Taming Complexity with Levels

Peter J. Denning
5/25/19

To tame the complexity of the kernel, we will arrange the software as a series of levels layered one on top of the next, like a stack of blocks. Each block manages a specific set of objects, such as files or processes. Each block is a small universe with its own logic that does not depend on any higher level and, although it depends on lower levels, it does not require you to understand the lower levels.

The idea of approaching complex systems as hierarchies of levels is not new. A good place to start to appreciate the power of this way of looking is with the famous 10-minute video "Powers of Ten" by Charles and Ray Eames.

http://www.eamesoffice.com/education/powers-of-ten-2/

The second video on that page is a journey through three dozen different scales of distance, starting with a family having a picnic in Chicago seen through a window 1 meter on a side. Every ten seconds, the field of view expands by a factor of 10. Around 10^{24} the entire known universe has shrunk to a single point of light.

One of the aspects of viewing the universe this way is that we can position ourselves at a particular scale and develop explanations for phenomena visible at that scale, without regard to the details of other scales. In other words, each scale is a world of its own, with its own rules, that can be studied without regard to lower or higher scales. For example, we can study traffic patterns in Chicago from a scale of kilometers without knowing how individual cars work at a scale of centimeters. Astronomers can study collisions of galaxies at a scale of 10^{21} without knowing the physics of individual stars. Chemists can study molecules at a scale of 10^{-9} without regard to the quarks and forces that hold atoms together.

In computer science we do the same thing with software. We organize its functions to work at different scales – we call them *abstractions* – and pay no attention to the inner details of lower-level functions – we call that *information hiding*. We can do this with the OS kernel, which can be viewed as 9 levels of functions, each with its own time scale.

Our map of the kernel functions is included as the file *OS_Levels.pdf*. Please refer to it frequently as we discuss the kernel throughout this book.

To put the principle of levels to work, we need to agree on some ground rules for how the levels are structured.

Abstract machines: Each level can be viewed as an abstract machine. An abstract machine is an entity that carries out defined computational operations on a particular kind of object. For example, the abstract machine for managing files – the file manager – provides a set of operations create, delete, open, close, read, and write files. It allows users to perform only those operations on files, and no other operations. Although a file appears to be a contiguous object – a linear array of bytes – it is implemented by a series of fixed-size records scattered around the disk. The details of the records are completely hidden from the user.

We can think of an abstract machine a set of software consisting of a user interface, programs for the functions visible at the interface, and internal data that keeps track of the "state" of the machine and of all the objects it manages. In addition, a machine contains invisible programs (we call them "daemons") that do internal housekeeping operations. In some cases, a machine contains hardware units that it manages as part of its function – for example, the abstract machine that provides an interface from user to external disks controls the disks.

Some of the components of an abstract machine are elements of smaller, more primitive abstract machines. For example, the abstract machine "file system", which manages user files, may implement some internal housekeeping functions, such as compacting files on the disk, as internal background threads.[1] The threads themselves are implemented on a lower level abstract machine that specializes in threads. The programmers of the file system can ignore the details of how threads are implemented and simply use the operations defined at the process manager machine interface, such as "create thread" or "delete thread".

The embedding of lower level abstract machines within higher level machines creates the level structure we have mentioned. Other ground rules:

Visible interface: A machine M has an interface I that consists of a set of functions F1,…,Fn. The visible functions are implemented as programs within the abstract machine. In operating systems we call the visible interface an API for "Application Program Interface". Some of the visible functions can actually be part of the interface of the next lower level abstract machine. An abstract machine can also hide lower level functions that do not make sense above its level.

Nesting: An abstract machine can use components from lower level machines but not from higher level machines. No part of an abstract machine can depend on anything in a higher level.

Downward call upward return: If an abstract machine M needs a function provided by a machine M', M' must be nested with M. In other words, M can call downward to a lower level machine M' but cannot call upwards. When it is done with a task, the lower level machine can return its values upwards.

Incremental testing: The software can be brought up and tested one level at a time. Once testing (or formal proof) has established that a level is working properly, we can go to work at installing the next higher level. This greatly simplifies the process of building the system and getting it to work.

[1] In operating systems we define a thread as the trace of a CPU through an instruction sequence. A process is a self-contained virtual machine containing one or more threads operating in the process's private memory. In our kernel hierarchy, processes as virtual machines are at level 8 and threads are at level 2. The file system is at level 6 and can see level 2 but not level 8.

These structure rules sometimes seem annoying. For example, when implementing a file system, we want to have directories that list all a user's files. We define directory management to be a higher-level abstract machine but not a component of files. What happens if we create a directory and want to store it in a file? At first glance this looks like a contradiction to the downward call rule: the file manager (level 6) cannot call the directory manager (level 7). How do we organize this so that the call to create a directory does not come from the file system? The answer is to have the "create directory" operation call the file manager to allocate a file to hold the directory. This works because the file manager does not worry about directory management. In operating systems, we can always find a way to organize the software to avoid upward calls.

You can see the implications of these rules in the levels map. The functions of a given level can be composed of components from lower levels. Each level manages a set of operating-system objects. Each level adds its set of objects to the repertoire managed by the whole operating system. At the last level (shell) all the components of the kernel are present.