MICROKERNEL SYSTEM AS BASIS FOR SYSTEM LIBRARY BASED ON GENERIC COMPONENTS

Adam Hlavatovič and Tibor Krajčovič

Slovak University of Technology, Faculty of Informatics and Information Technologies
Ilkovičova 3, 812 19 Bratislava, Slovak Republic
Tel.: +421 2 60291111 Fax: +421 2 654 20 587
e-mail: {hlavatovic, tkraj}@fiit.stuba.sk

Abstract: In this paper we present an idea of system library, based on generic components within microkernel system in the area of embedded systems. The paper describes basic Exokernel structure and functionality with focus on Exokernel ability to separate high level abstraction from kernel itself. Equally class hierarchy based Choices framework is briefly described. A Choices divides parts of operating system into class hierarchies. We meditate on weaknesses of both approaches, with respect to performance, configuration at the level of design and reimplementation in a case of hardware architecture changing that is common in embedded systems. Our system library is based on techniques of generic programming in combination with policy based design and design patterns. Generic component library allows one to combine advantages and suppress disadvantages for both Exokernel and Choices. Using library it's possible to build fully specialized operating system in embedded systems.

Keywords: Generic components, system library, embedded systems, Exokernel, Choices, specialized systems

1 INTRODUCTION

An increasing number of functions and complexity of applications has increasing requirements on embedded systems. Past two decades operating systems has become an integral part of still growing number of devices based on some available type of microprocessor.

Performance changing in the last decade allows us use general purpose operating systems, even in the area of embedded systems. Using such kind of systems have some advantages, relatively stable, well known and working code decrease efforts and develop time. Using general purpose operating systems in comparing with specialized ones impose a significant performance penalty. This performance penalty may be tens of percent big (Ron Brightwell & Hudson, 2003).

On the other hand reusability issues with specialized systems are well known. They're caused by a direct using of hardware dependent parts of system.

In the area of embedded systems, there are used a lot of different hardware architectures with different properties and level of performance. Changing from one hardware architecture to another is not rare, commonly because a new generation of product. An old hardware is replaced with new ones, more powerful, which allows add some new functions to the product. New functions require new applications in operating system or at least update the old ones. Adding a new functionality also means change requirements on system itself.

Requirements on operating system for certain hardware are broadly defined by applications running on it. In product design, especially in embedded systems, often these applications are already known in advance. They're depending on product functionality. Requirements on operating system for certain product are therefore also already known in advance. An ideal operating system meets only these requirements and nothing more.

Due to a large number of possible combinations of requirements on operating system it's clear that one operating system can't meet the each possible requirements combination. Instead of a one operating system, we need many, in ideal case for each possible combination one. What is in fact unworkable. Through it all, there is a way, how such kind of systems can be generated with the least effort.

Policy based, generic component library give as an opportunity to generate type safe design fragments. By changing implementation of each policy to meet our requirements, we're able to adapt these design fragments. Using design fragments created by this way it's possible to create desired operating system specialized by applications running on it.

Adopting some basic ideas from exokernel systems Aegis and Xok, we can enrich the system, by system configuration at the level of user applications. In light of two different types of user applications it looks like, as if each application ran in its own operating system specialized for its needs.

2 RELATED WORK

A lot of effort has already been devoted to operating system research. The direct result of this effort is a number of different architectures and approaches to operating system design.

MACH (Accetta, et al., 1986) the first generation of microkernel systems known for its performance lack in compared with then monolithic kernels. These issues were resolved in the next microkernel generation. L4 (Liedtke, 1995), represents the second microkernel generation and due to its implementation in assembly language it achieves excellent performance. Later, high level language implementation L4Ka::Pistachio (Liedtke, et al., 2001) was created. Exokenel (Engler D. R., 1998), separating high level abstraction from kernel itself or SPIN (Bershad, et al., 1995) operating system blurs the distinction between kernel and application. VINO (Seltzer, et al., 1996) an extensible operating system largely derived from NetBSD. Or portal based microkernel system Pebble (Gabber, 1999).

Except all mentioned operating systems, frameworks were also created. They're trying to provide methodology for developing operating systems easily. Like class hierarchy based framework Choices (Campbell, Islam, Raila, & Madany, 1993) or component based framework OSKit (Ford, 1997) providing common components already separated out. Or another component based framework Think (Fassino, 2002) implemented in Java.

3 CHOICES AS A CLASS HIERARCHY BASED FRAMEWORK

Choices is written as an object oriented framework for operating systems. Using Choices a various number of operating systems with different complexity can be build. It's not only an operating system or framework itself, but a complex methodology how object oriented operating systems can be designed. Except framework itself, this methodology consists of relationship between parts of the framework, data diagrams and control flow diagrams and by interfaces and class hierarchies (Campbell & Islam, A technique for documenting the framework of an object-oriented system, 1993). Choices framework is based on a class hierarchy. This hierarchy defines fundamental concepts of whole operating system. These concepts are further defined by Choices sub-frameworks. Sub-frameworks are implemented by class hierarchy. Replacing certain parts of hierarchy or even whole hierarchy, system policies can be easily customized. So system itself can be build by a large number of pre-implemented class hierarchies.

Entire Choices framework is implemented in C++ language. Fundamental language feature used by a framework is single inheritance with dynamic binding. Every class within choices framework is inherited from a super-class and predefines inherited methods if they're required.

There are three basic abstractions, on the top of Choices design. Memory object, process and domain (Madany, Campbell, & Kougiouris, 1991) are those abstractions. Below, at the level of sub-frameworks Choices consist of sub-frameworks for persistent storage, device management, message passing or virtual memory (Kougiouris, 1991).

Considerable amount of Choices is focused on lack true run time object oriented programming support for C++ language (Interrante & Linton, 1990). The support can be divided into four basic parts a) automatic memory management, b) type representation, gives run time information about user defined types, c) dynamic code loading, allows one to load binary modules at run time, COFF (Gircys, 1988) a ELF (Press, 1993) formats are currently supported and d) class level debugging support.

Choices disadvantages

Generally, frameworks tend to cumulate code redundancy across layers of system abstractions. Such kind of redundancy is one of the factors that decrease overall system performance. Class hierarchy based frameworks tends to lock an application into a specific design, loss of static type safety and combinatorial explosion of the various design choices (Alexandrescu, 2001).

Dynamic binding structures are always less efficient in compared with static binding structures. Single inheritance with dynamic binding impose significant performance penalty that can be 14% to 49% big (Driesen & Holzle, 1996) in compared with static binding. Performance penalty is caused mainly due to a worse code optimization and not only by dispatch mechanism itself. Moreover, Choices implements some new structures with dynamic binding into the system kernel, automatic memory management, run time class information and dynamic code loading. All mentioned have negative impact on overall system performance.

Especially unpleasant is that, using those new structures is enforced by the kernel itself and it's not possible to customize them easily if it's required.

4 EXOKERNEL XOK/AEGIS

Exokernel is a type of microkernel system. The main idea is to make kernel as small and simple as possible. Moreover, exokernel brings an idea that kernel itself shouldn't create any high level of abstraction, but only export low-level primitives. Low-level primitives exported by kernel are focused on (a) how to give applications control over the resource and (b) how to protect it.

Exokernel is a system based on a few basic rules (a) separate protection and management (b) expose hardware, allocation and resource revocation (c) protect fine-grained units.

Applying these basic rules on a system design, we get operating system kernel, Exokernel, consisting only of parts responsible for safety and safe resource sharing. High level management and abstractions are implemented by applications themselves.

There must still be some minimal abstraction inside the exokernel. For example the CPU is represented as a linear vector, where each element corresponds to a time slice. Time slices can be then allocated by environments (base abstraction of process). Based on this minimalist CPU abstraction it’s possible to create a wide range of system schedulers with different level of complexity. Each application can implement its own scheduling policy.

Exokernel implementations

Same as microkernel system, also exokernel systems are highly dependent on certain hardware architecture. Xok and Aegis (Engler, Kaashoek, & Jr., Exokernel: an operating system architecture for application-level resource management, 1995) are both exokernel system implementations. System Aegis is the older one for MIPS architecture and Xok is the newer one for x86 architecture. Xok is based on Aegis and therefore they both have a lot of common. They both provide the same primitives for CPU and memory sharing and protection, base environment abstraction, exceptions distribution and primitives for inter process communication. Aegis was initially designed for network devices and therefore lacks disk support. Disk support is already integrated in Xok system. Support allows applications to access disk blocks rather than a files (don't forget lack of high level abstraction) and guarantees their protection against unauthorized access.

Exokernel advantages

Exokernel is unique by allowing the simultaneous running of different, same type resource managers, e.g. system schedulers mentioned above. This, in compared with ordinary operating systems even microkernel where 'standard' resource manager is enforced by system kernel, allows us specialize a resource manager by each application itself. Positive effect on overall system performance can be tens percent big (Brightwell, 2003), thanks to such kind of specializations. Kernel - running in privileged mode, to user application - running in unprivileged mode, division also means system reliability increase. In the case of an error in library only applications using its services are affected. Neither kernel itself nor others user applications aren't affected by this error.

On the other hand, disadvantage of Exokernel is that applications and libraries directly using kernel primitives are hardware dependent. This hardware dependency seems essential for achieving maximum system performance (Liedtke, 1995).

Generally, whole kernel and user applications directly using its primitives must be reimplemented, when switching to new hardware architecture. In fact most of used architectures are similar to each other, except formal changes in instruction set, so only some parts of system must be really redesigned and reimplemented. In embedded systems hardware change is very common and therefore reimplementation is a double pain. The problem could be partially coped with a generic component library design.

5 GENERIC COMPONENT LIBRARY

Generic component library (Alexandrescu, 2001) is based on policy based design. This method is trying to assemble a class with complex behavior out of many little classes, called policies. Each policy takes care of only one behavioral or structural aspect. Policy based design is similar to Strategy design pattern (Gamma, Helm, Johnson, & Vlissides, 1995), but differ in that policies are compile time bound.

There is a sub-framework for system schedulers in the Choices framework. This sub-framework is defined by an abstract class ProcessContainer to keep processes. ProcessContainer class defines three member functions add() - inserts a new process, remove() - removes and returns process reference, and isEmpty() - checks if container is empty. All system schedulers must inherit from ProcessContainer.

FIFOScheduler is one of the ProcessContainer abstract class implementation. Class NonLockedFIFOScheduler is another ProcessContainer abstract class implementation. They both differ only in container access control policy, otherwise they are the same. The structure of system scheduler sub-framework doesn't allow this to be sufficiently reflected to the implementation of both classes. Both classes FIFOSchedulr and NonLockedFIFOScheduler are therefore implemented independently each other.

If we require a different structure of stored processes e.g. LIFO or some kind of tree structure, we must create a new ProcessContainer class implementation for each new structure we require. If we involve requirement for container access control policy or requirement for ability to store not only processes but also its proxy, whole situation becomes intractable due to a combinatorial explosion of the various design choices.

One of the nice features of policy based generic library is that combinatorial explosion issue can be elegantly solved. Simple policy based implementation of ProcessContainer class could look like:

```
template <
        typename T,
        template <class> class container_type,
        template <class> class locking_policy
>
class process_container
        : public locking_policy<T> {
private:
        container_type<T *> _procs;  //!< processes
public:
        void add(T * proc) {
                lock lck(*this);
                ...
        }
        ...
};
```

Where, template parameter locking_policy is implementation of container access control policy. Function remove() can be implemented by the same way as function add(). At the beginning, locking_policy<T>::lock object is created to take care of container access control.

Access control policy implementation at the level of object for process_container class could looks like:

```
// object level locking
template <typename T>
class object_lockable {
public:
        class lock {
        public:
                lock(T & obj) {
                ...
                }
        };
};
```

Implementations of process_container class with required functionality can be created by using typedef keyword, this way:

```
typedef process_container<process, object_lockable>
        fifo_scheduler;
```

or

```
typedef process_container<process, non_lockable>
        non_locked_fifo_scheduler;
```

Where, both new types are fully equivalent to both original classes FIFOSCheduler and NonLockedFIFOScheduler.

Each policy can be seen as a generated type behavioral requirement. Implementations such kind of policies are then implementations of generated type requirements. Policy based generic component library allow us to generate type that can be any combination of our requirements. Moreover, there isn't any performance penalty due to static binding.

In general, using this method of design we can generate systems that meet our specific requirements.

6 CONCLUSION

In this paper we briefly describe the structure and functioning exokernel system, class hierarchy based Choices framework and policy based generic component library. In order to create a system able to reflect already known in advance information about application types, running on the system. This type of information is really common in embedded systems, because is already known in advance what sort of applications will be running on system. Policy based generic component system library over microkernel system exokernel allow us to obtain system configuration ability at the level of user applications.

Exokernel system has been chosen due to his ability to run several different resource policy implementations in the same time. What can be understood as system configuration at the level of user application. In light of two different types of user applications it looks like, as if each application ran in its own operating system specialized for its needs. Disadvantage of this approach is high level of certain hardware dependency at the level of libraries and user applications. When switching to a new hardware is therefore necessary to reimplement hardware dependent parts of the system. Negative effect of reimplementation can be partially suppressed by using frameworks, such as Choices.

Generally, frameworks give us a higher degree of system abstraction. This abstraction contains hidden information in the form of framework structure and interactions between its parts. Information hidden in abstraction are portable through a wide range of hardware architectures. Frameworks also give us an ability of high level of system configuration in the time of its design and high level of reusability of source code at the level of classes. This features are especially advantageous in embedded systems where is often necessary to implement similar functionality for various hardware architectures. Disadvantages of class hierarchy based frameworks are, that such kind of approach to application design tends to lock an application into a specific design, loss of static type safety and leads to combinatorial explosion of the various design choices. Another disadvantage is an increased performance penalty due to dynamic binding of generated code.

Enumerated disadvantages of class hierarchy based frameworks can be partially suppressed by using policy based generic component library. Generic component library also brings high degree of system abstraction such as class hierarchy based frameworks. Moreover, thanks to policies, it offers higher degree of configuration and also solves loss of static type safety. With a small number of user defined types it's possible to handle out combinatorial explosion of the various design choices. Due to a static binding of generated code by the mechanism of templates in C++ language, there isn't any performance penalty such as in case of dynamic binding.

We believe that with a combination of generic component library over microkernel system, it’s possible to create a system able to reflect already known in advance information about application types, running on the system. This will allow creating specialized systems with high performance and reliability for various number of different hardware architectures.

ACKNOWLEDGMENTS

This work was supported by the Grant No. 1/0822/08 of the Slovak VEGA Grant Agency.

REFERENCES

Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., a iní. (1986). Mach: A New Kernel Foundation for UNIX Development., (s. 93-112).

Alexandrescu, A. (2001). *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc.

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., a iní. (1995). Extensibility safety and performance in the SPIN operating system. (s. 267-283). ACM.

Brightwell, R., Riesen, R., Underwood, K., Hudson, T. B., Bridges, P., & Maccabe, A. B. (2003). A Performance Comparison of Linux and a Lightweight Kernel. *Cluster Computing, IEEE International Conference on , 0*, 251.

Campbell, R. H., & Islam, N. (1993). *A technique for documenting the framework of an object-oriented system.* University of Illinois at Urbana-Champaign.

Campbell, R. H., Islam, N., Raila, D., & Madany, P. (1993). Designing and implementing Choices: an object-oriented system in C++. *Communications of the ACM , 36*, 117-126.

Driesen, K., & Holzle, U. (1996). The direct cost of virtual function calls in C++. *SIGPLAN Not. , 31*, 306-323.

Engler, D. R. (1998). *The exokernel operating system architecture.* Massachusetts Institute of Technology.

Engler, D. R., & Kaashoek, M. F. (1995). Exterminate all operating system abstractions. (s. 78). IEEE Computer Society.

Engler, D. R., Kaashoek, M. F., & Jr., J. O. (1995). Exokernel: an operating system architecture for application-level resource management. (s. 251-266). ACM.

Fassino, J.-P., Stefani, J.-B., Lawall, J., Muller, G., & R&d, F. T. (2002). THINK: A Software Framework for Component-based Operating System Kernels., (s. 73-86).

Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., & Shivers, O. (1997). The Flux OSKit: a substrate for kernel and language research. (s. 38-51). ACM.

Gabber, E., Small, C., Bruno, J., Brustoloni, J., & Silberschatz, A. (1999). The pebble component-based operating system. (s. 20-20). USENIX Association.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addisson-Wesley.

Gircys, G. R. (1988). *Understanding and using COFF.* O'Reilly & Associates, Inc.

Interrante, J. A., & Linton, M. A. (1990). *Runtime Access to Type Information in C++.* Stanford University.

Kougiouris, P. (1991). *A Device Management Framework for an Object-oriented Operating System.* The University of Illinois at Urbana-Champaign.

Liedtke, J. (1995). On micro-kernel construction. *SIGOPS Oper. Syst. Rev. , 29*, 237-250.

Liedtke, J., Dannowski, U., Elphinstone, K., Liefl\"ander, G., Skoglund, E., Uhlig, V. (2001). The L4Ka Vision. *The L4Ka Vision* .

Madany, P. W., Campbell, R. H., & Kougiouris, P. (1991). Experiences building an object-oriented system in C++. (s. 35-49). Prentice-Hall, Inc.

Panos, R. J., & Madany, P. (1991). Choices, Frameworks and Refinement. *Object-Orientation in Operating Systems* , 9-15.

Press, C. U. (1993). *System V application binary interface (3rd ed.).* Prentice-Hall, Inc.

Seltzer, M., Seltzer, M. I., Endo, Y., Endo, Y., Small, C., Small, C. (1996). Dealing With Disaster: Surviving Misbehaved Kernel Extensions., (s. 213-227).