:

a.out (Assembler OUTput) — The oldest Unix format, but did not have adequate support for dynamic linking and thus is no longer commonly used.

COFF (Common Object File Format) — An older Unix format that is also no longer used, but forms the basis for some other executable file formats used today.

PE (Portable Executable) — The Windows executable format, which includes a COFF section as well as extensions to deal with dynamic linking and things like .NET code.

ELF (Executable and Linkable Format) — The modern Unix/Linux format.

Mach-O — The Mac OSX format, based on the Mach research kernel developed at CMU in the mid 1980s.

Though a.out is not used any longer,[6] it serves as a good example of what types of things an executable file might need to contain. The list below lists the seven sections of an a.out executable:

1. exec header
2. text segment
3. data segment
4. text relocations
5. data relocations
6. symbol table
7. string table

The *exec header* contains the size information of each of the other sections as a fixed-size chunk. If we were to define a structure in C to hold this header, it would look like Listing 3.1.

```
struct exec {
        unsigned long   a_midmag; //magic number
        unsigned long   a_text;
        unsigned long   a_data;
        unsigned long   a_bss;
        unsigned long   a_syms;
        unsigned long   a_entry;
        unsigned long   a_trsize;
        unsigned long   a_drsize;
};
```

Listing 3.1: The a.out header section.

The *magic number* is an identifying sequence of bytes that indicates the filetype. We saw something similar with ID3 tags, as they all began with the string "TAG". Word documents begin with "DOC". The loader knows to interpret this file as an a.out format executable by the value of this magic number.

The *text segment* contains the program's instructions and the *data segment* contains initialized static data such as the global variables. The header also contains

6 When using `gcc` without the `-o` option, you will notice it produces a file named a.out, but this file, somewhat confusingly, is actually in ELF format.

```
public class StringTable {
        public static void main(String[] args) {
                String s = "String␣literal";

                if(s == "String␣literal") {
                        System.out.println("Equal");
                }
        }
}
```

Listing 3.2: String literals are often deduplicated.

the size of the *BSS section* which tells the loader to reserve a portion of the address space for static data initialized to zero.[7] Since this data is initialized to zero, it does not take up space in the executable file, and thus only appears as a value in the header. The two relocation sections allow for the linker to update where external or relocatable symbols are defined (i.e., what addresses they live at).

The *symbol table* contains information about internal and external functions and data, including their names. Since the linker may need to look things up in this table, we want random access of the symbol table to be quick. The quickest way to look something up is to do so by index, as with an array. For this to work, however, we need each record to be a fixed size. Since strings can be variable length, we want some way to have fixed-sized records that contain variable-sized data. To accomplish this, we split the table into two parts. The symbol table with fixed-sized entries, and a *string table* that contains only the strings. Each record in the symbol table will contain a pointer to the appropriate string in the string table.

Java Content

The string table will also contain any string literals that appear in the program's source code. This is another example of deduplication. In Listing 3.2, we see a common beginner's mistake in Java. The == operator tests for equality, but when applied to objects the equality it tests for is that the two objects live at the same address. Obviously, we wanted to use the `.equals` method. But if we compile and run this, what would we see? The output is:

```
Equal
```

7 A reputable link on the meaning of bss states that it is "Block Started by Symbol." See `http://www.faqs.org/faqs/unix-faq/faq/part1/section-3.html`.

Why did we get the right answer? Java class files are executable files too. And to save both storage space and network bandwidth when transferred, Java deduplicates string literals during compilation. When the class file is loaded into memory, the string literal is only loaded once. Thus the references compare as equal since both are pointing to the same object in memory. One small change and it will break. If we change the initialization of `s` to:

```
String s = new String("String␣literal");
```

We are now constructing a new object in memory that will live in a different memory location and our comparison will fail.

This is not specific to Java. In C, we have similar concerns. If we declare a string variable as:

```
char s[] = "String␣literal";
```

we get a variable `s` that points to the string literal. It is unsafe (and generally a compiler warning) to modify the string literal by doing something like `s[0] = 's';`. This is prevented in Java because `String` objects are immutable.

3.3 Linking in Java

When a Java program is compiled, an executable file called a `.class` file is produced. This has the standard parts including code, data, string table, and references to other classes that it depends on. One interesting thing to note is that Java has no concept of static linking. All references to code that lives in other class files are resolved while the program runs (dynamic loading).

A natural thing to assume then would be that the `import` keyword indicates that you want to link against a particular package. However, this is not the case. The `import` keyword simply indicates to the compiler the fully qualified path to a particular object, such as `ArrayList` being in `java.util`. This is mostly a scope issue, but it does enable the compiler to produce the appropriate name for the imported class, which is then, in turn, used by the dynamic class loader to find it.

3.4 Terms and Definitions

The following terms were introduced or defined in this chapter:

Back-Patching Rewriting a jump to a dynamically loaded library so that it jumps directly to the code rather than via an intermediate table.

Compiler A tool for converting a high-level language (such as C) to a low-level language such as machine code.

Deduplication The elimination of duplication, often by storing something once and replacing each copy with a reference or link to it.

Executable File or simply "Executable" — A file containing the code and data necessary to run a program.

Header File A file containing function and data type definitions that refer to code outside of a given file.

Library Collections of code that accomplish common tasks provided by a compiler writer, system designer, or other third party.

Linker A tool for combining multiple sources of code and data into a single executable or program.

Loader The portion of the Operating System that is responsible for transferring code and data into memory.

Macro A simple or parameterized symbol that is expanded to some other text.

Object File A file containing machine code, but calls to functions that are outside of the particular source file it was generated from are unresolved.

Preprocessor The program responsible for expanding macros.

von Neumann Architecture A commonly-used computer architecture where the code and data must both be resident in main memory (RAM) in order to run a program.

4 | Function Pointers

Both code and data live in memory on a computer. We have seen how it is possible to refer to a piece of data both by name and by its address. We called a variable containing such an address a *pointer*. But in addition to being able to point to data, we can also create pointers to functions. A **function pointer** is a pointer that points to the address of loaded code. An example of function pointers is given in Listing 4.1.

This example simply displays "3" upon the console. The interesting thing to note is that there is no direct call to `f()` (there is no `f(…)` in the code), but there is a call to `g()` which seems to have no body. However, we do use `f()` as the right-hand side of an assignment to `g`. The odd declaration of `g` makes it look like a function prototype. However, with experience, it becomes apparent that it is a function pointer because of the fairly unique syntax of having the asterisk inside of the parentheses. The function pointer `g` is now pointing to the location where the function `f()` lives in memory. We can now call `f()` directly as we always could by saying `f(3)`, or we can dereference the pointer `g`.

Remember that with arrays, the name of an array is a pointer to the beginning of that array. There is no need to use the dereference operator (`*`) because the subscript notation (`[` and `]`) does the dereference automatically. The same is true for functions. The name of the function is a pointer to that function. We do not need to dereference it with a `*` because the `(` and `)` do it automatically. Thus as the argument to `printf()`, `g(3)` dereferences the pointer `g` to obtain the actual function `f()` and calls it with the parameter 3.

4.1 Function Pointer Declarations

The most complicated part of function pointers is their declaration. Care needs to be taken to distinguish the fact that we have a function pointer, rather than a

```
#include <stdio.h>

int f(int x) {
    return x;
}

int main() {
    int (*g)(int x);

    g = f;
    printf("%d\n",g(3));
    return 0;
}
```

Listing 4.1: Function pointer example.

function that returns a pointer. For example:

```
int *f();
```

means that we have a function `f()` that returns a pointer to an integer.

```
int (*f)();
```

means we have a function pointer that can point to any function that has an empty parameter list and returns an integer. The difference is in the parenthesization.

```
int *(*f)();
```

means we have a function pointer that can point to any function that has an empty parameter list and returns a pointer to an integer. If we forget the parentheses:

```
int **f();
```

we are declaring a function that takes no parameters but returns a pointer to an integer pointer.

Why do function pointer declarations need to be so hard? The answer lies in code generation. To correctly set up the arguments to the function, the compiler needs to know exactly how many are required and what size (indicated by the type) they are. This means that there is only one correct way to have declared `g` in Listing 4.1.

```
void qsort( void *base,
        size_t num,
        size_t size,
        int (*comparator)(const void *, const void *)
);
```

Listing 4.2: `qsort()` definition.

4.2 Function Pointer Use

Function pointers primarily get used in two particular ways. Both ways depend on the ability to defer knowledge of what function we are calling until the very moment we make the call. The first use, and the most common from a high-level programmer's perspective, is to pass a function pointer as an argument to another function. The second use is to make an array of function pointers to use as a jump or call table. This is a technique that can be used to accomplish dynamic linking.

4.2.1 Function Pointers as Parameters

The easiest way to explain the motivation for passing a function a pointer to another function is by example. Let us imagine we are writing a function to sort some data. While we are coding our algorithm, we find some line that requires us to do a comparison. If we are comparing the primitive data types, we can do comparison by simply using the $<$ or $>$ operators. But what about a more complex data type, such as a structure? Is there any way we could compare them? Additionally, if our function takes the array to sort as a parameter, what type do we define it as?

The C Standard Library includes a function called `qsort()` that seeks to be a generic sorting routine that anyone can call on any array, regardless of what type of data it contains. The declaration for `qsort()` is given in Listing 4.2.

The first parameter is of type `void *`, which means it is a pointer to anything. This solves our problem of how to declare the parameter but presents a new problem. We use typed pointers because C is able to determine the size of the data at the particular address from the size of the type. But a void pointer could be pointing to anything, and thus we need to do something extra to tell `qsort()` the size of each element. We pass that size as the third parameter. The second parameter is the length of the array we want to sort, since there is no way to query the length of an array from its pointer in C.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NUM_NAMES 5
#define MAX_LENGTH 10

int main() {
    char names[NUM_NAMES][MAX_LENGTH] = {
        "Mary","Bob", "Fred",
        "John", "Carl"};
    int i;

    qsort(names, NUM_NAMES, MAX_LENGTH, strcmp);

    for(i=0;i<NUM_NAMES; i++) {
        printf("%d:␣%s\t", i, names[i]);
    }
    return 0;
}
```

Listing 4.3: `qsort()`ing strings with `strcmp()`.

The final parameter looks complex but, based on our earlier discussion, it should be evident that this is a function pointer (the * inside the parentheses gives it away). The function we pass to `qsort()` should return an integer, and take two parameters that will point to two items in our array which this function is supposed to compare. The return values for `comparator` need to be handled as:

$$\texttt{comparator} = \begin{cases} < 0, & \text{if the first parameter is} < \text{the second} \\ 0, & \text{if they are equal} \\ > 0, & \text{if the first parameter is} > \text{the second} \end{cases}$$

This rule might remind us of the return values for `strcmp()`. In fact, `strcmp()` makes an obvious choice for sorting an array of strings. The only requirement is that all the strings need to be a fixed size for this to work. Since sorting requires swapping elements, `qsort()` must be told as a parameter the size of the elements in the array in order to rearrange the array elements correctly. Listing 4.3 gives an example. When we run this code, we get the expected output indicating a successful sort:

```
0: Bob    1: Carl    2: Fred    3: John    4: Mary
```

One interesting thing to note is that this code compiles with a warning:

```
(11) thot $ gcc qs.c
qs.c: In function `main':
qs.c:15: warning: passing arg 4 of `qsort' from incompatible pointer type
```

This message is telling us that there is something wrong about passing `strcmp()` to `qsort()`. If we look at the formal declaration of `strcmp()`:

```
int strcmp(const char *str1, const char *str2);
```

we see the parameters are declared as `char *` rather than the `void *` that the function pointer was declared as in Listing 4.2. This is one warning that is all right to ignore. In the old days of C, there was no special `void *` type, and a `char *` was used to point to any type when necessary.

To do something more complicated like sorting an array of structures, we will need to write our own comparator function. Listing 4.4 shows an example. The program sorts the structures first by age and then by name if there is a "tie." The output is the following:

```
18: Bob    18: Fred    20: John    20: Mary    21: Carl
```

4.2.2 Call/Jump Tables

A **call table**, or **jump table**, is basically an array of function pointers. It can be indexed by a variety of things, for instance, by a choice from a menu to decide what function to call. Compilers may generate such a table to implement `switch` statements.

Of specific interest to us is how a linker might use a call table to do dynamic linking. Each dynamically linked library exports some public functions. If we give these functions a number, referred to as an **ordinal**, we can use this ordinal to index a table created by the linker in order to find a specific function's implementation in memory.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NUM_STUDENTS 5

struct student {
    int age;
    char *name;
};

int mycmp(const void *a, const void *b) {
    struct student *s1 = (struct student *)a;
    struct student *s2 = (struct student *)b;

    if(s1->age < s2->age)
        return -1;
    else if(s1->age > s2->age)
        return 1;
    else
        return strcmp(s1->name, s2->name);
}

int main() {
    struct student s[NUM_STUDENTS] = {
        {20, "Mary"}, {18, "Bob"}, {18, "Fred"},
        {20, "John"}, {21, "Carl"}};
    int i;

    qsort(s, NUM_STUDENTS,
                sizeof(struct student), mycmp);

    for(i=0; i<NUM_STUDENTS; i++) {
        printf("%d:␣%s\n", s[i].age, s[i].name);
    }
    return 0;
}
```

Listing 4.4: `qsort()`ing structures with a custom comparator

4.3 Terms and Definitions

The following terms were introduced or defined in this chapter:

Call Table or Jump Table — A table of function pointers.

Function Pointer A pointer that points to code rather than data.

Ordinal A number used to refer to a particular function in a dynamically linked library.

5 | Processes & Address Spaces

After the loader has done its job, the program is now occupying space in memory. The program in memory is referred to as a **process**. A process is the Operating System's way of managing a running program and keeping it isolated from other processes on the system. Associated with each process is a range of legal addresses that the program may reference. These addresses form what is known as an **address space**. To make sure that a programmer does not interfere with any other running process (either by accident or maliciously), the Operating System needs to ensure that one program may not change the code or data of another without explicit permission.

One way to solve this problem of protection is to have the computer enforce strict limits on the range that an address in a process may take. At each instruction that references a memory address, that address is checked against this legal range to ensure that the instruction is only affecting the code or data in that process. However, this incurs a performance penalty since the CPU has to do extra checking every time there is a memory operation.

Modern Operating Systems take a different approach. Through a technique referred to as **Virtual Memory**, a process is given the illusion that it is the only one running on the computer. This means that its address space can be all of the memory the process can reference in the native machine word size. On a 32-bit machine with 32-bit pointers, a process can pretend to have all $2^{32} = 4$ gigabytes to itself. Of course, even the most expensive high-end computers do not have 4GB of memory per process of physical RAM, so the Operating System needs to work some magic to make this happen. It makes use of the hard disk, keeping unneeded portions of your program there until they need to be reloaded into physical memory.

Figure 5.1 shows a diagram of what a typical process's address space contains. The loader has placed all of the code and global variables into the low addresses. But this does not take up all of the space. We also need some dynamic storage

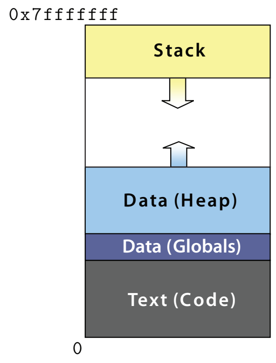


Figure 5.1: A process has code and data in its address space.

for function invocations and dynamically generated data. Functions will keep their associated storage in a dynamically managed section called the **stack**. Other dynamic data, which needs to have a lifetime beyond that of a single function call, will be placed in a structure called a **heap**.[1]

Notice that the stack and the heap grow toward each other. In a fixed-sized region, the best way to accommodate two variable sized regions is to have them expand toward each other from the ends. This makes sense for managing program memory as well, since programs that use a large amount of heap space likely will not use much of the stack, and vice versa. Chapter 6 and Chapter 7 will discuss the stack and the heap at more length.

5.1 Pages

Memory management by the Operating System is done at a chunk size known as a **page**. A page's size depends on the particulars of a system, but a common size is 4 kilobytes. The Operating System looks at what chunks you are using and those that you do not have allocated. It is clear that smaller programs will have large chunks of unallocated space in the region between the stack and the heap. These unallocated pages do not need to take up physical memory.

1 Unfortunately, the term "heap" has two unrelated meanings in Computer Science. In this context, we mean the portion of the address space managed for dynamic data. It can also refer to a particular data structure that maintains sorted order between inserts and deletes, and is commonly used to implement a priority queue.

Since the Operating System manages the pages of the system, it can do some tricks to make dynamic libraries even more convenient. We already motivated dynamic libraries by saying they can save disk storage space by keeping common code in only one place. However, from our picture of process loading, it would appear that every shared library is copied into every process's address space, wasting RAM by having multiple copies of code loaded into memory. Since physical memory is an even more precious resource than disk storage, this seems to be less of a benefit than we initially thought. The good news is that the Operating System can share the pages in which the libraries are loaded with every process that needs to access them. This way, only one copy of the library's code will be loaded in physical memory, but every program can use the code.

5.2 Terms and Definitions

The following terms were introduced or defined in this chapter:

Address Space The region of memory associated with a process for its code and data.

Page The unit of allocation and management at the Operating System level.

Process A program in memory.

Virtual Memory The mechanism by which a process appears to have the memory of the computer to itself.

6 Stacks, Calling Conventions, & Activation Records

The first type of dynamic data we will deal with is local variables. Local variables are associated with function invocations, and since the compiler does not necessarily know how many times a function might be called in a program, there is no way to predict statically how much memory a program needs. As such, a portion of memory needs to be dedicated to holding the local variables and other data associated with function calls. Data in this region should ideally be created on a function's call and destroyed at a function's return. If we look at the effect of having a function call other functions, we see that the local variables of the most recent function call are the ones that are most important. In other words, the local variables created last are used, and are the first ones to be destroyed; those created earlier can be ignored. This behavior is reminiscent of a stack.

To create a stack we need some dedicated storage space and a means to indicate where the top of the stack lives. There is a large amount of unused memory in the address space after loading the code and global data. That unused space can be used for the stack. Since practically every program will be written with functions and local variables, the architecture will usually have a register, called the **stack pointer**, dedicated to storing the top of the stack.

With storage and a stack pointer, we can make great strides in managing the dynamic memory needs of functions. When a function is compiled, the compiler can figure out how many bytes are needed to store all of the local variables in that function and then write an instruction that adjusts the stack pointer by that much on every function call. When we want to deallocate that memory on function return, we could adjust the stack pointer in the opposite direction.

Other than local variables, what information might we want to store on the stack? Since the concept of a stack is so intrinsically linked with function calls, it

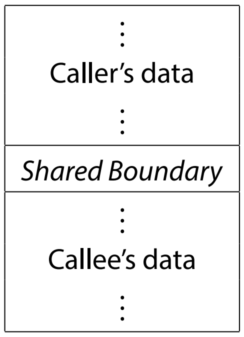


Figure 6.1: The shared boundary between two functions is a logical place to set up parameters.

seems to make sense that the **return address** of the function should be stored on the stack as well. If we examine the caller/callee relationship, we see that their data will be in adjacent locations on the stack. Figure 6.1 shows the two stack entries and the boundary between them. If the caller function needs to set up arguments for the callee, this boundary seems a natural place to pass them.

On machines with many registers, some registers may be designated for temporary values in the computation of complicated expressions. These temporary registers may be free for any function to use, and thus, if a function wants a particular register to be saved across an intervening call to another function, the calling function must save it on the stack. This is referred to as a **caller-saved** register. Other registers may be counted upon to retain their values across intervening function calls, and if a called function wants to use them, it is responsible for saving it on the stack and restoring them before the function returns. These are **callee-saved** registers. In general, the stack is used to save any architectural state that needs to be preserved.

We now have the following pieces of data that need to be on the stack:

1. The local variables — including temporary storage, such as for saving registers

2. The return address

3. The parameters

All of these together will form a function's **activation record** (sometimes called a **frame**). Not every system will have all of these as part of an activation record. How

	MIPS	x86
Arguments:	First 4 in `%a0`–`%a3`, remainder on stack	Generally all on stack
Return values:	`%v0`–`%v1`	`%eax`
Caller-saved registers:	`%t0`–`%t9`	`%eax`, `%ecx`, & `%edx`
Callee-saved registers:	`%s0`–`%s9`	Usually none

Figure 6.2: A comparison of the calling conventions of MIPS and x86

parameters are passed, be it on the stack or in registers, is part of a "contract" the system abides by, called the **calling convention**.

6.1 Calling Convention

For one function to call another, they must first agree on the particulars of where parameters will be passed and where return values will be stored. Each system will have a different calling convention depending on its particular details, such as the number of general purpose registers. Figure 6.2 summarizes two different architectures' calling conventions.

Beyond an architecture or system's designers specifying the details of a calling convention, individual programming languages may specify how some data is to be exchanged. Let us start with a look at a C program written for an `x86` processor running Linux and the assembler output generated by `gcc`. Figure 6.3 shows the source and output for a `main()` function that calls a function `f()` with one parameter.

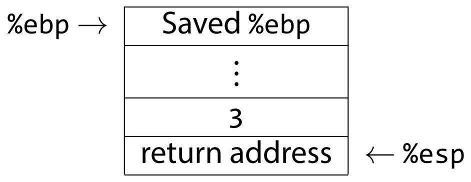
If we begin to trace the first five instructions of the `main()` function, we notice that they are all related to the management of the two stack-related registers, `%esp`, the stack pointer, and `%ebp`, the base pointer. In this context, `%ebp` is being used to keep track of the beginning of the activation record, and is sometimes referred to as the **frame pointer**. The `andl $-16, %esp` instruction is a way to do data **alignment**. On many architectures, it is faster to access data that begins at addresses that are multiples of some power of two because of memory fetches and caches. The `subl $16, %esp` allocates some memory on the stack for the parameter to `f()`. This is a subtract because the stack starts from a high address and grows towards lower addresses. All figures in this book have been drawn with that detail in mind. Thus, a `push` is a subtract and a `pop` is an add. Whenever a function call is made,

```c
#include <stdio.h>

int f(int x) {
    return x;
}

int main() {
    int y;

    y = f(3);

    return 0;
}
```

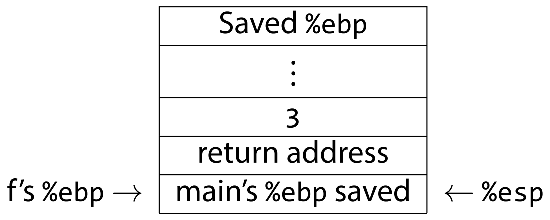
```
f:      pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        leave
        ret
main:   pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $3, (%esp)
        call    f
        movl    %eax, -4(%ebp)
        movl    $0, %eax
        leave
        ret
```

Figure 6.3: A function with one parameter.

the `call` instruction pushes the return address onto the top of the stack and then jumps to the subroutine. Below is a diagram of the current state of the stack (the dots indicate extra space left unused due to alignment).



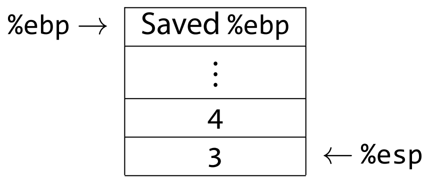
When we enter into `f()`, we again set up an activation record by first saving `main()`'s frame pointer and then adjusting the frame pointer to point to the bottom of `f()`'s activation record.



At this point, `f()` wants the value of its parameter, which is stored above the current base pointer. The instruction `movl 8(%ebp), %eax` accesses it, which is two words (8 bytes) away. There is no need for any calculation in this simple function, and the return value is directly placed into `%eax`. The `leave` instruction restores `main()`'s frame by popping the activation record, setting `%esp` to `%ebp`, and then restoring `%ebp` to the old value. The `ret` instruction pops the return address off the stack and returns back to `main()`.

While a lot is going on in these few instructions, everything seems fairly straightforward. An activation record is created for `main()`, the parameter is placed on the stack, and the function `f()` is called, which sets up its own frame and does the work. Deallocating the frame is simple due to the `leave` and `ret` instructions.

Let us now look at the code in Figure 6.4, which illustrates a function that takes two parameters. Much the same is going on in Figure 6.3; the only difference is we push two parameters instead of one. The stack looks like:



```c
#include <stdio.h>

int f(int x, int y) {
        return x+y;
}

int main() {
        int y;

        y = f(3, 4);

        return 0;
}
```

```
f:      pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        leave
        ret
main:   pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $4, 4(%esp)
        movl    $3, (%esp)
        call    f
        movl    %eax, 4(%esp)
        movl    $0, %eax
        leave
        ret
```

Figure 6.4: A function with two parameters.

when we make the call to `f()`.

Something that is peculiar here is that the two parameters seem somehow backwards. They have been pushed in reverse order from the way they are written. This particular quirk in the C calling convention is introduced entirely for the support of one very common function:

```
int printf(const char *format,...);
```

The `printf()` function takes a variable number of arguments. As such, there is no way for it to know how far down the stack to look for data. The only way it might be able to know is to look inside the format string, which is what it does. The number of scan codes inside the format string tells `printf()` how many arguments to expect. Because of this, you must always be sure to pass as many arguments as you have scan codes, or `printf()` might start printing other values off the stack. By pushing the arguments in reverse order, the closest argument to `%ebp` is the format string, always at a fixed offset in a predictable location. This is why the C calling convention pushes arguments in reverse order.

6.2 Variadic Functions

A **variadic function** is a function that takes a variable number of arguments. In C, a variadic function declaration is easy to recognize due to the ellipses (...) in the function declaration. The `stdarg.h` header file provides several macros to help deal with the parameter list. Listing 6.1 shows an example of a variadic function that turns its parameter list into an array. The `va_start` macro takes the last required parameter and initializes the `va_list` variable `ap`. The `va_arg` macro facilitates advancing the pointer to walk the stack by the size of the appropriate type (which must be known by the function). It casts the data at that address back to the appropriate type and advances the pointer automatically.

Java Content

Java and other languages with built-in support for dynamic arrays often expose the parameters of a variadic function as an array. For example:

```
public static void printArray(Object ... objects) {
        for (Object o : objects)
                System.out.println(o);
}

printArray(3, 4, "abc");
```

displays its parameters on the screen, one per line.

```c
#include <stdio.h>
#include <stdarg.h>

int *makearray(int a, ...) {
    va_list ap;
    int *array = (int *)malloc(MAXSIZE*sizeof(int));
    int argno = 0;
    va_start(ap, a);
    while (a > 0 && argno < MAXSIZE)
    {
        array[argno++] = a;
        a = va_arg(ap, int);
    }
    array[argno] = -1;
    va_end(ap);
    return array;
}

int main() {
    int *p;
    int i;
    p = makearray(1,2,3,4,-1);

    for(i=0;i<5;i++)
        printf("%d\n", p[i]);

    return 0;
}
```

Listing 6.1: A variadic function to turn its arguments into an array.

```
void f(char *s) {
    gets(s);
}

int main() {
        char input[30];

        f(input);
}
```

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $40, %esp
    andl    $-16, %esp
    subl    $16, %esp
    leal    -40(%ebp), %eax
    movl    %eax, (%esp)
    call    f
    leave
    ret
```

Figure 6.5: A program with a buffer overrun vulnerability.

6.3 Buffer Overrun Vulnerabilities

The code in Figure 6.5 has a problem. The code allocates an array of thirty characters, then calls `gets()` to ask the user to enter some input. The `gets()` function takes only one parameter, and from our knowledge of C, we know that there is no way to tell from that one pointer parameter just how big the array is that was passed. This means that a malicious person could enter something much larger than 30 characters. Since `gets()` has no idea when to stop copying input into our array, it keeps going. As the assembly listing shows, the array was allocated on the stack. When `gets()` exceeds that space, it starts writing over the rest of the data on the stack, corrupting it.

If our malicious user notices that the program crashes with long input, he or she may suspect that the input is overwriting the activation record on the stack. By carefully crafting the input string, the user can deliberately overwrite specific offsets on the stack, including the return address of `gets()`. With the return address modified, when `gets()` returns, code will start executing at whatever location the user entered.

With some more careful crafting of the input string, the return address can be modified so that it points to the location in memory where the array is stored. Cleverly, the user can place machine code as the input, and when `gets()` returns, it jumps via the return address to the code that the attacker entered. If this program had been a system service, running as a superuser on the machine, the attacker could have injected code to make themselves a superuser too. This type of attack is known as a **buffer-overrun vulnerability**.

Great care must always be taken with arrays on the stack. Overrunning the end of the buffer means writing over activation records, and possibly subjecting the system to an attack. *Always check that the destination of a copy is large enough to hold the source!*

6.4 Terms and Definitions

The following terms were introduced or defined in this chapter:

Activation Record The local variables, parameters, and return address associated with a function's invocation.

Alignment Making sure that data starts at a particular address for faster access due to memory fetches or the cache.

Callee-Saved A piece of data (e.g., a register) that must be saved by a called function before it is modified and restored to its original value before the function returns.

Caller-Saved A piece of data (e.g., a register) that must be explicitly saved if it needs to be preserved across a function call.

Calling Convention An agreement, usually created by a system's designers, on how function calls should be implemented — specifically regarding the use of registers and the stack.

Frame Another name for an activation record.

Frame Pointer A pointer, usually in a register, that stores the address of the beginning of an activation record (frame).

Return Address The address of the next instruction to execute after a function call returns.

Stack A portion of memory managed in a last-in, first-out (LIFO) fashion.

Stack Pointer The architectural register that holds the top of the stack.

7 | Dynamic Memory Allocation Management

While the stack is useful for maintaining data necessary to support function calls, programs also may want to perform dynamic data allocation. Dynamic allocation is necessary for data that has an unknown size at compile-time, an unknown number at compile-time, and/or whose lifetime must extend beyond that of the function that creates it. The remaining portion of our address space is devoted to the storage of this type of dynamic data, in a region called the **heap**.

As is often the case, there are many ways to track and manage the allocation of memory. There are trade-offs between ease of allocation and deallocation, whether it is done manually or it is automatic, and the speed and memory efficiency need to be considered. Also as usual, the answer to which approach is best depends on many factors.

This chapter starts by describing the major approaches to allocation and deallocation. We first describe the two major ways to track memory allocation. The first is a **bitmap** — an array of bits, one per allocated chunk of memory — that indicates whether or not the corresponding chunk has been allocated. The second management data structure is a **linked list** that stores contiguous regions of free or allocated memory. A third technique, the **Buddy Allocator** attempts to reduce wasted space from many allocations. The chapter also describes an example implementation of `malloc()`, the C Standard Library mechanism for dynamic memory allocation.

7.1 Allocation

The two operations we will be concerned with are *allocation*, the request for memory of a particular size, and *deallocation*, returning allocated memory back to the system for subsequent allocations to use. The use of the stack for function calls led us to create activation records upon function invocation and to remove them from the stack on function return. In essence, we were allocating and deallocating the activation records at runtime — the very operations we are attempting to define for the heap.

The question is then, why is the stack insufficient and what is different about the heap? As the name implies, the stack is managed as a FIFO with allocation corresponding to a push operation and deallocation corresponding to pop. This worked for function calls because the most recently called function, the one whose activation record was allocated most recently and lives at the top of the stack, is the one that returns first. Deallocations always occur in the opposite order from the allocations. New allocations always occur at the top of the stack, and with the stack growing from higher addresses to lower ones by convention, this means that all space above the stack pointer is in-use. All space at lower addresses is free or not part of the stack.

Thus, allocation is simply moving the top of the stack, and deallocation is moving it back. But for objects whose lifetime is not limited to the activation of a particular function, this order requirement is too restrictive. We would like to be able to allocate objects A, B, and C, and then deallocate object B. This leaves an unallocated region in the middle that we may wish to reuse to allocate object D.

In this section, we are considering this more general case of allocation: the possibility that we have free space in between allocated spaces. We need to track that space and to allocate from it. With that in mind, the simple dividing line between free and used space that the stack pointer represented is insufficient and we need to use a more flexible scheme.

7.1.1 Allocation Tracking Using Bitmaps

What we wish to do is to ask for an arbitrary piece of memory, is this space in use or is it free? At the heart of it, this question is answered via a single boolean value that is mutually exclusive: yes, it is being used or no, it is free. That information can be stored in a single bit per piece of memory. For an entire region of memory, we can combine all of the individual bits into a single array of bits called a **bitmap**.

When a modern computer user thinks of the term bitmap, he or she invariably recalls the image format. This is appropriate, since if we were to create a format for storing black and white images, we might devote one bit per pixel and treat the image as a large array of pixels. The same concept applies to managing memory

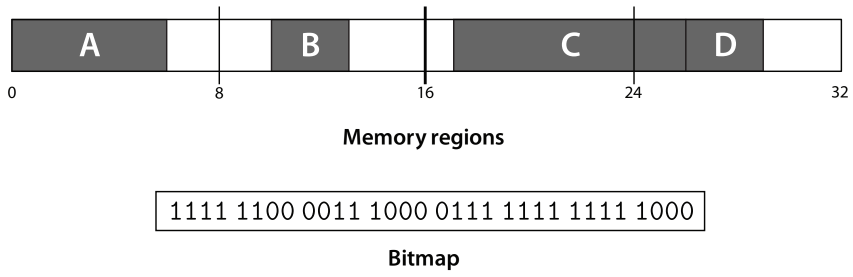


Figure 7.1: A bitmap can store whether a chunk of memory is allocated or free.

using bitmaps. We store an array of bits where a `0` indicates an unallocated chunk of memory, and a `1` indicates a chunk has been allocated for some purpose. Figure 7.1 shows a region of 32 chunks. Several parts are allocated to A, B, C, and D, with the remainder of the chunks free. Below it is the corresponding bitmap.

Bitmaps have several significant disadvantages that make them undesirable for common use. The first major drawback is the space requirement. A part of the region of memory that we are managing will have to be devoted to storing the bitmap itself. If the unit of allocation is very small, for instance a single byte, the bitmap will need to be large (one bit per byte). This means that for every eight bytes we will need one more byte for the bitmap. One out of nine bytes (11%) will be wasted in management overhead.

To reduce the size of the bitmap then, we could try to reduce the number of bits necessary by having one bit represent more space. This will result in increasing the smallest unit we can allocate. Instead of each bit tracking a single byte being allocated or free, each bit will represent a contiguous chunk of memory. Now, for a memory region of size S and a chunk size of size c, we will only need $\frac{S}{c}$ bits.

While this will reduce the size of the bitmap, a new problem arises: It is impossible to allocate less than a single chunk, since a single bit cannot represent partial allocations — only whole ones. This leads to the problem of **internal fragmentation**, which is wasted space due to an allocation unit being bigger than our need. When we allocate a contiguous region of memory, on average the very last chunk will be half full. This could result in a large amount of wasted space over the lifetime of a program.

A second disadvantage to using bitmaps is the difficulty in finding a large enough free space to hold a given allocation. The challenge lies in how to discern that there are the right number of zeros in a row. If the empty space is at the end, the search

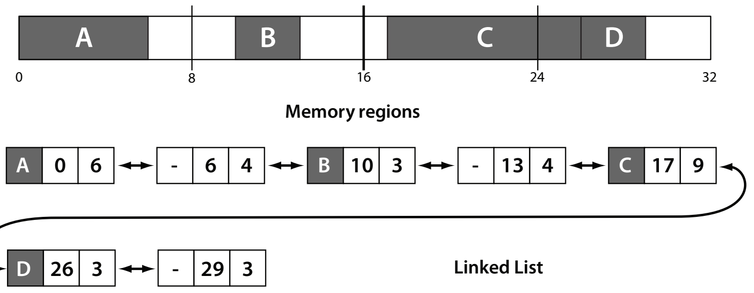


Figure 7.2: A linked list can store allocated and unallocated regions.

may be slow. Practically, it would involve a lot of bit shifting and masking. As such, using bitmaps for any significant tracking of dynamic memory allocations is unlikely, but bitmaps do often find a use in tracking disk space allocation.

7.1.2 Allocation Tracking Using Linked Lists

The potentially huge size of a bitmap relative to the memory region it was managing drove us to chunk memory and introduce internal fragmentation. However, there are two properties of a bitmap that might allow us to reduce its overall size while avoiding the necessity of chunks. One observation that we may make is that frequently the bitmap may contain many zeros. This would be the case when the region is new and there haven't been many allocations, when there have been a lot of deallocations, or when a region is simply much larger than the current dynamic memory needs. Noticing this, we can take a page from matrices. When a matrix has mostly zero entries, we call it **sparse**. In the CS world, we find that sparseness can apply to various data structures including our bitmap. A space-saving idea with a sparse data structure is to only store the elements that are nonzero. A linked list is a sparse data structure that supports storing a variable number of elements with dynamic inserts and deletes. We can omit the zero entries and infer that anything not in the linked list is free space.

While the choice to use a sparse data structure is likely a good one, there are two issues that prevent us from stopping here. The first is that sparseness is good for size, but we need the linked list to easily support fast allocations that come from the free space. If the information about unallocated space is not directly stored in the linked list, we must infer the necessary space is available and of the proper size. This motivates having additional linked list nodes that represent free space, but in doing so, we have lost the sparseness that made a multi-byte linked list node a reasonable thing to store over a single bit in a bitmap.

The solution here is that our observation of sparseness in the bitmap was accurate, but it did not go far enough in describing the situation. Not only are many of the bitmap entries zeros, those zeros are also likely to be next to other zeros in long contiguous *runs*. When we have an allocation, that also exists in the bitmap as a run of ones. A key observation is that the number of ones or zeros in a particular run is actually an encoding of that regions's length, albeit in unary (base 1). Unary turns out to be the worst of all bases in terms of compactness because to represent the number n, you need a string of length n. If we instead consider base two (binary), the length of a string of bits needed to represent the value n is only $\log_2(n)$.

This logarithmic rather than linear growth gives us a better scheme for storing the size of a particular allocation. Instead of denoting a size as a run of n ones or zeros in a bitmap, we could simply store the size in a normal variable in memory. A 32-bit integer would only be enough space to store information about 32 chunks when used as a bitmap, but because of the slow growth of logarithms, those same 32 bits can store a value of 2^{32}, or about four billion! Even if we decided not to use chunks but track memory at the byte granularity, an often unreasonable choice for a bitmap, we could still represent allocations of up to 4GB in size.

The removal of runs is a simple form of compression known as **run-length encoding** or RLE. Using RLE to compress allocations and free spaces allows us to have one linked list node per allocated or free contiguous region. Figure 7.2 shows the same region and allocations as in Figure 7.1, but with the nodes of a linked list corresponding to the free and allocated spaces. The first number is the starting index of the contiguous space; the second is the size of the allocation.

Like a bitmap, the linked list needs to be stored in the same memory it is managing. With the bitmap, since the size is known ahead of time (it is simply the size of the region divided by the size of the chunk), space can be reserved before any allocations are done. The bitmap will not grow or shrink as long as the region and chunk do not change size. The linked list, on the other hand, has a number of nodes that is proportional to the number of allocations and free spaces. This number changes as the region is used.

To store the linked list, we could reserve the worst-case size in advance, much as we did with the bitmap. The worst case length of the linked list would occur when there is one node per minimum unit of allocation. This could occur for a number of different scenarios, such as with a full region of individual unit-size allocations or where unit allocations are separated by unit-sized free spaces. In this case, we would have n list nodes just as we had n bits in the bitmap. However, while each entry in the bitmap only required a single bit's worth of storage, how large might a list node be? We need to store the size, the start, and links for the linked list. For faster deallocation support, we probably want this to be a doubly-linked list, requiring us to have two node pointers. Assuming all of these fields are four bytes in size, we would need $4 \times 4 = 16$ bytes or $16 \times 8 = 128$ bits. Thus, to reserve space for the worst case scenario, we would need $128n$ bits where the bitmap only needed n bits. The linked list is 128 times the size!

This is horrible and we may wonder how we began by trying to reduce the size of a data structure but ended up making it 128 times worse. The answer is in the worst case scenarios. They were the worse case because they eliminated the runs that were the bases of our compression. When our assumptions are not valid, our end result is likely to come out worse. The good news is that our assumptions are valid in the *typical* case. Such degenerate linked lists are not likely to result from the normal use of dynamic memory.

While we have convinced ourselves the linked list is still a valid approach, we still need a good solution for where to store the elements of the linked list. Reserving the space in advance is not feasible. A better solution might be to think of the memory-tracking data structure as a "tax" on the region of memory we are tracking. For a bitmap, we pay a fixed-rate tax off the top — before we have even used the region. For paying that tax, we never have to pay again. For the linked list however, we could instead pay a tax on each allocation. Every time we get a request for dynamic memory, we could allocate a bit extra to store the newly-required list node. For instance, if we get a request for 100 bytes, we actually allocate 116 bytes and use the additional space to store one of the nodes we described above.

7.1.3 Allocation Algorithms

Searches through the linked list are necessary to find a region to satisfy an allocation request, but the integer size comparisons are easier for the computer compared to the bit matching needed for bitmaps. Whichever technique we use, when an allocation request is made there may be many free spots that could accommodate the request. Which one should we choose? Below is a list of various algorithms from which we could pick:

First fit Find the first free block, starting from the beginning, that can accommodate the request.

Next fit Find the first free block that can accommodate the request, starting where the last search left off, wrapping back to the beginning if necessary.

Best fit Find the free block that is closest in size to the request.

Worst fit Find the free block with the most left over after fulfilling the allocation request.

Quick fit Keep several lists of free blocks of common sizes, allocate from the list that nearest matches the request.

First fit is the simplest of the algorithms, but suffers from unnecessary repeated traversals of the linked list. Each time first fit runs, it starts at the beginning of the list and must search over space that is unlikely to have any free spaces due to having allocated from the beginning each prior time the function was called. To avoid this cost, we could remember where the last allocation happened and start the search from there. This modification of first fit is called **next fit**.

Both of these algorithms take the first free block they find, which may not be ideal. This strategy may leave uselessly small blocks or prevent a later request from being fulfilled because a large free block was split when a smaller free spot elsewhere in the list might have been a better fit. This wasted space between allocations is **external fragmentation**.

To avoid external fragmentation, we may wish to search for the best fit. The **best fit** algorithm searches the entire list looking for the free space that is closest in size to the request. This means that we will never stop a large future request from being fulfilled because we took a large block and split it unnecessarily. However, this algorithm can turn out to be poor in actual usage because we end up with many uselessly small leftovers when the free space is just slightly larger than the request. This is guaranteed to be as small as possible whenever an exact fit is not found, due to the difference between the free space and the allocation being minimized by our definition of "best". Additionally, best fit is slow because we must go through the entire linked list, unless we are lucky enough to find a perfect fit.

To avoid having many small pieces remain, we could do the exact opposite from best fit, and find the **worst fit** for a request. This should leave a free chunk after allocation that remains usefully large. As with best fit, the entire list must be searched to find the worst fit, resulting in poor runtime performance. Unlike best fit, which could stop early upon finding a perfect fit, the worst fit cannot be known without examining every free chunk. Despite our intuition, simulation of this algorithm reveals that it is not very good in practice. An insight into why is that after several allocations, all of the free chunks are around the same, small size. This is bad for big requests and makes looking through the whole list useless as every free chunk is about equal size.

An alternative to the search-based algorithms, **quick fit** acknowledges that most allocations come clustered in certain sizes. To support these common sizes, quick fit uses several lists of free spaces, with each list containing blocks of a predetermined size. When an allocation request is made, quick fit looks at the list most appropriate for the request. Performance is good because searching is eliminated: With a fixed number of lists, determining the right list takes constant time. Leftover space (internal fragmentation) can be bounded since an appropriately-sized piece of memory is allocated. If the lists were selected to match the needs of the program making the allocation, this would leave very little wasted space. Additionally, that leftover space should not harm future large requests because the large requests would be fulfilled from a different list. One issue with quick fit is the question of whether or not to coalesce free nodes on deallocation or to simply return them to their appropriate list. One solution is to provide a configurable parameter to the allocator that says how many adjacent small free nodes are allowed to exist before they are collapsed into one. This ensures large unallocated regions as well as enough of the more common smaller regions.

The two likely “winners” of the allocation battle are next fit and quick fit. They both avoid searching the entire list yet manage to fulfill requests and mostly avoid fragmentation. The GNU `glibc` implementation of `malloc()` uses a hybrid approach that combines a quick fit scheme with best fit. The writers claim that while it is not the theoretically best performing `malloc()`, it is consistently good in practice.

7.2 Deallocation

The other important operation to consider is deallocation. This is where the true distinction against stack allocation is drawn. Whenever we free space on the stack, we reclaim only the most recently allocated data. The heap has no such organization, and thus deallocations may occur regardless of the original allocation order. Since the stack is completely full from bottom to top, the only bookkeeping necessary is an architectural register to store the location of the top. The heap, on the other hand, will inevitably have “holes” — free spaces from past deallocations — that will arise. Keeping track of the locations of these holes motivates the use of a data structure such as a bitmap or linked list. In this section, we look at the various approaches a

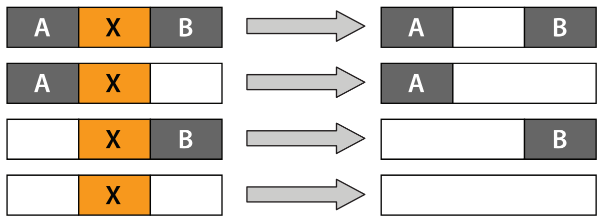


Figure 7.3: Coalescing free nodes on deallocation.

system might take to deallocation.

7.2.1 Using Linked Lists

When we allocate some memory, our linked list changes. The free node is split into two parts: the newly allocated part and the leftover free part. Eventually deallocations happen, and it is time to release a once-used region of memory. Figure 7.3 shows the four scenarios that we might find when doing a free operation. The topmost shows a region being deallocated (indicated with an 'X') that has two allocated neighboring regions; in this case, we simply mark the middle region as free. The second and third cases show when the left or right neighbor is free. In this case, we want to **coalesce** the free nodes into a single node so that we may later allocate this as one large contiguous region. The final case shows both of our neighbors being free, and thus we will need to coalesce them all.

To facilitate coalescing nodes, we may want to use a doubly linked list, which has pointers to the next node as well as the previous node. Note that we do not want to coalesce allocated nodes because we would like to be able to search the linked list for a particular allocation by name (or address, if that is what we are storing as the "name").

7.2.2 Using Bitmaps

For all of the flaws of using bitmaps for dynamic memory management, deallocation of a bitmap-managed region is surprisingly simple: the appropriate bits switch from `1` (allocated) to `0` (deallocated). The beauty of this approach is that free regions are coalesced without any explicit effort. The newly freed region's zeros naturally "melt" into any neighboring free space.

7.2.3 Garbage Collection

Up until this point, we have been assuming that requests for deallocation have come directly from the user. Forgetting to explicitly deallocate space can lead to **memory leaks**, where a dynamically allocated region cannot be explicitly freed because all pointers to it have either been overwritten or gone out of scope. It requires the diligence of the programmer to avoid leaking memory, a tedious task that might seem to lend itself to being automated.

For deallocation to be done automatically, the system needs to know that a chunk of dynamically allocated memory is no longer used. A clue to how this might be determined is in the previous paragraph. If we have leaked the memory — that is, we have no valid pointers to it — then we can reclaim that space. The system is now faced with a fundamental problem: If there are no pointers to a region, how does the system itself (in Java's case, the Virtual Machine) find it? There are several approaches that might work. The JVM could keep an internal list of pointers to every object that is allocated. Another alternative is to walk the stack looking for object references.

Once all of the data items that are unreachable are discovered, the task of freeing their space, called **garbage collection**, starts. Since it takes time to find all of the "garbage," the collection process is not usually on-demand in the same way that `free()` works in C. Except for the so-called *concurrent collectors*, garbage collectors run only when necessary — usually when the amount of free heap space has dropped below some threshold. When this threshold is hit, the program generally pauses and garbage collection begins. While there is a vast array of techniques by which to free used space, we will discuss three common strategies: *reference counting*, *in-place collectors*, and *copying collectors*.

Reference Counting

We have already determined that a dynamically-allocated object is garbage and can be collected when it has been leaked and there are no longer any valid references to the memory. Possibly the simplest way to determine this is to count valid links to an object and, when the count reaches zero, automatically free the memory. This strategy is known as reference counting and can be implemented relatively easily even in native code.

Each object needs a reference count variable associated with it. This variable is incremented or decremented as the program runs. It will need to be updated:

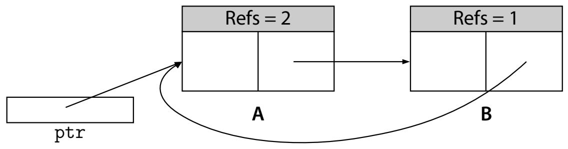


Figure 7.4: Reference counting can lead to memory leaks. If the pointer `ptr` goes out of scope, the circularly linked list should be collected. However, each object retains one pointer to the other, leading to neither having the requisite zero reference count for deallocation.

1. When a reference goes out of scope.

2. When a reference is copied (explicitly with assignment or implicitly in parameter passing, for example).

When a reference goes out of scope, the reference count on the associated object must be decremented. Copying references affects both sides of the assignment. The left-hand side (often called an *l-value*) might have been referring to an object prior to the assignment. This reference is now going to be lost from the overwrite, so the original object's reference count must be decremented. The right-hand side of the assignment (predictably called the *r-value*) is now going to have one more reference to it and the associated counter must be incremented accordingly.

When an object's reference count reaches zero, the object is garbage and can be collected. This might happen while the program is running (making it a concurrent collector) or at periodic breaks in the program's execution (a stop-the-world collector). The act of garbage collection can be as easy as freeing the object with whatever heap management operation is available.

A problem that can arise in a reference counting garbage collector is, remarkably, that it can leak memory. If a data structure has a cycle, such as in a circularly linked list as shown in Figure 7.4, there can be no way to collect the data structure. With a cycle, there is at least one reference to each object that remains even after all references from the program code are gone. Since the reference count never reaches zero, the objects are not freed. Possible solutions to this problem include detecting that the objects are part of a cycle or by using one of the other garbage collection algorithms.

In-place Collectors

Another approach to garbage collection is via an in-place collector. The process is comprised of two phases: a *mark* phase and a *sweep* phase. During the mark phase, all of the references found on the stack are followed to find the objects to which they refer. Those objects may contain references themselves. As the algorithm traverses this graph of references, it marks each object it encounters as reachable, thus indicating it is not to be collected. When every reference that can be reached has its associated object marked, the algorithm switches to the sweep phase.

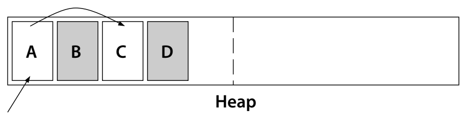
In the sweep phase, all unmarked objects are freed from the heap. All that remains are the reachable objects. When the deallocation is finished, all of the marked objects are reset to unmarked so that the process may begin all over again when the garbage collector is invoked the next time.

This *mark and sweep* approach is simple and relatively fast. It avoids cycles because encountering an object we have already seen can be detected as the object will be already marked as seen. It suffers from a significant problem, however. The newly freed space might be between objects that are still alive and remain in the heap. We now have holes that are small and scattered throughout memory, rather than a big free contiguous chunk of memory from which to allocate new objects. While there might be a significant fraction of space that is free, it might be fragmented to the point of being unusable. This is once again, external fragmentation, and was the motivation behind coalescing the adjacent free nodes in a linked list management scheme.

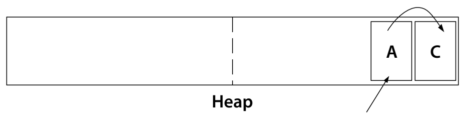
Copying Collectors

To fix the fragmentation issue of the in-place collector, a garbage collector could compact the region by moving all of the objects closer together. This would constitute a third *compaction* phase and is actually unnecessary. We can avoid a third phase by combining deallocation and compaction into a single pass through the heap.

Copying garbage collectors such as the *semispace* collector typically divide the heap into two halves and copy from the full half into the reserved, empty half. Figure 7.5 shows an example. In Figure 7.5a, objects B and D have been designated unreachable and should be freed. Rather than explicitly do this freeing and be left with two small holes in memory, objects B and D are left untouched. Objects A and C are referenced and thus alive. A copying collector will move these live objects to the reserved half of the heap, placing them contiguously to avoid wasting space.



(a) Objects B and D are unreachable and the heap is nearly half full.



(b) Objects A and C are moved to the reserved half, and the original half is marked as free.

Figure 7.5: Copying garbage collectors divide the heap in half and move the in-use data to the reserved half, which has been left empty.

Figure 7.5b shows the resultant heap.

While copying all the live objects seems like it should be more expensive, the reduction of fragmentation in the free space usually negates the cost of copying. An allocation that has to search through a list or bitmap to find space is slow, whereas a contiguous region is simple to dole out.

To accommodate a copying garbage collector, some restrictions on references need to be made. Since the addresses that data items live at can change during the execution of the program, there can be no hard-coded addresses. Additionally, references must be kept distinct from integer types since when the system moves an object, all references must be able to be found before they can be properly updated. Indirection may be employed through a "table of contents," where references are indices into a table that has real addresses that are updated as needed. This, however, incurs additional cost whenever a dereference occurs. While garbage collection makes life easier for the programmer, it does come at the cost of efficiency.

7.3 Linked List Example: `malloc()`

The C Standard Library provides a heap allocator called `malloc()`. When the loader creates an address space for the process, the typical layout is that code and global data start at a low address and extend as needed. The end of this fixed-size portion is represented as the symbol `_end`. The stack, by convention, starts at a high address and grows downward. The space between `_end` and the stack pointer can be used as the heap. `malloc()` and the Operating System denote the maximum space of the heap by the symbol `brk`. Figure 7.6 shows the relation of these symbols to each other.

The break can be set via a system call `brk()` or a library wrapper `sbrk()`. When

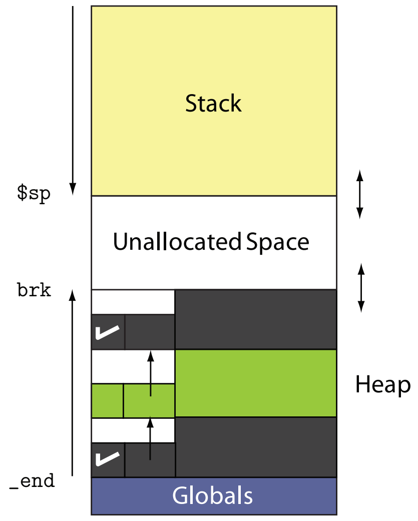


Figure 7.6: Heap management with `malloc()`.

`malloc()` gets a request for allocation that cannot fit, it extends `brk`. The heap is exhausted if the break gets too near the top of the stack. Likewise, the stack may be exhausted (usually from deep recursion) if it gets too close to `brk`.

Typically `malloc()` uses a linked list allocation strategy to track free and allocated space. One of the issues with linked lists is the question of where to store the list. An implementation of `malloc()` might store the linked list inside the heap, with each node near the allocated region. This allows calls to `free()` to easily access the size field of the node in the list corresponding to the region to reclaim. The drawback to this is that any over- or under-run of a heap-allocated buffer may overwrite the list, resulting in a corrupted heap.

Not all implementations of `malloc()` adjust the value of `brk`. The GNU implementation in `glibc` uses `mmap()` for allocations beyond 128KB. The `mmap()` system call requests pages directly from the Operating System. Some `malloc()`s use only `mmap()` for allocation.

7.4 Reducing External Fragmentation: The Buddy Allocator

The concern over external fragmentation has lead to various algorithms being developed beyond those used with the basic bitmap or linked list data structures. One particularly good algorithm for reducing external fragmentation is called the **buddy allocator**. Consider the case that we have a free memory region of 2MB and that we wish to allocate 4KB. The buddy algorithm looks for a free space reasonable to hold the allocation request. Right now, there is a single free space of size 2MB which is too large to reasonably allocate to this space. The buddy allocator takes the region and splits it into two allocations of half of the original size. So in this case, the algorithm turns our space into two allocations of 1MB each. This is still too much, and so one of the 1MB regions is split into two 512KB, and again split to 256KB, and so on, until finally we split an 8KB space into two 4KB regions. We now mark one of them as in-use and return it for the user.

Free regions of a particular size are linked together, using a portion of the region as a list node as in the implementation of `malloc()` described in the previous section. This way the left over regions from previous splits can be handed out quickly. We don't need to keep a list node for an allocated region, however. All that is necessary is a single bit that indicates that the region is free. Practically, we may still wish to reserve space for an entire list node for when the space is later freed.

We don't need to keep a list of allocated space because of a nice property that our scheme has. Every time a region was split into two, the two regions have addresses with a specific relationship. For a block of size n at address 0, when split it becomes two blocks of size $\frac{n}{2}$, one with address 0 and the other at address $\frac{n}{2}$. If we started with n being a power of two, then $\frac{n}{2}$ is one less power of two. This means the regions that were created have addresses that differ only by the value of a single bit, in the position of the new size.

Given an address L of a region of size 2^k, we can find its "buddy" (the other node split from the parent node) by inverting the kth bit. The bitwise XOR (eXclusive-OR) operator computes this for us: `buddy = L ^ size`. Note that this only works if the first region began at address `0` and was a size that is a power of two. If the starting address was something else, as it often will be in a practical application, we can simply subtract the actual starting address from L before performing the XOR operation.

If a block and its buddy are both free after deallocation, they can be safely coalesced back into a node of size 2^{k+1}. The free buddy will be removed from the linked list of free spaces of size 2^k and the combined space will be inserted into the free list of size 2^{k+1}. Since this may result in both a region and its buddy being free at size 2^{k+1}, we can repeat this process with progressively larger regions until the newly coalesced region's buddy is not free or only the original entire free space is left.

7.5 Terms and Definitions

The following terms were introduced or defined in this chapter:

Bitmap An array of bits indicating which chunks are allocated.

Coalesce To combine two or more nodes into one.

External Fragmentation Free space that is too small to be useful, a result of deallocation without compaction.

Garbage Collection Automatic deallocation of dynamic memory that occurs when a memory region is no longer needed.

Heap A region of a process's address space dedicated to dynamic data whose lifetime extends beyond that of the function that creates it.

Internal Fragmentation Wasted space due to the minimum allocation unit being too large.

Memory Leak When a dynamically allocated region cannot be deallocated because all pointers to it have been lost.

Run-length Encoding A simple compression scheme where n values in a row can be replaced by the number n in any base greater than one. For example, the string "AAAAA" could be compressed to "5A".

8 | Operating System Interaction

THE **Operating System** (OS) is a special process on a computer responsible for two major tasks: *managing resources* and *abstracting details*. An OS manages the shared resources on the computer. These resources include the CPU, RAM, disk storage, and other input and output devices. The OS is also useful in abstracting the specific details of the system away from application programmers. For instance, an OS may provide a uniform way to print a document to a printer regardless of its specific make and model.

The core process of the OS is called the **kernel**. The kernel runs at the highest privilege level that the CPU allows and thus can perform any action. The kernel is responsible for management and protection; it should be the most trusted component on the system. The kernel runs in its own address space that is referred to as **kernel space**. Programs that are not the kernel, referred to as **user programs**, cannot access the memory of the kernel and run at a lower privilege level. This portion of the computer is called **user space**.

Application programmers access the facilities that the Operating System provides via **system calls**. System calls are functions that the Operating System can do, usually related to process management and I/O operations.

8.1 System Calls

The particular system calls an Operating System provides depend on the particular OS and version. Because of this, most application programmers access system calls via a library **wrapper function**. Wrapper functions provide an OS-neutral way to do common tasks such as file and console I/O. The C Standard Library contains many wrapper calls as part of the `stdio` package.

Figure 8.1 shows the steps involved in executing a library call that needs to call the Operating System. If we have a call to `printf()` in our program, it is compiled

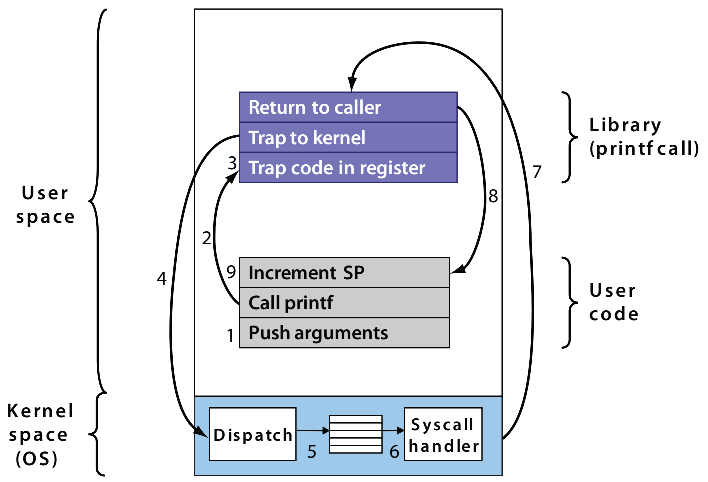


Figure 8.1: A library call usually wraps one or more system calls.

```
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 7), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                 MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x2a95557000
write(1, "Hello world!\n", 13Hello world!) = 13
exit_group(0)
```

Figure 8.2: A "Hello world" program run through `strace`.

to a `call` to the C Standard Library. In the library, the code to interpolate the arguments is run, and the final output string is prepared. When it is finally time to display the string, the library makes a system call to do the actual I/O operation. The Unix/Linux utility `strace` provides a list of system calls made during the execution of a program. In Figure 8.2 we see the system calls made by a "hello world" program using `printf()`.

The Unix and Linux Operating Systems provide a `write()` system call that interacts with I/O devices. The first parameter being the value 1 indicates that the output should go to the `stdout` device. Section 8.1.2 will detail the I/O calls provided by Unix and Linux systems.

8.1.1 Crossing into Kernel Space from User Space

Figure 8.1 also illustrates that a system call forces the CPU to switch modes from dealing with a process running in user space to the kernel running in kernel space. User space applications cannot cross this boundary themselves but instead issue a **trap** instruction that signals to the CPU that a **context switch** into the kernel should occur. A context switch occurs whenever the CPU switches from running one process to another. To resume the suspended process, the state of the machine — called a process's **context**, which includes the registers, open files, and stack — must be saved. To run the new process, its context must be restored. Context switches are very time consuming and all attempts to avoid doing more than absolutely necessary are made.

Whenever the CPU receives a trap or **interrupt**, it transfers control via an array of function pointers indexed by interrupt number called the **interrupt vector**, which is set up by the Operating System when it boots up. Under Linux the system call trap is `int 0x80`.

Upon entering the OS via a trap, the CPU is now in **kernel mode** and can perform the privileged operations of the OS, such as talking to the system I/O devices. The dispatcher selects the appropriate system call routine to execute based on the value of a register when the interrupt was sent. The kernel can now perform the requested operation.

8.1.2 Unix File System Calls

Listing 8.1 gives an example of how to write a program that writes text to a file without using the C Standard Library calls. Unix and Linux systems provide the primitive I/O operations of `open()`, `read()`, `write()`, and `close()` to operate on files. These operations end up being used to perform almost every I/O operation due to the traditional Unix paradigm of treating all devices as files in the file system.

While C programs that operate on files make use of a `FILE *` to track open files, the Unix system calls designate an integer as a *file descriptor,* which represents an open file. When a program begins, three file descriptors are automatically opened. They are:

Descriptor	C Name	Usage
0	`stdin`	Standard Input: Usually the keyboard
1	`stdout`	Standard Output: Usually the terminal
2	`stderr`	Standard Error: Usually the terminal

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main() {
        int fd;
        char buffer[100];
        strcpy(buffer, "Hello,␣World!\n");

        fd = open("hello.txt", O_WRONLY | O_CREAT);
        write(fd, buffer, strlen(buffer));
        close(fd);
        exit(0);

        return 0;
}
```

Listing 8.1: Using the Unix system calls to do a “Hello World” program.

Notice that `stdout` and `stderr` both display upon the screen by default. They are separate streams, however, and may be redirected or piped independently of each other.

Another thing to notice about Listing 8.1 is the second parameter to `open()`. Two macros are bitwise-ORed together. If we look for the definitions of these in the header files, we see them as:

```
#define O_RDONLY        0
#define O_WRONLY        1
#define O_RDWR          2
#define O_CREAT         16
```

which are all powers of two. This technique is common when we want to send several flags that affect the operation of a function. Each separate bit in an integer can be seen as an independent boolean flag. Bitwise-ORing them together allows the programmer to specify one or more flags simultaneously. In this example the flags to open the file for writing and to create it if it does not already exist are set. The implementation can check whether a particular flag is set by bitwise-ANDing the parameter with the same constants. This technique is also very commonly seen in the functions Microsoft Windows provides.

8.1.3 Process Creation with `fork()` and `execv()`

No survey of important system calls under Unix/Linux would be complete without discussing some issues of process management. A programmer may frequently wish to spawn off a new process to do some additional work concurrently. The system call to do this is `fork()`. Listing 8.2 shows an example of `fork()`'s usage. When `fork()` is called, the Operating System creates a clone of the original process that is identical in every way except for the return value of the `fork()` call. In the original process, denoted the parent, the return value is the **process ID** — the number that the OS uses to keep track of processes — of the child. In the child process, the return value is zero.

If we run the program, we might try to predict its output. The problem with doing this is that there is usually no guarantee that two separate processes will run in any particular order. At the very least, we can be sure that there will be four lines printed to the screen: the two inside the `if`/`else` and each process's copy of the "Hi from both" line. The other guarantee is that the lines will print in the proper relative order in terms of a single process. That is, there is no way to see both "Hi from both" lines before "Hi from the child" and "Hi from the parent" have each

```
#include <stdio.h>
#include <unistd.h>

int main() {
        if(fork()==0) {
                printf("Hi␣from␣the␣child!\n");
        }
        else {
                printf("Hi␣from␣the␣parent\n");
        }

        printf("Hi␣from␣both\n");
        return 0;
}
```

Listing 8.2: An example of process creation using `fork`.

been displayed. On the test run, the following output was seen:

```
Hi from the child!
Hi from both
Hi from the parent
Hi from both
```

This indicates that the child process ran and completed before the parent process resumed its execution.

The other common use of `fork()` is to launch a separate program entirely. The family of `execv()` functions all wrap around the `execv()` system call, which embodies the loader we described in Chapter 3. The unusual thing about `execv()` is that it needs a process to be created for it. The system call itself will not create a process; rather it will replace the process that called it with the new program to be loaded. This means that `execv()` often comes shortly after a call to `fork()`. Listing 8.3 shows an example of a program that launches the `ls` program. The parent, in the `else`, waits for all child processes to complete before it continues by using the `wait()` function.

```
#include <stdio.h>
#include <unistd.h>

int main() {
        if(fork()==0) {
                char *args[3] = {"ls", "-al", NULL};
                execvp(args[0], args);
        }
        else {
                int status;
                wait(&status);
                printf("Hi␣from␣the␣parent\n");
        }
        return 0;
}
```

Listing 8.3: Launching a child process using `fork` and `execvp`.

8.2 Signals

A **signal** is a message from the Operating System to a user space program. Signals are generally used to indicate error conditions, in much the same way that Java Exceptions function. A program can register a handler to “catch” a particular signal and will be asynchronously notified of the signal without the need to **poll**. Polling is simply the action of repeatedly querying (e.g., in a loop) whether something is true.

Figure 8.3 shows a list of the OS signals on a modern Linux machine. You can generate a complete list for your system by executing the command `kill -l`. Most signals tend to fall into a few major categories. There are the error signals, which indicate something has gone awry:

SIGILL The CPU has tried to execute an illegal instruction.

SIGBUS A bus error, usually caused by bad data alignment or a bad address.

SIGFPE A floating point exception.

SIGSEGV A segmentation violation, i.e., a bad address.

There are several ways to tell a program to forcibly exit on a system:

```
SIGHUP     SIGINT   SIGQUIT   SIGILL   SIGTRAP  SIGABRT
SIGBUS     SIGFPE   SIGKILL   SIGUSR1  SIGSEGV  SIGUSR2
SIGPIPE    SIGALRM  SIGTERM   SIGCHLD  SIGCONT  SIGSTOP
SIGTSTP    SIGTTIN  SIGTTOU   SIGURG   SIGXCPU  SIGXFSZ
SIGVTALRM  SIGPROF  SIGWINCH  SIGIO    SIGPWR   SIGSYS
```

Figure 8.3: The standard signals on a modern Linux machine.

SIGINT Interrupt, or what happens when you hit CTRL+C.

SIGTERM Ask nicely for a program to end (can be caught).

SIGKILL Ask meanly for a program to end (cannot be caught).

SIGABRT, SIGQUIT End a program with a core dump.

The remaining signals send information about the state of the OS, including things like the terminal window has been resized or a process was paused.

8.2.1 Sending Signals

Signals can be sent programmatically by using the `kill()` system call. The name is somewhat misleading, since any of the signals can be sent by using it, but most often the signals that are sent seem to be related to process termination. The code in Listing 8.4 makes the program stop with the `SIGSTOP` signal. The program will not resume until it receives a `SIGCONT` signal. The `getpid()` call asks the OS for the current process's id. If you know the process id of another process, you can send it signals as well.

It is useful to be able to send termination signals via the command line in a Unix shell, and the command `kill` allows you to do this. You need to pass the process id, but that can be obtained using the `ps` command. By default, `kill` with a process id argument will send that process a `SIGTERM` signal, which a process can choose to ignore. If a process is truly crashed, it is better to send the `SIGKILL` signal. You can specify the signal to send by saying `-NAME`, where name is the name portion of a signal, like `SIGNAME`. You may also specify a signal's numerical value, and you will often see a forcible termination of a process done like:

```
kill -9 process_to_kill_pid
```

```c
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

int main() {
        pid_t my_pid = getpid();
        kill(my_pid, SIGSTOP);
        return 0;
}
```

Listing 8.4: Signals can be sent programmatically via `kill`.

8.2.2 Catching Signals

Much like Java Exceptions, signals can be caught by a program. However, since many of the signals indicate something has gone terribly wrong, great care has to be taken in how a caught exception is dealt with. Any of the termination signals, if caught, should only be used as an opportunity to clean up any open or temporary files and exit gracefully. Memory state may be corrupted, so any attempts to continue will just make the program fail further down the line. A few signals are useful to catch, like `SIGALRM`. Listing 8.5 gives an example.

In this program, we set up a signal handler by using the `signal()` function. It takes two parameters, the first is the signal to listen for, the second is a function pointer of a function to call upon receipt of the signal. The alarm signal allows you to specify a timeout upon which the program will be notified that the time has elapsed. In our example, `alarm()` function tells the OS to send a `SIGALRM` in one second. This is not a precise time, as it may be beyond one second that the handler is called. The program in Listing 8.5 makes a countdown timer from ten to one and then exits.

One final signal of particular interest is `SIGTRAP`. This is the *Breakpoint Trap* signal. Debuggers such as `gdb` listen for this signal to retake control of an executing process in order to examine it.

The Intel `x86` instruction set defines all interrupts with a two-byte instruction encoding. The `int` opcode is `0xCD` followed by the interrupt number. For example, the Linux system call trap (`int 80`) would be `0xCD 0x80`. However, there is a special one-byte encoding for the breakpoint trap, `int 3`. It is the opcode `0xCC`.

Debuggers place breakpoints by overwriting existing instructions, since in-

```
#include <unistd.h>
#include <signal.h>

int timer = 10;

void catch_alarm(int sig_num) {
        printf("%d\n",timer--);
        alarm(1);
}

int main() {
        signal(SIGALRM, catch_alarm);

        alarm(1);
        while(timer > 0) ;
        alarm(0);
        return 0;
}
```

Listing 8.5: `SIGALRM` can be used to notify a program after a given amount of time has elapsed.

serting them would require rewriting the code. With the x86's variable-length instruction set architecture, a two-byte breakpoint might overwrite more than one instruction. This would be problematic if a particular breakpoint was skipped over and the target of a jump was the second byte of the breakpoint. To avoid this problem, the breakpoint trap is given a special one-byte encoding. Remembering this encoding may come in handy if you ever are dealing with low-level code and want to insert a breakpoint by hand.

8.3 Terms and Definitions

The following terms were introduced or defined in this chapter:

Context The state of a process, necessary to restart it where it left off. Usually includes the state of the registers and other hardware details.

Context Switch The act of saving the context of a running process and restoring the context of a suspended process in order to change the currently running program.

Interrupt A CPU instruction or signal (the voltage kind) issued by hardware that interrupts the currently executing code and jumps to a handler routine installed by the Operating System. On Intel x86 computers, there is no distinction in name between an interrupt and a trap; both are referred to as interrupts.

Interrupt Vector An array of function pointers indexed by interrupt number used to call an OS-installed handler routine.

Kernel The core process of an Operating System.

Kernel Space The Operating System's address space.

Operating System A program that manages resources and abstracts details of hardware away from application programmers.

Poll The act of repeatedly querying some state in a loop.

Process ID The number that the OS uses to keep track of processes.

Signal A message from the Operating System, delivered asynchronously, that usually indicates an error has occurred.

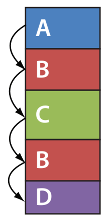
System Call A function that the Operating System provides to user programs to interact with the system and/or perform I/O operations.

Trap A software interrupt, usually used to signal the CPU to cross into kernel space from user space. **See also:** *interrupt*

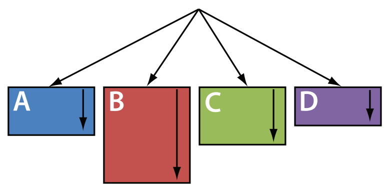
User Program Any application that is not part of the Operating System and runs in User Space.

User Space The unprivileged portion of the computer in which user programs run.

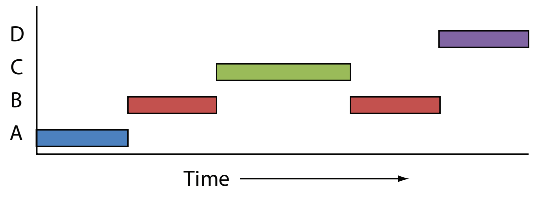
Wrapper Function A function, typically part of a library, that provides a generic way to do a system-specific common task such as file and console I/O.



(a) The CPU, with only one program counter, sees a contiguous stream of instructions.



(b) Each process, with its own isolated address space, appears to have a dedicated program counter and is never interrupted.



(c) Multiprogramming as viewed over time on a single CPU.

Figure 9.1: While the CPU sees just one stream of instructions, each process believes that it has exclusive access to the CPU.

9 | Multiprogramming & Threading

In Chapter 5, we introduced the abstraction of a running program in memory called a process. One of the significant parts of that abstraction was the process's view that it had a large amount of RAM all to itself as part of its address space. Another part of the process abstraction is the idea that a process has the entire CPU to itself. On a large supercomputer or a small embedded device, this might be the case. In general, however, a modern Operating System has to manage multiple processes all competing for the shared resource of the CPU. Figure 9.1 illustrates the differing perspectives that the CPU and the processes have. The CPU sees one unified stream of instructions, whereas each process believes it has exclusive access to the machine.

An observation about running processes is that they frequently will need to perform some type of I/O operation. Whenever an I/O device is accessed, there is a delay until it is ready to transmit or receive data. During this delay, a process cannot proceed and is said to be **blocked**. For instance, imagine a very fast typist typing at 120 words per minute. Two words per second is fast for a person, but modern computers can do billions of operations per second. Most of the time, the computer is waiting for the user to do something. If during this waiting time the computer could do other work, it could provide the illusion of doing multiple tasks at the same time. This abstraction is called **multiprogramming**. Figure 9.1c shows the progress of processes over time. Note the gap in B's execution while C runs. Process B should be totally unaware that it was not in control of the CPU for a portion of time.

To change from one application to another, we need to save the current application's CPU state, which we defined before as its context. Just as we did a context switch when we wanted to enter the kernel to perform a system call, when one process is paused and another is begun, the Operating System does a context switch.

It is also possible that a program is **CPU-Bound**, meaning that it does not do very much I/O but rather is computing something intensive, using as much of the CPU as it can get. In these cases, a CPU-bound process would stop other processes from being able to run. To prevent such a process from starving out the others, periodically the hardware will issue a timer interrupt. This timer interrupt will cause the Operating System to run, and then it can determine whether the program should continue or should be paused to let another process have the CPU. This process of pausing a running program after a period of time is called **preemption**.

Figure 9.2 shows the life cycle of a process. When a process is created, it enters a queue of processes on the system that are available to run, called the *ready queue*[1]. When a process is in the running state it eventually will yield the CPU, either because it performs an I/O operation and transitions to the blocked state or because it is preempted and goes back into the ready queue. At this point, a component of the Operating System called the **scheduler** chooses a ready process (from the queue) and

1 While this queue may be managed in FIFO order, we will use 'queue' just to imply a set of waiting objects.

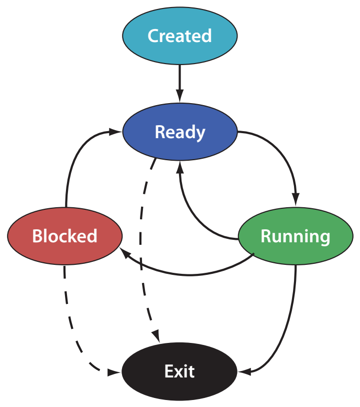


Figure 9.2: The life cycle of a process. Dashed lines indicate an abnormal termination.

Per Thread	Per Process
Program counter	Address space
Registers	Open files
Stack & stack pointer	Child processes
State	Signals & handlers
	Accounting info
	Global variables

Figure 9.3: Thread State versus Process State.

lets it run. Processes eventually finish voluntarily or because of an error (indicated by the dashed transitions in Figure 9.2).

Sometimes, we as programmers know that a process will be **I/O Bound**, and we would explicitly like to do some other task related to the process simultaneously. If we launch another process to do work in parallel, we have two issues. The first issue is that we have no guarantee from the scheduler when a particular process will run, so we may not be doing the other work while the first process is blocked. The second problem is that because of the isolation an address space provides, it is very cumbersome and slow to share information between processes. Ideally, we'd like the ability to run multiple streams of instructions that share a single address space so that sharing data is as easy as loads and stores.

We can have multiple streams of instructions in a single process's address space by having a mechanism called a **thread**. A thread is a stream of instructions and its associated context. A thread's context should be small, since the Operating System will still manage the process as a whole. For instance, a list of open files is part of a process's context but not an individual thread's. Figure 9.3 shows a list of what might be part of a process's context and what context needs to be stored per thread. If we define a stream of instructions as a function, we can more easily see what state we need to store. A function needs the machine registers and a stack of its own. This will form a minimal thread context.

9.1 Threads

Every process has one thread by definition. An application must explicitly create any additional threads it needs. Support for threading can come from two sources. In **user threading**, a user-space library provides threading support with minimal, if any, support from the Operating System. In **kernel threading**, the Operating System has full support for threads and manages them in much the same way as it manages processes.

9.1.1 User Threading

With user threading, we assume the Operating System has no explicit support for threads. A library containing helper functions will take care of thread creation and maintenance of the threads' state. There are two major hurdles to making user threading work. Imagine two processes are running on a system, one of which wants 100% of the CPU, the other wants 10%. Since the Operating System can preempt the greedy process, the one that needs only a little bit of CPU time can get it.

Imagine now two threads of a process that have the same characteristics. The greedy thread runs and runs until the process containing both threads is interrupted. When the process resumes execution, the greedy thread continues to run. The process abstraction prevents the process from knowing it was ever stopped. While each process was protected from the others on a system, there is no protection from the Operating System for threads since it does not know they are even there!

This may seem to be the downfall of user threading. However, if we go back to the original motivation for having threads at all, it was that we wanted to do work in a cooperative way and that the isolation afforded to us by an address space was too much. Since threads live within a single application, they can be expected to cooperate. A user threading library could then supply a `yield()` call whereby one thread voluntarily gives up the CPU, and the threading library can pick a different thread to run. Application programmers writing a multithreaded program only need to be aware that they should call `yield()` at appropriate times. This explicit yielding even gives a bit of extra power to the developer, because it becomes easy to make one thread be more important than the others by having it yield less frequently.

Having solved that problem, let us tackle a second. Imagine that a thread is executing some code and comes upon an I/O operation. The operation traps into the kernel, and finding the data not yet ready, the kernel moves the process into the blocked state. One of the original motivations was that during these times of being blocked, we would like to run some other code to do some useful work, so we might assume that another thread will get to run. But remember, the kernel knows nothing of the threads and has put the entire process to sleep. There is no way any other code in the process can run until the I/O request has finished.

Though we found a reasonable way around the yielding problem, this seems the death knell for user threads. There is no way to avoid the inevitable block that will happen to the process when the I/O operation cannot be completed. The only way around it would be if there was some facility by which going into a blocked state could be prevented. If an Operating System has a facility for non-blocking I/O calls, the user threading library could use them and insert a yield to run a different thread until the requested data was ready.

Unix/Linux systems have a system call named `select()` or `poll()` that tells whether a given I/O operation would block. It has the added side effect of doing the actual I/O request. Since `select()` can be non-blocking, the threading library could provide its own version of the I/O calls. A thread would use the library's routines, and when in the library, the library could make a call to `select()` to see if the operation would block. If it would, it can put that thread to sleep and allow another thread to run. The next time we are in the library, via a `yield()`, a `create()`, or an I/O call for another thread, we can check to see whether the original call is ready, and if so, unblock the requesting thread.

Since an Operating System needs to have non-blocking I/O support to make user threading work, it is arguable that user threads actually require no explicit support. The `select()` call is useful for more than just threads, including checking to see whether any network packets have arrived. While this minimal level of functionality is required, we will see with kernel threads that the level of OS support is far beyond a single system call.

9.1.2 Kernel Threading

Kernel threading is the complete opposite of user threading. With kernel threads, the Operating System is completely in charge of managing threads. The OS has a system call that creates threads. The scheduler in the OS knows that when one thread is blocked, another thread from the very same process could still run. A CPU-bound thread can be preempted without any need for the programmer to put in explicit calls to a `yield()` function. In short, kernel threads are everything that user threads were not. However, kernel threads also may be slower to create.

In user threads, any thread operation was a library call, and since no process boundaries were crossed, no context switches needed to be done. But anytime a thread is created in kernel threading, the OS must update its internal record-keeping, and a context switch into the kernel must be done. Context switches are expensive, so creating many threads will be significantly slower. Thread and process switching will both require context switches with kernel threading. These issues may not be a problem for a program using a small number of threads, but a Web server for a high-demand Web site may be spawning new threads for every request a hundred times per second. In that case, kernel threads may have too much overhead.

Ultimately, when it comes time to make a multithreaded program, you will be at the mercy of the system you are developing for. If it has kernel threads, your life may be easier, but performance may not be as good as possible. The good news is that you can always use user-threading libraries on a system with kernel threads. Use the tool that works best for the situation.

9.2 Terms and Definitions

The following terms were introduced or defined in this chapter:

Blocked A process that is unable to continue because it is waiting for something, usually an I/O request to complete.

CPU-Bound A process that primarily needs to do computation and rarely needs to do an I/O operation.

I/O Bound A process that spends most of its time blocked.

Kernel Threading Operating System support for thread management.

Multiprogramming Part of the process abstraction where a process appears to have the CPU entirely to itself, even when there are multiple processes on a single machine.

Preemption Interrupting a process because it has had the CPU for some amount of time in order to allow another process to run.

Scheduler The portion of the Operating System responsible for choosing which process gets to run.

Thread A stream of instructions and its associated context.

User Threading Threading done via a user space library that provides thread support with minimal, if any, support from the Operating System.

10 | Practical Threading, Synchronization, & Deadlocks

To facilitate multithreaded programming on a wide range of Operating Systems, a standard library was developed to hide the implementation details of writing multithreaded programs. The POSIX group created a standard library called `pthreads` that allows for the creation and management of multithreaded programs without concern for the underlying implementation — that is, whether user threading or kernel threading is supported.

With an abstraction such as the `pthreads` library, it is possible to write portable threaded programs that run on a variety of systems. In this chapter, we will examine the functionality the library provides. Once we are able to create and manage threads, we will consider issues of **synchronization**: The need for many threads that share data to safely modify that data. Likewise, we need to examine the issue of **deadlocks**. Deadlocks arise when two or more threads are unable to make progress while waiting on each other to do some task.

10.1 Threading with `pthreads`

The most basic operations for a threading library to provide are functions for creating and destroying threads. The `pthread` library provides several functions to aid us with these tasks. All processes consist of one thread of execution already, and so any additional threads we want to create will be in addition to this original thread. Thread creation is done via the `pthread_create()` function.

Listing 10.1 shows an example of running the `do_stuff()` function in two threads. The main thread is "recycled" to run the function in addition to the newly spawned thread. `pthread_create()` takes four parameters. The first is a pointer to a variable of type `pthread_t` that is set to a unique identifier for the newly created

```
#include <stdio.h>
#include <pthread.h>

void *do_stuff(void *p) {
    printf("Hello␣from␣thread␣%d\n", *(int *)p);
}

int main() {
    pthread_t thread;
    int id, arg1, arg2;
    arg1 = 1;
    id = pthread_create( &thread, NULL,
            do_stuff, (void *)&arg1 );
    arg2 = 2;
    do_stuff((void *)&arg2);
    return 0;
}
```

Listing 10.1: Basic thread creation.

thread. This identifier is "opaque," meaning that we do not know what type this identifier is (integer, structure, etc.) and we should not depend on its being any particular type or having any particular value. The second parameter controls how the thread is initialized, and for most simple implementations it can be set to `NULL` to take on the defaults.

The third and fourth parameters specify the stream of instructions to run in the new thread. First comes a pointer to a function containing the code to run. This function can, of course, call other functions, but it could be considered analogous to a "main" function for that particular thread. The signature of the function must be such that it takes and returns a `void *`. Because of the strict type checking done on passing function pointers in terms of return values and parameters, this function needs to be as generic as possible while still having a well-defined prototype. The advantage in using a `void *` is that it can point to anything, even an aggregate data type like an array or structure. In this way, no matter how many arguments are actually needed, the function can receive them. The final parameter is the actual parameters to pass to this function, which can be `NULL` if unnecessary.

```
int main() {
    pthread_t thread;
    int id, arg1, arg2;
    arg1 = 1;
    id = pthread_create(&thread, NULL,
            do_stuff, (void *)&arg1);
    pthread_yield();
    arg2 = 2;
    do_stuff((void *)&arg2);
    return 0;
}
```

Listing 10.2: Inserting a yield to voluntarily give up execution.

Compiling this program requires an additional command-line option to `gcc`. Since the `pthread` library is not part of the C Standard Library, it is not linked against the program automatically. Adding the `-pthread` switch tells the linker to include the appropriate code and data.

Executing the resulting program may lead to some interesting results. On one system, the following output from Listing 10.1 was seen:

```
Hello from thread 2
```

It would appear that the newly created thread is not run. If we did a similar test using `fork()` instead, the output of both would be seen. So what is the difference? When the `main()` function returns, the process is over. Since `main()` terminates relatively quickly, the other thread never gets a chance to run. With two processes (from `fork()`), each will not terminate until its respective `main()` finishes, guaranteeing that each will print its output before completing.

Some control over when a thread runs or when the process terminates is necessary. From the discussion of threading in the previous chapter, we know that it is possible to voluntarily yield control to a separate thread. The `pthread` library exposes this through the `pthread_yield()` function. If we rewrite the `main()` function as in Listing 10.2, we get the following output:

```
Hello from thread 1
Hello from thread 2
```

However, this is no guaranteed solution. While the `pthread_yield()` is a suggestion to let another thread run, there is no way to *force* this to happen. The other

```
int main() {
    pthread_t thread;
    int id, arg1, arg2;
    arg1 = 1;
    id = pthread_create(&thread, NULL,
            do_stuff, (void *)&arg1);
    arg2 = 2;
    do_stuff((void *)&arg2);
    pthread_join(thread, NULL);
    return 0;
}
```

Listing 10.3: Waiting for the spawned thread to complete.

thread(s) might be blocked, or the scheduler might simply ignore the yield. A better solution is to force the process to wait until the other thread completes.

Listing 10.3 illustrates the better way to ensure threads complete. Calling the `pthread_join()` function blocks the thread that issued the call until the thread specified in the parameter finishes. The second parameter to the `pthread_join()` call is a `void **`. When a function needs to change a parameter, we pass a pointer to it. Passing a pointer-to-a-pointer allows a function to alter a pointer parameter, in this case, setting it to the return value of the thread the join is waiting on. We can, of course, choose to ignore this parameter, in which case we can simply pass `NULL`. Note, however, joining the threads still does not guarantee they will run in any particular order before the call to `pthread_join()`.

The moral of the threading story is that, unless explicitly managed, threads run in no guaranteed order. This lesson becomes even more important when we begin to access shared resources in multiple threads concurrently. When we need to manipulate shared objects, we may need to ensure a particular order is preserved, which leads to the next topic: Synchronization.

10.2 Synchronization

Imagine that there are two threads, Thread 0 and Thread 1, as in Figure 10.1. At time 3, Thread 0 is preempted and Thread 1 begins to run, accessing the same memory location X. Because Thread 0 did not get to write back its increment to

	Thread 0	**Thread 1**
1	`read X`	
2	`X = X + 1`	
3		`read X`
4		`X = X + 1`
5	`write X`	
6		`write X`

Figure 10.1: Two threads independently running the same code can lead to a race condition.

memory, Thread 1 has read an older version. Whichever thread writes last is the one that makes the update, and the other is lost. When the order of operations, including any possible preemptions, results in different values, the program is said to have a **race condition**.

Determining whether code is susceptible to race conditions is an exercise in Murphy's Law.[1] Race conditions occur when code that accesses a shared variable may be interrupted before the change can be written back. The obvious solution to preventing a race condition is to simply forbid the thread from being interrupted during execution of this **critical region** of code. Allowing a user-space process to control whether it can be preempted is a bad idea, however. If a user program were allowed to do this, it could simply monopolize the CPU and not allow any other programs to run. Whatever the solution, it will require the help of the Operating System, as it is the only part of the system we can trust to make sure an action is not interrupted.

A better solution is to allow a thread to designate a portion of its code as a critical region and control whether other threads can enter the region. If a thread has already entered a critical region of code, all other threads should be blocked from entering. This lets other threads still run and do "non-critical" code; we have not given up any parallelism. The marking of a critical region itself must not be interruptible, a trait we refer to as being "atomic." This atomicity and the ability to make other threads block means that we need the Operating System or the user-thread scheduler's help.

Several different mechanisms for synchronization are in common use. We will focus on three in this text, although a fourth, known as a *Monitor*, forms the basis

1 "Anything that can go wrong, will go wrong."

	Thread 0	Thread 1
1	`lock mutex`	
2	`read X`	
3	`X = X + 1`	
4		`lock mutex`
5	`write X`	
6	`unlock mutex`	
7		`read X`
8		`X = X + 1`
9		`write X`
10		`unlock mutex`

Figure 10.2: Synchronizing the execution of two threads using a mutex.

for Java's support for synchronization. The `pthread` library provides support for *Mutexes* and *Condition Variables*. *Semaphores* can be used with the inclusion of a separate header file.

The `pthread` library provides an abstraction layer for synchronization primitives. Regardless of the facilities of the Operating System, with respect to its support for threading or synchronization, mutexes and condition variables will always be available.

10.2.1 Mutexes

The first synchronization primitive is a **mutex**. The term comes from the phrase **Mut**ual **Ex**clusion. Mutual exclusion is exactly what we are looking for with respect to critical regions. We want each thread's entry into a particular critical region to be exclusive from any other thread's entry. A mutex behaves as a simple lock, and thus we get the two operations `lock()` and `unlock()` to perform.

With a mutex and the lock and unlock operations, we can solve the problem of Figure 10.1. Figure 10.2 assumes a mutex variable named `mutex` that is initially in an unlocked state. Thread 0 comes along first, locks the mutex, and proceeds to do its work up until time 4, when it is preempted and Thread 1 takes over. Thread 1 attempts to acquire the mutex lock but fails and is blocked. With no other threads to run, Thread 0 resumes and finishes its work, unlocking the mutex. With the mutex now unlocked, the next time that Thread 1 is scheduled to run it can, as it is no longer in the blocked state.

The `pthread` library provides a simple and convenient way to use mutexes. A

```c
#include <stdio.h>
#include <pthread.h>
int tail = 0;
int A[20];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void enqueue(int value) {
    pthread_mutex_lock(&mutex);
    A[tail] = value;
    tail++;
    pthread_mutex_unlock(&mutex);
}
```

Listing 10.4: Using a `pthread_mutex` to protect an enqueue operation on a shared queue.

mutex is declared of type `pthread_mutex_t` and can easily be initialized to start off unlocked by assigning `PTHREAD_MUTEX_INITIALIZER` to it. Locking and unlocking are done via the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. Listing 10.4 shows an example of protecting a shared queue using a mutex.

10.2.2 Condition Variables

Sometimes we would like to do more than just protect a region. Imagine two threads working together, one of which is producing items into a fixed-size buffer, the other consuming them from that same buffer. This is a classic problem of synchronization known as the *Producer/Consumer Problem*. Figure 10.3 gives a pseudocode implementation. Since the buffer is fixed-size, we must be careful not to overrun or underrun the bounds. We want to stop the producer when the buffer is full and stop the consumer when it is empty. Let us then assume that we have a `sleep()` call to put the current thread to sleep and a corresponding `wakeup()` that will wake up a particular sleeping thread by notifying the scheduler that the thread is no longer blocked.

If we examine the consumer, we see the conditional `if(counter==0)` and the action `sleep()`. Here, obviously, the code is being guarded from the possibility of underrun. Possibly less obvious is the `if(count==N-1)` followed by the `wakeup(producer)`. If `count` is currently one less than the maximum, we know the

Shared Variables

```
#define N 10;
int buffer[N];
int in = 0, out = 0, counter = 0;
```

Consumer

```
while(1) {
  if(counter == 0)
    sleep();
  ... = buffer[out];
  out = (out+1) % N;
  counter--;
  if(counter == N-1)
    wakeup(producer);
}
```

Producer

```
while(1) {
  if(counter == N)
    sleep();
  buffer[in] = ... ;
  in = (in+1) % N;
  counter++;
  if(counter==1)
    wakeup(consumer);
}
```

Figure 10.3: A pseudocode implementation of the Producer/Consumer problem. Note that this code has an unresolved synchronization problem.

```
void *producer(void *junk) {
    while(1) {
        pthread_mutex_lock(&mutex);
        if( counter == N )
            pthread_cond_wait(&prod_cond, &mutex);
        buffer[in] = total++;
        printf("Produced:␣%d\n", buffer[in]);
        in = (in + 1) % N;
        counter++;
        if( counter == 1 )
            pthread_cond_signal(&cons_cond);
        pthread_mutex_unlock(&mutex);
    }
}
```

Listing 10.5: The producer function using condition variables. The consumer function would be similar.

buffer was full before we consumed an item, and if the buffer was full, the producer is asleep, so wake it up.

However, there is a subtle problem here. Imagine that the consumer is running, executes the `if(counter==0)` line, and finds the buffer empty. But right before the sleep is executed, the thread is preempted and stops running. Now the producer has a chance to run, and since the buffer is empty, successfully produces an item into it. The producer notices that the `count` is now one, meaning the buffer was empty just before this item was produced, and so it assumes that the consumer is currently asleep. It sends a wakeup which, since the consumer has not yet actually executed its sleep, has no effect. The producer may continue running and eventually will fill up the buffer, at which point the producer itself will go to sleep. When the consumer regains control of the CPU, it executes the `sleep()` since it has already checked the condition and cannot tell that it has been preempted.[2] Now both the consumer and the producer threads are asleep and no useful work can be done. This is called a **deadlock** and is the subject of Section 10.3.

There are two ways to prevent this problem. The first is to make sure that the check and the sleep are not interrupted. The second is to remember that there was a

2 In fact, it does not need to have been preempted if these two threads were running on separate cores or processors.

wakeup issued while the thread was not sleeping and to immediately wake up when the next sleep is executed in that thread. The first way is implemented via condition variables, the second solution is a semaphore.

A **condition variable** is a way of implementing sleep and wakeup operations in the pthread library. A variable of type pthread_cond_t represents a "condition" and acts somewhat like a phone number. If a thread wants to sleep, it can invoke pthread_cond_wait() and go to sleep. The first parameter is a condition variable that enables another thread to "phone it" and wake it up. A thread sleeping on a particular condition variable is awoken by calling pthread_cond_signal() with the particular condition variable that the sleeping thread is waiting on. Condition variables can be initialized much the same way that mutexes were, by assigning a special initializer value to them appropriately called PTHREAD_COND_INITIALIZER.

While this enables us to wait and signal (sleep and wake up), we still have the issue of possibly being interrupted. This is where the second parameter of pthread_cond_wait() comes into play. This parameter must be a mutex that protects the condition from being interrupted before the wait can be called. As soon as the thread sleeps, the mutex is unlocked, otherwise deadlock would occur. When a thread wakes back up it waits until it can reacquire the mutex before continuing on with the critical region. Listing 10.5 shows the producer function rewritten to use condition variables.

10.2.3 Semaphores

The semaphore.h header provides access to a third type of synchronization, a **semaphore**. A semaphore can be thought of as a counter that keeps track of how many more wakeups than sleeps there have been. In this way, if a thread attempts to go to sleep with a wakeup already having been sent, the thread will not go to sleep. Semaphores have two major operations, which fall under a variety of names. In the semaphore.h header, the operations are called wait and post, but they can also be known as lock and unlock, down and up, or even P and V. Whenever a wait is performed on a semaphore, the corresponding counter is decremented. If there are no saved wakeups, the thread blocks. If the counter is still positive or zero, the thread can continue on. The post function is an increment to the counter and if the counter remains negative, it means that there is at least one thread waiting that should be woken up.

One way to conceptualize the counter is to consider it as maintaining a count of how many resources there are currently available. In the Producer/Consumer

```
void *producer(void *junk) {
    while(1) {
        sem_wait(&semempty);
        sem_wait(&semmutex);
        buffer[in] = total++;
        printf("Produced:␣%d\n", buffer[in]);
        in = (in + 1) % N;
        sem_post(&semmutex);
        sem_post(&semfull);
    }
}
```

Listing 10.6: The producer function using semaphores. The consumer function would be similar.

example, each array element is a resource. The producer needs free array elements, and when it exhausts them it must wait for more free spaces to be produced by the consumer. We can use a semaphore to count the free spaces. If the counter goes negative, the magnitude of this negative count represents how many more copies of the resource there would need to be to allow all of the threads that want a copy to have one.

Semaphores and mutexes are very closely related. In fact, a mutex is simply a semaphore that only counts up to one. Conceptually, a mutex is a semaphore that represents the resource of the CPU. There can only be one thread in a critical region that may be running, and all other threads must block until it is their turn.

Semaphores can be declared as a `sem_t` type. There is no way to have a fixed initializer, however, because a semaphore can initially take on an integer value rather than being locked or unlocked. A semaphore is initialized via the `sem_init()` function, which takes three parameters: the semaphore variable, the value `0` on all Linux machines, and the initial value for the semaphore. Listing 10.6 shows the producer/consumer problem solved by using semaphores. Notice that they can even replace the mutex, although we could use a mutex if we wanted. The `semempty` (initialized to `N`) and `semfull` (initialized to `0`) semaphores count how many empty and full slots there are in the buffer. When there are no more empty slots, the producer should sleep, and when there are no more full slots, the consumer should sleep.

10.3 Deadlocks

The formal definition of a deadlock is that four things must be true:

Mutual exclusion Only one thread may access the resource at a time.

Hold and wait When trying to acquire a new resource, the requesting thread does not release the ones it already holds.

No preemption of the resource The resource cannot be forcibly released from the holding thread.

Circular wait A thread is waiting on a resource that is owned by a second thread which, in turn, is waiting on a resource the first thread has.

For our purposes, we will truly worry about the circular wait condition. This means that we must be careful about how and when we acquire resources, including mutexes and semaphores. If we do something as simple as mistakenly alter the order of the semaphores from Listing 10.6 to be:

```
sem_wait(&semmutex);
sem_wait(&semempty);
```

our program will instantly deadlock. If there are no empty slots, the thread does not release the semaphore used for mutual exclusion so that the other thread may run and consume some items.

Ensuring that your code is deadlock-free can sometimes be a difficult task. A simple rule of thumb can help you avoid most deadlocks and produce code that spends as much time unblocked as possible:

Always place the mutex (or semaphore being used as a mutex) around the absolute smallest amount of code possible.

This is not a perfect rule, and surely there is a counter-example to defeat it. No rules will ever replace understanding the issues of synchronization and using them to illuminate the potential problems of your own code.

10.4 Terms and Definitions

The following terms were introduced or defined in this chapter:

Critical Region A region of code that could result in a race condition if interrupted.

Deadlock A program that is waiting for events that will never occur.

Race Condition A region of code that results in different values depending on the order in which threads are executed and preempted.

Synchronization The protection against race conditions in critical regions.

11 | Networks & Sockets

In this chapter we will examine the basics of having two or more computers talk to each other over an electronic or radio-frequency connection. Having computers connected to a **network** is almost taken for granted in this day and age, with the quintessential network being the Internet. There are plenty of other networks, from telephones (both cellular and land-line) to the local-area networks that share data and applications in businesses and homes.

We will start with an introduction to networking basics from a programmer's perspective. We focus on the makeup and potential issues of network communication, and how they affect the performance and reliability of transmitting and receiving data. We then move to the de facto standard for programming network applications: **Berkeley Sockets**. Berkeley Sockets is an **Application Programming Interface** (API). An API is an abstraction, furnished by an Operating System or library that exposes a set of functions and data structures to do some specific tasks.

11.1 Introduction

A network is a connection of two or more computers such that they can share information. While networking is ubiquitous today, some details are important to understand before network-aware applications can be adequately written.

Networks, like Operating Systems, can be broken up into several layers to abstract specific details and to allow a network to be made up of heterogeneous components. There is a formal seven-layer model for networking known as the *OSI model* that serves as a set of logical divisions between the different components into which a network can be subdivided. The Internet uses five of these layers and results in the diagram shown in Figure 11.1. The bottom-most layer represents the actual electronic (physical) connection. While wireless networks are common, most networks will consist of a closed electrical circuit that sends signals and needs to

Application (DHCP, DNS, FTP, HTTP, IRC, POP3, TELNET ...)
Transport (TCP, UDP, RTP ...)
Internet (IP)
Data Link (ATM, Ethernet, FDDI, Frame Relay, PPP ...)
Physical Layer (Ethernet physical layer, ISDN, Modems, SONET ...)

Figure 11.1: The Internet Layer Model.

Bits	**0–3**	**4–7**	**8–15**	**16–18**	**19–31**
0	Version	Header Length	Type of Service	Total Length	
32	Identification			Flags	Fragment Offset
64	Time to Live		Protocol	Header Checksum	
96	Source Address				
128	Destination Address				
160	Options				
192–	Data				

Figure 11.2: Layout of an IP packet.

resolve issues of message collision. On top of this layer comes an agreement on how to send data by way of these electronic signals. The data is organized into discrete chunks called **packets**. How these packets are organized needs to be standardized for communication to be intelligible to the recipient. This standard agreement on how to do something is known as a **protocol**. The protocol governing the Internet is appropriately known as the *Internet Protocol* (IP).

The Internet Protocol defines a particular packet, as illustrated in Figure 11.2. The first 192 bits (24 bytes) form a header that indicates details such as the destination and source of the packet. To identify specific computers in a network, each computer is assigned a unique **IP address**, a 32- or 128-bit number. Because of the way IP addresses are allocated, it often works out that many computers share a single IP address, but the details of how this works are beyond the scope of this text.

Ethernet Header	IP Header	Protocol Header	Application Header	Data

Figure 11.3: Each layer adds its own header to store information.

The different sizes of IP addresses come about as a result of two different standards. IPV4 is the current system using 32-bit addresses. Addresses are represented in the familiar "dotted decimal" notation such as `127.0.0.1`. As networked devices keep growing, an effort to make sure that every device can have a unique address spawned the IPV6 standard. With this standard's 128-bit addresses, there is little chance of running out anytime in the foreseeable future.

Packets sent via IP make no guarantees about arrival or receipt order. As a theoretical concept, such a guarantee is impossible to make. Imagine that Alice sends a message to Bob and wants to know that Bob receives it, so she asks Bob to send a reply when he gets it. A week passes, and Alice hears nothing from Bob and begins to wonder. However, she is met with an unanswerable question: Did Bob not get her message, or did she not get Bob's reply?

The good news in regard to this conundrum is that modern networks are usually reliable enough that "dropped" packets are rare. A protocol that does nothing to guarantee receipt is known as a **Datagram** protocol. The term Datagram comes from a play on telegram, which also had no guarantee about receipt.

While mostly reliable communication might be adequate for some uses, the majority of applications want reliable, order-preserving communication. Email, for instance, would be useless if the message arrived garbled and with parts missing. Since we assume a mostly-reliable network, we can do better. A protocol implemented on top of IP called **Transmission Control Protocol** (TCP) attempts to account for the occasional lost or out-of-order packet. It does this through acknowledgment messages and a sequence number attached to each packet to indicate relative order. To do this, TCP needs to add this sequence number to the packet, and so it adds its own header following the IP header. Figure 11.3 illustrates the concatenation of headers done by each layer. (Note that the figure assumes the data link layer is Ethernet.)

Some applications, like streaming audio or video, or data for video games, can tolerate the occasional lost or out-of-order packet. Not worrying about receipt or order allows for larger amounts of data to be sent at faster rates. In fact, no connection is even necessarily made between the sender and the recipient. The most common

Server	Client
socket()	
bind()	connect()
listen()	
accept()	
send() and recv()	
close()	

Figure 11.4: The functions of Berkeley Sockets divided by role.

of these so called **connectionless** protocols is UDP, or the **User Datagram Protocol**. Using UDP provides nearly raw access to the IP packet without the overhead, or guarantees, associated with TCP.

The topmost layer is the *Application Layer*. This is implemented on top of TCP or UDP and consists of a protocol for programs to talk to each other. The protocol might be binary, like the Oscar protocol for AOL's Instant Messenger, or it might be text such as the famous Hypertext Transfer Protocol (HTTP) used by Web browsers to ask for Web pages from Web servers.

While IP addresses are convenient for computers to store and manipulate, they are not generally easily remembered by humans. People would rather have a name or other word to associate with a particular computer. On the Internet, each Web site has a unique *Domain Name* that corresponds to a particular IP address of the Web server. The World Wide Web provides a set of servers to facilitate translating a name into an IP address, a process known as domain name resolution. A **Domain Name Server** (DNS) provides a way to look up a particular IP address based upon the parts of a domain name.

11.2 Berkeley Sockets

Unix-like Operating Systems try to interact with devices, files, and networks in a uniform fashion by treating them all as part of the filesystem. By doing this, the programmer's interaction with the I/O device is uniform: The device can be opened, read or written, and closed. **Berkeley Sockets** serve to implement this abstraction for network communication.

The functions of the Berkeley Sockets API are listed in Figure 11.4. A socket is an I/O device representing a connection to a computer via a network. The `socket()`

call creates a file descriptor representing the connection to be used by the other functions. Berkeley Sockets distinguish between a listening server and a connecting client. Listing 11.1 gives the code for a server that simply replies with "Hello there!" to any program that connects to that particular machine and port pair. A **port** is an application-reserved connection point on a particular IP address.

Due to the fact that a network communication failure is much more likely than failures with many other I/O operations, even a simple program ends up with many lines of code. If something goes wrong, every function will return a negative number and set `errno`, the global error code, to the appropriate value. Using `perror()` converts `errno` into a (sometimes) useful error message and prints it to the screen. For the `send()` and `recv()` functions, the returned value indicates how many bytes were actually sent. If the data is too large, multiple calls may be needed to handle it.

We can connect to the server in Listing 11.1 by using `telnet`, which emulates a terminal and connects to a specified address and port. If the server is running on the local machine, the following output would be seen:

```
(1) thot $ telnet localhost 1100
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello there!
Connection closed by foreign host.
```

Berkeley Sockets also support connectionless protocols like UDP. The `sendto()` and `recvfrom()` calls take extra parameters that specify the address of the recipient or the sender. There is no need to do anything other than set up a socket to use them.

11.3 Sockets and Threads

The theme of Chapter 9 was that threads are useful when a program wants to do tasks in parallel. However, if all of those threads are CPU-bound, we may not see any performance advantage on a single-processor machine. Fortunately, we quickly realized that many modern programs are I/O-bound, and while the I/O operations are blocked, another thread could be scheduled to run.

While the I/O operations on a local machine may seem slow to the CPU, they are nothing compared to the delay incurred by doing network communication. Thus,

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>

#define MYPORT 1100

int main() {
    int sfd, connfd, amt = 0;
    struct sockaddr_in addr;
    char buf[1024];

    if((sfd=socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket␣failed");
        exit(EXIT_FAILURE);
    }

    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(MYPORT);
    addr.sin_addr.s_addr = INADDR_ANY;

    if(bind(sfd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("Bind␣failed");
        exit(EXIT_FAILURE);
    }
    if(listen(sfd, 10) < 0) {
        perror("Listen␣failed");
        exit(EXIT_FAILURE);
    }
    if((connfd=accept(sfd, NULL, NULL)) < 0) {
        perror("Accept␣failed");
        exit(EXIT_FAILURE);
    }

    strcpy(buf, "Hello␣there!\n");
    while(amt < strlen(buf)) {
        int ret = send(connfd, buf+amt, strlen(buf)-amt, 0);
        if(ret < 0) {
            perror("Send␣failed");
            exit(EXIT_FAILURE);
        }
        amt += ret;
    }
    close(connfd);
    close(sfd);
    return 0;
}
```

Listing 11.1: A server using sockets to send “Hello there!”.

we frequently see programs that access remote machines written in a multithreaded fashion. One such application is the multithreaded Web server.

A Web server does not necessarily meet the traditional idea of a multithreaded application, since each page request is likely to be independent and there is no real advantage to sharing a single address space. However, the real benefit comes from the idea of a main thread that accepts connections and then spawns a worker thread to take care of the I/O operations. The thread creation cost should be much cheaper than the overhead needed to create a full process, and thus the server can utilize CPU time more effectively. Other performance enhancements can also be incorporated. With a single address space, the server is free to make a shared cache in memory of frequently accessed files, reducing the need for disk I/O.

The more obvious marriage of threads and sockets comes from the frequent need to do asynchronous, bidirectional communication. Consider writing a simple instant messaging program that can talk to another program across a network. Which of the two instances of the program is the server and which is the client? Both programs want to send data at the request of the user. If the program was written in a single threaded fashion, it would need to have a sequence of `send()`s and `recv()`s, but their order dictates who can talk and who must listen.

The solution to this problem comes by having two separate threads, one devoted to sending and the other to receiving messages. This way, the receiving thread can remain safely blocked and the user can send any number of messages without delay.

11.4 Terms and Definitions

The following terms were introduced or defined in this chapter:

Application Programming Interface/API An abstraction, furnished by an Operating System or library, which exposes a set of functions and data structures to do some specific tasks.

Internet Protocol (IP) Address A number that represents a particular computer on the Internet.

Network A group of computers wired to talk to each other.

Packet The unit of data transmission on a network.

Port An application-reserved connection point on a particular IP address.

Protocol An agreement on how data should be sent.

Socket An abstraction representing a connection to another computer over a network.

A | The Intel x86 32-bit Architecture

The Intel x86 32-bit architecture is an example of a **Complex Instruction Set Computer** (CISC). While more recently designed CPUs have a simplified set of minimal operations, a CISC computer has many different instructions, special purpose registers, and complex addressing modes.

Figure A.1 gives a list of the general purpose registers. The first six can be used for most any purpose, although some instructions expect certain values to be in a particular register. `%esp` and `%ebp` are used for managing the stack and activation records. The program counter is `%eip`, which is read-only and can only be set via a jump instruction. The results of comparisons for conditional branches are stored in the register `EFLAGS`.

The registers `%eax`, `%ebx`, `%ecx`, and `%edx` each have subregister fields as shown in Figure A.2. For example, the lower (least-significant) 16 bits of `%eax` is known as `%ax`. `%ax` is further subdivided into two 8-bit registers, `%ah` (high) and `%al` (low). There is no name for the upper 16-bits of the registers. Note that these subfields are all part of the `%eax` register, and not separate registers, so if you load a 32-bit quantity and then read `%ax`, you will read the lower 16-bits of the value in `%eax`. The same applies to the other three registers.

Operations in x86 usually have two operands, which are often a *source* and a *destination*. For arithmetic operations like `add`, these serve as the addends, and the addend in the destination position stores the sum. In mathematical terms, for two registers a and b, the result of the operation is $a = a + b$, overwriting one of the original values.

`%eax`	Accumulator
`%ebx`	Base
`%ecx`	Counter
`%edx`	Data
`%esi`	String Source
`%edi`	String Destination
`%esp`	Stack Pointer
`%ebp`	Base or Frame Pointer
`%eip`	Instruction Pointer
`EFLAGS`	Flag register

Figure A.1: The 32-bit registers.

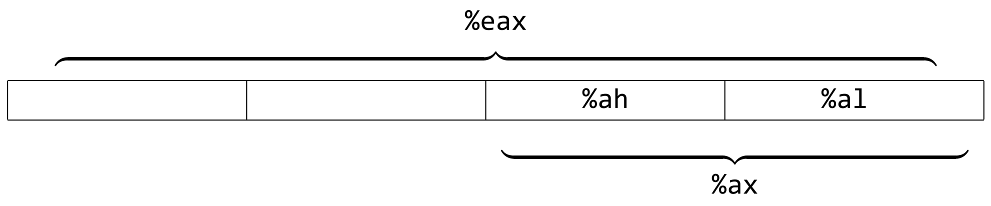


Figure A.2: `%eax`, `%ebx`, `%ecx`, and `%edx` have subregister fields.

`sub`	Subtract
`add`	Add
`and`	Bitwise AND
`push`	Push a value onto the stack
`pop`	Pop a value off of the stack
`mov`	move a value
`call`	call a function
`leave`	clean up a stack frame
`ret`	return from a function
`lea`	compute an address (pointer)

Figure A.3: The instructions used in this book.

A.1 AT&T Syntax

AT&T syntax is used by default in `gcc` and `gdb`. In AT&T assembler syntax, every operation code (opcode) is appended with the type of its operands:

`b`	byte (8-bit)
`w`	word (16-bit)
`l`	long (32-bit)
`q`	quad (64-bit)

After the opcode, the first operand is the source and the second operand is the destination. Memory dereferences are denoted by `( )`. Listing A.1 gives an example of a "hello world" program as produced by `gcc`.

A.2 Intel Syntax

Intel assembler syntax is the default syntax of the Intel documentation, Microsoft's compilers and assemblers (MASM), and NASM – the Netwide Assembler. Instead of appending the operand size to the opcode, Intel syntax uses C-like casts to describe the size of operands. The type sizes are spelled out:

```
.file   "asm.c"
        .section  .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "hello␣world!"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp ;1111 1111 1111 0000
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $0, %eax
        leave
        ret
```

Listing A.1: Hello world in AT&T assembler syntax.

```
main:
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        and     esp, -16 ;1111 1111 1111 0000
        sub     esp, 16
        mov     DWORD PTR [esp], .LC0
        call    puts
        mov     eax, 0
        leave
        ret
```

Listing A.2: Hello world in Intel assembler syntax.

`BYTE`	1 byte
`WORD`	2 bytes
`DWORD`	4 bytes (double word)
`QWORD`	8 bytes (quad word)

Intel syntax orders the operands completely in reverse from the AT&T convention. The first operand is the *destination*, the second operand is the source. Dereferences are denoted by `[ ]`. Listing A.2 gives a sample of the same "hello world" program as in Listing A.1 rewritten in Intel syntax.

A.3 Memory Addressing

One of the biggest surprises in the x86 instruction set for RISC assembly programmers is the memory addressing model. Architectures such as MIPS require that all Arithmetic/Logical Unit (ALU) operations have only registers as operands. Moving to and from memory requires explicit *load* and *store* instructions. However, most x86 instructions may take one operand as a memory location, as long as the other (if necessary) is a register. You may not have both a source and a destination that are in memory.

Memory addresses can be constructed from four parts: a signed offset (constant), a base (register), an index (register), and a scale (constant: 1, 2, 4, or 8). The resulting address is determined as: $base + index \times scale + offset$. As an example, we can choose `%ebx` as the base, `%eax` as the index, 4 as the scale, and 16 as the offset. In AT&T syntax this would be expressed as: `16(%ebx,%eax,4)`. In Intel syntax, it would be written as: `[ebx+eax*4+16]`. If the offset is zero or the scale is one it can be omitted, as can either of the registers if they are unnecessary.

A.4 Flags

Suppose we have a conditional statement in C such as `if(x == 0) { ... }` which we could translate into x86 using the compare and jump-if-equals instructions as:

```
    cmpl   $0, %eax
    je     .next
    ; ...
.next:
```

One thing that is not immediately apparent in the code is how the branch "knows" the result of the previous compare instruction. The answer is that the compare instruction has a side-effect: It sets the %eflags register based on the result of the comparison.

The %eflags register is a collection of single-bit boolean variables that represent various pieces of state beyond the normal result. Many instructions modify %eflags as a part of their operation. Some arithemetic instructions like addition set flags if they overflow the bounds of the destination. The conditonal jumps consume the state of various flags as the condition on which to branch. In the above example, the jump-equals instruction actually checks the value of the special zero flag (ZF) that is part of %eflags. In fact, the je instruction is actually a psuedonym for the jz instruction: jump if the zero flag is set.

The side-effect of an operation setting flags can lead to confusing code. Consider the listing:

```
        test   %eax, %eax
        je     .next
        ; ...
    .next:
```

This is functionally equivalent to the version that used cmpl above. The test instruction computes the bitwise-AND of the two arguments, in this case both are the register %eax. Since anything AND itself is going to be itself, this seems to be a no-op. But test takes the result of that AND and sets the ZF based on it. To learn about these side-effects, it is always handy to have the instruction set manual nearby.

A good compiler will probably generate the listing that uses test rather than cmp since the immediate 0 takes up 4 bytes of representation that are not needed in the encoding of test. The smaller code is generally preferred for performance (caching) reasons.

A.5 Privilege Levels

To keep user programs and the kernel separate, x86 processors have four different privilege rings that processes may run in (Figure A.4). The kernel itself runs as the most privileged process in ring 0. Ring 1 is usually reserved for drivers, which can be thought of as kernel processes. Ring 2 is used for either drivers or user libraries. The final ring is for unprivileged user programs. The general protection rule is that a process running in a particular ring may access the data of the rings above but not

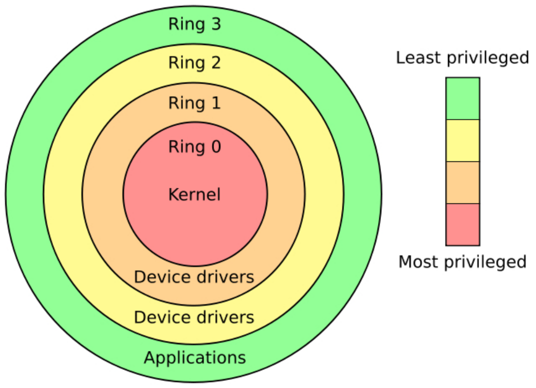


Figure A.4: Privilege rings on an x86 processor.

below it. This allows the OS to modify anything and to be protected from malicious or buggy user programs.

B | Debugging Under gdb

Often you will want to be able to examine a program while it executes, usually to find a bug or learn more about what a program is doing. A **debugger** is a program that gives its user control over the execution and data of another program while running.

With the very first programs a new programmer writes come the first problems, known as "bugs." These are logical errors that the compiler cannot check. To track down these bugs, a concept known as *debugging*, beginners often use print statements. While such statements certainly work, this approach is often tedious and time-consuming due to frequent recompiling and re-execution. Adding print statements can also mask bugs. Since these are additional function calls, they provide legitimate stack frames where before an array out of bounds problem might have been a segmentation violation. With multithreaded programs, print statements can change timings and context switch points resulting in a different execution order that may hide a concurrency issue or race condition.

Print statements generally come in two kinds: The "I'm here" variety, which indicates a particular path of execution, and the "x is 5" variety that examine the contents of variables. The first type is an attempt at understanding the decisions that a program makes in its execution, i.e., what branches were taken. The path of a program through its flowchart representation is known as its **control flow**. The second type explores data values at certain points of execution.

We can do both of these things with a debugger, without the need to modify the source code. We may, however, choose to modify the executable at compile-time to provide the debugger with extra information about the program's structure and correspondence to the source. Remember that a compiled executable has only memory locations and machine instructions. Gone are symbols like `x` or `count` for variables; all that remains are registers and memory addresses. For a debugger to be more helpful, we can choose to add extra information to the executable while compiling that equates those particular addresses back to the human-readable names they originally had. For `gcc`, the `-g` flag indicates that the compiler should augment the symbol table and executable code with debugging information.

Debuggers come in all shapes and sizes. Some might be part of an Integrated Development Environment (IDE); others might be stand-alone. They can be graphical or text-based. For this chapter, we will use the stand-alone, textual debugger `gdb`. All of the concepts and commands we establish will be portable to other debuggers, usually without many, if any, differences. We will begin our discussion of the role of the debugger by discussing how to examine (and stop) the flow of control in a program.

B.1 Examining Control Flow

The simplest way to control the execution of a program is to make it pause. We can ask a program to stop by having the debugger insert a **breakpoint** at a specific location. Locations can be specified in many ways, such as:

Function Name Execution will stop *before* the specified function is executed.

Line Number Execution will stop *before* the specified line of code is executed.

Absolute Address Execution will stop *before* the instruction at that address is executed.

The first two specifications require the executable to have additional information that may not necessarily be there. The line number information requires the executable to have been built with debugging information. The function names are usually part of the symbol table even without a special compilation but the symbol table may be "stripped" out after compilation. Specifying an absolute address always works, but gives no high-level language support.

Running `gdb` is as simple as specifying an executable to debug on the command line. If your program requires command line arguments, they can be specified by using the `--args` command line option, or by issuing the `set args` command once `gdb` has started up.

Once in `gdb`, the command to place a breakpoint is `break`, which can be abbreviated just by typing `b`. For example, a breakpoint could be placed at the `main()` function by typing `b main`. When the program is run via `run` or `r`, the program will immediately stop at the `main()` function. If there is debugging information, a breakpoint can be set at a particular file and line number, separated by a colon: `b main.c:10` sets a breakpoint at line 10 in the `main.c` file.

It is also possible to put a breakpoint at an arbitrary instruction by specifying its address in memory. The syntax is `b *0x8048365`, where the hexadecimal number needs to be the start of an instruction. It is up to you to ensure that this address is valid. If it is not aligned to the start of an instruction, the program might crash. Section 8.2.2 gives some insight on how breakpoints are implemented and why placing a breakpoint in the middle of a multibyte instruction would be catastrophic to the program.

Once the program is stopped, you will probably want to examine some data, the topic of the next section. One useful thing to check, however, is what the call stack contains, i.e., what function calls led to the current place. This is called a **backtrace** and can be seen via the `backtrace` (abbreviated `back` or `bt`) command.

When you are finished with your examination, you have either found your bug and want to stop debugging, or you will need to continue on. Typing `run` or `r` allows you to restart the program from the beginning. The `quit` command exits `gdb`. If you want to `continue`, you can simply issue the command or the shortcut `c`. Continuing runs the program until it hits another breakpoint or the program ends, whichever comes first.

You may also want to step through the execution of the program, as if there were a breakpoint at each line. If the program was built with debugging information, you can use the commands `next` and `step`. These two commands are identical except for how they behave when they encounter a function call. The `step` command will go to the next source line inside the called function. The `next` command will skip over the function call and stop at the source code line immediately following the call. In other words, it will not leave the current function. Both `step` and `next` can be abbreviated with their first letter.

If the source code is not available and the program was not built with debugging information, `step` and `next` cannot be used. However, there are parallel commands for operating directly on the machine instructions. The `stepi` command goes to the next machine instruction, even if that is inside a separate function, whereas `nexti` skips to the next instruction following a `call` without leaving the current function. The abbreviation for `stepi` is `si` and for `nexti` is `ni`.

Each of the above control flow instructions (i.e., `continue`, `next`, `step`, `nexti`, and `stepi`) take an optional numerical argument. This number indicates a *repeat count*. The particular operation is performed that many times before control is returned to the debugger.

The technique of last resort is to interrupt the program with a `SIGINT` signal (see Section 8.2) by pressing CTRL+C. The signal will be handled by `gdb` and it will return control back to the user. One word of caution, however: it may be somewhat surprising where execution has stopped (it could be deep in a bunch of library calls), so it may be helpful to use `back` to get a backtrace and possibly some locations for regular breakpoints.

B.2 Examining Data

With execution stopped and control transferred back to the debugger, most likely you will want to examine data values in memory. In `gdb` there are are two primary commands for examining data, `print` and `x`. The `print` command will display the value of an expression, which can be written using C-style syntax:

```
(gdb) print 1+2
$1 = 3
```

If the program was built with debugging symbols, you can write your expressions in terms of actual program variables to see what they contain. If your program is without such symbols, you can always look at register values. For example, on an x86 platform, you could display the contents of the register `%eax` by prefixing it with a dollar sign (`$`) like so:

```
(gdb) print $eax
$1 = 10
```

With the ability to use casts and dereferences, the `print` command is likely all that you need. However, the relative frequency with which you will want to look at the contents of some memory locations is high enough that there is a dedicated examine command, `x`. With the `x` command, the specified argument must be an address. By default, the contents are dumped as a hexadecimal number in the machine's native word size. It is possible to also specify a format, which acts as a typecast for the data. The type is specified by a single letter code following a forward slash. For instance:

```
(gdb) x 0x8048498
0x8048498 <_IO_stdin_used+4>:   0x6c6c6548
(gdb) x/d 0x8048498
0x8048498 <_IO_stdin_used+4>:   1819043144
```

```
(gdb) x/x 0x8048498
0x8048498 <_IO_stdin_used+4>:   0x6c6c6548
(gdb) x/s 0x8048498
0x8048498 <_IO_stdin_used+4>:    "Hello, world!"
```

Note that all four `x` commands operate upon the same address, but each has a different data interpretation. With `/d`, the number is printed out in decimal, rather than the hexadecimal (also obtainable via `/x`). Specifying `/s` will treat the address as the start of a C-style string and will attempt to print data until it encounters a null character.

The above example also illustrates a few of the output features of `gdb`. The first column is the address that is being examined, but in between the $<$ and $>$, `gdb` attempts to map this address back to the nearest entry in the symbol table. In some cases, this can be quite useful, since with debugging symbols, individual variable names will be identified even when you know only an address (say from the contents of a pointer). Without full debugging symbols, or with a stripped executable, the output might not be correct, so common sense must always be used when interpreting the output.

B.3 Examining Code

Whenever a breakpoint is encountered and control is returned to `gdb`, the current source line will be displayed if it is available. When the program contains debugging information and the source files are available, the debugger can operate in terms of the source code. You can see the source code around the current instruction pointer by using the `list` command. You can also specify a file and line number, as with breakpoints, or a function name.

When the source is unavailable, you can only see the machine instructions by using the command `disassemble`. Like `list`, `disassemble` will, by default, attempt to disassemble the instructions around `%eip`. You can additionally specify a region of memory addresses to dump. Care must be taken to ensure that the first address is actually the valid start of an instruction; otherwise, on a variable-length instruction set architecture, the disassembler could get confused.

One final trick is especially useful when CTRL+C is used to stop the program. Using the examine command with the `/i` format, you can disassemble an individual instruction at a certain location. To disassemble the instruction at the current instruction pointer location, do:

```
x/i $eip
```

B.4 gdb Command Quick Reference

Command	Abbrv.	Description
`help`		Get help on a command or topic
`set args`		Set command-line arguments
`run`	`r`	Run (or restart) a program
`quit`	`q`	Exit `gdb`
`break`	`b`	Place a breakpoint at a given location
`continue`	`c`	Continue running the program after hitting a breakpoint
`backtrace`	`bt`	Show the function call stack
`next`	`n`	Go to the next line of source code without entering a function call
`step`	`s`	Go to the next line of source code, possibly entering a new function
`nexti`	`ni`	Go to the next instruction without entering a function call
`stepi`	`si`	Go to the next instruction, possibly entering a new function
`print`		Display the value of an expression written in C notation
`x`		Examine the contents of a memory location
`list`		List the source code of the program
`disassemble`	`disas`	List the machine code of the program

B.5 Terms and Definitions

The following terms were introduced or defined in this chapter:

Backtrace A list of function calls that led to the current call; a stack dump.

Breakpoint A location in code where execution should stop or pause, usually used to transfer control to a debugger.

Control Flow The path or paths possible through a region of code as a result of decision (control) structures.

Debugger A program that controls and examines the execution and data of another program.

References For Further Reading

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] Jeff Bonwick and Sun Microsystems. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[3] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3rd edition, 2005. Available from: `http://lwn.net/Kernel/LDD3/`.

[4] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 1: Basic Architecture. 2007. Available from: `http://www.intel.com/design/processor/manuals/253665.pdf`.

[5] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2A: Instruction Set Reference, A-M. 2007. Available from: `http://www.intel.com/design/processor/manuals/253666.pdf`.

[6] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2B: Instruction Set Reference, N-Z. 2007. Available from: `http://www.intel.com/design/processor/manuals/253667.pdf`.

[7] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley Longman, 2nd edition, 1999. Available from: `http://java.sun.com/docs/books/jvms/`.

[8] Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. *The GNU C Library Reference Manual*. Free Software Foundation, 0.11 edition, 2007. Available from: `http://www.gnu.org/software/libc/manual/pdf/libc.pdf`.

[9] Mark Mitchell, Jeffrey Oldham, and Alex Samuel. *Advanced Linux Programming*. New Riders Publishing, 2001. Available from: `http://www.advancedlinuxprogramming.com/alp-folder`.

[10] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 3rd edition, 2007.

[11] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with gdb*. Free Software Foundation, 9th edition, 2006. Available from: `http://sourceware.org/gdb/current/onlinedocs/gdb.html`.

[12] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001.

Index

A page number appearing in **bold** indicates an end-of-chapter definition.

Colophon

col·o·phon, *noun*, 1: an inscription
placed at the end of a book or manuscript
usually with facts relative to its production

—Merriam-Webster's Collegiate Dictionary, Eleventh Ed.

This text was typeset in LaTeX 2_ε using `lualatex` under the MiKTeX 2.9 system on Windows 7. LaTeX is a macro package based upon Donald Knuth's TeX typesetting language.[1] LaTeXwas originally developed by Leslie Lamport in 1985.[2] MiKTeX is maintained and developed by Christian Schenk.[3]

The typefaces are Minion Pro, Myriad Pro, and Consolas. Illustrations were edited in Adobe® Illustrator® and the final PDF was touched-up in Adobe® Acrobat®.

About the Author

Jonathan Misurda is a Lecturer in the Department of Computer Science at the University of Pittsburgh, where he did his Ph.D. in the Software Testing aspect of Software Engineering. His heart lies, however, in Computer Science Education.

Jonathan's hobbies and interests can vary seemingly on the month, but his experiences in preparing this text have sparked an interest in Graphic Design and Typography. He is always on the lookout to learn how to better present the topics he explains and the information he presents. His goal is to make both his teaching and his book look better and to express their content more clearly. To this end he jokes that he is in his "Knuth" phase.

1 http://www-cs-faculty.stanford.edu/~knuth/
2 http://www.latex-project.org/
3 http://www.miktex.org/