Computer software architecture

Bryan Ruddy

31 May 2022

In order to employ computer hardware for useful purposes, it needs to be equipped with software that can work with that hardware. Except when working with the very simplest microcontroller systems, it would be a hopelessly enormous job to write all of the required software from scratch. Instead, the software applications we write are supported by a range of other software, which taken together forms the software part of a computer system.

After completing this module, you should be able to identify the main software elements of a computer system along with their functions, understand the ways in which computing resources can be organized and shared, and understand where to look for information needed to develop software as part of a computer system.

The challenge

Computer hardware presents the programmer with a set of capabilities: one or more processor instruction sets to perform calculations and make decisions, places to store information, ways to send information, ways to gather data, and ways to create physical changes in the world. However, all of these capabilities exist at only a very basic level. The hardware in a computer monitor lets you set the colour and intensity of individual pixels, but it's up to the software to set these pixels in ways that make sense to human eyes. A hard drive can store a collection of trillions of binary bits, but it's up to the software to organize those bits into meaningful data. A processor can perform hundreds of different instructions, but few useful problems can be solved without re-framing them and expanding them to be in terms of those instructions alone.

In a very simple computer system, like a microcontroller in a car that measures acceleration and decides whether to inflate an airbag, you may be able to do all these jobs yourself while writing your code directly in assembly language. However, this quickly becomes a tedious waste of your time—why should you write your own code to convert text data into pixel values for display, when nearly every computer needs to do the same job? Furthermore, we usually need to share data with software written by others—how could you expect your computer's storage to be usable if every programmer involved in writing code for it organized the bits in it in their own, different way?

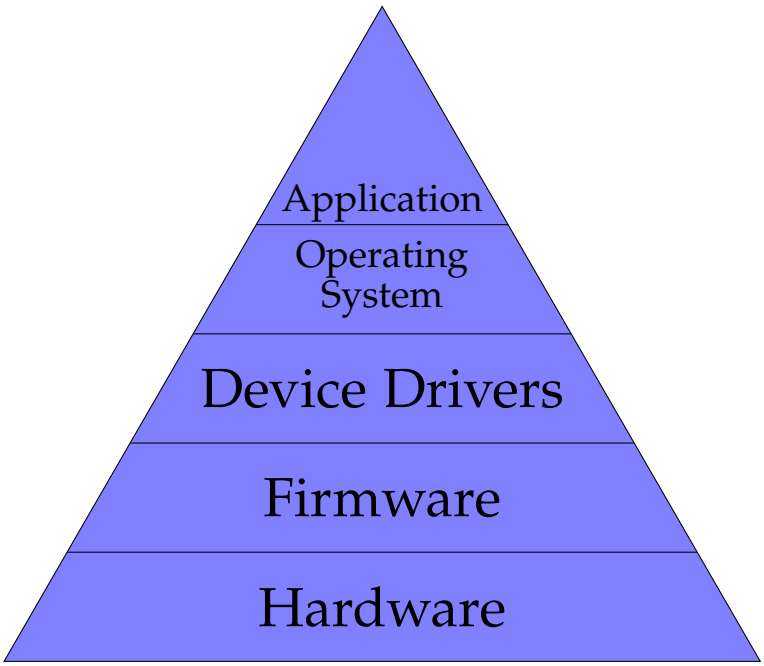
As you can see, there are two big problems that need to be solved to enable the use of complex software and complex computers. The first is simple efficiency; we can do much more if we can re-use code written by others to do a job we need done, rather

than writing it ourselves. The second is standardization; when we all agree to do things the same way we can work together without conflicts and re-use more code for further gains in efficiency.

Copy-and-paste can be a simple and tempting alternative to writing code ourselves, but this is a risky approach that is prone to errors and can expose you and your employer to legal risk—this is not the way!

Software layers

In order to make sense of complex software, we need to make use of abstraction. As you have seen earlier in this course, abstraction is vital in reducing huge systems down to pieces a single person can understand and work with. In the case of whole computer systems, we use abstraction to divide the software into layers that run atop the hardware. Each layer need only interface with the layers above and below it in the stack. A computer system does not need to include distinct pieces of software for each layer; for instance, a small *embedded system* like the BBC micro:bit may combine the firmware, device drivers, operating system, and application software into a single program, though we still may be able to identify different parts of that program with different layers of software.



The general software hierarchy in a computer system.

Application software

Application software is the reason we have a computer in the first place—it is the software that performs some desired task, whether that be a computation, something involving the organization of data, or some physical activity. Computers may be general-purpose, designed to run many different kinds of applications, or special-purpose and focused on bringing both the right hardware and the

right software to bear on a particular problem. Regardless, it is the computer systems engineer's job to ensure that the application software can be developed without its developers requiring detailed knowledge of the computer hardware. The lower layers of software present in the stack help to create this scenario.

However, there are some common problems that must be solved within the application software, regardless of the operating system or other software layers. Often the approaches to solving these problems depend on the programming language used to write the application, but sometimes the language doesn't offer the answer.

Programming languages

At this point in your academic careers, you have learned at least three different programming languages: C and Matlab, in ENGGEN 131, and Python in this course. You will have seen that, while each language offers many similar capabilities, there are other ways in which they are very different to work with.

There are two main kinds of programming language. The first kind is a *compiled* language, where the code you write is processed by a program called a compiler that translates it to machine code, which can be stored for later use and transferred to other computers. C is perhaps the most common compiled language. The other is an *interpreted* language, where the code you write is translated to machine code every time you run it by a program called an interpreter. Matlab and Python are examples of interpreted languages. Programs written in interpreted languages are generally slower to run, but offer more flexibility by eliminating the compilation step.

Regardless of the programming language type, most languages provide a *standard library* that can accomplish common tasks for you, with minimal coding required. Python has a particularly extensive standard library, as does Matlab, but even the C standard library can be very helpful. These standard libraries represent a basic form of code re-use.

Memory management

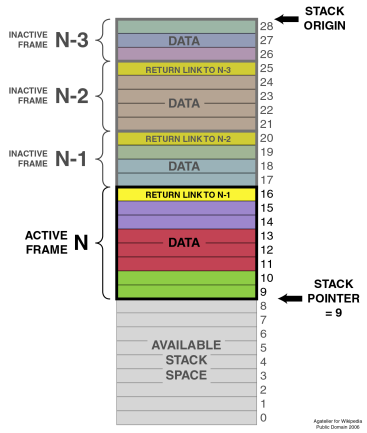
Your applications are responsible for managing their own memory. (The operating system gives out memory for applications to use, but doesn't interfere with how they manage it.) There are three main approaches to memory management within applications: static, stack, and heap. Static memory allocation, the simplest form, only applies to programs written in compiled programming languages. Here, the memory requirements are determined at the time the program is compiled, and no memory management takes place while the program is running. This is very simple and reliable, but can be inflexible and difficult to work with. It is typically used for global variables and objects, in conjunction with other memory management approaches for other data.

Python, being an interpreted language, cannot perform static memory allocation.

Stack memory allocation determines memory requirements for

each function call, and allocates memory in a continuous block that grows with each function call. Each function call has its own *stack frame*, and code within that function typically cannot access any data from other stack frames. Each stack frame also stores the starting point of the previous stack frame, to allow for simple restoration of state when the function returns. Stack memory allocation is simple and straightforward, but has a few limitations. First, because each function call gets its own stack frame, functions that use recursion can consume very large amounts of memory. Second, the total amount of space available for the stack is fixed, so if this runs out the stack will overflow and the program will crash or corrupt data.

Heap memory allocation is the most flexible form, and allows for fully dynamic memory allocation as the program runs. Arrays can change size, objects can be created and destroyed, and other complex activities are possible. To do this, a program requires a *memory manager* to track the usage of blocks of memory, and to make memory available as needed. Memory managers typically form part of the standard library in a programming language. In C, a very simple memory manager is typically used that requires programs to manually allocate and de-allocate memory as needed. In Python, the interpreter provides a memory manager that handles memory allocation and de-allocation automatically behind the scenes, and the standard library includes tools for controlling the memory manager. Memory is freed using a method called *garbage collection*, whereby the interpreter keeps track of whether any references to a variable in memory exist, and de-allocates the memory when there are no references to it. This is easy to use, but can be problematic if you use complex linked lists and allow variables to contain circular referewnces to each other.



Stack memory organization. [1]

Firmware

Firmware is the software layer that lives closest to the hardware: it typically runs on a set of dedicated processors to make storage, data transfer, and I/O hardware devices function. The firmware running on these devices then provides a standardized interface that the main processor can use to operate them.

In one common appearance of firmware, those with Windows PCs may have had to interact with the BIOS, the *basic input/output system*, either when building a computer from parts or when something has gone wrong. The BIOS is responsible for coordinating the start-up of all of the computer hardware components, and for loading enough of the operating system from long-term storage to RAM so that it can begin executing on the main processor. It also can directly operate key pieces of hardware (keyboard, display, storage, networking) to assist with the set-up process.

New PCs use a system called UEFI (unified extensible firmware interface), though the concept is similar.

Device drivers

Device drivers are pieces of software that run on a main processor and operate a specific hardware device. Their purpose is to translate between the communications interface used by the firmware on the device, which is likely to be specific to that kind of device and even to the specific hardware version, and a standard set of functions that can be accessed by other software without regard for the specific details of the hardware. In many cases, this also includes the implementation of complex behaviour that will be activated through the standard interface. For instance, a display driver doesn't just command the display to set particular pixels to particular colours and brightnesses, it may also include functions that take text input and generate the correct pixel patterns to show the text characters on the screen.

Operating systems

Operating systems (OSs) are collections of software that manage the general access to hardware resources by application software, and also provide a variety of standard interfaces and services for use by application software. Operating systems can be designed to operate a single piece of application software at a time (*single-tasking*) or to permit several applications to run simultaneously (*multi-tasking*). They can also be designed to enforce boundaries between multiple users on a computer, either while allowing one user to work at a time (a typical desktop or mobile phone OS) or while allowing multiple users to work simultaneously (a *server* OS). The hardware resources they control can all be closely connected to a single main processor or group of processors, or they can even belong to a number of physical computers connected together in a data network (a *distributed* OS).

As you can imagine, operating systems span a wide range of complexities, from a simple single-tasking *embedded* OS that runs a single application on a dedicated piece of hardware through to complex distributed OS coordinating many processors and serving many simultaneous users, each running multiple applications simultaneously. Regardless, they are all designed to solve the same basic problems around resource management. The core problem is that each individual piece of hardware can only do one thing at a time, and even that only when it is ready to do it. The OS sits between the application and the hardware, directing the needs of each application to the best and most available hardware resource and blocking application execution when a necessary resource is unavailable.

Sharing the processor

Perhaps the most obvious piece of hardware that needs management is the processor; each processor core can only run one se-

quence of instructions at a time. In a single-tasking operating system, typically used on a computer with a single processor core, the user application is in control, and the operating system code runs only when required or when no user application is running. In *cooperative multi-tasking*, the operating system has a standard mechanism whereby user applications can incorporate waiting points, where they pause and allow the operating system to pass execution on to a different application. This system, more common up to the 1990s but still used in some contexts, depends on user applications to include enough pauses to share well, and to not accidentally incorporate infinite loops that freeze the whole computer.

Some processors can be run in such a way as to appear to the operating system as twice as many cores as there are physical cores, using clever hardware tricks, but the general idea holds.

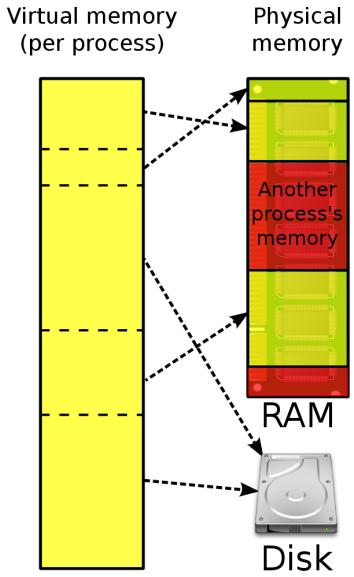
In *preemptive multi-tasking*, the operating system can forcibly pause the execution of a user application in order to pass the processor on to a different program, saving the first application's state until it next is given a turn on the processor. Most processors include a special feature, called an *interrupt*, to facilitate this process. The operating system includes a mechanism to schedule processor time among the applications running, either based on slices of time or based on input or output events; in a *real-time* OS, this scheduler guarantees that a program has the time required to process an input before it receives a new input. Preemptive multi-tasking is the standard for modern operating systems; even when only one user application is running, the operating system itself runs tasks in the background for maintenance or to process input and output. A preemptive multi-tasking OS can also allow for multi-threading, where individual applications can perform parallel processing across several cores while otherwise sharing resources.

Python uses cooperative multi-tasking via the async/await mechanism, which you may have seen when using the micro:bit simulator.

You explored the benefits of multi-threading in the performance laboratory exercise.

Sharing the RAM

RAM, the storage device with the best performance for the data being actively processed by an application, is another limited resource that the operating system must help manage. Even in a single-tasking OS, the OS itself will use some memory, and if an application changed that memory the operating system would stop working. In multi-tasking operating systems, without memory management programs could interfere with each other, leading to data corruption, crashes, and security problems.



Virtual memory organization. [2]

Most modern operating systems organize the RAM into *pages*, blocks of memory each allocated to a single program. From the program's perspective, all of the available memory looks like one contiguous block. The operating system uses some special hardware features of the processor to map this *virtual memory* to discontinuous parts of the physical RAM, or even to storage on other devices (like the hard drive) if there isn't enough physical memory. Because of this mapping, it is not possible for one application to access memory pages belonging to a different application; if the application tries to access a virtual memory address beyond what it has been allocated, the operating system pauses the appli-

cation until it has allocated more pages to support the request. If the application tries to access an invalid virtual memory address, the operating system will generate a *segmentation fault* and the program will crash, without actually accessing the invalid memory. If the computer is running out of available RAM, pages that are only infrequently used may be *swapped* to a hard drive or other slow storage device to make room.

Note, however, that the operating system itself still must have access to all of the memory, and so it is still possible for a malfunction of the operating system to cause data corruption or security problems. There are many people working very hard to find ways to cause such problems, so that they can steal financial data, access passwords, or install unwanted software on your computer that can harm it (i.e. ransomware) or steal your resources (i.e. Bitcoin miners). Operating systems can take some special precautions to reduce the impact of these errors. The most basic one, supported by most modern processors, is to mark memory pages so that the processor cannot execute any of the data present in them as code.

Sharing other resources

All of the other pieces of hardware in a computer system have the same sharing problem, but the ways in which applications access them are less predictable than for the processor and RAM. For these devices, as well as any software components in the operating system that could allow multiple applications to change the same location in memory, we need to make sure that only one application at a time can have access. (This is especially true in multi-threaded applications, where the threads all can access and change the same memory.)

The most basic way to do this is through a mechanism called *mutual exclusion*: the operating system provides a special shared variable called a *semaphore* that signals whether the resource is available, and the processor provides a special *atomic test-and-set* instruction that can test a semaphore value and change it all in a single operation. When an application wants to use a resource, it uses the atomic test-and-set instruction on the semaphore, and proceeds only if that instruction actually changed the value, i.e.e that another application had not already set it. When the application is done with the resource, it restores the semaphore to its default value.

It is very difficult to make mutual exclusion work correctly on a multi-tasking computer. If two threads attempt to access the resource simultaneously, there can be a *race condition* where the computer enters an unexpected state that depends on exactly which clock cycle each thread began its access attempt. Atomic instructions can alleviate this risk, but aren't always available. It is also possible for two threads to be in a situation where each is waiting to access a resource controlled by the other, called a *deadlock*. In

this case, both threads are stuck and cannot continue, blocking resource access and program execution alike. It is almost impossible to completely eliminate deadlocks, but operating systems attempt to include mechanisms to alleviate them. The simplest approach is to require each thread to release its reserved resources before taking a new resource.

Services

In addition to managing resource sharing, operating systems also provide some common, standardized and shared services. One such service is the networking stack, which we discussed last week in detail; the network and transport layers are typically provided by the operating system, while the data link layer is provided by the firmware and device driver. Another common service is security, with operating systems providing robust encryption and authentication services, while working to prevent unauthorized access via the network.

A major service we all interact with daily is the file system. Operating systems provide standardized approaches to data organization on long-term data storage devices like hard drives, so that applications can save data and have it easily accessed on other computers and/or by other programs. Different operating systems use different file systems, though there are shared standards used by all operating systems for removable storage like USB flash drives and DVDs.

In the days of floppy disks, removable storage also used different file systems for different operating systems, which caused big problems when you needed to get data from an IBM computer running DOS to an Apple computer!

Software interfaces

How do we access all these operating system features? Well, the operating system developer needs to provide documentation for the functions they provide. This documentation is called the *specification* for an application programming interface, or API. Besides core services, the API for an operating system typically also provides common code for things like graphical user interfaces, so that all the programs running on an operating system look and act similarly. Operating systems aren't the only kind of software that provides an API, however—almost anything can provide one, including software libraries, other applications, device drivers, and even websites! This is the most effective way to solve the original problem stated at the start of these notes, that of writing applications efficiently by re-using standard code.

APIS can be designed to be accessed in a number of different ways. Some operating system APIs, like the Win32 API that lies at the heart of Microsoft Windows, are primarily intended to be accessed using languages like C or C++, and can even be used directly from assembly code. An API for a web service, like Google Drive, typically operates using specially-formatted HTTP requests and responses. In general, the API for a piece of software is usually

The tiny executable file mentioned in the notes on binary representations does just this to make a message box appear.

designed to work most closely with the language or approach that software was written in, not the language you want to use.

This situation can cloud the benefits of APIs, since it can take a lot of work to adapt one from the styles of one programming language to a different one. In Python, fortunately, most common APIs have been wrapped in Python code by other programmers so you can simply `import` them. Most do not come installed with Python itself, so you use a package management tool like Conda or pip to download them from the Internet and install them. The docstrings in Python code can be used directly as the API specification, making the information you need easy to find.

Figure sources:

[1] from `https://commons.wikimedia.org/wiki/File:ProgramCallStack2_en.png` (Public domain)

[2] By Ehamberg, from `https://commons.wikimedia.org/wiki/File:Virtual_memory.svg` (CC BY-SA 3.0)