Shag/OS:
An Object-Oriented Micro-Kernel-Based Distributed Operating System
Winter '96 Status Report

Frank Barrus
shaggy@csh.rit.edu

Rochester Institute of Technology
March 11, 1996

ABSTRACT

Shag/OS is an object-oriented distributed operating system designed around a micro-kernel with dynamic system modules (objects) handling all of the duties normally performed by old monolithic kernels. The status of the project is discussed in this paper, along with lessons learned and future expectations.

1. Current Status Overview

Quite a lot of work has been done on the project so far, involving research, design, development and testing of the microkernel, device drivers, utilities, parsers and even an interface description language (IDL). An extensive amount of progress has been made over the past several months, however there still remains a fair amount of work to do.

2. Problems and Solutions

Many problems were encountered while developing this system, far too numerous to list here. A few of the major ones, and some of their solutions, are presented below:

2.1. Objects

The biggest stumbling block and time consuming aspect of this project has been the design of a good object model, on which all other aspects of the system are based. In order to be considered reasonable, it has to meet the following criteria:

- Minimal memory overhead:

  Objects should be as small as possible, containing only vital information. Optional information, such as timestamps, security control, debugging information, etc, should be just that, optional, and should not consume space if not used.

- Minimal runtime execution overhead:

  Calling a dynamic object should not take a significant amount of time. This is complicated by the fact that method names and attribute names must be dynamically matched. This is discussed in more detail in the section on labels.

- Compatibility with existing C++ calling scheme

  Calling a dynamic object should be able to look exactly like calling a static C++ object. And this should be able to be achieved without having specific interfaces for every type of object or function, as this would defeat the dynamic nature of the system.

- Simple definitions of dynamic objects

  New dynamic C++ objects should be able to be defined in a way that is simple and differs very little from the way of defining static C++ objects, except when the new dynamic features are used. Some sort of interface description language or preprocessor is needed to convert dynamic C++ code to regular C++ code by encapsulating the static objects within dynamic objects.

- Dynamic run-time type checking

  Since, in theory, any type of object could be passed as a parameter to any method of a dynamic object, type checks must be done at run time, as efficiently as possible.

- Superclasses and Forwarders

  Some means of falling back on a superclass for more generic methods must be allowed, as well as forwarders that intercept any calls for methods that are supported by the forwarder (or all methods, if so desired) The forwarders are crucial to security since they are used to verify the permission of the caller to access the object before handing the call off to the object.

- Hierarchy of errors

  Some sort of means needs to be provided, similar to the (as-yet) incompletely and inefficiently implemented C++ exception handling system, of providing a hierarchy of errors that can result, and a means of passing that error back to a level that can handle it.

- Compatibility between architectures

  Different machines with different byte-orderings and different storage formats, and even different implementations of the object system, should be able to communicate with each other.

- A name

  Some sort of simple, but descriptive and catchy, name needs to be found for the whole object model being used for this system, so that it can be referred to easily and distinguished from other systems.

Most of these problems have now been solved, on paper (or whiteboard) at least, but the implementation is still being developed, since it involved not only some fancy headers and libraries, but also the development of a utility to generate the interfaces from new "dynamic C++" code.

This utility, DECO (Dynamic Encapsulator for C++ Objects), reads normal C++ files, but pays special attention to a new keyword, "dynamic:" in C++ classes, and builds the dynamic interfaces necessary to access those components of the class. (dynamic elements are automatically public)

For example:

```
class File {
private:
    int fd;
public:
    File(int fdesc) : fd(fdesc) {}
dynamic:
    int write(char *buf, int len) const;
    int write(obj o) const;
};
```

The first 'write' can only be called with an object that is a character pointer followed by one that is an integer, and the second write can be called with any object type. The second form of 'write' is responsible for doing its own type checking if necessary.

The definition:

```
objdef File obj_File;
```

can be used to declare a type 'obj_File' that refers to objects, the same as type 'obj', except that when assigned to it ensures that the type is 'File', otherwise it becomes a null object.

This new object is exceedingly more efficient, since the exact type is known, and all method calls can be processed by calling the most precise method available. Calls with dynamic parameters are still dynamically checked, but method names do not need to be matched. In addition, a pointer to the actual data used by the static class of type 'File' can be obtained, since the type is known, allowing direct access and no loss in efficiency from standard C++.

In addition to the fully dynamic object, and the dynamic one whose static type is known, a third type will be available for objects that are likely to be of a known type, but don't have to be. This will allow for direct calls when the type matches, but still allow dynamic calls to be made when they don't, at the additional runtime cost of a quick check before each call. This will provide maximum efficiency in the most commonly used cases while still providing the complete flexibility of a dynamic system.

The generic structure of the dynamic C++ objects and the calls made between them allows the dynamic code to be used not only in objects that are loaded by Shag/OS, but also in native Unix/Linux code (and potentially other operating systems, for that matter) without having to include or use specific header files. Dynamic objects can be polled and asked to list their methods and attributes, so that this information can be obtained at run-time, and the calling program can adapt to what is available.

The exact implementation of the underlying dynamic object structure does not need to be known to use the dynamic objects, therefore new, more efficient implementations could later be used without rewriting any code. Code will have to be recompiled to take advantage of the new structure, of course, but existing modules with the old code are still compatible with the new ones, just as machines with different implementations can talk to each other. The conversions are handled in the communication.

To provide compatibility between systems, a method, currently called 'flat' is called to generate a flat and consistent memory image of the object, doing all necessary byte-order translations as well as numerical conversions for floating point. This flat image is not truly entirely flat, as it can be constructed of multiple memory segments, and will still contain references to any dynamic objects held within the object being flattened. In this manner, only the static part of the object is transferred to another system (or to another process through shared memory if available), and the dynamic object references still refer to the original objects, although they now use proxy objects to access the original objects on the other system.

As far as a name goes, until something better is found, the objects shall have to be referred to as DECO objects, since they are produced by the DECO pre-processor.

2.2. Labels

There are two major complications with dynamic calls, matching the method name, and matching the data types. The data types are fairly easy to match, involving no more than a few compares of some pointers on each object. (this is for the cases where exact matches are needed because the object's class is expected to be locally available. For cases where the type must match but the class is in a remote system, things get a bit more hairy)

However, the method name is a bit more complex, since it needs to match across multiple systems, and a string compare is too slow. Therefore, some sort of unique identifier needs to be associated with each name, however these should not be globally defined ahead of time, because that limits the future flexibility of the system, especially in cases where attribute names are concerned in addition to method names.

One possibility would be to have a central registry, where, upon initialization, each object sends the names it needs identifiers for, and gets a unique identifier for each one. Unfortunately, although this sort of central scheme works fine locally, it leaves a lot to be desired in a distributed system on the network. Also, even locally it eats up storage for each name, which can only be freed by keeping track of which process is using which names and freeing them when the process terminates, or at some other appropriate time. This also involves extra communication before real communication can even begin.

I tried many techniques that involved generating unique identifiers, and then propagating them through the network by resorting to string compares when an id match was not found, and then updating the id's to match. The problem was always that the system could be spoofed by the chance that a unique id on one machine was not guaranteed to be unique on another, and there was no easy way to ensure that unique identifiers could be asynchronously created on different machines without resorting to long identifiers, which defeated the whole purpose since I was only dealing with short strings (guaranteed to be unique if they meant different things, and the same if they meant the same thing) to begin with.

In order to finally continue development of a functional, although less than perfect, system, I discarded the label idea altogether for now, although I did so in such a way that it should be able to be seamlessly integrated back into the system later when a reasonable solution is found.

Instead, I decided that since I was going to compare 64-bit id's, I might as well pack the entire string into those 64 bits (8 bytes) if it's less than 8 characters in length, making sure to pad the excess with null characters to ensure it can match other identical strings. That way, no arbitrary-length string compare is needed. Instead, a simple comparison of a 64-bit number will check for a match for method or attribute names. For strings longer than 8 characters, the actual strings will have to be passed and compared in the traditional way.

In most cases, the short strings will be used, since most commonly used method names will be shorter than 8 characters. (the long length of many C identifiers usually comes from the fact that they need a lot more distinguishing information, but in an object oriented system, the reference to the correct object provides much of the necessary disambiguation, and therefore the method names can usually be kept short and simple)

For those strings that are longer, there will be a delay in processing of each method call, but for now that will have to suffice, in order to get the system functional. A more efficient scheme can be added as a future optimization, perhaps by applying a hashing function to the string (a 64-bit hashed value should be fairly unique, thereby limiting the number of string compares that need to be performed) or by finding a solution to the label problem. In any case, the new solution will be able to be applied without changing any code that uses it, and therefore from now on the rest of the system can be developed without worrying about this point any longer.

3. Lessons Learned

One of the most important lessons I think I've learned is that although I must keep efficiency as a concern in my design process, I must not let it get in the way of developing a working product. After all, one that works, even somewhat inefficiently, is better than one that doesn't exist, and improvements can always be made later.

Unfortunately, I learned that lesson a little bit too late, after many long months of designs, redesigns, coding, testing, and code re-writes. I was always striving for the optimal solution, in terms of fast, small, and simple, and would rework parts of the project to death in order to achieve this, often at the expense of having a wholly working product. The result is that I learned a lot, and got to try out many new ideas, and made many pieces far more efficient than they would have been otherwise, but the project as a whole fell way behind schedule.

Many of the improvements I made up front could have been merely considered in the design, to leave room for them in the future, and could have been added after more of the system as a whole was functional, even if it was somewhat bloated and slow. (by my standards, not the rest of the world, since it is quite small and fast in comparison to many of the large systems in use in the commercial world, but then, I'm a perfectionist with my own work, and I expect it to be smaller and faster than *everything* else out there, and I'm not happy until that's true)

From this point on, I'm trying to let go of many of my ideals, and pull together all that I've written and all that I've learned, and produce a working system, which can be fine-tuned and optimized later, after everything is functional.

4. Technical Experience Gained

Through the process of increasing the efficiency of my code, I learned an extensive amount about the low-level internal workings of C++, and, more specifically, about the GNU g++ implementation. I feel quite competent in being able to write efficient code, both in terms of speed and size. I must have written thousands of short test programs to try out various features and see the assembly code generated by different variations.

I've also been following the issues concerning the draft standard for the C++ language and the C++ libraries, and keeping my code up to date with the latest revisions, (as supported by my compiler, currently GCC 2.7.2) as well as considering the full scope of the language when adding my dynamic extensions.

I also learned a lot more about networking, and communication issues between different architectures, as well as different techniques for remote procedure calls.

Through writing the CParse module (described later in this document) and working on the DECO pre-processor, I've put my parsing theory to use, and gained further insight into the syntactical structure of C/C++ programs, as well as how to handle the interpretation/compilation of it.

5. Under Development

The following aspects of the system are still under development:

5.1. Virtual Memory

The virtual memory system is functional and supports demand paging, but still needs to have shared memory finished, as well as a dynamic allocator for the stack. It also needs to be improved to work better with the new object system, to aid in fast communication, by sharing memory rather than copying it.

5.2. Device Drivers

Simple drivers exist for the video display and the keyboard, but an ethernet driver still needs to be completed and tested. Eventually, drivers for the disk drives and serial and parallel ports will be needed, but they are outside the current scope of this project.

5.3. Networking

The basic structure of a network interface layer and a datalink layer have been designed, but they have not yet been implemented, since the exact implementation depends upon finalizing the layout of objects and the communication methods used between them.

5.4. Tasks

Simple tasks exist right now for the kernel. No user-level tasks exist yet, but they differ from the kernel tasks only in virtual location and permissions, so they should be able to be added fairly easily later. The task system needs to work closely with the object system so most of the work is being held off here.

5.5. Threads

Simple multi-threading has been written, but all threads are currently kernel level. User-level threads will be implemented once the object-system is complete, since they need to use the object-system to call objects in other tasks or in the kernel, and without these calls, they would not have permission to access anything.

5.6. Loadable Device Drivers

The original ’ld’ equivalent which was used for dynamically loading object modules has been temporarily scrapped in favor of the new Elf32 module, which dynamically loads and links modules using Elf-format executable files. This has been tested under Linux, and will be called by the microkernel to load all system objects and device drivers. The old dynamic loader used ’a.out’ format object files, and had many complications as a result, which have now been resolved with the Elf format.

5.7. Importing Device Drivers

Work is being done to enable existing device drivers from other operating systems (currently Linux) to be used with the Shag/OS operating system, thereby immediately giving it a large amount of device support. Linux already supports loadable device drivers, so all that is necessary is a compatible interface to the modules. They will not run as efficiently (or be as dynamically useful) as native Shag/OS device drivers, but will have to suffice until native ones can be written.

5.8. Testing Environment

Due to the flexible layout of the system, and the object-oriented communication structure, I was able to build a local testing environment that provides just the basic features needed for testing other objects under the Unix (Linux) environment. This allowed me to test higher level functionality before all the low-level details were completed, and also sped up development because I didn’t have to build an entire system and load it on my test machine every time I wanted to debug a new object. Of course, the very nature of this test environment is still changing as I’m redefining what an object consists of and how it communicates, but this will still save me a lot of future testing time.

5.9. DECO (Dynamic Encapsulator of C++ Objects)

The DECO utility is a pre-processor for dynamic C++ files. It leaves standard C++ code alone, and allows the quoting of literal lines that will not be processed, and searches for additional keywords, such as 'dynamic:' in a class structure, and adds code for the dynamic interface of such objects. An object does not need to contain any dynamically accessible attributes or methods in order to be used dynamically. An object that contains the keyword 'dynamic:' without any specified dynamic elements will still be dynamically encapsulated for the purposes of passing it between different environments. In this sense, in can be used for doing traditional static RPC's where the exact type of each parameter must be known ahead of time, along with the exact function being called.

But for objects with dynamic methods, an interface is built to each of these which can check the types of all the parameters at run-time to find the correct function to call, if there is one. Also, a generic calling method, dynamically check the id of the method it is called with, and will call the appropriate member function.

Functions are also generated for accessing attributes and for looking up the names of supported methods and attributes.

The DECO program can either be used as a front-end to a C++ compiler, to pre-process the dynamic components of the C++ code, or it can be used independently to convert dynamic header files ('*.dh') to standard header files ('*.h'). In the former sense, a shell script can combine all of the processes (CPP, DECO, C++), and thus act as a dynamic C++ compiler ('D++'? 'C+=2' ?). In the latter sense, it operates more as an IDL.

DECO is the secondary core of the new object system (with the primary core simply being the model itself and the methods used to store objects and communicate between them), because without DECO writing new dynamic objects would involve too much extra coding and understanding of the object system for simple objects. With DECO, dynamic C++ objects are no more complicated than the basic static C++ object that most people are already familiar with.

Too much time was spent trying to find other means to simplify the interface for new dynamic objects before I finally decided to write a new extension to the C++ language (or an IDL, depending upon how you look at it). The idea of DECO solved nearly all of the complications I had been having. Now it just needs more time to be completed. Luckily, using the already complete CParse module (see below) that shouldn't take much longer.

On a side note, due to the fact that the object system took most of my development time, along with the development of DECO, I will be making my dynamic object model, and the DECO utility, the focus of my Master's Thesis.

6. Completed Sub-projects

Not all of these sub-projects were directly part of the original plan, but in some form all of these were useful to the completion of the project.

6.1. fconfig (file configure)

This utility is used to configure files in the build area from source files in the source tree. It takes information about the current machine architecture being configured for, and it creates links to all relevant source files, along with a new source file that includes them so that the most machine specific details override the more generic ones.

Multiple source trees are allowed, and nothing is modified in the source directories, so that local files override generic ones that are part of a distribution, and the distribution can reside on read-only media, and still be compiled from without copying the source tree over.

It also searches the source files for '#pragma' commands that allow each source file to add necessary compiler flags or to override or remove flags that would cause conflicts.

The dependencies for every file are checked, and a make rule with all the correct flags and dependencies is generated for each target.

6.2. mkheader (make headers)

This utility allows for putting standard headers on all source files, and updating them to keep track of the date of creation, and the last modification date (not counting the header update).

6.3. genofs (generate offsets)

Given a C/C++ header file, and a list of relevant data structures, this utility will build a file of #define commands, suitable for inclusion in an assembly source file, of the offsets of each element within the structures.

6.4. SOLO (Shag/OS LOader)

SOLO is used for loading the microkernel and all the other files needed to start the operating system. It is specifically designed for the PC, and is written almost entirely in assembly code. It provides basic console, serial, and file I/O functions, as well as a simple shell from which to load the kernel, other programs, or to use for automated system start-up shell scripts.

SOLO understands both the a.out and ELF file formats, and initializes virtual memory to load the executable at the address it was created to run at. It is also capable of booting other programs (such as DOS/Windows/etc) from any partition, and can boot Linux directly as well.

It is designed to stay resident in a small portion of memory (32K, including buffers) so that if the kernel exits, either from a shutdown or a crash, the low-level routines in SOLO can hopefully recover and provide some clue as to the cause of the crash. Simple debugging commands are built into SOLO, and all console output is logged in a separate area of memory that can usually survive a reboot, to insure that critical debugging output, or the result of a kernel crash, is not lost.

6.5. Elf32 (Executable and Linking Format, 32-bit)

The Elf32 module is used for dynamic loading Elf-format executable files. It handles relocation to any address, and loading and linking of dynamic shared libraries and modules.

This program is used by the microkernel to load all system objects (device drivers) and user-level programs. The Elf32 module itself is statically linked, as the kernel only knows how to load static Elf files.

Switching the system over to use Elf was a major change, but one that was definitely necessary. The option to generate position independent code (PIC) is fully supported with Elf, and is used to create Elf shared libraries. For Shag/OS, this is also used to generate dynamic object modules, allowing them to be not only loaded at any address in memory, but shared with other tasks by mapping in a shared copy (using virtual memory) at any available virtual address, without having to either reserve a fixed address range for each object or duplicate and relocate the object.

Upgrading to Elf involved not only writing a new C++ class for dealing with Elf binaries, but also converting the boot loader to load an Elf microkernel. At the moment, that process is incomplete, so the microkernel must still be compiled in a.out format, although the rest of the system can be composed of Elf objects. Soon, the microkernel should be able to be an Elf object as well.

6.6. CParse

This is a C++ Class that provides full CPP/C/C++ parsing, performing a lexical analysis that can be controlled by the program can using it, including CPP preprocessing, and keyword tokenizing. It also includes the extra tokens needed for DECO, and is used by both DECO and OSH (the object-oriented shell).

It is based around a state-machine and uses only memory buffers, so there is no dependence on any type of underlying file structure, or any operating system or library specific details.

6.7. Keyboard

The keyboard device driver has generic tables of keyboard key codes, along with state information and a translation table for converting various key combinations to ASCII (or other character representation) results. This allows it to be customized for any variety of keyboards, including international ones.

6.8. Console Device

The console driver is a PC-specific simple output device that supports virtual consoles, and allows switching between them. No attempt is made to emulate any particular terminal, since that is handled by a separate object.

6.9. VT Emulation

The VT-100/200 emulation driver is a separate object, not directly integrated with the console device. This allows other forms of emulation to be swapped in while still using the same console device. The also allows the generic VT object to be used with devices other than the standard console, while still providing the same consistent emulation, and without replicating code.

6.10. Testing

An extensive amount of time has been spent testing the parts that have already been written, both to insure their correct functionality and interoperability with other components, and to try to increase their efficiency, both in terms of minimizing memory consumed and maximizing speed of execution.

7. Project State and Code Overview

There are currently over 37,000 lines total comprising the project, of which around 30,000 are currently used. This includes shell scripts, makefiles, header files, libraries, utility programs, device drivers, user-level programs, and of course the microkernel itself.

Although more code is still being added, this number is expected to start shrinking sometime in the near future, when 'DECO' will allow replacing a lot of static code with dynamic code, improving modularity and decreasing redundant code by allowing better sharing of common objects and code.

Even now, however, despite the large amount of source code, the largest executable file currently produced, the microkernel, weighs in at only 38K. This is expected to grow as the memory management and IPC routines are expanded, but the size should remain under 64K. The boot loader is 6K, most device drivers are under 8K, and due to shared libraries and the use of objects for much functionality that was traditionally performed by local functions, even programs such as the shell shouldn't be much larger than 32K. This should leave plenty of room on the system for I/O buffering and large applications that demand more memory, allowing smaller machines to run more than they are usually capable of.

The layout of the project is self-documenting, as there are '.purpose' files in each directory stating the purpose of the files in that directory. Also, any machine-specific code code is placed in a 'm-*machine_type*' subdirectory. Therefore, it is fairly easy to see the layout of the whole project, and the amount of code in each section, automatically with the 'showtree' shell script.

For those directories with machine-dependent code, a separate list is made, showing 'mi:' for machine-independent code, and 'mi-*machine_type*:' for machine-dependent code. Note that 'pc' and 'i386' are two different things, since the former is a hardware architecture and the latter is a processor architecture, and they can be independently interchanged with other architectures.

Current Code State of Shag/OS:

Shag/OS: an Object-Oriented MicroKernel Operating System -- 4 files, 363 lines, 9253 bytes
  Config: Configuration files -- 10 files, 146 lines, 4206 bytes
  bin: Executable binaries (and shell scripts) -- 8 files, 296 lines, 5934 bytes
  bugs: Files with GCC bugs for reporting to GNU -- 2 files, 50 lines, 1396 bytes
  deadcode: Old versions of code, testing code, and other garbage -- 3 files, 236 lines, 5945 bytes
    save: unknown: missing .purpose -- 9 files, 2246 lines, 53112 bytes
    tmp: unknown: missing .purpose -- 2 files, 99 lines, 37131 bytes
    misc: unknown: missing .purpose -- 59 files, 2762 lines, 170311 bytes
  -- total: 73 files, 5343 lines, 266499 bytes
  doc: Assorted documentation -- 16 files, 1424 lines, 39435 bytes
  drv: Device drivers -- 1 file, 4 lines, 60 bytes
    console: Video console output -- 3 files, 160 lines, 3790 bytes
    -- mi: 1 file, 20 lines, 612 bytes
    -- m-pc: 2 files, 140 lines, 3178 bytes
    ether: Ethernet -- 4 files, 126 lines, 3837 bytes
    ide: IDE Hard-drive device driver -- 3 files, 341 lines, 7414 bytes
    ip: Internet Protocol -- 2 files, 34 lines, 931 bytes
    kb: Keyboard device driver and mapping utilities -- 5 files, 776 lines, 15202 bytes
    serial: Serial I/O -- 3 files, 348 lines, 7844 bytes
    sfs: Static File System -- 3 files, 233 lines, 4701 bytes
    tcp: Transport Control Protocol -- 2 files, 34 lines, 931 bytes
    tty: Basic tty driver with editing -- 2 files, 328 lines, 8215 bytes
    vtemu: VT-terminal emulation (vt100, vt200) -- 6 files, 1070 lines, 24231 bytes
  -- total: 34 files, 3454 lines, 77156 bytes
  fconfig: File configuration utility -- 2 files, 895 lines, 20740 bytes
  lib: Libraries -- 7 files, 361 lines, 11610 bytes
  -- mi: 5 files, 265 lines, 8127 bytes
  -- m-i386: 1 file, 35 lines, 1366 bytes
  -- m-pc: 1 file, 61 lines, 2117 bytes
    c++: C++ specific code -- 5 files, 204 lines, 5872 bytes
    -- mi: 4 files, 204 lines, 5872 bytes
    -- m-unix: 1 file, 45 lines, 1640 bytes
    class: Various class definitions -- 8 files, 1366 lines, 38497 bytes
    hw: Hardware-specific -- 4 files, 204 lines, 7702 bytes
    io: Input/Output -- 1 file, 172 lines, 3280 bytes
    object: Dynamic Object-Oriented services -- 13 files, 1436 lines, 34380 bytes
    old: Old out-of-date code (unused) -- 6 files, 621 lines, 19557 bytes
    posix: Posix-complient code -- 24 files, 2002 lines, 54872 bytes
    -- mi: 20 files, 1515 lines, 45359 bytes
    -- m-i386: 4 files, 487 lines, 9513 bytes
      sys: System specific -- 5 files, 210 lines, 6943 bytes
    -- total: 29 files, 2212 lines, 61815 bytes
    stdc++: Standard C++ -- 4 files, 522 lines, 14848 bytes
    sys: System-specific -- 2 files, 124 lines, 3470 bytes
  -- total: 79 files, 7267 lines, 202671 bytes
  objtest: Routines for testing objects -- 6 files, 577 lines, 13490 bytes
  sys: System files -- 1 file, 18 lines, 496 bytes
    a_out: a.out-format binary control -- 8 files, 2273 lines, 60563 bytes
    boot: Boot loader -- 27 files, 5684 lines, 117233 bytes
    -- mi: 2 files, 175 lines, 4353 bytes
    -- m-pc: 25 files, 5509 lines, 112880 bytes
    elf: Elf-format binary control -- 22 files, 2916 lines, 77972 bytes
    mkern: MicroKernel -- 37 files, 3807 lines, 102136 bytes
    -- mi: 14 files, 1129 lines, 30448 bytes
    -- m-i386: 13 files, 1338 lines, 38743 bytes
    -- m-pc: 10 files, 1340 lines, 32945 bytes
  -- total: 95 files, 14698 lines, 358400 bytes
  ubin: User-level binary programs and utilities -- 1 file, 20 lines, 612 bytes
    osh: Object-Oriented Shell -- 1 file, 1504 lines, 27936 bytes
  -- total: 2 files, 1524 lines, 28548 bytes
  util: Utilities for compilation, configuration, and development -- 10 files, 1844 lines, 49085 bytes
-- total: 341 files, 37881 lines, 1076813 bytes

8. Milestones

Here is a list of expected future milestones for 1996:

March 22 The object system will be complete and functional, and will allow fully dynamic calls between objects. (but much of the interface will have to be coded by hand until DECO is complete)

March 29 The DECO pre-processor will be complete and will allow automatic generation of all dynamic interfaces.

April 12 The object system will be incorporated into the microkernel, and, combined with the Elf32 code, should allow dynamic loading of modules.

April 19 The virtual memory system will be upgraded to properly work with the new object concepts of threads and tasks, and to aid in their communication.

May 3 The ethernet device driver will be completed, and will be used with a simple communications protocol to talk to objects on other machines. (or on the same machine with a loopback if other compatible machines are not available for testing)

May 10 Test objects will be created so that a demonstration should be possible with communication between objects residing on different machines, including talking to system devices on remote machines in the same manner as on the local machine, without needing any knowledge of where the objects reside, thus demonstrating the distributed nature of the operating system.

9. Conclusion

I’ve still got a heck of a lot of work ahead of me, but things are starting to really come together now, and it’s great to look at the project as a whole and see how much I’ve accomplished already, and that gives me the strength I need to forge ahead and see it to completion.