Shag/OS

A Small, Dynamic, Object-Oriented

MicroKernel-based Operating System

An independent-study project proposal

by Frank Barrus

shaggy@csh.rit.edu

September 5, 1995

History:

The original version of Shag/OS, which reached version 0.34, was developed as an independent study project for Ken Reek. It was completed in the early spring of 1992, as the result of six months of design, hardware research, development, and debugging, while taking full-time classes. It was designed to be easily portable, and was implemented on both Intel 386/486-based PC architectures and MicroVAX's.

Nearly all of the code was written in strict ANSI C, with only minimal support from assembly language. Only about one thousand lines of code varied between architectures, thus aiding its portability. That number includes the code for the basic device drivers needed. (keyboard, video, serial, and hard-drive)

Overview:

The entire operating system is based around the concept of an object-oriented microkernel, which basically handles the primitive functions of memory management, scheduling, inter-object communications, and managing a virtual namespace for identifiers and locating objects. All of these primitives can be extended to have additional functionality through the use of objects that override certain features.

All devices take the form of objects that are then loaded either into user space or kernel space. Additional objects can be layered on top of the low-level device objects to add features, or override existing functionality.

The object-oriented approach minimizes redundancy, makes the system more flexible, and provides a consistant interface to all types of devices, files, networks, and other more abstract entities.

Compact Size:

Shag/OS v0.34 had a fully functional microkernel (consisting of virtual memory management, scheduling, interprocess communication, and all low level interrupt support) of only 32K. Even with all the drivers loaded for the keyboard, video, serial I/O, and hard drive, and the basic terminal drivers and the filesystem, the code consumed less than 64K, and could run a shell and other simple programs with only an additional 200K of RAM. It had more than enough room to spare, and ran sufficiently fast, on a bottom-of-the-line 1MB 386SX-16 PC.

Shag/OS v1 should have a microkernel that is still under 32K, although an upper bound of 64K will be considered acceptable, if the extra code is deemed necessary. Complete with standard drivers, the code should consume less than 128K of RAM, and should require no more than 512K total RAM for a functional system. Therefore, a 2MB machine should be more than sufficient for most general purpose uses, although of course some applications, due to their own memory requirements, may spend a lot of time paging in that amount of memory. (such as gcc, or X-Windows, which are much larger than that)

For contrast, most UNIX's for PC's consume at least 300K for the kernel code alone, and usually need a machine with at least 4MB of RAM to even run, and 8MB or more to run efficiently.

Current Goals and Features:

For version 1.0 of Shag/OS, the entire system has been redesigned, using the experience gained from the last version to make improvements. The new version is completely in C++ (with some embedded assembly), and will have increased modularity, and hopefully a decrease in the number of bugs, due to the improved structure of the system. The following goals are expected:

- Small

  The microkernel should be under 64K (for most CISC architectures), and basic device drivers should not add much additional overhead.

- Dynamic buffering

  All unused system memory should be able to be used for buffers to increase system performance when it is not being used by applications.

- Simplicity and redundancy reduction

  Most complex objects should be broken down into smaller, simpler ones with well defined interfaces, that can be combined in numerous ways to do more complex tasks.

- Networking

- Distributed object interface

  Objects should be able to communicate with other objects without knowledge of whether the other object is local or on another machine connected by a network. This allows any application to be distributed if it is broken down into smaller objects, and the working environment can be distributed by using remote resources and having them appear local.

- Portability

  The entire system is designed with portability in mind, such that during compilation, the most machine-specific variation of each portion of code is used. A hierarchical structure allows a fallback to completely general machine-independent code whenever something more specific isn't found, while allowing for optimizing portions of the code for specific machines, or general types of architectures. Very little machine-specific code should be necessary to port the micro-kernel, although lots of machine-specific code can later be written to increase efficiency.

- Virtual Memory

  Most object communcation in Shag/OS is based around the usage of virtual memory management. In addition to allowing objects to have memory regions that are much larger than physical memory, with the least recently used portions paged out to storage devices, it is also used for dynamically remapping regions of memory to different objects when passing memory regions as parameters. All buffers are passed to other objects by the means of a memory descriptor, and when accessing that region locally, memory pages are simply shared. Over the network, other means are used to access that region, but the method is transparent to the objects communicating, keeping the interface consistent for local and remote objects.

- Dynamic Scheduling

  All timestamps are kept as 64 bit values, accurate to the nearest nanosecond. The actual accuracy will vary based on the timers that exist on each system, but this allows a consistent notion of time across a wide range of systems. Shag/OS is somewhat of a real-time system, in the sense that events can be scheduled to occur at any time interval that can be resolved by the hardware, and different scheduling algorithms can be applied to different groups of threads, depending upon their needs. The actual scheduling used by a thread can be altered at any time, either in the parameters used, or the entire algorithm. The rate at which the system clock is updated can be dynamically altered at any time while the system is running.

- Loadable Device Drivers

  Since all device drivers are just objects in the virtual namespace, they can be swapped for other objects at any time. This means that while testing, for instance, an ethernet driver, any time a new one is compiled, it can be simply loaded and tested, without rebooting the system.

• Shared Libraries

In Shag/OS v0.34, linking with shared libraries was done entirely at execution time, with a dynamic loader, that insured that there was never any version mismatch for the library. As long as the function existed, and followed the same interface as was expected, it was accessible.

Shag/OS v1.00 will use a similar method to insure that only one current version of each shared library ever needs to exist, although old versions may still be loaded and accessed if needed for compatability reasons.

• Debugging

Shag/OS v0.34 gave full symbolic tracebacks whenever any process, or even the kernel, faulted. This time, a lot of the debugging will be made even easier, since every function has a special header that provides runtime checking of all the parameters to make sure they are all within reason, and allowing for directly finding where a program went wrong, rather than waiting until a side effect surfaces. All of this will be able to be turned off completely at compile time for time-critical code once it has been debugged.

• Standards

Shag/OS v0.34 was designed to comply with Posix standards for the standard C interface, and v1.00 will designed with Posix standards and the new C++ library standards in mind.

• Windowing System

A simple extensible windowing system, based on using the multiple-inheritance of objects to its fullest extent to make a large and powerful system based on small and simple components, might be easily developed on top of the base system.

Research:

Lots of research has been done into techniques used by other operating systems, including Mach, Ameoba, Chorus, and Plan9. During the course of this research, I found many ideas similar to ones I was using, as well as discovering new ideas. Seeing the results of using ideas similar to my own helped me to decide which ones were more likely to more successful when implemented.

Platform:

All current development is being done on the 32-bit Intel 80x86 architectures. The code, and development environment, is designed for maximum flexibility for easily porting to other architectures, with a minimal amount of code changes, or with many code overrides later to increase system performance.

Independent Study Goals:

Because the scale of this project is so large, I hope to have just the following implemented on the Intel-based PC architecture during the course of this independent study:

• Object-oriented micro-kernel (vm, threads, ipc)

• Dynamic object loading, linking, and unloading (needed for device drivers)

• Console device driver (text-only, with multiple virtual consoles)

• Keyboard device driver (with loadable key mappings)

• Ethernet device driver

• Network objects

• Transparent communication between distributed objects

Changes in v1.00 from v0.34:

The new version of Shag/OS, to be done for this independent study project, will differ from the previous independent study project in the following ways:

- Completely rewritten - learning from the old version

  By rewriting it from scratch, many of the ideas learned through the development of the old version can be used to improve the new version without bloating it with ancient code. This also makes it easier to try out new ideas and base the system around them without worrying about interfacing them with obsolete code.

- Written entirely in C++

  The original version was written in C, and had many strange preprocessor hacks to make the object-oriented interface work in a simple, consistent manner. Writing the new version from scratch in C++ eliminates many of these problems, as well as providing a more robust and flexible structure to the code.

- Networking

  This time, a simple ethernet driver will be added, along with a simple network protocol to allow reliable communication between remote systems.

- Distributed Object Interface

  The new calling interface used for calling object methods is designed to be fast on a local machine as well as transparently distributed. This will allow objecs to reside on any machine on a network, and still communicate as if they were all on the local machine. Virtual memory and demand paging will be used to increase the efficiency of this.

- BIOS Interface

  Allows calling system ROM routines at any time (provides protected/real mode interface on PC systems), thus providing instant compatability with all devices supported by the firmware, until specific device drivers can be implemented. This solves the chicken and egg problems of trying to boot up a system when the drivers that are needed are not yet loaded, since the ROM routines for reading the disk, accessing the console, etc, can be called until the real device drivers are loaded. This lets the microkernel remain small, because it doesn't need to have large chunks of code dedicated to the purpose of bootstrapping the system.

Relevant Experience:

- JakeDev

  In 1990, as a Computer Science House member, I was the project leader, and main developer, for a project to write an operating system for Jake, a spare VAX 11/780 that was owned by the House. It had simple memory management, multi-tasking, device drivers for the console, the Massbus, and a hard-drive, as well as a Unix-like shell.

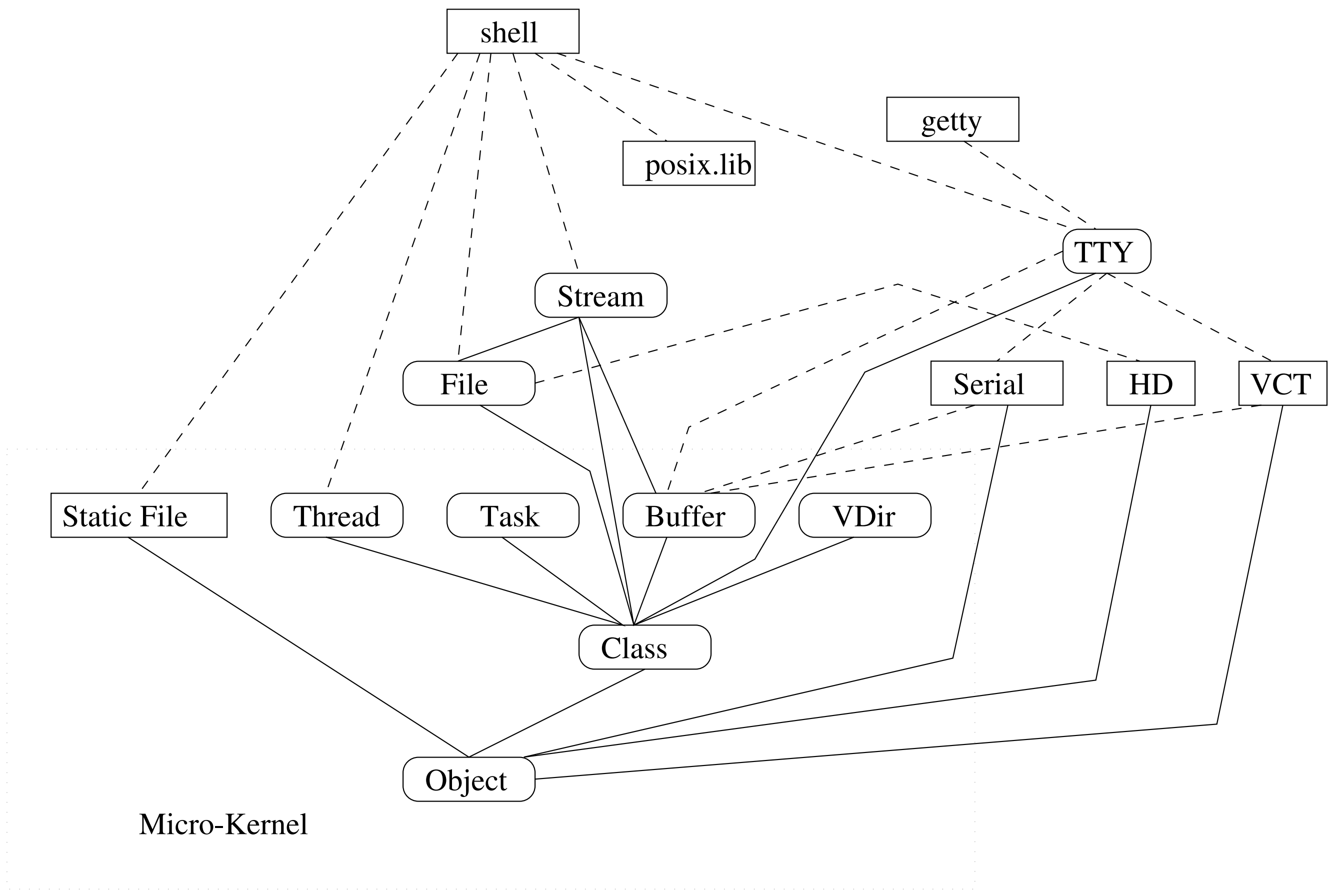
- Operating Systems Lab

  I completed OS Lab with Warren Carithers in the fall of 1991, and did the work on the memory management, device-independent I/O, serial device driver, and buffering.

- Operating Systems Independent Study

  In the fall of 1992, I worked on the early version of Shag/OS in C as an independent study project for Ken Reek. This work continued through winter quarter.

Shag/OS v0.34 Structure:



Explanations for diagram:

- Square Boxes: Objects

  These represent entities that are complete unto themselves. They are not registered as a class, and therefore cannot be inherited from. However, other objects can interface with specific instances of them, and utilize and extend their capabilities. In most cases, although one of each type is shown, many could actually exist (as in the case of device drivers, such as the Serial I/O driver). In the case of device drivers, there is an underlying class used to create each instance, but it isn't exported to the rest of the system.

- Rounded Boxes: Object Classes

  These represent classes that can be inherited from and that instances of can be created. Most of the classes shown have one or more instances created, for example the Thread class, which has an instance created for every thread running on the system. Only the class is shown in the diagram, not the individual threads. When an interface line is shown connected to a class, it shows connection to any number of the instances of that class, not the class itself.

- Solid Lines: Inheritance

  These lines show that a class or object inherits from the class below it, taking on all its properties and code as part of itself.

- Dashed Lines: Interfaces

  These lines show communication, usually with an object calling the object below it.

- VCT - Virtual Console Terminal

  This device object incorporates both the video display for the console and the interface for the keyboard. The driver supports any number of virtual consoles, which can be switched between at any time.

- HD - Hard drive

- TTY - Terminal Driver

  This driver is a higher-level device driver, that uses either the serial ports or the VCT as the low-level portion.

- VDir - Virtual Directory

  These are directory structures maintained in memory for a virtual namespace, used for mapping in all other filesystems as well as special objects, such as devices.

- Static File

  These are temporary files used for getting the system up and running. The kernel is linked to a static filesystem that has all startup files, and these are accessed until a real filesystem is loaded.

Potential Shag/OS v1.00 Structure: