Component-based General-purpose Operating System

Martin Děcký

Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic.

Abstract. Software components as reusable building blocks for software are becoming widely used in software engineering. However, so far there has been little attempt to apply this programming and design paradigm in the domain of general-purpose operating systems. This paper analyzes several nice properties of component systems which might be beneficial in operating system design, the obstacles which emerge when we want to use components as a building blocks of an operating system, proposes a component model which targets these obstacles, metrics to determine the rate of success of this approach and concludes with current work in this field.

Introduction

An operating system is an essential part of many larger software systems. But it is also a complex software system by itself. Several operating systems, which development process is public (i.e. Linux, FreeBSD), show some common features. The development starts with a simple (not trivial) design and the first implementation prototype, which is debugged to achieve some expected properties (especially stability and efficiency). This code is then used as a "solid ground" for a more complex elaboration of the original design, which in turn is used for code refactoring. The new code is debugged again and the iteration continues.

This approach can be called "simple design, smart code". It relies less on the quality of the initial design and more on rigorous coding and complex algorithms. The purpose of this paper is to propose a different approach, merely "smart design, simple code". Although the authors of some operating systems would say that their system uses smart design from the beginning, it is only part of the specification and the implementation follows it roughly.

Component systems have the ability to structure a software system into autonomous parts, which interact via a strictly defined interface (a black-box model). The system can be described in the architecture of the components (their binding and composition) while keeping the implementation of a single component rather simple and straightforward. This brings better encapsulation of different concerns and better manageability. Components have already been successfully used in many large enterprise systems and also in several domain-specific operating systems.

But even if a general-purpose operating system can be seen as a very important enterprise system and the benefits of components will be equally appreciated in this context, there is no working general-purpose component-based operating system. This paper describes the obstacles which have prevented such a system in the past (notably the difference between some low-level requirements of operating systems and the high-level nature of component systems) and proposes an approach how to solve them.

A *general-purpose operating system* (shortly OS) is understood in a broader sense for the purpose of this paper (than, for example, just the kernel). It is the whole software layer joining the hardware and *end-user applications*. This includes kernel, syscall API, system libraries providing process, thread and memory management, inter-process communication and synchronization, networking, distributed computing, etc. These are basically the services which are provided commonly to all end-user applications.

In the traditional view there is a line drawn between the OS and the end-user applications. With components it is possible to forget this distinction on operating system logic and business application logic and merge them into a single architecture. This will be yet another benefit of a component-based operating system.

The structure of this paper is as follows: First we look at the complexity of state-of-the-art OSes (section 2), then we consider the benefits of the component approach in more detail (section 3). We analyze the most severe obstacles of bridging component systems and operating systems (section 4) and consider a proposed *component model* dealing with these obstacles (section 5) and *component framework* as the implementation (section 6). Finally we discuss the metrics which should tell us whether the selected approach and implemented prototype was successful (section 7) and conclude with current, future and related work (section 8).

Look at OS complexity

Although the business logic of all but the most low-level computer systems is placed in end-user programs, the functionality of these programs always heavily depends on the properties of the underlying operating system. It is impossible to guarantee the stability of an end-user program running in an unstable operating system. Even achieving outstanding application performance is difficult with an ineffective operating system.

Most of the existing general-purpose operating systems are fairly complex, but this complexity is the product of iterative development process. There are many examples of misconceptions, anti-patterns in the design, some strange code pieces which no longer serve their original purpose, etc., in almost all widely used OSes. The high costs of maintainability of these systems have usually consequences like the the documentation being out-of-sync with the implementation, etc.

It is worth noting that there is a difference between the iterative process of initial development and further incremental changes to the OS (which serve the purpose of implementing new features). But even in the latter case, it is very nice to be able to implement an incremental change without doing any significant changes to the already designed and coded parts of the OS

Designing a complex system from scratch and being able to foresee all the obstacles of the implementation is possible when there is some foundation (like a component system) which allows to see the system in an "almost working" state even before writing a single line of code. Unfortunately, an operating system itself creates the foundation for other software systems and in many cases the only required interface available is the interface to the bare hardware.

Benefits of component-based OS

Let us first look on the benefits of component approach in general. Components have basically three main properties:

Architecture design: Structuring a software problem into components (a black-box model) is a good way how to design the architecture of the system without dealing with many implementation details. The definition of required and provided interfaces, bindings, cardinalities and composite components allows to check the feasibility of the design as well as several extra-functional properties of the system prior to having the implementation finished.

Encapsulation of concerns: Components serve as a natural unit of granularity of the implementation. No matter if a given component model defines the components as objects, sets of objects, methods or blocks, even a complex structure can be modeled as a composite component, thus defining clear boundaries between different parts of the system and limiting unwanted and unnecessary interaction.

Design and implementation reusability: A properly implemented component can be used repeatedly in many contexts (provided its interface fits the given purpose). A component design pattern or a component factory can be used repeatedly in the design of the system (with or without a fixed implementation), knowing its features and limitations.

All these properties guarantee that a component-based software system is fundamentally less prone to programming bugs and allows simpler formal verification of the code (because many functional and extra-functional properties can be explicitly specified by the architecture). Another general benefit is manageability. This goes beyond the initial implementation phase and affects the incremental changes (implementing new features). Again good design and encapsulation of concerns helps the code to be easily manageable and reusability limits the effort of coding similar design patterns again and again.

Aside from these nice features, which are common to all component-based software systems, there can be some OS-specific benefits, too. Perhaps the most elegant benefit would be the possibility of *automated architectural decisions*. There is no single best solution for many architectural questions concerning operating system design. For example, is it better to implement a graphic windowing subsystem in kernel or in user-space? Is it better to have file system drivers as kernel modules or as user-space servers?

The right answer to these and similar questions is highly context-specific. There are some clear reasoning rules, but most of the time there are just fuzzy trade-offs between efficiency and security, maintainability and ease of deployment, etc.

The idea behind automatic architectural decisions is to define a set of extra-functional requirements on the operating system in a certain usage context and decide the deployment of the given components in shared or separate address spaces, with different privileges and in different compositions with respect to

these requirements. Virtually the same components will then be usable in various deployment contexts and will be composed using automatically generated connectors.

Obstacles involving components in OSes

Many traditional component frameworks (i.e. Fractal [6], SOFA 2.0 [14]) are implemented primarily in Java. The choice of Java was perhaps motivated by two main reasons:

Language features: The Java language combines static and strong class typing with a natural concept of class interfaces and reflection.

Run-time features: The Java Virtual Machine provides not only the backend for many of the language features (introspection, garbage collector, threading, synchronization, etc.), but also the Java Run-time Environment, which provides many useful classes and methods ready to be used by the component framework.

Most of the general-purpose operating systems are written in C or C++. We can identify many possible reasons for this: Java did not exist in the time of the creation of the given OS, the C/C++ compilers have a longer development history and can be seen as mature, etc. Perhaps the most important reason lies in the properties of C/C++. It is a programming language which was primarily created for writing operating systems. When doing some raw hardware manipulations (e.g. tricks as explicit typecasting of bitwise-precise defined data structures, executing highly platform-dependent low-level code and unsafe pointer arithmetics or pointer generation), C and C++ have just the right means to do it (similarly as Java has the right means for implementing component frameworks).

These fundamental differences between the highly structured and safe world of typical component frameworks and sometimes very low-level and unsafe world of operating systems is the greatest barrier between them. A component model on the operating system level has to bridge these worlds.

We can choose to prefer the high-level view and implement the component framework for example in Java or .NET, having the Virtual Machine doing the interaction with the hardware. However the Virtual Machine has to be significantly reduced and changed, as the typical VM implementations rely on many services provided by the underlying operating system (and there will be no one in our case).

Another choice would be to adopt a component model which requires almost no run-time support and uses C (like Koala [4]). But as this component model is targeted on embedded devices and our goal is a general-purpose operating system, this would probably lead to a lot of complex code in the components.

We seek a good balance between safety, introspection and the possibility of low-level tricks. The framework might support features such as garbage collector, synchronization and so on, but there should be always a possibility to take full control. A programming language more or less compatible with C might be helpful in the initial phase because of easier comparison with the legacy systems.

Component model with nanokernel

The proposed component model tries to overcome the obstacles from previous section by concern separation. As already noted, some of the features of Java (and thus inherited into many component models) might be quite useful in the context of an operating system (strong typing, interfaces, introspection, garbage collector, easy synchronization, set of standard classes and methods), but in certain situations also the low-level features of C/C++ might be helpful (typecasting, pointer arithmetics, pointer generation, explicit destructors, non-standard usage of synchronization primitives). It is a matter of efficiency and maintainability to be able to use the right tools for the right task. Thus the proposed component model (schematically illustrated on Fig. 1) separates the low-level features and high-level features. In more detail:

Nanokernel: An operating system kernel running in privileged processor mode. The kernel should however provide and implement just a very limited functionality (context switching, physical memory management, spinlocks, interrupt and exception handling) which utilize the low-level features like pointer generation, etc. This set of features should be even significantly smaller than the set of features implemented by a typical microkernel. Basically even microkernels implement some complex functionality while the nanokernel should only implement atomic actions.

Despite its small size, the nanokernel should still be implemented as a properly designed OS core, in a portable way (clear separation of architecture independent and architecture dependent code). The *Hardware Abstraction Layer* is marked as **HAL** on the figure.

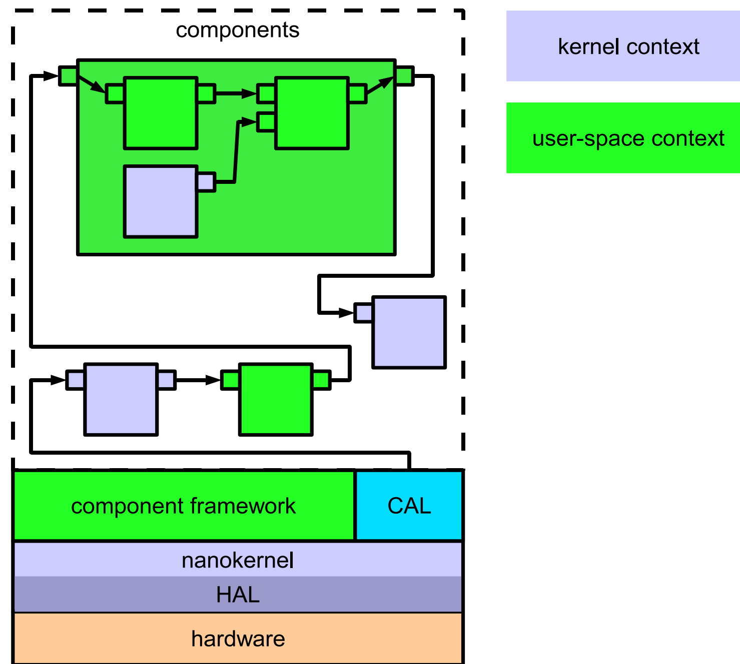

Figure 1. Overview of the component model with nanokernel.

Component framework: The framework is the run-time support for the components. It manages component bindings, composition, deployment and lifecycle. It also contains means by which the components can define scheduling of their execution, etc.
CAL: An abbreviation for *Component Abstraction Layer*. This part of the system is a glue between the nanokernel primitives, the component framework and the component space. Its purpose is to create primitive components encapsulating the basic kernel functionality (using nanokernel atomic actions) and export interfaces which can be used by other components.
Components: Components providing high-level operating system functionality (scheduling, virtual memory management, passive synchronization, thread and process management, etc.) and possibly even business functionality. There is no strict line between components running in CPU's kernel and user-space mode, but each component can run in a separate or shared address space with either kernel or user-space privileges.

The figure shows both primitive components running with kernel privileges and user-space privileges as well as an example of a composite component. The bindings of the interfaces are also shown, the interfaces on the left are required, on the right are provided. However, some other aspects which might be involved in the final framework are not shown (i.e. generated connectors).

There is no strict line between privileged and unprivileged components, there is also no strict separation of components providing core OS functionality and end-user business functionality. In fact, the model can be seen as the integration of the operating system and end-user applications. Legacy applications can still be supported by a component providing emulation layer for legacy APIs (POSIX, Win32, etc.).

Component framework

The component framework defines several properties of the component model on an implementation level. First of all, it defines the components from the programming language point of view.

A common unit of granularity of operating system design is a subsystem (i.e. memory management subsystem, process & thread management subsystem, etc.). However the boundaries of the subsystems are very vague, they do not have a strict interface between them and tend to grow into each other in the implementation. Thus subsystems as such do not have the suitable granularity of components. Each subsystem manages several entities, which are for example threads, processes and CPUs in the case of process & thread management, address mappings in the case of memory management, etc. The

granularity of these entities is on the other hand too small – they are instances of classes which are created and destroyed quite often.

There are some requirements on the components (formulated in [1]) which should be respected:

Uniform perspective: All features of the system have to be accessible via components, which should be the primary first class entities. Other first class entities can be present as well (connectors, adaptors, etc.). The component world should not be able to directly interfere with the low-level structure of the component framework or the nanokernel.
Open-endedness: There should be no line drawn between components that are implementing OS features and components that are implementing end-user business logic.
Dynamic deployment: Except for certain specific contexts (embedded use) the component model has to be dynamic, allowing not only the addition and removal of components in the architecture, but also supporting dynamic reconfiguration of the running system.

As object-oriented programming is a well-understood concept which supports (given a proper programming language and run-time support) the requirements above, the prototype component framework defines components as objects. However, the basic difference between object-oriented and component-based programmings is the fact that components do not model each single entity in the system, but arbitrary black boxes, which the system can be composed from.

Success metrics

As with any approach which prescribes how a complex software system should be designed and implemented, not only theoretical benefits and drawbacks have to be analyzed, but there has to be also a quantitative measure of the overall soundness of the new approach.

This measure serves two purposes: As an absolute value, it can tell us whether we have really achieved the goals we set earlier. A comparison of this value with the measure of more traditional approaches can tell us whether it is worth to start completely from scratch. We propose manageability metrics based on a comparison case study of initial OS development and of a set of incremental changes. The metrics itself will be then a vector consisting of several distinctive measures, from trivial ones to very sophisticated.

Let us consider a change we want to rate. One of the simplest measures would be number of written/changed lines of code, net time spent on implementing a certain incremental change and so on.

A more sophisticated measure should deal with the number and size of forced architectural changes. Finally, the most elaborate measures would have to deal with the number of anti-patterns (measured by a tool like SISSy [2]) which will be induced in the code despite very rigorous programming, the change of Halstead complexity, cyclomatic complexity, etc. Other maintainability measures ([13]) can be considered, too.

Current and future work

The current work is being done in several parallel activities. An experimental research operating system HelenOS [3] has been developed and is being enhanced to host prototypes of the component framework.

HelenOS is a research operating system created at Charles University in Prague, Faculty of Mathematics and Physics, originally as part of the Software project course and based on previous work of Jakub Jermář on SPARTAN kernel. The author of this paper is a founding HelenOS team member.

One of the several goals of HelenOS is to have a working, yet simple and easily extensible platform for operating system research. The source code of HelenOS (and SPARTAN kernel) is written from scratch and is thus not cluttered by the various "hacks" which are present in many established operating systems (i.e. Linux) and which render them almost unusable for any major architectural redesign. Another benefit of having a system written from scratch is the ability to calibrate the success metrics, both on the initial development and on any (real or artificial) incremental change.

Currently, an early prototyping of the component framework in HelenOS is in progress. Objective C has been chosen as the implementation language mostly because it is a strict superset of C, which makes it very suitable for doing the success metrics calibration (a similar incremental change can be made in the plain C variant and in the component-based variant of HelenOS). Objective C also nicely joins the high-level features and the presence of low-level C constructs as discussed earlier.

Related work

TinyOS [5], application-specific multicore SoC [7] and reconfigurable component-based OS in THINK [8] are examples of existing component-based OSes. These systems are targeted on embedded devices and industrial controllers.

However the primary goal of the approach described in this paper is to create a general-purpose desktop/server operating system, running natively on standard microcomputer hardware, with excellent maintainability and also other treasured features (i.e. source code portability, etc.). There are just two noteworthy related works in this context: *Go!* [9] is a component-based operating system designed on the principle of object-level isolation and running the code of all components in a single privileged address space (similarly as Singularity [10]). It is no longer actively developed.

The BITS operating system project [11] is an approach with some similarities with the one described in this text, but unfortunately the system appears to be only designed in theory. Some of the authors' decisions are questionable (the incorporation of as many traditional Unix concepts as possible). There has been a huge research and development of component models in the recent 10 years and we believe that with the new knowledge it is now possible to reach a working implementation.

Conclusion

The paper hints problems of the development process of many general-purpose operating systems ("simple design, smart code") and proposes a way how components can be used to improve it ("smart design, simple code"). The component approach should also improve other properties of operating systems (reusability, maintainability), but these benefits do not come for granted. Many obstacles have to be solved to bridge the low-level world of operating systems and high-level world of component systems.

The proposed approach is a component model comprising of separated low-level part (a nanokernel), high-level part (component framework and components) and a glue part (Component Abstraction Layer). A success metrics and a methodology for its calibration has been proposed.

The current work is focused on implementing the proposed model and framework as a variant of HelenOS research operating system. This is also motivated by the fact that the HelenOS system was written from scratch and thus can be easily used to calibrate the success metrics.

Acknowledgments. I would like to thank my supervisor Doc. Petr Tůma for sharing his time, knowledge and experience with me. The presented work was partially supported by the Grant Agency of the Czech Republic project GD201/05/H014.

References

[1] Yokote, Y., Teraoka, F., Tokoro, M.: A Reflective Architecture for an Object-Oriented Distributed Operating System, Sony Computer Science Laboratory Inc., Japan, 1989

[2] Trifu, M., Trifu, A.: SISSy – a Tool for Structural Investigation of Object-Oriented Softwarre Systems, FZI Forschungszentrum Informatik, Software Engineering Group, Germany, http://sissy.fzi.de/SISSy/CMS/-Documentation/SISSy_Overview.pdf, 2006

[3] HelenOS project, http://www.helenos.eu/

[4] Koala Component Model, http://www.extra.research.philips.com/SAE/koala/

[5] TinyOS community forum, http://www.tinyos.net/

[6] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: The Fractal Component Model and Its Support in Java, Software Practice and Experience, Special issue on Experiences with Auto-adaptive and Reconfigurable Systems, 36, pp. 11 – 12, 2006

[7] Cescirio, W., Baghdadi, A., Gauthier, L., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S., Jerraya, A. A., Diaz-Nava, M.: Component-based design approach for multicore SoCs, Proceedings of the 39th Design Automation Conference, pp. 789 – 794, 2002

[8] Polakovic, J., Özcan, A. E., Stefani, J.-B.: Building reconfigurable component-based OS with THINK, Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, Dubrovnik, Croatia, pp. 178 – 185, 2006

[9] Go!, http://www.doc.ic.ac.uk/˜jamm/research/go/overview.html

[10] Singularity, http://research.microsoft.com/os/singularity/

[11] BITS, http://www.soi.city.ac.uk/˜patty/bits.html

[12] Caromel, D., Delbé, C., di Costanzo, A., Leyton, M.: ProActive: an integrated platform for programming and running applications on grids and p2p systems, Computational Methods in Science and Technology, issue 12, pp. 69 – 77, 2006

[13] Welker, K. D., Oman, P. W.: Software Maintainability Metrics Models in Practice, Crosstalk, Journal of Defense Software Engineering, issue 8, pp. 19 - 23, 1995

[14] Bureš, T., Hnětynka, P., Plášil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proceddings of SERA 2006, Seattle, USA, IEEE CS, 2006