Runtime system

From Wikipedia, the free encyclopedia

A **runtime system**, also called **run-time system**, primarily implements portions of an execution model. This is in contrast to the runtime lifecycle phase of a program, during which the runtime system is in operation. Most languages have some form of runtime system, which implements control over the order in which work that was specified in terms of the language gets performed. Over the years, the meaning of the term 'runtime system' has been expanded to include nearly any behaviors that are dynamically determined during execution.

Contents

Overview

Every programming language specifies an execution model, and many implement at least part of that model in a runtime system. One, debatable, way to define a runtime system is that any behavior that is not directly the work of a program is runtime system behavior. This definition includes as part of the runtime system things such as putting parameters onto the stack before a function call, the behavior of disk I/O, and parallel execution related behaviors.

By this definition, essentially every language has a runtime system, including compiled languages, interpreted languages, and embedded domain-specific languages. Even API invoked stand alone execution models such as Pthreads have a runtime system that is the implementation of execution model's behavior.

Most scholarly papers on runtime systems focus on the implementation details of parallel runtime systems. A notable example of a parallel runtime system is that of Cilk, a popular parallel programming model.[1] In addition, the proto-runtime toolkit was created to simplify the creation of parallel runtime systems.[2]

In addition to the execution model behavior, a runtime system may also perform support services such as type checking, debugging, or code generation and optimization.[3]

The runtime system is also the gateway by which a running program interacts with the **runtime environment**, which contains not only state values that are accessible during program execution, but also active entities that can be interacted with during program execution like disk drives and people, via keyboards. For example, environment variables are features of many operating systems, and are part of the runtime environment; a running program can access them via the runtime system. Likewise, hardware devices such as a DVD drive are active entities that a program can interact with via a runtime system.

A unique application of a runtime environment (RTE) is within an operating system (OS) that *only* allows that RTE to run, meaning from boot until power-down the entire OS is dedicated to only the application(s) running within that RTE. Any other code that tries to run or any failures in the application(s) break the RTE which breaks the OS which stops all processing and requires a re-boot. If the boot is from read-only memory, an extremely secure,

simple, single-mission system is created. For example, this an easy way to create a never-needs-patching, can-never-be-modified Internet of Things device. In this case, the IOT could not be used for other purposes (e.g. a botnet) but nor can it be patched to prevent exploiting vulnerabilities to force a reboot.

Examples

As a simple example of a basic runtime system, the runtime system of the C language is a particular set of instructions inserted into the executable image by the compiler. Among other things, these instructions manage the processor stack, create space for local variables, and copy function-call parameters onto the top of the stack. There are often no clear criteria for deciding which language behavior is considered inside the runtime system versus which behavior is "compiled". In this case, the reason that C's stack behavior is part of the runtime system, as opposed to part of a keyword of the language, is that it is systematic, maintaining the state of the stack throughout a program's execution. The systematic behavior implements the execution model of the language, as opposed to implementing semantics of particular keywords which are directly translated into code that computes results.

Another example, which illuminates the nature of a runtime system, is the case of using an application programming interface (API) to interact with a runtime system. The calls to that API look the same as calls to a regular software library, however at some point during the call the execution model changes. The runtime system implements an execution model different from that of the language the library is written in terms of. A person reading the code of a normal library would be able to understand the library's behavior by just knowing the language the library was written in. However, a person reading the code of the API that invokes a runtime system would not be able to understand the behavior of the API call just by knowing the language the call was written in. At some point, via some mechanism, the execution model stops being that of the language the call is written in and switches over to being the execution model implemented by the runtime system. For example, the trap instruction is one method of switching execution models. This difference is what distinguishes an API-invoked execution model, such as POSIX threads, from a usual software library. Both POSIX threads calls and software library calls are invoked via an API, but POSIX threads behavior cannot be understood in terms of the language of the call. Rather, POSIX threads calls bring into play an outside execution model, which is implemented by the POSIX threads runtime system (this runtime system is often the OS kernel).

Advanced features

Some compiled or interpreted languages provide an interface that allows application code to interact directly with the runtime system. An example is the `Thread` class in the Java language, which allows code (that is animated by one thread) to do things such as start and stop other threads. Normally, core aspects of a language's behavior such as task scheduling and resource management are not accessible in this fashion.

Higher-level behaviors implemented by a runtime system may include tasks such as drawing text on the screen or making an Internet connection. It is often the case that operating systems provide these kinds of behaviors as well, and when available, the runtime system is implemented as an abstraction layer that translates the invocation of the runtime system into an invocation of the operating system. This hides the complexity or variations in the services offered by different operating systems. This also implies that the OS kernel can itself be viewed as a runtime system, and that the set of OS calls that invoke OS behaviors may be viewed as interactions with a runtime system.

In the limit, the runtime system may provide services such as a P-code machine or virtual machine, that hide even the processor's instruction set. This is the approach followed by many interpreted languages such as AWK, and some languages like Java, which are meant to be compiled into some machine-independent intermediate representation code (such as bytecode). This arrangement greatly simplifies the task of language implementation and its adaptation to different machines, and improves efficiency of sophisticated language features such as

reflection. It also allows the same program to be executed on any machine without an explicit recompiling step, a feature that has become very important since the proliferation of the World Wide Web. To speed up execution, some runtime systems feature just-in-time compilation to machine code.

At the other extreme, the physical CPU itself can be viewed as an implementation of the runtime system of a specific assembly language. In this view, the execution model is implemented by the physical CPU and memory systems. As an analogy, runtime systems for higher-level languages are themselves implemented using some other languages. This creates a hierarchy of runtime systems, with the CPU itself – or actually its inner digital logic structures that determine things like program counter advancement and scheduling of instructions – acting as the lowest-level runtime system.

A modern aspect of runtime systems is parallel execution behaviors, such as the behaviors exhibited by mutex constructs in Pthreads and parallel section constructs in OpenMP. A runtime system with such parallel execution behaviors may be modularized according to the proto-runtime approach.

History

Notable early examples of runtime systems are the interpreters for BASIC and Lisp. These environments also included a garbage collector. Forth is an early example of a language that was designed to be compiled into intermediate representation code; its runtime system was a virtual machine that interpreted that code. Another popular, if theoretical, example is Donald Knuth's MIX computer.

In C and later languages that supported dynamic memory allocation, the runtime system also included a library that managed the program's memory pool.

In the object-oriented programming languages, the runtime system was often also responsible for dynamic type checking and resolving method references.

See also

- Run time (program lifecycle phase)
- Execution model
- Programming model

References

1. Blumofe, Robert D.; et al. (1995). "Cilk: An efficient multithreaded runtime system" (http://dl.acm.org/citation.cfm?id=209958). ACM.
2. Open Source Research Institute; et al. (2011). "The Proto-Runtime Toolkit" (http://opensourceresearchinstitute.org/pmwiki.php/PRT/HomePage).
3. Andrew W. Appel (May 1989). "A Runtime System" (https://users-cs.au.dk/hosc/local/LaSC-3-4-pp343-380.pdf) (PDF). Princeton University. Retrieved 2013-12-30.