CHAPTER 2

Programming

In Chapter 1, we learned how the CPU fetches, decodes, and executes instructions and that those instructions sit on a persistent storage device until the CPU is turned on and transfers them to main memory. Of course, someone has to write the instructions in the first place, which we call "programming." So simply put, programming is the act of writing instructions for a computer to do some specific task. In this chapter, we will explore the different types of programming languages you can use, along with the advancements that have been developed over the years to make programming easier.

One of the most interesting aspects of computer science and software in general is how we are continuously inventing new programming languages. In the end, they are all converted to machine language appropriate for the ISA of a given CPU, but how this process is done varies. For example, in some cases the program is converted to machine language once. In other cases, the program may be converted every time it is about to be executed, in which case you need a program that does the conversion on the fly.

Deciding which programming language to use can be daunting when you look at the landscape of possibilities, and it's not always a black-and-white decision; often it comes down to personal preference.

Let's jump into the fundamentals of programming languages so you have a grounding in the basic concepts that are shared by almost all languages. There are entire books written about a single programming language, so we will touch on the basics and give you some good references for learning more.

© Paul D. Crutcher, Neeraj Kumar Singh, and Peter Tiegs 2021
P. D. Crutcher et al., *Essential Computer Science*,
https://doi.org/10.1007/978-1-4842-7107-0_2

Programming Language Fundamentals

It is possible to program a computer using the computer's native machine language. However, machine language is essentially a stream of binary numbers, which are difficult to read and extremely difficult to write. Listing 2-1 shows the machine language in hexadecimal format for a simple program. Can you tell what it's doing?

Listing 2-1. Machine Language for a Simple Program

```
Address     Instruction
00000098    B800000000
0000009D    B904000000
000000A2    BE00000000
000000A7    BF00000000
000000AC    6AF5
000000AE    E800000000
000000B3    6A00
000000B5    6800000000
000000BA    6A0C
000000BC    6800000000
000000C1    50
000000C2    E800000000
000000C7    6A00
000000C9    E800000000
```

No? That's not surprising! Obviously, we need a better way to program the computer, and that's where programming languages come into play. One of the first languages developed is called "assembly language." Assembly language is very close to machine language in terms of the instructions and syntax, so it is referred to as a "low-level" language.

Hello, World!

When you are learning a new programming language, it's common practice to write a program that prints "Hello, World" to the screen. This will enable you to understand the minimal amount of work you have to do to get the program to compile and output a message. Knowing how to output a message from your program is important because you may need to print messages from your program to help you debug it when it isn't working as intended. Let's look at printing "Hello, World" using assembly language in Listing 2-2.

Listing 2-2. "Hello, World" Using Assembly Language

```
STD_OUTPUT_HANDLE equ -11
NULL    equ 0

global  main
extern ExitProcess, GetStdHandle, WriteConsoleA

section .data
hello db "Hello, World", 0
hellol equ $ - hello

section .bss
dummy   resd 1

section .text
main:
        mov eax, 0
        mov ecx, 4
        mov esi, 0
        mov edi, 0

        push    STD_OUTPUT_HANDLE
        call    GetStdHandle
        push    NULL
        push    dummy
        push    hellol
        push    hello
        push    eax
        call    WriteConsoleA

        push NULL
        call ExitProcess
```

There's a lot going on in this example! You can see it uses a variable to represent a memory location (e.g., "`hello`"), specifies blocks of data (e.g., "`section .data`") and code (e.g., "`section .text`"), uses a label to represent the memory address of the start of the program (e.g., "`main:`"), and also leverages Windows operating system routines (e.g., "`GetStdHandle`," "`WriteConsoleA`," "`ExitProcess`"). There's also a section called "`section .bss`" where you declare variables that should be initialized to 0. This is obviously easier to read than raw machine language, as you can see, but it is structured in a particular way. Can you guess why that is?

Since the example isn't in machine language, the CPU can't execute the instructions directly. We need a special program called a compiler to convert the assembly language code into machine language.

Compile, Link, and Load

Unlike the machine language example that was dumped from memory, the assembly language example is text that you must save to storage as a file. The instructions in the file need to be converted to machine language and put into memory so the CPU can execute them. As depicted in Figure 2-1, this process is typically broken down into three phases: compile, link, and load.

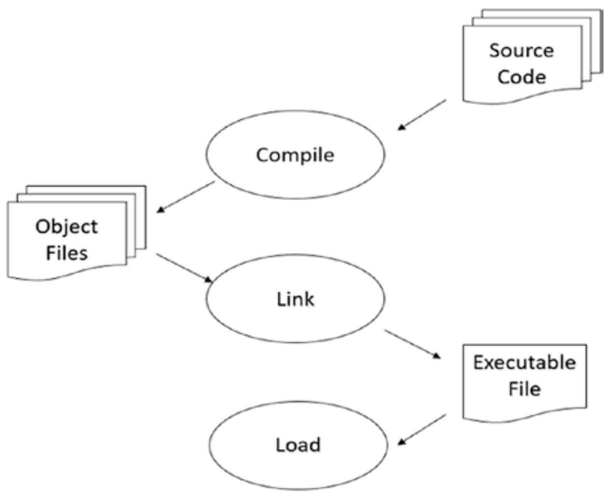


Figure 2-1. *Compile, Link, Load*

A compiler is a program that handles the task of taking the assembly instructions and converting them to machine language. The compiler verifies the syntax of the code relative to the language it is written in and generates machine language instructions for the CPU it will execute on. Much of the syntax in the assembly language example is there for the sake of the compiler so it can generate the appropriate machine code, for example, having to distinguish between data and code using "`section .data`" and "`section .text`," respectively. The compiler output will contain the values for global variables that are initialized to specific values (like "`hello`"), the code, a list of variables that should be initialized to 0, and references to functions that the compiler expects to come from some other source, like the output from another compiler or from the operating system. The binary files that the compiler generates are called object files. In Linux, object files have a "`.o`" extension, while Microsoft Windows object files have a "`.obj`" file extension.

A program called a linker takes multiple compiled object files and puts them together to create an executable program that can be loaded by a particular operating system. The linker's job is to make sure all the references in the object files are resolved before generating the executable program. It's common to get errors when linking a program typically indicating an incorrect or missing reference to a function or variable that you expected to import from some other source.

The ability to link object files from different sources together is powerful because it enables us to share and reuse code modules. When you create code that you want to reuse in multiple programs, you can have the compiler generate the object file and then use that object file when linking to other programs. We call this type of code a "static library." You can use different programming languages to generate them as long as the machine language they generate is compatible. For example, you could have two languages with a complier for each; the compilers need to use compatible mechanisms for passing parameters to functions on the stack so the code in their object files can call functions in other object files.

Static libraries are great for reusing code, but they have one drawback. If you update a static library because you've added functionality or fixed a problem in the code, you need to recreate the executable file for all the programs you've written that use that static library. Dynamic libraries were invented to fix this problem. You need to use special operating system calls to load dynamic libraries instead of linking the machine code directly into your program. This means you can update the dynamic library without having to recompile your original program – as long as the interfaces to the functions don't change! For now, just know that there are two types of libraries and that using dynamic libraries is a powerful, yet tricky, mechanism for reusing code.

Operating systems, like programming languages, are designed to make it easier to write programs. In the assembly language example, there are routines you can call to do work for you, like writing information to the console using the Microsoft Windows `WriteConsoleA` function. Another service the operating system provides is loading and executing your program. The operating system needs to know a few things about your program, like which part of it holds data (variables and default values), which part has instructions, and which instruction should be executed first. It will then put the data and instructions in memory and update the instructions to use appropriate memory locations. The operating system has a special program called a “loader” that handles this process. The loader expects the program to be stored as a file on a media device, like a hard drive, in a specific format, called the “executable file format.” There are several executable file formats that have been developed over time, such as the Executable and Linkable Format (ELF), which is used by Linux (and many other operating systems). Microsoft Windows uses the Portable Executable (PE) format.

Separating the process into compiling, linking, loading, and executing phases is very flexible. For example, you could write compilers for many different languages that target the same linker. The compiler focuses on converting the intermediate instruction format to different types of instruction set architectures. It can also optimize the instructions for those specific architectures and create specific executable file formats. Having a program that has a specific output format that another program can work with is a very important concept in programming. Imagine how much more work it would be if every time someone came up with a new programming language, they had to write the compiler and linker and the executable file format, as well as load it and execute it! By breaking this process up into steps, it saves a lot of time and enables sharing of code between programs.

High-Level Languages

Let’s compare our “Hello, World” assembly language example to an example written in the relatively old but popular “C” language. C became popular in the 1980s after Brian Kernighan and Dennis Ritchie published their edition of C in 1978. Their version included the standard input/output library, additional data types, and compound assignment operators. The following sample in Listing 2-3 is a simple "Hello, World" program in C, and as you can see, it is very different than assembly language!

Listing 2-3. "Hello, World" in the C Programming Language

```
#include <stdio.h>
int main() {
   printf("Hello, World!");
   return 0;
}
```

There are special keywords that you use so that the compiler can do its job. For example, the "`#include`" keyword tells the compiler to include another file, in this case "`stdio.h`," which is the C standard input/output library header file. Header files are used separate from the code files (which typically end in .c for the C language). They allow the compiler to understand how to call functions in other libraries without having to look at the code itself. The header file lists the names and parameters for functions that are available for use from the code file (as well as variable names and macro definitions). The brackets "`<`" and "`>`" tell the compiler to look for that file outside of the current folder by using the "include path," which is an operating system environment variable that we won't cover here. Every executable program in C must have a function called "`main`." Brackets ("`{`", "`}`") are used to group lines of code together. "`printf`" is a function that is defined in "`stdio.h`" that prints data to the screen. Parameters to the `printf` function are specified inside parentheses. A semicolon is used to specify the end of a string of commands.

The use of parentheses, brackets, and semicolons is all part of the C language syntax. The syntax is the rules for combining language-specific symbols in the correct order that the compiler will be able to understand.

Remember, the compiler is just another program, so strict rules are necessary to make it easier to convert the language into machine language code through procedural programming mechanisms as we're describing here.

Let's take a deeper look at the compilation process for a high-level language like C. Figure 2-2 shows how a compiler breaks down the compilation process in terms of preprocessing, lexical analysis, parsing, building a symbol table, and generating the code.

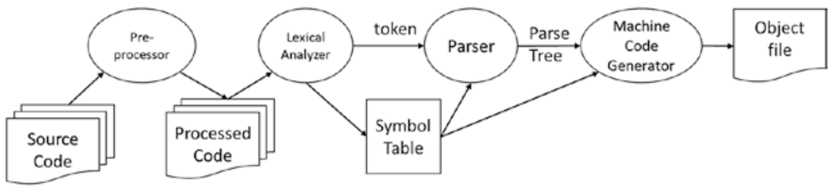


Figure 2-2. *Compilation Process*

The preprocessor looks for specific identifiers in the source code and resolves those to create a file that can be scanned by the next step in the process. In the C language, preprocessor directives start with "#", for example, "#include." The preprocessor will load the file specified by the #include so it becomes part of the source file.

Once the preprocessing is done, the lexical analyzer scans the source file to identify tokens. As it is identifying tokens (e.g., "int" is a keyword, "main" is an identifier, etc.), it updates the symbol table. If there are characters that are not allowed based on the syntax of the program, the lexical analyzer will throw an error. An advanced analyzer may try to recover from the error so it can continue with the compilation process.

The parser does the syntax analysis. It receives the tokens from the lexical analyzer and determines if they are in the appropriate order based on the syntax of the language. Parsers may generate what's called a "parse tree" or an "abstract syntax tree." The parse tree (trees are discussed in Chapter 3) is a representation of the input that conforms to the grammar of the language, and you can generate a version of the original source by walking the tree in the right order. Having a tree-oriented representation of the source code allows the compiler to make multiple passes when generating the machine language without having to reparse the original source. You could also imagine a compiler that creates the parse tree and then uses that to generate multiple output files for different types of processors without having to retokenize and parse the original source code. The parser is also responsible for detecting and reporting syntactical errors (like missing a semicolon), semantic errors (like trying to assign the wrong type of value to a variable), and logical errors (like an infinite loop or unreachable code). Compiling high-level languages is a complex topic, so if you're interested in more detail, we encourage you to read some of the references we've cited at the end of this chapter.

Hopefully you are starting to appreciate why C is considered a high-level language as compared to assembly and machine languages! Since the early 1950s, many high-level programming languages have been created. Fortran, LISP (List Processing), Algol, COBOL (Common Business-Oriented Language), BASIC, Pascal, Smalltalk, SQL (Structured Query Language), Objective-C, C++, Perl, Java, Python, Visual Basic, R, Java, PHP, Ruby, JavaScript, Delphi, C#, Scala, Groovy, Go, PowerShell, and Swift are a few of the more popular languages. Once you understand several of the key programming paradigms, you'll see how many of these languages have quite a bit in common.

Programming Paradigms

So far, we've looked at machine language, assembly language, and C "Hello, World" examples (you may have guessed by now that the first machine language example was a version of "Hello, World"). We recognize that machine language and assembly language are low-level programming languages, and we know how programs are compiled, linked, and loaded. High-level programming languages abstract away the machine language entirely, and the compilation process is broken down into several phases.

Within the classification of high-level programming languages, though, there are several different programming paradigms you should be aware of: imperative, declarative, object-oriented, interpreted, parallel, and machine learning (ML). Learning about these programming paradigms helps you recognize the common elements of many high-level programming languages. Let's take a closer look at each one.

Imperative Programming

Imperative programming is the oldest programming paradigm. Imperative program languages are constructed through a series of well-defined commands in a specific order, and the program flow is controlled by loops and branches. Imperative programs can be broken down into additional programming styles: structured, procedural, and modular.

Structured programming adds sequences, selection, and iteration operations to solve problems with nonstructured imperative programs. Procedural programming is when you divide the program into a small set of procedures, or functions, while modular programming is where you break down the program into a set of modules (files) that can be tested independently of each other.

Imperative programming is typically easier to read and relatively easier to learn because you can easily follow the execution flow, which is why most people learn an imperative programming language first. However, the programs are often much larger, relative to other paradigms, when trying to solve more complex problems. Some alternatives, like functional programming, which is considered a declarative programming paradigm, can do a lot more with less code but are typically harder to learn and read.

Most of the examples we've studied so far have been imperative, so we won't revisit them here. The C programming language is considered an imperative programming language, as well as COBOL, Pascal, Fortran, and many others.

Declarative Programming

With declarative programming, instead of programming based on the steps you go through to arrive at the solution, the program is written by describing the end result. It's also done at a higher level of abstraction. Functional programming is a common type of declarative programming.

In functional programming, the primary rule is that a function has no side effects. It cannot rely on data outside of the function; it can only operate on the data passed to it as parameters. Here's an example of an imperative programming function that violates that rule:

```
int a = 0;

void increment() {
    a = a + 1;
}
```

In this simple example, the `increment` function takes no arguments, and it is incrementing a variable that is declared outside of the function. This is a valid function in an imperative language like C, but if you're adhering to functional programming rules, you would implement the function this way:

```
int increment( int a ) {
    return a + 1;
}
```

This "increment" example is considered a "pure" function because it only operates on its parameters and thus there can be no side effects like setting the value of a variable outside of the function, and it doesn't keep track of anything between calls. It simply operates on the parameters that are passed to it and nothing else.

Another type of function is one that takes other functions as parameters or returns a function as a result. These are called "higher-order" functions. Consider the following Python code that prints the length of each string in a list. The map function takes a function name as the first parameter and a list of objects (we cover object-oriented programming in the next section) as the second parameter. It simply applies the function to each object in the list and returns the result as a special type of object called an iterator. You then pass the iterator object, which will walk through all of the elements in the data structure, from the map function to a list function to create a list of objects:

```
print( list( map( len, ["programming", "is", "fun"] ) ) )
```

The output looks like this:

```
[11, 2, 3]
```

Here we are able to accomplish the task in one line of code! However, it's not as easy to understand what is going on, is it? The flow of the code isn't obvious because it's about the operations you are performing on the data (in this case, a list of words). To understand it, you read the code from the inside out, so to speak, and also have to understand what the function is going to do, which isn't always obvious.

You have to think differently when writing declarative code, but it can be very powerful. For example, it is easier to execute the operations in parallel. In this case, it's possible to execute the "`len`" command for each parameter on a different CPU at the same time, which would be very fast!

Writing this code in an imperative way is much different. Let's look at the imperative version, again using Python:

```
word_lengths = [0,0,0]
word_list = ["programming", "is", "fun"]
for i in range(len(word_list)):
    word_lengths[i] = len(word_list[i])
print(list(word_lengths))
```

There are several more lines of code in this example, but it is a little bit easier to follow the flow of execution. However, since the "`for`" loop operates each command sequentially, it's not as easy for the system to execute the instructions in parallel.

Object-Oriented Programming

Object-oriented programming is an evolution of procedural programming that introduces some very important concepts such as encapsulation, abstraction, inheritance, and polymorphism.

In object-oriented programming, encapsulation is achieved by defining classes of objects. A class defines the private variables that only the methods of that class can act upon, protected variables that only derived classes can access, and public variables the functions and methods outside of the class can access. All of the code that operates on those variables is encapsulated within the class definition. Code external to the class can only use the public mechanisms to interact with an instance of the class. An instance of a class is called an object. For example, in C++, you can define a Vehicle class that has a public method for getting the capacity of the vehicle, but have private and protected properties and methods that are not visible outside of the class:

```
class Vehicle {
  private:
    int access_count = 0;
  protected:
    int capacity = 0;
  public:
    int get_capacity() {
      ++access_count;
      return capacity;
    }
};
```

In this example, the Vehicle has a private variable that increments every time “get_capacity” is executed. However, the capacity variable is set to 0 and is “protected,” not “private” like the “access_count” variable. This means classes that derive from the Vehicle class (like a car or bus) can manipulate the capacity variable but not “access_count.”

Inheritance is when you define a new “child” class based on the definition of an existing “parent” class. The child class can add additional methods and properties or override the parent implementation and/or add new functionality. We’ve defined a Vehicle class. Now let’s inherit from it to create two new classes, Car and Bus:

```
class Car: public Vehicle {
    public:
        Car() { capacity = 4; }
}

class Bus: public Vehicle {
    public:
        Bus() { capacity = 20; }
}
```

We’ve introduced a new C++ concept in this example called the “constructor.” The constructor has the same name as the class being created. The constructor is called automatically when the object is created. In this example, when you create a Car, it initializes the capacity variable to 4, but when you create a Bus, it initializes the capacity variable to 20. Note that neither class defines the capacity variable because it was defined in the Vehicle parent class. Because the Vehicle class has already specified the function to get the capacity of the vehicle, the child class doesn’t have to do anything other than initialize the variable in its constructor. When you create a Bus or Car, you can call those functions that are defined by the Vehicle class, like this:

```
Bus aBus;
int capacity = aBus.get_capacity();
```

We can use the same vehicle example to describe polymorphism, which means having many forms. When you write code that deals with instances of the `Vehicle` class, you can access the public get_capacity method. It doesn't matter if the object is a bus or car because they both inherit from the Vehicle class. The implementation of get_capacity is different, though, depending on whether or not the object is a car or bus. In this case your code is dealing with vehicles, but they can have different forms. Here's an example where we create a Bus but treat it as a Vehicle:

```
Bus aBus;
Vehicle* aVehicle = &aBus;
int capacity = aVehicle->get_capacity();
```

We declared a variable called "`aVehicle`" that is a "`Vehicle*`". That's special syntax in the C language to specify that the "`aVehicle`" variable is the memory address of another variable that inherits from the Vehicle class. I can "point" that variable at an instance of a `Bus` object, as in this example, using the "`&`" operator. The ampersand tells the compiler to use the address of `aBus` and then assign it to `aVehicle`. Later, we can change `aVehicle` to be the address of the `Car` object. This is how we enable polymorphism in C++. We write our code using the `aVehicle` variable, and depending on what address it is assigned to, it could be a `Car` or a `Bus`.

Now that we've covered the primary concepts common to object-oriented programming languages (encapsulation, inheritance, and polymorphism), we can move on to the interpreted programming paradigm.

Interpreted Programming

Instead of compiling your source code into an executable file, you can use a program called an interpreter and either type in the commands directly at a prompt or put them in a source file and have the interpreter execute it. Interpreters are able to execute the high-level code instructions as they read them instead of compiling and linking into an executable program. The interpreter itself is an executable program that reads the code and interacts with the operating system to do what the code says. Python is the interpreter for the, you guessed it, Python programming language! Let's look at an example of a "hello world" program in Python:

```
print("hello world")
```

Whoa, it's just a single line of code! However, you do have to run this example from the Python program from the command line, which will load and print a prompt (">>>") when it's ready for input, like this:

```
C:\python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC
v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> _
```

Interpreted languages like Python are very powerful. You can easily evaluate code using the interpreter and do rapid testing because you don't have to perform compile/link/load.

JavaScript is another interpreted language that is commonly executed by web browsers like Google Chrome and Microsoft Edge. Instead of running an interpreter from the Windows command line prompt or a Linux terminal, JavaScript is executed by an interpreter in the browser. The script sits on a web server waiting to get downloaded by the browser. In Figure 2-3, the browser requests a page from a web server, which is an HTML document that contains the JavaScript code. JavaScript can be embedded in HTML, or there can be a reference to a JavaScript file in the HTML file. For this example, it's embedded in the HTML file.

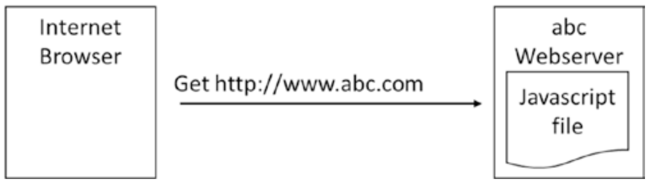


Figure 2-3. *Browser Getting a Page from a Web Server*

In Figure 2-4, the browser receives the HTML file containing the JavaScript code from the server. Now the browser has a copy of the script and can start interpreting it.

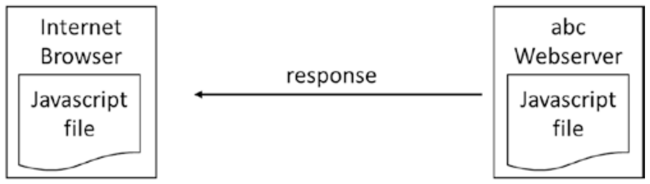


Figure 2-4. *Browser Receiving a Page from a Web Server*

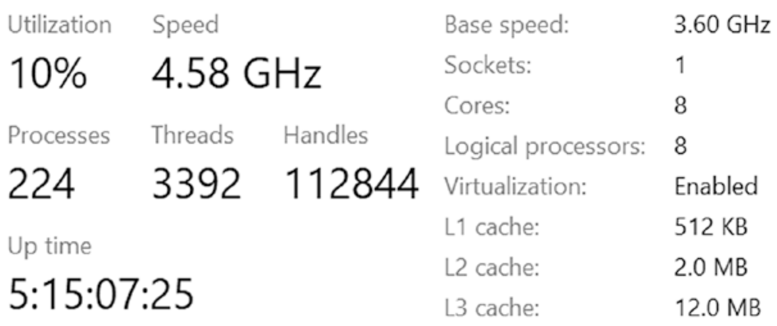
It's very important to understand the context in which your program is executing. For this example, the JavaScript code is executing in the browser, despite its origin being on the server. The script can interact with the browser itself, or it can make calls over the Internet back to the server it was downloaded from to get data or have the server do work on its behalf. This is a very different environment compared to a program that is running entirely on the same machine.

It's also important to realize that interpreted languages can also be imperative, declarative, and object-oriented. Python and JavaScript are both object-oriented and interpreted languages, as well as supporting declarative and imperative mechanisms!

Parallel Programming

Parallel programming is when your program takes advantage of multiple CPU cores or Graphics Processing Units (GPUs) to run routines at the same time. A GPU is also a type of processor, but it is designed to run parallel tasks more efficiently. As we saw previously, declarative programming lends itself well to parallel execution. Imperative programming languages add additional mechanisms to help write code that executes in parallel.

First, we should understand how processes work and how they're scheduled by the operating system to run on a CPU. When your program is loaded by the operating system, it's launched as a new process. Your program uses CPU registers and accesses memory as it executes instructions, and there's also the stack that it uses to keep track of function parameters, local variables, and return addresses. If there's just one process running on the system, these resources are used exclusively by that one process, but rarely is that the case. In an operating system like Linux or Windows, there are almost always many more processes loaded into memory than the number of CPU cores that can execute them. Figure 2-5 is a screenshot of the Windows Task Manager's CPU performance screen. On this one machine, there are 225 processes loaded, but only eight cores!



Utilization	Speed	
10%	4.58 GHz	
Processes	**Threads**	**Handles**
224	3392	112844
Up time		
5:15:07:25		

Base speed:	3.60 GHz
Sockets:	1
Cores:	8
Logical processors:	8
Virtualization:	Enabled
L1 cache:	512 KB
L2 cache:	2.0 MB
L3 cache:	12.0 MB

Figure 2-5. *Windows Task Manager CPU Performance Information*

The operating system is responsible for scheduling all of these processes on the different cores. In this case, it's possible to run up to eight processes at the same time, one on each core, but we likely need to give CPU time to more than eight processes to keep all aspects of the system running smoothly. The operating system has to use a technique called time slicing to give additional processes CPU time. In short, the operating system initializes the CPU to run a specific process by saving and restoring register values so that the process doesn't need to know it's being time-sliced. The operating system sets a timer on the CPU that will execute the scheduling code when it goes off. Because the operating system is handling this in the background, you don't really need to worry about what the operating system is doing to make this work.

The trick to hide the complexity of process switching from the process itself is memory mapping. With memory mapping, the process thinks it has access to all of physical memory, but in reality, the CPU translates the memory addresses that the process is referencing into physical addresses. Because the program is not using actual physical addresses, the memory that the program references is called "virtual memory." By using virtual memory, the process can assume its stack grows down from the top of memory at the same address every time it executes, but in reality, it is in different pages of physical memory. When the OS switches between processes, it needs to adjust the memory mapping. This is an expensive operation because the CPU has internal buffers that keep track of the mapping so that it happens very quickly. These buffers need to be flushed and get reinitialized when the process switch happens. Thus, a process will suffer a brief performance hit after a process is scheduled to start running.

Threads, on the other hand, are associated with one process and are faster to switch between than processes because the virtual memory map doesn't have to change. Figure 2-6 shows the relationship between a process and its threads.

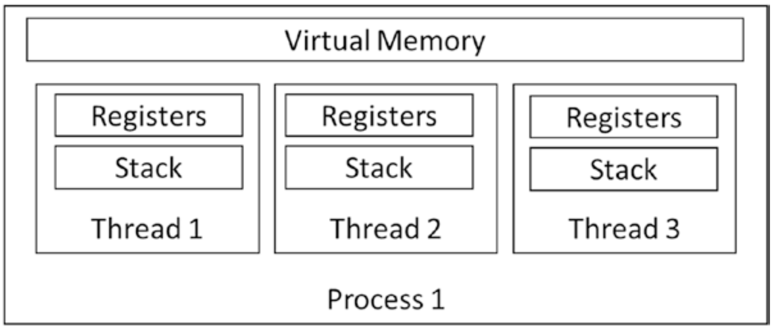


Figure 2-6. *A Process and Its Threads*

When using threads, you need to be very careful since they share resources, memory in particular, with other threads running in the same process. You can run into situations where two threads are trying to change the same memory location and overwrite values in unpredictable ways. To avoid these problems, you have to use techniques like locks and semaphores, discussed later in Chapter 4. You also have to be careful that your threads aren't waiting on each other, in which case they will wait forever, which is called a "deadlock." Writing multi-threaded programs is one of the most difficult programming techniques, but is extremely powerful if you get it right!

Machine Learning

Machine learning is a totally different programing paradigm. Instead of focusing on the flow of the program or writing functions, the computer learns from experience so that it can make predictions in the future. Machine learning is so fundamentally different than other programming paradigms that we decided to devote Chapter 8 to cover it in detail.

Summary

In this chapter, we learned that assembly language was one of the first programming languages. Assembly language introduced some key concepts like using variable names to represent memory locations. A process called compiling and linking is used to create executable programs. The operating system loads executable programs, so they are created in a format that the operating system understands. Operating systems make writing programs much easier by providing services, such as writing to the screen and loading your program into memory. There are many different types of programming techniques you can use to program the computer. We briefly covered imperative, declarative, object-oriented, interpreted and parallel programming.

References and Further Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc. 1986, 2006
- Keith D. Cooper, Linda Torczon. *Engineering a Compiler (Second Edition)*. Morgan Kaufmann, 2011
- John R Levine. *Linkers and Loaders (First Edition)*. Morgan Kaufmann, 1999
- Donald Knuth. *The Art of Computer Programming, Volume 1*. Addison-Wesley, 1968

- Mary Rose Cook. "A practical introduction to functional programming." Publish date not known, retrieved March 2021 <https://maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming>
- Brian Kernighan, Dennis Ritchie. *The* C *Programming Language (Second Edition)*. Pearson, 1988
- Mark Lutz. *Programming Python (Third Edition)*. O'Reilly Media, 2006