Operating System Kernel Automatic Construction

Xiaohua Jia
Dept of Computer Science
The University of Queensland, Australia 4072

Mamoru Maekawa
Graduate School of Information Systems
University of Electro-Communictions, Tokyo, Japan

abstract

Each year a large amount of resources have been devoted into porting operating system kernels from one machine to another. This paper proposes a new system which is able to construct operating system kernel automatically based on kernel functions specification and hardware interface specification. The system can increase the efficiency and productivity of porting operating systems or building new systems. It can also generate more reliable kernels than human programmers.

1. Introduction

The operating system (OS) kernel is the core of an OS. It interacts directly with hardware and provides a hardware independent interface. It contains some fundamental functions, such as process management, inter-process communication and scheduling, so that the rest of the OS can be built as user processes running on top of it.

The OS kernel is the most critical part of an OS. It is critical in the sense of efficiency and reliability. The kernel functions are the support for all the activities of the OS and therefore are the most heavily used. The design and programming of the kernel must be error free. Any hidden bugs would cause disastrous results in real situations.

The OS kernel is the most time consuming part in the development of an OS. The kernel interacts with computer hardware directly. Kernel programming involves a lot of hardware details, such as probing the status of a device, reading data from a physical device port, and so on. It is regarded as the most nasty and error prone part of an OS program. Moreover the non-determinacy of interrupts makes the kernel debugging and testing extremely difficult. Kernel programming requires a lot of effort and time by experienced system programmers.

The OS kernel needs to be reproduced more often than the rest of the OS. Since the kernel is hardware dependent, if an OS is to be ported from one machine architecture to another, the kernel needs to be re-implemented on the other machine architecture. There is usually no need to change the other parts of the OS because they are designed to be hardware independent. Each year there are many OSs being moved to new machines due to the requirement of running these OSs on them or simply due to the machine hardware upgradings. Thus there is a large amount of effort being spent on porting the OS kernels.

Although the theory and techniques of building OSs have been established for several decades, most of the researches concentrate on performance issues of various part of the OS, such as scheduling algorithms, file systems, memory management, and so on. Few researches address the issues of automatic OS construction. In contrast, tools for automatic compiler generation have been widely used since 1980's, such as YACC[4], GMD[3] and Eli[2]. There is a great need to develop similar tools as compiler's construction for generating OSs.

This paper proposes a new system which is able to generate the OS kernel code automatically based on the hardware interface specification and the kernel functions specification. This system has the following advantages:

(1) It is much more efficient and productive for porting or building OSs. It reduces (or eliminates) the human effort in programming on hardware directly and takes less time to generate a kernel for a specific hardware architecture.

(2) The code generated by this system is more reliable (containing less errors). Human programming is usually more error prone than the machine. This is particularly true when programming directly on tedious and complicated hardware.

(3) The generated kernel would have a better modularity of the code structure, which makes the debugging and maintenance easier. This is because the code generated by the system follows some rules or regulations. It would be easy to find some common characteristics of the procedures or code segments from a generated program.

(4) The method of automatic kernel construction promotes an uniform kernel interface. The kernel interface is not only hardware independent, but OS independent. The kernel functions are *generic* to all general OSs. If the kernel interface is made standard, the portability of the rest parts of an OS can be easily achieved.

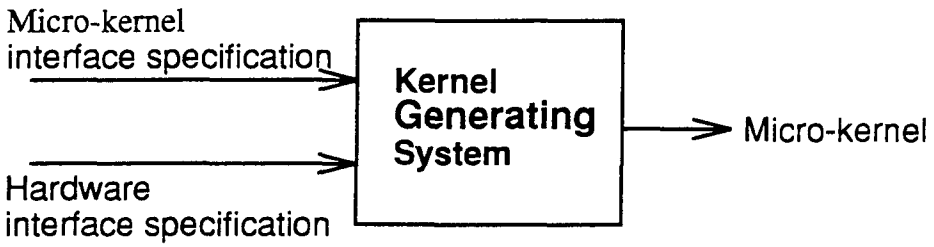
2. System Overview

Most of the early systems are monolithic systems, such as the Unix system. The entire operating system exists merely as a large program, known as the kernel, consisting of a set of procedures. As the functionality of the OS is constantly extended due to new requirements from applications and new I/O devices connected to the system, the kernel becomes extremely large and complicated. The Unix system is such an example. It is difficulty to imagine that this type of operating systems can be automatically constructed because of the variety of the OS functions and their complexity.

Microkernel is a new structure for building OSs. It basically separates the traditional OS functions into two parts, the microkernel and server processes. The microkernel only provides the minimum services to support processes running. The major OS system functionalities, such as file systems, directory management and memory management are implemented as processes (called OS servers) running on the top of the microkernel, the same as user processes. Many new operating systems have been developed by using the microkernel technology, such as Amoeba system [5], Mach [1], Plan 9 [3] and so on. For simplicity, we use kernel to refer microkernel in the following descriptions.

Although the functions of OSs differ greatly from one OS to another due to their different objectives, their kernel functions are more or less the same. This is because the kernel provides the most fundamental functions that every OS would use. The kernel functions are limited (unlike the functions of an OS) and generic. It is possible to standardize the kernel functions and the kernel interface. Once its functions and interface are defined by a formal method, the kernel can be automatically constructed.

We propose a Kernel Generating System (KGS) as shown in the diagram below. It takes the formal specification of hardware interface and the functional specification of the kernel interface. As the output of the system, it automatically generates the OS code which is specially for the target hardware and performs the required functions specified in the kernel interface. For ease of further debugging and maintenance, the generated kernel code is in a high level programming language, the C language for example.



The kernel interface consists of a set of primitives (or system calls) which are trapped into the kernel through software interrupts. The OS servers can use those primitives to obtain from the kernel the services, such as inter-process communication, physical memory allocation, input/output, and so on. The hardware interface is the specification of hardware architectures, such as interrupting support, memory address translation support, I/O subsystems, and I/O devices.

3. Kernel Functions Specification and Hardware Independent Code Generation

Although much research has been done on developing microkernel systems, there still does not exist a consensus about the functionality of the microkernel. The size of microkernel varies from less than 10K to over 100K. It is necessary to identify a minimum set of kernel functions which are powerful enough to support the variety of OS functionalities required by applications. The kernel is supposed to contain the following functions:

(1) inter-process communications,
(2) low level memory management,
(3) low level process management and scheduling, and
(4) primary I/O operations.

The above kernel functions are the part which is visible (which can be accessed by) to the rest of the OS. The other part which directly interacts with the hardware and drives the I/O devices is discussed in the next section.

The interface of the kernel consists of a set of operations and data type definitions. An interface specification language is needed to define the syntax of the interface operations and to specify the semantics of the operations. The syntax of an operation includes the information such as operation name, parameters and their types and the type of return value. The semantics of an operation specifies the functionality

of the operation, i.e. the state changes caused by the operation. For example, the IDL (Interface Definition Language) used in many distributed systems such as DCE [6] and CORBA [8] can be used to specify the syntax of the kernel interface.

Once the kernel functions and the interface are defined, the algorithms or structures used to implement these kernel functions can be designed and made standard. Therefore the KGS can always generate the kernel with the standard interface and performing exactly the specified functions. The program code generated by the KGS for hardware independent kernel functions would be the same for any hardware architectures. This idea can be extended further to the whole OS code. If the OS interface and its functionality are standardized and formally defined, theoretically, the whole OS code (not only the kernel code) can be automatically generated in the similar way.

4. Hardware Interface Specification and Hardware Dependent Code Generation

The most critical and difficult part of the system is to generate the hardware dependent code of the kernel for a particular hardware architecture. The hardware dependent code of the kernel includes interrupt handlers, I/O device drivers, and so on.

After investigating various types of I/O devices and their device drivers in different OSs such as the Unix, MS-DOS and Amoeba system, we find that there is a great similarity in designing and implementing the device drivers in different systems. Even the device drivers for different types of I/O devices follow the similar program structure. Furthermore, as the progress of microprocessors, nowadays almost all the I/O devices are controlled by a dedicated processor (called controller). By using a well defined interface, the CPU communicates with the controller which further controls the actual I/O hardware. The I/O controllers make the hardware dependent code easier to be generated.

To enable the KGS to generate device drivers for any specific hardware, the hardware interface must be formally specified. The existing hardware specifications (or notations) are usually for hardware design and testing like some VLSI specification notations. They do not address the semantics of a hardware interface, such as interface operations, interrupt signals, and so on. The required hardware specification language is for the purpose of code generation. It must be able to specify both static information and dynamic functionality. The static information includes the type of the device, addresses of control registers, addresses of data registers, interrupt number (or vector address), and so on. The dynamic functionality of a device includes the operations performed on the device, the conditions under which an operation should be performed, the status changes when an operation is performed, and so on.

It is regarded as a difficult task to construct programs from general specifications, but the specification language used here is a very specialized and limited one. The interrupt handler or device driver can be generated from the specification. For example, the following is a simple specification of an input device:

interrupt number: 09H
data port: 60H
status port: 62H
code for data available: 01H
code for error: 03H
Data-Available⊢READ
READ → RESET

The above specification only gives readers a very rough idea about what a hardware interface specification is like. It is not formal and complete at all. The notation "Cond⊢Op" means that condition "Con" must be true before operation "Op" can be performed. The notation "Op1→Op2" means that operation "Op1" must be always followed by operation "Op2". An interrupt handler routine for this input device can be constructed in the following pattern:

(a) check the status port whether data is ready. If so, goto (b); else return.
(b) move data from the data port to a buffer.
(c) reset status so that the device can interrupt again.

For different types of input devices, their port addresses, status codes and operations for reading data are different from each other. However, the principles of reading data from devices and the operation structures are similar to each other, which makes the program automatic generation possible.

5. Project Status and Open Problems

The proposed system is still under the development. We have been tried to use the Z language [9] to specify some of the kernel functions and to generate C programs from the specifications. At the same time, we have been designing a hardware interface specification language which suits our requirements for code generation. We have investigated various hardware devices and their programming interfaces. A prototype of a language which is similar to functional logic specification has been developed for this purpose. As the project running forward, we have the following major problems to be solved.

Firstly, reducing complexity of hardware interface specifications. One of the important objective of this system is to set OS programmers free of writing tedious programs for driving hardware devices. If the hardware specification is as equal complex as the hardware programming, the system would fail to achieve its original goal. However, if the hardware specification is not detailed enough or not complete enough, the system can hardly generate the code which works correctly.

Secondly, formalizing the interactions of various modules in the kernel. Modules in the kernel, each being specified independently, need to interact with each other for data transfer, synchronization, and so on. For example, an interrupt handler of a device need to communicate with the corresponding device driver for data transfer. will be accessed by Another example is that several modules which access shared data, such as information about processes, have to be mutual exclusive when accessing the shared data. It is difficult to specify these module interactions and to have some uniform methods to implement them.

Thirdly, flexibility of choosing algorithms for implementations. Some functions can be implemented by using different algorithms, which would present different features to the rest of the system. For example, there are many process scheduling algorithms, memory management algorithms. If users are allowed to choose a particular algorithm, then how to specify (or identify) the impact (or restrictions) of this algorithm to the rest of the system in order to make all the parts working consistently.

Finally, efficiency of the generated code. This is another major concern of the system. As we mentioned in the introduction, the OS kernel must be efficient because it is the most heavily used part in the OS. We use an uniform way to generate interrupt handlers, device drivers. Therefore code optimization based on special characteristics of a hardware architecture is another difficult and important topic.

The proposed system is a new attempt for building OSs. It has a great significance for OS development. It benefits both OS researches and industries.

References

[1] Accetta M., Baron R., Golu B D., Rashid R., Tevanian A., and Young M., "Mach: A New Kernel Foundation for UNIX Development," In Proc. Summer 1986 USENIX Conf., 1986, pp.93-112.

[2] Gray, R.W., Heuring, V.P., Levi, S.P., Sloane, A.M. & Waite, W.M., "Eli: A Complete, Flexible Compiler Construction System," Communications of the ACM 35 (February 1992), 121-131.

[3] Grosch, J., Emmelmann, H., "A Tool Box for Compiler Construction", Compiler Generation Report No. 20, GMD Forschungsstelle, an der Universitat Karlsruhe, Germany, 1990.

[4] Johnson, S.C., "YACC -- yet another compiler compiler," Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[5] Pike P., D. Presotto, K. Thompson and H. Trickey, "Plan 9 from Bell Labs", In Proc. of the Summer 1990 UKUUG Conference, Landon, July, 1990.

[6] Rosenberry W., D. Kenney and G. Fisher, "Understanding DCE", O'Reilly & Associates, Inc., 1992.

[7] Tanenbaum A.S., "Distributed Operating Systems", Prentice Hall Int., 1995.

[8] The Object Management Group, "The Common Object Request Broker: Architecture and Specification", OMG Document NO 91.12.1, Revision 1.2, 1993.

[9] Diller, Antoni., "Z: an introduction to formal methods", JOHN WILEY & SONS, 1990.