Towards Object Orienting Linux Kernel Structures

Rushikesh Joshi, Nishit Desai, Kanika Nema, Anil Gracias
Department of Computer Science and Engineering
Indian Institute of Technology, Bomaby

Abstract

Object oriented programming has many advantages over conventional procedural programming for constructing systems with highly modular and extensible abstractions. This paper outlines a process for objectification of Linux kernel structures. An inside-out approach is adopted with which kernel structures can be objectified bottom up. Five levels of objectification are defined. Kernel files move from level 0 to level 4 as they mature in terms of degree of objectification. The paper also discusses a proof of concept objectification of message queues of linux 2.6 version. The kernel carrying partly objectified code has been tested.

1 Introduction

Examples of operating systems that use object orientation for kernel implementations are Apertos [12], Ameoba [10, 9], Choice [3] and Chorus [2]. These systems have been mostly built object oriented grounds up. On the other hand, we observe that a *big-bang* approach to modern large scale non-object oriented code that is in wide circulation is infeasible. Applying objectification to existing Linux kernel requires a different approach owing to its size, implementation complexity, and process manageability issues. In this paper, a process for objectification of linux kernel structures is described. The objective of the process is to carry out a step-by-step objectification of kernel structures without any change to functionality. The process permits an inside-out objectification, while the kernel remains compilable and bootable at all times in presence of partial objectification. The inside-out approach is essentially a bottom-up approach, in which, the deepest structures are objectified before the outer layers get objectified. The approach is discussed in detail in Section 1. Due to this approach, a kernel is usable even when it is partly object oriented.

Essentially, the process is managed around objectification of non-object-oriented kernel files. As objectification progresses, kernel source files move from the lowest object maturity level (level 0) to higher levels. For example, the first level crossing from level 0 to level 1 for a chosen C source file can be achieved by making the file compatible with C++. In this stage, the code is made compatible specifically for the GNU C++ compiler. The 5-level model is further explained in Section 3. Level tagging of kernel files not only guides the objectification process, but also serves as a progress indicator. When all files reach the highest level of objectification, the kernel can be said to be object oriented.

Object extraction from data structures and procedures is guided by an analysis of data structure types, variables and their usage. Procedures may use data for different purposes such as for creating new entities, as input parameters, as output results or as global variables with different read/write access patterns. Interconnectivity metric based [5], global and type based [8], evidence driven [7] and concept analysis [11] are examples of object extraction techniques that have been found relevant, for the structures in the scope of our study. However, though the degree of objectification itself may be subjectively qualified, the process does permit object oriented refactoring of level 4 files. As a proof of concept, message queues of Linux IPC in Kernel 2.6 [1] were objectified. The partly objectified kernel has been tested. The experiment has been outlined in Section 5.

This paper proceeds as follows. Section 2 outlines the inside-out approach. The five levels of objectification maturity are described in Section 3. Section 4 presents techniques for bringing kernel files to level 1. Objectification of message queues of Kernel 2.6 is described in Section 5.

2 The Inside-out Approach

We use an *inside-out approach* [6] for objectification, which is similar to the incremental cluster-by-cluster approach of Zou et al [13]. In this approach, objectification of kernel implementation can be achieved in parts, file by file. As shown in Figure 1, a structure S, when objectified to S', may be simultaneously in use by both the objectified and non-objectified parts of the kernel.

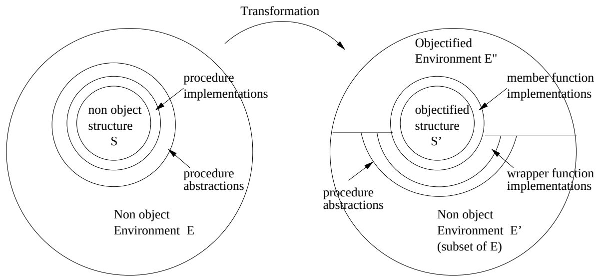

Figure 1: Objectification of a Non-object Structure

Some of the intermediate layers disappear as more and more parts get objectified. For example, in the figure, when E' is objectified, the wrapper function implementation layer can be dissolved. Early candidates for objectification are those structures that are mostly independent from the rest of the code. As the kernel gets incrementally objectified thus, it remains compilable and bootable even when some of its parts are purely procedural while the rest may be object oriented.

However, before the real benefits of objectification can be tapped, it is necessary to first bring the candidate code to a level in which the code is compilable by a C++ compiler. Also such code should be able to link with the rest of the interacting procedural code. In the next section, we will present the transformation process in detail, covering the five levels of objectification maturity of kernel source files.

3 Objectification Levels

As mentioned above, five levels of objectification are used to indicate the extent to which a particular file in the kernel has been objectified. At all levels, all files are required to be compilable by appropriate (gcc or g++) compiler to produce a bootable kernel image. Besides guiding the objectification process, the five levels described below can also be used to track the progress of the objectification process.

level 0 (Procedural) A file at level 0 is the conventional C file in the existing kernel.

level 1 (G++ Compatible) A file at level 1 is made g++ compatible by solving all compatibility issues as described in Section 4. A file at level 1 also links with the rest of the kernel. The file need not contain any object oriented features, but once brought to this level, it is capable of objectifying itself. It can be noted that all header files included in a file that has reached level 1, are also at level 1.

level 1.x (Client of Objects) A file at level 1.x has reached level 1 already, and hence outgoing procedural calls can be replaced by member function calls on objects located in other files that have reached levels 3 or 4.

level 2 (Local Objectification) A file at level 2 has only local objectification, i.e., the external visibility is still procedural. To achieve level 2 maturity, wrapper functions are used as shown in Figure 1.

level 3 (Dual Use) At this level, some of the dependent code of objectified file reaches level 1.x or above, while the rest of the dependent code may still be at level 0. Thus it is required to maintain two interfaces to the objects located in this file. In Figure 1, while E' uses S' through its procedural interface, E'' uses the same through its objectified interface.

level 4 (Full Objectification) A file at level 4 is used only by files that are at level 1.x or above. Wrapper functions of files at level 4 are dissolved as they are not needed. Further object oriented refactoring such as structural adjustments can be applied on a group of files that have reached level 4.

4 Migration to Level 1

In this section we discuss various compatibility issues that we encountered while bringing kernel files to level 1.

4.1 C++ Reserved Keywords

Many of the C++ reserved keywords have been used in the existing C source of the linux kernel. The reserved keywords of C++ which are not reserved in C are listed in Table 1. The usages of these keywords need to be renamed.

throw	bool	catch	class	delete
false	friend	virtual	namespace	new
operator	private	protected	public	template
true	try	using	typename	this

Table 1: Additional Keywords in C++

```
static inline void list_add_tail
(struct list_head *new,
          struct list_head *head)
{
    __list_add(new, head->prev, head);
}
(a) C Source:linux-2.6.14/include/list.h
----------------------------------------
static inline void list_add_tail
(struct list_head *new_changed,
          struct list_head *head)
{
    __list_add(new_changed, head->prev,
    head);
}
(b) Equivalent C++ Source
```

Figure 2: Renaming keywords

While renaming, care needs to be taken to ensure that the new identifier introduced is not already used.

```
struct ctl_table_header *tmp;
tmp = kmalloc
     (sizeof(struct ctl_table_header),
                          GFP_KERNEL);
(a) C Source: linux-2.6.14/kernel/sysctl.c
-----------------------------------------
struct ctl_table_header *tmp;
tmp = (struct ctl_table_header *)kmalloc
      (sizeof(struct ctl_table_header),
                           GFP_KERNEL);
(b) Equivalent C++ Source
```

Figure 3: Explicit Typecasting required in C++

It is worthwhile noting that a brute force method of renaming may be incorrect due to multiple usages of a name in different scopes. Hence renaming of a given variable is applied within its scope boundary. A renaming example from file *linux-2.6.14/include/list.h* is provided in Figure 2.

4.2 Typecasting

4.2.1 Implicit Typecasting

In C, typecasts may be done implicitly by the compiler. The C++ standard has deprecated this feature and requires typecasts to be done explicitly, which produces better type-safe code. The linux kernel source liberally uses void pointers to handle data and relies on the C compiler to do the appropriate typecasts. For example, *kmalloc* allocates required memory and returns a void pointer which is implicitly typecasted by the C compiler to the appropriate type. Figure 3 shows an example case from file *linux-2.6.14/kernel/sysctl.c.*

4.2.2 Enumerated Types and Integers

In C, integer values can be freely used instead of enumerated values for assignment to variables of enumerated types. In C++, such a conversion requires an explicit typecast or direct use of values defined as part of the enumerated type. An example from file linux-2.6.14/kernel/sched.c is in Figure 4.

```
enum idle_type
{
      SCHED_IDLE,
      NOT_IDLE,
      ....
};
.....
enum idle_type itype;
itype = 0;
(a) C source:linux-2.6.14/kernel/sched.c
-----------------------------------------
enum idle_type itype;
itype = (enum idle_type) 0;

(b) Equivalent C++ Source
```

Figure 4: Typecast for Enumerated type

4.2.3 Void Pointers in Arithmetic Expressions

C++ does not allow the use of void pointers in arithmetic expressions. In C, arithmetic operations done with void pointers are carried out implicitly as byte arithmetic. An equivalent code in C++ would require void pointers to be explicitly typecasted to char pointers when used inside arithmetic expressions.

4.3 Conflicting Names

4.3.1 Multiple Declarations

In C, a global data item may be declared several times as long as there is a maximum of one such declaration having an initialization. In C++, global data can be defined exactly once. C treats the initialized occurrence as a definition. C++ treats the second occurrence as a new declaration in itself and finds this redeclaration illegal. The conflicting declaration should be removed in order to get the code compiled by g++.

4.3.2 Type Name Conflicts

In C, the name of an instance can be the same as the name associated with a type using typedef, provided both do not lie in the same scope. In C++, the instance name cannot be the same as the name assigned to a type with typedef. We require to rename the type or the instance name. Such code fragment is shown in Figure 5.

```
typedef int (Func)(int,int);
struct S {
     Func *Func;
};

(a) C Source
-----------------------------------------
typedef int (Func_type)(int,int);
struct S {
     Func_type *Func;
};
(b) Equivalent C++ Code
```

Figure 5: Function Typedef in Name Conflicts

4.4 Function Definition Style

In C, an alternate style of function definition may be used, wherein, the argument types are specified after the argument list. This style of function definition is no longer supported in C++. The code can be compiled by adopting the standard style for function definition as shown in Figure 6.

```
int Func(a,b,c)
int a;
float b;
char c;
{
  ...
}
(a) C Source
-----------------------------------------
int Func(int a, float b, char c){
          ...
}
(b) Equivalent C++ Source
```

Figure 6: Old style of Function definition

4.5 Unlocatable Definitions

In C, nested or embedded definitions are visible in the scope where the enclosing data type is defined. In C++, these embedded definitions are not visible. To solve the problem the embedded definition is to be moved outside the enclosing data structure. In Figure 7, struct X has been moved out for visibility.

```
union U {
    struct X {
       ...
    } x;
    ....
};
size_X = sizeof(struct X);
(a) C Source
-------------------------------------------
struct X {
      ...
};
union U {
    struct X x;
    ...
};
(b) Equivalent C++ Source
```

Figure 7: Inaccessible Embedded definition

4.6 Non-Trivial Initializers

These are some extensions provided by gcc compiler in addition to the default ways of initializing instances of structures and array of elements. However, such forms of non-trivial initializations are not supported by g++ and need to be appropriately recoded.

4.6.1 Labeled Initializers:

The labels and initializing values may be placed in any order and only the elements that need to be initialized are to be specified. Under g++, it is not possible to use the label-initializers in any order. The order must be same as that in the structure definition. For proper ordering of the initializers, we may have to add some preceding initializing elements with their default values as shown in Figure 8.

```
struct A {
    float f;
    char c;
    int i;
};
struct A alpha = {c:'a'};
(a) C Source
-----------------------------------------
struct A {
   float f;
   char c;
   int i;
};
struct A alpha = {f:0.0,c:'a'};
(b) Equivalent C++ Source
```

Figure 8: Labeled Initializers

4.6.2 Range Index Initializers:

Gcc provides range index initializers to initialize a range of elements in an array to a particular value. The values after this specific form of initialization are then assigned to successive indexes. To have the code compiled by g++, we rewrite the code fragment by explicitly specifying the value repeatedly for the entire range and add default values for element s preceding the range. For example, int a[20]={[1 ... 5]=1,5} needs to be respecified as int a[20]={0,1,1,1,1,1,5,}.

4.6.3 Index Initializers:

Another form of array initialization provided by gcc is for assigning a value to a specific element by indicating its index. The compiler calculates the size of the array from the highest index specified. To code without using this extension, we need to calculate the size of the array, assign the correct value from the mentioned index and for the values unspecified default values are to be assigned. For example, int A[]={[2]2,[4]10} needs to be reimplemented as A[5]={0,0,2,0,10}.

4.7 *goto* Crossing Initialization

In C++, a jump cannot cross an initializing statement. To circumvent this problem we have to split initializing statement into declaration and initialization. This scenario is depicted in Figure 9.

```
goto xlabel;                 goto xlabel;
if(condition)                if(condition)
{                            {
    int y = 0;                    int y; y =0;
    xlabel:                       xlabel:
    ...                           ...
};                           };
(a) C Source                 (b) Equivalent C++
                                   Source
```

Figure 9: Jump Crosses Initialization

4.8 Linking Level 0 and Higher Level Files

It is tedious to link between C++ code and C code due to name mangling done by the C++ compiler. Name mangling is done by C++ compiler to support features like overloading of functions. But, C compilers do no such name mangling. Name mangling as done by the GNU C++ compiler and C compiler is shown in Table 2. The mangled function name carries information about the parameter list of the function. Therefore same functions will be referenced differently from C and C++ compiled object files.

To resolve this, C++ language provides a linkage specification with which we can declare that a function or object follows the linkage conventions for a some other supported language. The default linkage for objects and functions is C++. g++ compiler also supports C linkage, for gcc compiler.

Program	Unmangled (C style)	Mangled (C++ style)
int funcA() { }	funcA	funcA__Fv
int funcB(char c, int i) { }	funcB	funcB__Fci
int funcC(void *vp) { }	funcC	funcC__FPv
main() { }	main	main

Table 2: Name Mangling done by GNU compiler

To access C code from C++ code, we have to specify C style linkage for the C code which is accessed

```
int temp(int i){
    ...
}
int main(){
   ...
}
(a) C source - kernel.c
---------------------------
extern "C" {
     int temp(int);
}
int C_access(){
       temp(10);
       any C++ code
}
(b) C++ source - new.cpp
```

Figure 10: Accessing C code from C++ code

from C++ code. As shown in Figure 10, to access a function compiled by the C compiler, the function in C++ file needs to be declared with C linkage *extern "C"* [4].

Similarly, for accessing C++ code from C code will require the code to be declared within *extern "C"*.

In both cases, final linking has to be done by g++ compiler.

If there is some need to use extern "C" in a header file that is used in both C code and C++ code, then we have to use __cplusplus macro to eliminate this problem because "extern C" can not be used in C code.

5 Objectification of Message Queues

The files related to System V IPC message queues in linux 2.6.14 have been taken for proof-of-concept implementation. There are two files *ipc/msgutil.c* and *ipc/msg.c* related to message queues. Functions defined in *msgutil.c* are used by *msg.c*. Using the *inside-out* approach, first *msgutil.c* was chosen for objectification. It was converted to level 1 using the techniques described in Section 4. For successful compilation and linking of the kernel, the top level makefile has to be modified. As a next step, *msgutil.c* was converted to level 2. Local objectification was achieved by defining a class and wrappers were implemented for member functions. Since functions defined in *msgutil.c* are used only by *msg.c* file, the file *msg.c* was brought to level 1. File *msg.c* was modified such that all wrappers in *msgutil.c* were removed bringing it to level 4 [1]. At this point, file *msg.c* was at level 1.x and file *msgutil.c* was at level 4. File *msg.c* was locally objectified with wrapper functions for incoming calls. To bring *msg.c* at level 4, we need to bring all users of this file to level 1.x. Structures can be made into classes when, all users of the structures are at level 4. But, till such time classes are inherited from structures. Figure 11 shows one such class used by *ipc/msg.c*. In all, four classes were found in *msg.c*. Each class is the subclass of a structure. Another class containing global variables and methods operating on these global variables was extracted.

Also, it should be noted that files at level 4 with system call implementation, will continue to need wrapper interfaces for the system calls. This is because the user of system call functions in the kernel is the system call table, defined in architecture specific assembly files. Thus, system calls can be implemented in two ways. One alternative is that they can be made class member functions, in which case, wrappers are needed. The other approach is that they are not included in a class. Since, linux provides the **asmlinkage** macro, the interface to system calls need not be modified in the second approach.

```
class class_msg_queue:public msg_queue
{
public :
  void freeque (struct msg_queue *, int);
  void expunge_all (struct msg_queue *,
                                      int);
  int pipelined_send (struct  msg_queue *,
                             struct msg_msg *);
  void ss_add (struct msg_queue *,
     struct msg_sender *);
  int convert_mode (long *, int);
}
```

Figure 11: Extracted class: *linux-2.6.14/ipc/msg.c*

6 Conclusion and Future Work

The inside-out approach has been found for gradual iterative objectification of the linux kernel. Five levels of objectification have been defined to indicate the extent to which a particular file in the kernel has been objectified. This level model guides the objectification process, and serves as a progress indicator. Problems which occur when linux kernel is compiled by g++ have been enumerated. As a proof-of-concept implementation, the System V IPC message queue code has been objectified in linux 2.6.14. Test cases have been run to verify the correctness of objectified part of the kernel. The future work involves work in the areas of automatic objectification at various levels, and dynamic kernel memory allocation for objects.

[1] Level 3 was not required as, calls to msgutil.c are only by msg.c

References

[1] The Linux Kernel Archives. Website. `http://www.kernel.org/`.

[2] Paulo Amaral, Rodger Lea, and Christian Jacquemot. Implementing a modular object oriented operating system on top of Chorus. Technical Report CS/TR-92-57, Saint-Quentin-en-Yvelines (France), 1992.

[3] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of object-oriented operating system design. Technical Report R-89-1510, Urbana, Illinois (USA), 1989.

[4] Stephen Clamage. Mixing C and C++ Code in the Same Program. Website, 1998. `http://developers.sun.com/prodtech/cc/articles/mixing.html`.

[5] Jean-Francois Girard and Rainer Koschke. A comparison of abstract data types and objects recovery techniques. *Sci. Comput. Program.*, 36(2-3):149–181, 2000.

[6] Anil Gracias. Towards Re-engineering of Linux Kernel from Procedural to Object-Oriented. Master's thesis, Department of Computer Science, Indian Institute of Technology Bombay, 2002.

[7] Kostas Kontogiannis and Prashant Patil. Evidence driven object identification in procedural code. In *STEP '99: Proceedings of the Software Technology and Engineering Practice*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[8] Sying Syang Liu and Norman Wilde. Identifying objects in a conventional procedural language: an example of data design recovery. In *ICSM '90: Proceedings of the IEEE International Conference on Software Maintenance*, pages 266–271, San Diego, CA, USA, 1990. IEEE Computer Society.

[9] Andrew S. Tanenbaum and Sape J. Mullender. An overview of the amoeba distributed operating system. *SIGOPS Oper. Syst. Rev.*, 15(3):51–64, 1981.

[10] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[11] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 246–255, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[12] Yasuhiko Yokote. The apertos reflective operating system: the concept and its implementation. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 414–434, New York, NY, USA, 1992. ACM Press.

[13] Ying Zou and K Kontogiannis. Incremental transformation of procedural systems to object oriented platforms. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, 3-6 Nov. 2003.