Visual OS: Design and Implementation of a Visual Framework for Learning Operating System Concepts

James H. Hill
Institute for Software Integrated Computing
Vanderbilt University
Box 1829 Station B
Nashville, TN 37235
(615) 322-8489
j.hill@vanderbilt.edu

Aniruddha S. Gokhale
Institute for Software Integrated Computing
Vanderbilt University
Box 1829 Station B
Nashville, TN 37235
(615) 322-8754
a.gokhale@vanderbilt.edu

ABSTRACT
An operating system can be described as software composed of numerous components providing a distinct functionality, such as CPU scheduling, disk scheduling, virtual memory and paging, while working together to efficiently manage the hardware and resources of a computer system. Understanding their features and interplay can be a non-trivial task, in particular, for undergraduate students in Computer Science studying operating systems. To aid in their understanding of the OS dynamics, a number of aids including textbooks, journals and simulators exist. Although these aids suffice to understand simple OS concepts, some algorithms, such as paging, synchronization and process control, and their interactions are too complex to understand without a way to visualize these interactions and operations.

This paper provides three contributions to the R&D on visualizing the dynamics of operating systems. First, we describe the design architecture of Visual OS, which is our OS visualization engine. Second, we describe how we used software design patterns to make our framework extensible to accommodate a variety of OS features that cater to different domains, such as generic computing, real-time systems and embedded systems. Finally, we describe how we have used Visual OS for a programming assignment in a senior level OS class at Vanderbilt University.

Categories and Subject Descriptors
D.4.2 **[Operating Systems]**: General. D.4.8 **[Operating Systems]**: Performance – *simulation, measurements.* D.3.3 **[Programming Languages]**: Language Constructs and Features – *abstract data types, classes and objects, frameworks, inheritance, patterns, polymorphism.*

General Terms
Design, Experimentation, Measurement, Performance

Keywords
Operating Systems, Patterns, Simulators, Frameworks, Education Tools, Visualization Tools, Visual OS, Operating System Concepts.

1. INTRODUCTION
Every professor of an operating system course has his or her methods to simplify the understanding of its concepts for undergraduate Computer Science students. Some use programming assignments that emulate OS concepts, while others use simulators that visualize the concepts. Because an operating system is composed of many components that interact with one another, and many tools have a limited scope of component interaction, we have developed Visual OS – an experimental research application aimed to aid in the teaching of operating system concepts through visualization by removing the overhead and difficulty of interacting with actual hardware, or portions of an operating system not available at user level development. This paper describes significant research enhancements to previous efforts in building educational frameworks, and how Visual OS was applied to learn various aspects operating system concepts.

The remainder of the paper is organized as follows: Section 2 describes the design architecture and challenges we faced when designing Visual OS; Section 3 describes design patterns used when implementing the Visual OS architecture, and a project for a senior level operating system course using Visual OS; Section 4 discusses related operating system simulators; and Section 5 provides concluding remarks.

2. VISUAL OS DESIGN ARCHITECTURE
In any application, designing the architecture is one of the most important parts of the development process. When designing the architecture for Visual OS, we desired several key features, which are listed below:

- The architecture should be extensible and scalable.
- The architecture should provide a common interface among components.
- The architecture should be easy to learn and operate.

- The architecture should allow run-time configuration.

We discuss each one of these desired properties and their associated technical challenges in Section 2.2 of the paper.

2.1. Architecture Design and Layout

The architecture design and layout for Visual OS is similar to an actual operating system. It is composed of a bootstrap, kernel, user interface, and several managers, e.g. process, memory, etc. The bootstrap is responsible for loading the kernel and starting the system. It also loads resources used by the kernel, such as the user interface and default managers if specified. But most importantly, the bootstrap is responsible for assembling the loaded pieces of the system so they may function properly.

As illustrated in Figure 1, there are three levels in the system. The top level is composed of the user interface, or any component used to manipulate the underlying system. The middle level is the system level. The bottom level is the component, device, and manager (CDM) level. The system level acts a liaison, or bridge, for the upper and lower levels; and contains a run-time configuration (RTC) manager for loading, configuring, and unloading objects in the lower level. The user interface is used for invoking commands upon the system. The commands are then delegated to the appropriate component, manager, or device by the system. The user interface is also used to observe the state of the system and its makeup which is achieved via callbacks and system queries.

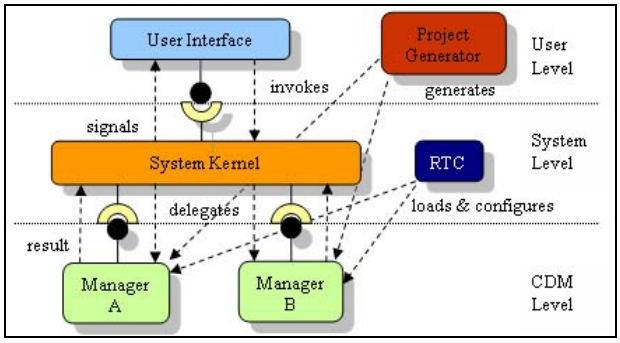

Figure 1. Three level diagram of the Visual OS architecture in its assembled form.

2.2. Technical Challenges

This section describes the challenges associated with the architecture design and layout when developing a visualization framework for studying operating system concepts.

2.2.1. Extensibility and Scalability

Designing an R&D simulator for OS concepts is a complex undertaking. In order to guarantee consistent evolution of concepts in operating systems, we needed the architecture to be scalable so new concepts, or features, could be added without breaking any existing infrastructure.

2.2.2. Common Interfaces

We wanted all components of similar type to share a common interface so we could create families of components. This is desired because we wanted to extract design patterns that occur within an operating system. More importantly, we wanted to create common interfaces that can be understood by the system, and be inherited by components.

2.2.3. Ease of Learning and Use

When designing the architecture for the system, we did not want to create one that was too complex to understand and, more importantly, to use. The users should be able to interact with the system without having to learn a complex API. In addition, the API should resemble other standard API, e.g. *malloc* and *free* from the standard C library and *CreateProcess* from the Win32 SDK.

2.2.4. Run-Time Configuration

We wanted our architecture to allow users to implement their own algorithms and visualize them without having to restart the system. We also wanted the system to allow users to unload and load strategies at run-time without having to recompile and relink the application since we would not provide the source code for Visual OS.

3. RESOLVING THE CHALLENGES IN DEVELOPING AN OS VISUALIZATION FRAMEWORK

This section describes how we addressed the technical challenges of developing a visualization framework for studying OS concepts. Our solution is based on a systematic application of software design patterns [4]. We also show how these patterns are put together to form the OS visualization framework.

3.1. Design Patterns in Visual OS

Visual OS was built using design patterns and implemented in C++. Figure 1 illustrated the assembled system after bootstrapping. Figure 2 illustrates the design pattern-oriented architecture of Visual OS shown in Figure 1. In the remainder of this section, we explain these patterns used to implement Visual OS and discuss how they resolved the technical challenges described in Section 2.2.

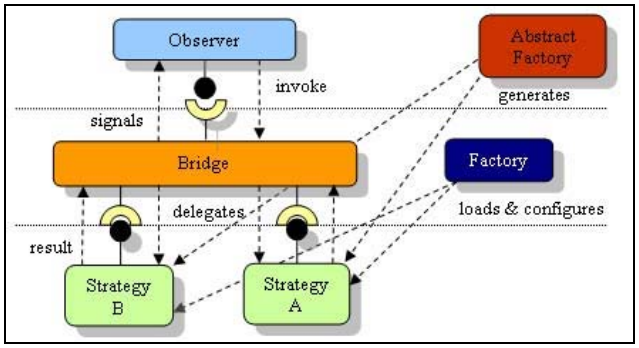

Figure 2. Design patterns in used to implement the various components Visual OS.

3.1.1. Bridge Pattern

The *bridge pattern* decouples an abstraction from its implementation so two objects can vary independently [4]. This pattern was used in the kernel of Visual OS to resolve the extensibility and scalability challenge – Section 2.2.1. As illustrated in the Figure 2, the kernel acts a bridge between the user interface and the lower level components, devices, and managers. Because the upper and lower levels communicate through the system kernel, the implementation can vary on either end. Therefore, we added a layer of abstraction and independence between the upper and lower levels simultaneously so rescaling the entire system does not destroy the current infrastructure.

3.1.2. Strategy Pattern

The *strategy pattern* defines a family of algorithms, encapsulates each one, and makes them interchangeable so they can vary independently from the clients using them [4]. This pattern was used to solve the common interface, extensibility, and ease of use challenge – Section 2.1.1, 2.2.2, & 2.2.3.

All objects in the system inherit a common base interface, and more specialized objects are derived from the common base to create subclasses. By combining component design and with subclasses, we defined specialized objects familiar to the system that provide a common interface for ease of use and extension. When we add extend the original subclasses to create newer subclasses, it will inherit a previously known interface and the system will know its base operations. In addition, when a user implements his or her original algorithm for a known subclass he or she must overload the common interface through polymorphism. When the component is imported into the system, the system knows how to manipulate it because of the common interface provided by the *strategy pattern*.

3.1.3. Factory Pattern

The *factory pattern* defines an interface for creating an object, but lets subclasses determine which class, or subclass, to instantiate [4]. This pattern was used internally in the run-time configuration to resolve the run-time configuration challenge – Section 2.2.4. When the user changes the strategy of the system, the factory within the run-time configuration is responsible for creating it for the system.

3.1.4. Observer Pattern

The *observer pattern* defines a one-to-many dependency between objects so that when one object changes states, all its dependencies are notified and updated automatically [4]. This pattern was used to resolve the ease of use challenge – Section 2.2.3. The user interface is implemented using the *observer pattern* because we wanted an efficient update method that did not make unnecessary and unwanted updates. When a component needs the user interface to update its information, it signals user interface. Therefore, users do not have to learn the logistics of manipulating the user interface. Instead, they focus primarily on implementing their algorithm.

3.1.5. Abstract Factory Pattern

The *abstract factory pattern* provides an interface for creating families of related or dependent objects without specifying their concrete class [4]. This pattern was also used to solve the ease of use challenge – Section 2.2.3. The *abstract factory pattern* was implemented in the project generator tool that is integrated into Visual OS. It allows users to generate different project types and shell code for implementing a strategy. Instead of handwriting the shell code, which can be error prone, the generator produces the bare necessities needed to implement original strategies, which is free of errors, and makes Visual OS easier to use.

3.1.6. Component Configurator Pattern

The *component configurator pattern* allows an application to link and unlink its component implementations at run-time without having to modify, recompile, or statically relink the application [7]. This pattern was used to resolve the run-time configuration challenge – Section 2.2.4. Because we chose a component based design, we already had the foundation for adding run-time configuration to the system. To add complete support for run-time configuration, we defined components in individual run-time libraries instead of static libraries[1]. Then the *component configurator pattern* was used to load and unload the components into the system.

3.2. Visual OS Undergraduate Class Project

During the Fall 2004 semester, we used Visual OS to teach the memory management portion of our undergraduate operating systems course. For three lectures, we discussed the concept of memory management and various strategies such as partitioning, paging and segmentation. The students were then assigned a programming project on the final lecture day. For the assignment, they were responsible for implementing three contiguous memory managers[2] using Visual OS.

When the students submit their work, a survey will be distributed to evaluate the programming assignment and Visual OS. It will contain questions asking the students to comment about the technical challenges we addressed when designing Visual OS. These comments will be used to further improve Visual OS, as well as define new challenges that must be addressed.

4. RELATED WORK

The power of *visualization* for teaching non-trivial concepts is well understood as evidenced by the number of projects that use visualization as a base for disseminating knowledge. Of these projects, there are some that allow users to create scripts, which the visualization tool interprets and displays the results. Although this is a good method for furthering the understanding of concepts in a subject like operating systems, students do not get

[1] Our component architecture and run-time configuration is similar to the COM/COM+™ technology created by Microsoft. We chose not to use that technology because (1) we did not need all the functionality it provides; and (2) it is not platform-independent, which was a requirement of our system.

[2] The students were responsible for implementing best-fit fixed-partition, dynamic-partition, and paging memory management schemes.

the requisite experience of implementing the learned concepts with a well known systems programming language, such as C++ [6].

To address this limitation, there are projects that attempt to incorporate a well known systems programming language by allowing users to implement their own concepts and evaluate them in a simulator. Unfortunately, these projects are *statically bound*, meaning they have predetermined state and set of tasks, and do not support run-time configuration [1] [5]. Moreover, the simulator only explores limited aspects of OS, but in reality there are many aspects – many of which have complex interactions with each other. Thus, these projects do not show how multiple artifacts, or components, of an OS coordinate with one another. Visual OS described in this paper attempts to overcome these shortcomings. Its evaluation and future enhancements are possible after we have gained sufficient insights using it in our OS class.

5. CONCLUDING REMARKS

In this paper, we have described the Visual OS architecture and design, the patterns incorporated into the system, and a class project at Vanderbilt University that uses Visual OS. As we are learning, the subject area of operating systems is vast and there will always be room for further development. In addition, as more operating system concepts want to be explored, the system will need to be expanded to meet those needs. As a result of these insights, we have defined and discussed a framework that will allow extension and scalability to occur without having any negative side-effects on the system's current infrastructure. More importantly, we have defined a framework that can be used at the undergraduate level to teach concepts that are non-trivial without the aid of a visualization tool.

6. REFERENCES

[1] Christopher, W. A., Procter, S. J., & Anderson, T. E. (1993). *The nachos instructional operating system*. Berkley, CA: University of California at Berkley.

[2] Crowley, Charles (1997). *Operating systems: A design-oriented approach*. Chicago: Irwin.

[3] Flynn, I. M. & McHoes, A. M. (2001). *Understanding operating system concepts (3rd. Ed)*. Pacific Grove, California: Wadsworth Group.

[4] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software.* Boston, MA: Addison-Wesley.

[5] Holland, D. A., Lim, A. T., & Seltzer, M. I. (2002). A new instructional operating system. *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, *33*, 111-115.

[6] Ontko, Ray (2001). *Modern operating system simulators* [Computer software and manual]. Retrieved September 10, 2004, from http://www.ontko.com/moss.

[7] Schmidt, D., Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-oriented software architecture: Vol. 2. Patterns for concurrent and networked objects.* New York: John Wiley & Sons, Ltd.

[8] Silberschatz, A., Galvin, P. B., & Gagne, G. (2003). *Operating system concepts (6th Ed.)*. Hoboken, NJ: John Wiley & Sons, Inc.