SOFTWARE—PRACTICE AND EXPERIENCE
*Softw. Pract. Exper.* 2007; **00**:1–16 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

# Object Oriented Wrappers for Linux Kernel ‡

D.Janakiram*, Ashok Gunnam*, N.Suneetha*, Vineet Rajani*, K.Vinay Kumar Reddy †

*Distributed & Object Systems Lab, IIT Madras, Chennai, India.*

**SUMMARY**

**Linux is an open source operating system, which has increased in its popularity and size since its birth. Various studies have been conducted in literature on evolution of the Linux kernel, which have shown that there are considerable maintenance problems arising out of the coupling issues in Linux kernel and this may hamper the evolution of the kernel in future. We propose an object oriented (OO) wrapper based approach to Linux kernel to provide object oriented abstractions to external modules. Since, the major growth in size of Linux kernel is in device drivers, our approach provides substantial benefits in terms of developing the device drivers in C++ though the kernel is in C. Providing reusability, extensibility features to device drivers improves the maintainability issues of the kernel to a large extent. The OO wrappers provide several benefits to module developers in terms of understandability, development ease, support for OO modules etc. The design and implementation of C++ wrappers for Linux kernel and the performance of a device driver reengineered in C++ are presented in the paper.**

KEY WORDS: *Linux . kernel. API. Object Oriented. wrappers. device drivers. In Kernel API. operating system*

## 1. Introduction

Linux is an open source operating system developed using C language. It is generally argued that performance is the most important quality factor for operating systems. Consequently, C and assembly language are used for their development and the programs are carefully optimized to achieve performance. Often the techniques used, although they improve the performance of the operating system, result in the degradation of maintenance.

‡You can download the code at
http://dos.iitm.ac.in/MOOL

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

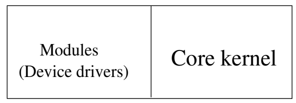


Figure 1. Linux kernel

The increasing attention on performance has resulted in increasing the complexity of the Linux kernel. Various studies [9], [10], [6] have been conducted in literature on the evolution of Linux kernel, which show evidence of considerable maintenance problems arising out of coupling in the kernel and this may hamper the evolution of the kernel in future. It has also affected the understandability of kernel due to lack of high-level abstractions.

One possible solution to the maintenance problem is to restructure the kernel using object oriented principles and re-implement the kernel in an OO language like C++. However, the Linux community is reluctant to adopt such an approach mainly citing performance problems with mainstream OO languages such as C++ [23]. There are attempts to use the OO principles like virtual tables for polymorphic function dispatch rather than use C++ language for implementing the kernel. Some of these arguments are well founded and we argue in this paper an acceptable approach could be to provide OO wrappers to kernel so that outside modules such as device drivers could be developed in C++.

Device drivers constitute a major part of the Linux kernel code. The device driver code lies outside the core kernel. In addition, the major evolution of kernel is in device drivers to support the evolving hardware trends. 57% of kernel code is observed to be of device drivers [4]. These drivers are developed either as kernel modules that can be linked statically or at runtime. So the kernel can be viewed as shown in figure 1, consisting of external kernel modules (like device drivers) and the core kernel.

One approach towards providing better maintainability to operating systems could be to develop the entire kernel and hence the operating system in an OO language. An OO operating system has many advantages like extensibility, reconfigurability and reusability [1], [5]. Development of programs, like device drivers on top of OO operating system brings out the advantages of object oriented programming in that each hardware device can be treated as an object. C++ as a suitable OO language for development of OS kernel is well argued [11], [1]. However, as most operating systems are developed in C, it may not be a feasible solution to re-engineer the whole kernel into C++, due to factors like portability of existing applications, human expertise, cost involved for this and so on.

An alternate and light weight solution is to provide OO wrappers for the core kernel. The underlying kernel remains the same but appears as an OO core kernel to external modules, like device drivers. We have applied this design to Linux kernel and provided C++ wrappers for the core kernel of Linux (2.6.9).

In this paper, we present the design, implementation and the performance evaluation of the C++ wrapper based Linux kernel. To evaluate the usefulness of C++ wrappers, some of the existing device drivers have been reengineered into C++. The performance evaluation of a set

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

of C++ drivers have shown that there is a performance degradation of less than 2%, which we believe is an acceptable overhead, given the advantages that are derived out of it.

## 2. Technical Challenges

1. Identifying abstractions
   An important feature of object orientation is *abstraction*. According to OO philosophy, abstraction should be based on data, i.e, the program should be structured around data rather than functionality. However in procedural paradigm, the programs are usually structured based on functionality ( for example, using flow charts during their design). As Linux is developed using a procedural language, it is structured based on functionality [6]. For this reason, it is necessary to identify the candidate cohesive objects from Linux kernel which are abstracted based on data.
2. C++ support
   The original Linux kernel does not provide C++ execution environment. Some of the C++ features like templates, exceptions, global and static objects are not supported in the kernel mode programming. Moreover some of the keywords specific to C++ such as *new, delete*, are used as identifiers in the Linux kernel. To handle these issues, an additional support is required to compile the C++ modules which use the original kernel header files. Also, the C++ libraries required for runtime support of the C++ modules, need to be provided.

## 3. Identification Of Objects

Many approaches for identification of abstractions from procedural code have been discussed in literature [18], [16], [17]. The common principle these works advocate is to identify the global data and data structures from procedural code and group them with the functions that use them. The rationale behind this kind of grouping is the central idea of OO design, structure the program based on data rather than functionality. According to this philosophy, all the procedures which use or modify same data belong to the same module, which in this case is a *class*.

Identification of the objects and the relationships between them, support for callbacks are the main requirements for identifying OO abstractions for the kernel.

The following are the set of heuristics that are used to identify the objects:

1. Identify the types (data structures) that are present in the code.
2. Identify the functions that operate on these types, i.e, either formal parameters or return types. The data structures which are used inside the body of the function are used to identify the relationships among objects.
3. Group the identified type and its appropriate functions into an abstraction.
4. If a function operates on more than one type and any of them is modified inside the function, the function is grouped with the type that gets modified.

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

5. If a function uses more than one data structure, and if no type is modified, then assign the function to the data structure which is the super type of other types, i.e, composite of other data structures.

For example, *mm_struct* is a data type, which stores information related to process' memory. Some of the procedures, which use this data structure either as formal parameters or return types, are *mm_alloc*, *exit_mm*, *handle_mm_fault*, *mmput*. So these functions are grouped together to form a class. Among these functions, *handle_mm_fault* accesses other data structures such as *vm_area_struct* in addition to *mm_struct*. The function assignment collision is resolved as specified in step 5. As *vm_area_struct* is a subtype of *mm_struct*, we assigned the function to *mm_struct*

### 3.1. Callback Objects

The Linux kernel uses function pointers to implement callbacks, i.e, the kernel calls the function pointed to by the function pointers when certain events like timer or new module insertion happens. Callbacks are important functions required by most of the external modules, like device drivers. So there should be a way to provide these callbacks in the wrappers in C++.

An easy way to identify callbacks is to identify the function pointers that are declared in the kernel. To be able to provide an OO way of callbacks and still be able to use the underlying kernel callback infrastructure, we used a technique, which is combination of template and strategy design patterns.

The kernel implements the callbacks by declaring function pointers in kernel in one of the following two ways

1. Inside a structure (for example file_operations)
2. As parameters for a function.

In case 1, a design which is based on template method pattern [14] is used. We declare a new class which contains virtual functions, each of which represents a corresponding function pointer declared in the structure. A C interface is defined and is statically registered with the function pointers of kernel. This C interface is a kind of template method. It calls the hooks which are virtual functions of the classes declared. These virtual functions are called in a polymorphic way. A user can define a sub class of the corresponding super class, define the virtual functions in it, and the callbacks are registered automatically.

For example, figure 2 shows a callback object around the file_operations structure.

For case 2, a strategy pattern is used. The strategy pattern states that the algorithm be encapsulated so that the actual algorithm needed can be assigned at runtime. In case 2, where function pointers are passed as parameters to another function, it is very much similar to an object oriented strategy pattern. Here there is no need to register a C interface to kernel statically.

For example, figure 3 shows how interrupt handlers of Linux kernel are provided in wrappers. The request_irq function of Linux kernel has a pointer to handler function, which the user defines at a later point of time. Here, that algorithm which is defined at runtime is the actual handler function, which in OO terms is the strategy. The context class, which in this case is

```
class CharDriver
{
public:
        virtual int devOpen();
        virtual int devRelease();
  ...
  ...
}*driver;
```

(a)

```
extern "C"
{
int open()
{
   driver–>devOpen();
}
int release()
{
   driver–>devRelease();
}

...
...
struct file_operations fops={
             .open=open,
             .release=release,
             ....
             };
}
```

(b)

Figure 2. Abstractions for function pointers inside a structure

InterruptHandler class has a reference to strategy, and delegates the request to appropriate algorithm at runtime based on the concrete strategy assigned to the reference.

## 4. Design

This section explains the design of C++ wrapper based core kernel and the design goals in accomplishing the task.

### 4.1. Design Goals

1. *Structure should be identifiable*
   The users should be able to use their operating system knowledge to guess the abstractions and the functions they perform. For this, the subsystems of the operating system should be directly identifiable in the wrappers. This makes the learning and understanding easier. To achieve this goal, we have created abstractions like MemoryManager, ProcessManager etc, which directly correspond to memory management and process subsystems of Linux kernel. It is sufficient for the users to understand these wrappers to perform most of the tasks with the subsystems.
2. *Facilitate house keeping within the wrappers.*

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

```
Handler *h;

class InterruptHandler
{
public:
    m_requestIrq( InterruptHandler h)
    {
        request_irq(irq, InterruptHandler::handler,flags, dev_name,dev_id);
    }

    static handler()
    {
        h->handler()
    }
}
```

(a)

```
class Handler
{
public:
    virtual irqreturn_t handler()
}
```

(b)

```
class MyHandler : public Handler
{
public:
    Irqreturn_t handler()
    {
        /*definition goes here*/
    }
}
```

(c)

Figure 3. Abstractions for function pointers used as parameters to a function

For an operating system, many a time, a function should be called only after performing certain operations like locking. Keeping track of locking and unlocking makes the programmer's task difficult. Therefore, it would be helpful if most of the house keeping is done with in the wrappers so that the programmer can concentrate on the primary task.

3. *Reusability*
   Reusability is one of the important characteristics of OO design. This requires the wrappers to be simple, so that the concerns are separated well. Our abstractions are made highly cohesive by localizing the interactions. So providing new wrappers that can provide extensions to existing ones, becomes easier.
4. *Performance overhead of wrappers should be less*
   Performance is one of the most important qualities required by the applications like kernel modules. When wrappers are used in the development of kernel modules, they introduce an extra level of indirection, i.e., an extra function call to do the same function, which creates a performance overhead. The wrapper system will not be useful if the performance overhead due to indirection is high.

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

*Prepared using* speauth.cls

### 4.2. Design Of Wrapper Based Kernel

Linux is a large monolithic kernel, where in different subsystems like memory management, process scheduling, inter process communication and device drivers run in a single executable file. The kernel provides system call interface to the applications for requesting various services of the kernel.

Linux provides support for loading and linking modules to the kernel at runtime. This, loadable module support, removes the need for recompiling the entire kernel for supporting additional modules. Device drivers and file systems are examples of loadable modules.

Although the kernel is monolithic, conceptually the kernel can be viewed as two parts, the core kernel, containing different subsystems like memory management, process scheduling, inter process communication, interrupt handling etc, and the other part the loadable modules, like device drivers.

The Linux kernel handles the device drivers in the following way. In Linux, the devices are considered as files. So when a user application wants to read or write to a device, it issues a read/write system call on the corresponding device file. The virtual file system (VFS) routes the system call to appropriate device function. The VFS provides callbacks for operations of device drivers. The device drivers define the callbacks and register them with the VFS. As every device is a kind of file, the interface of the callbacks for device drivers is provided by the virtual file system. For example, the *file_operations* structure which provides the interface for character drivers is defined in *fs.h* which belongs to virtual file system. So in the Linux kernel, the core kernel communicates with device drivers through the virtual file system, where as the device drivers communicate directly with the core kernel.

We provide wrappers for the core kernel of Linux. The design of our wrapper based kernel is shown in figure 4.

In the wrapper based kernel, its not possible to register the C++ callbacks with C kernel. So in our wrapper library, we define the C interface for callbacks of the kernel, which inturn calls the polymorphic driver methods of wrappers, and registers it with the VFS. So, when the kernel wants to communicate with a driver, it makes a call to the VFS, which in turn forwards it to wrappers. The wrapper does the processing and forwards it to appropriate driver. So in the wrapper based kernel, the communication from kernel to device drivers is through VFS and wrappers.

However, when the driver wants to communicate with the core kernel, it makes calls to appropriate wrapper methods, which inturn delegate the request to underlying Linux kernel. So the communication of drivers to the core kernel is through wrappers. This feature can further be exploited for storing the state information which can be used for shielding the device driver failure from kernel.

The design of wrappers to core kernel is based on the Facade pattern as discussed in [14], [15]. The intent of the pattern, "provide a uniform interface to a set of interfaces in the system. Facade defines a higher level interface which makes the subsystems easier to use" [14]. In simple words, it provides a higher level interface by abstracting out the details that hamper the understanding of the interface. There is no difference to the client in terms of functionality, but it only enhances the ease of use.

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

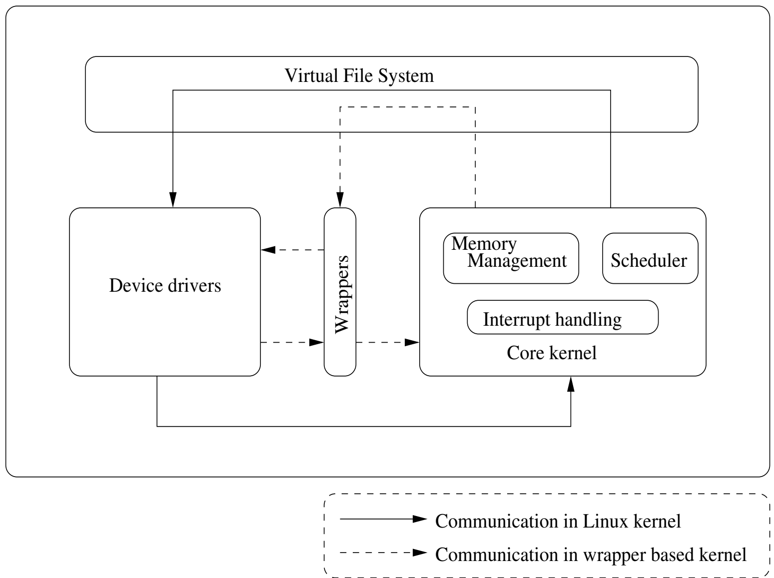


Figure 4. Design of wrapper based kernel

The kernel is abstracted into three major subsystems- memory, process and interrupt. These three subsystems are necessary for the device driver programmers. Some of the important wrappers and the corresponding services are listed in table I.

Memory management subsystem deals with the dynamic memory allocation for kernel usage. KernelMemoryAllocator wraps the functions that use kmem_cache data structure. This abstraction can be used to allocate memory for kernel data structures. The functionality provided by this abstraction is analogous to malloc and free of user space. MemoryManager wrapper provides the functionalities required to manage the memory of a process. NonContiguousMemory and Pager wrappers provide functionalities for dealing with virtual memory.

The Scheduler and Waitqueue wrappers provide the functionality for managing a process. The Waitqueue abstraction is widely useful for character drivers.

Timer and Interrupt management subsystems handle the timing measurement and interrupt handling tasks respectively. InterruptHanlder wrapper facilitates working with IRQ data structures, which is an important wrapper for all the device drivers. It contains many useful functions like request_irq, save_irq, can_request_irq, probe_irq_on, probe_irq_off, enable_irq, disable_irq. The irq handlers are defined as strategies (class Handler) whose instance is present in InterruptHandler class. A static handler function is statically registered in the wrapper framework. This handler function internally calls the polymorphic handler function of InterruptHandler which can be defined by user.

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

Table I. Wrappers for Linux kernel

| S.No | Subsystem | Wrapper | Wrapper Functions |
|---|---|---|---|
| 1 | Memory | KernelMemoryAllocator | kmem_cache_create, kmem_cache_reap, kmem_cache_shrink, kmem_cache_free |
| 2 | Memory | MemoryManager | mm_alloc, exit_mmap |
| 3 | Memory | VirtualMemoryManager | find_vma, vma_merge |
| 4 | Memory | NonContiguousMemory | vmalloc, vfree |
| 5 | Memory | Pager | alloc_pages, get_free_pages, free_pages, free_pages_bulk |
| 8 | Process | Waitqueue | init_waitqueue_head, init_waitqueue_entry, add_wait_queue, remove_wait_queue |
| 9 | Process | Scheduler | schedule_timeout_uninterruptible, schedule_timeout_interruptible |
| 10 | Interrupt | InterruptHandler | enable_irq, free_irq, disable_irq, local_irq_save, local_irq_restore, local_irq_enable requestIrq |
| 11 | Interrupt | Tasklet | tasklet_init, tasklet_disable_nosync, tasklet_disable, tasklet_enable |
| 12 | Timer | TimerManager | do_gettimeofday, update_times |

A simple class diagram of wrappers is shown in figure 5

## 5. Implementation

In this section we discuss the implementation of C++ wrappers for the Linux kernel.

A patch which gives the runtime support of C++ in Linux kernel is provided in [21]. The patch provides all the basic C++ features like virtual functions, exceptions, inheritance, in the Linux kernel with a minimal overhead. It reduces the developer's unnecessary effort in masking the variables in the Linux kernel that have names of C++ keywords like new, delete etc. If the user wants to use kernel API in C++ modules, the C header files should be enclosed between begin_include.h and end_include.h which are provided by the patch. But it is not a good practice to mix up C and C++ codes. So instead of using the kernel API directly, the C++ wrappers can be used.

The wrappers have been provided as include files in "*include/oowrap*". The wrappers are organized in a similar way to that of Linux source code organization. For example, the memory related wrappers are placed in "*oowrap/mm*", process related wrappers are placed in "*oowrap/process*". The process related functions are not placed separately in Linux kernel directory tree. The reason behind allocating a separate directory for process related wrappers is for the clear realization of the subsystems.

A simple wrapper "*InterruptHandler*" for interrupt handling is shown in figure 6. This is a wrapper around interrupt handling functions enable_irq, disable_irq, probe_irq_on, request_irq.

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

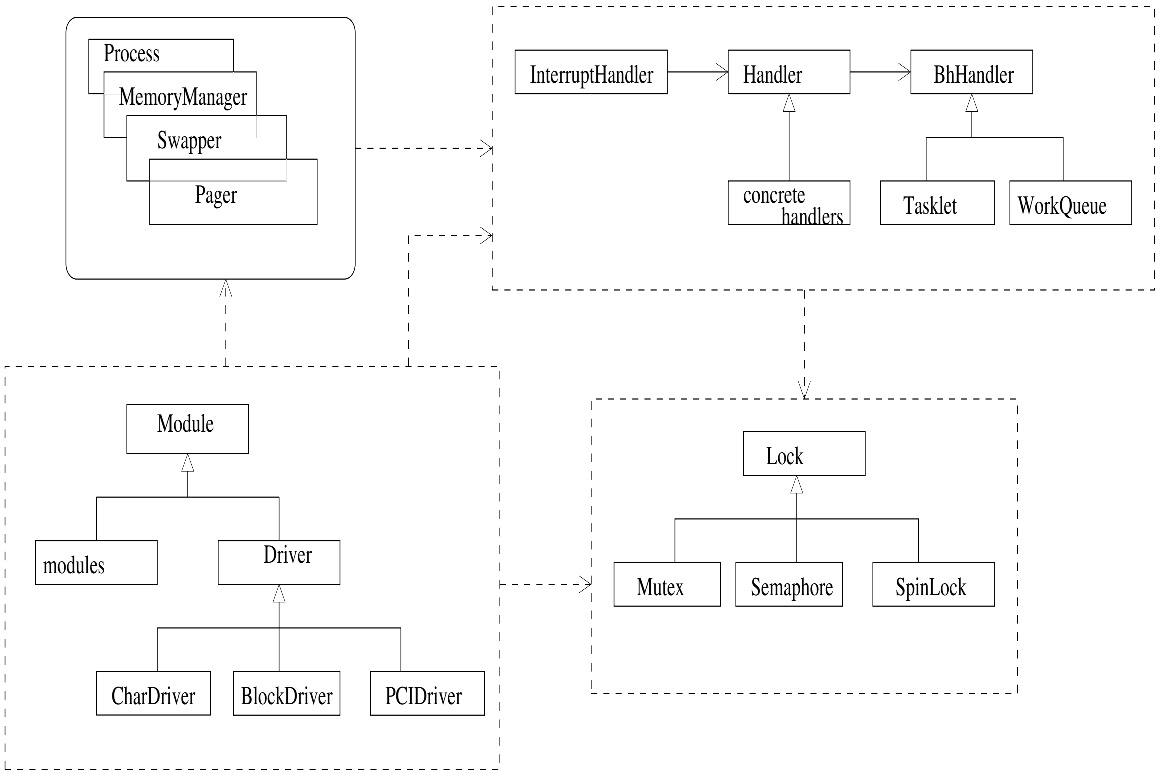


Figure 5. Class diagram of wrappers

If a developer wants to allocate memory for his kernel data structures, he creates an object of KernelMemoryAllocator and calls the alloc function in it, which forwards the call to the underlying kernel's kmalloc function.

The source code of Linux kernel along with wrappers is available for download in [20]

## 6. How To Use The Wrappers

The wrappers can be used for programming modules in an OO fashion. It provides the programmer the flexibility of programming completely in OO without mixing the procedural and OO paradigms. A simple example in figure 7 shows the flexibility of using C++ in kernel modules. This module does not give the complete code. It just shows how modules like device drivers can benefit by using C++ wrappers.

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

```
#include < begin_include.h >
#include < linux/slab.h >
#include < end_include.h >

class InterruptHandler
{
public:
      void  m_enableIrq(unsigned int irq)
      {
         enable_irq(irq);
      }
      void m_disableIrq(unsigned int irq)
      {
        disable_irq(irq);
      }

      void  m_freeIrq(unsigned int irq, void* dev_id)
      {
        free_irq(irq,dev_id);
      }
};
```

Figure 6. Implementation of wrappers

This example shows a simple driver program. Many vendors produce different drivers for a device. The applications may use any of these drivers. The Driver class defines the interface that different drivers should confirm to. ConcreteDriver1 and ConcreteDriver2 implement this interface. Applications can use any of these drivers by setting the reference of Driver to appropriate ConcreteDriver. *Terminal* is our wrapper which provides functions for printing onto the terminal.

The example shown is a polymorphic driver which can be switched at runtime based on the requirements of the user. The user has to write a driver that extends the *Driver* interface and define the abstract functions of *Driver*. Based on the concrete object the user assigns to the abstract *Driver* reference, the corresponding driver will be called.

To develop the same polymorphic driver in C, it needs considerably more development and testing effort than in C++. This is because, the developer should first implement a virtual table to support driver switching, and handle operations on this virtual table carefully while writing driver programs. This includes a lot of development and maintenance work. Also it is more prone to errors than the C++ version as it should be done manually, where as it is taken care by compiler if developed in C++.

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

```
#include<oowrap\Terminal.h>

class ConcreteDriver1:public Driver
{
public:
       virtual void devOpen()
       {
        Terminal  ob;
        ob.print("DEVICE FILE OPENED");
       }

       virtual void devRead()
       {
         Terminal ob;
         ob.print("READING FROM DEVICE");
         ....
         ....
       }

       virtual void devWrite()
       {
         Terminal ob;
         ob.print("WRITING TO DEVICE");
         ....
         ....
       }

       virtual void devRelease()
       {
         Terminal ob;
         ob.print("RELEASING DEVICE");
       }
}
```

(a)

```
class ConcreteDriver2 : public Driver
{
public:
       virtual vod devRead()
       {
 ....

       }
 ....
 ....

}
```

(b)

```
Driver* driver[2];
driver[0]=new ConcreteDriver1();
driver[1]=new ConcreteDriver2();
```

(c)

Figure 7. Usage of wrappers

## 7. Case Study: Network Device Driver

Device drivers are one of the main components that interact with the core kernel. In Linux, device drivers can be inserted at runtime as modules or can be compiled statically into the kernel.

We have re-engineered two network device drivers called ne2k_pci.c and 8210.c from C to C++. The ne2k_pci is device driver for PCI NE2000 clones and 8210.c is the driver for 8210

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

```
class ne2k_pci_OO
{

private:
       Terminal obj;
       InterruptHandler interrupt_handler_obj;
public:

       int ne2k_pci_open();
       int ne2k_pci_close();
       void ne2k_pci_block_input();
       void ne2k_pci_block_output();
        .........
        .........
        void ne2k_pci_get_drvinfo();
        int __devinit ne2k_pci_init_one ();

}ne2k_pci_obj;
```

Figure 8. A sample network driver reengineered to C++

series Ethernet cards. These drivers are used frequently when the network connection of the system is used, enabling us to estimate performance degradation caused due to wrappers.

The functions of the driver are grouped into a class structure and where ever the kernel interface is used, we replaced them with the instances of the appropriate C++ wrappers. A small part of the network driver code is shown in figure 8.

The reengineered driver uses two wrappers, the InterruptHandler and Terminal which are related to interrupt handling and printing related functions. The driver was completely written in C++, without using any of the kernel interface directly.

### 7.1. Evaluation of case study

Apart from being an OO interface to the Linux kernel, we identified certain qualities (in section 4.1) in the domain of operating systems, which improve the usability of the wrappers. Among these qualities, structure identifiability, house keeping within wrappers and reusability affect the development and maintainability effort of kernel modules. In this section, we explain how these qualities are achieved using wrapper based design. As performance is considered the most important quality factor for kernel modules apart from the other three, we provided detailed analysis of it in section 7.2

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

1. *Structure identifiability:*
   During the development of kernel modules, it becomes easy for the developer to use abstractions which represent the concepts of the domain (operating system). Hence the kernel is abstracted into a set of high-level wrappers, which directly correspond to different subsystems of the kernel. The developer just needs to learn a set of wrappers of the kernel to develop a module. This decreases the learning time which in turn reduces the development time. For example, to develop the network driver shown in figure 8, all that the developer needs to know about the kernel is the InterruptHandler (for handling interrupts) and Terminal (for printing driver debug information) wrappers.
2. *Housekeeping within wrappers:*
   During kernel module development, care should be taken to ensure certain aspects like right parameters are being passed to functions, the locking and unlocking related functions are called correctly before and after a specific function call and a lock has been achieved before using a data item etc. These aspects are repetitive and failing to do so may some times cause a severe corruption to kernel data structures and may even lead to a kernel crash. Ensuring these aspects during driver development complicates the developers work. So encapsulating them in the wrappers reduces considerable development effort.
   For example, in the network device driver, the parameter hdr is identified as critical parameter and its incorrect usage might lead to a kernel crash [8]. Care should be taken to pass correct value to this parameter. A housekeeping wrapper, called ne2kpci_filter was associated with the driver, which automatically checks the usage of this parameter and restricts improper usage. Similarly, locks can be obtained by the use of scoped locking pattern [7] in wrappers.
3. *Reusability:*
   The device driver hierarchy provides blackbox reuse, where in order to develop a new driver, the user just needs to extend it from an appropriate class of driver and define the specific parts of that driver. The reuse can also be achieved by abstracting the architecture specific parts of code and composing the driver with specific architecture object. In either case, the reuse achieved is considered better than the reuse achieved in procedural paradigm.

### 7.2. Performance Evaluation

We evaluated the performance of C driver in the original Linux kernel and C++ enabled Linux kernel. We also measured the performance of C++ driver in C++ enabled Linux kernel.

The performance of the C and C++ drivers is observed under both idle and load conditions. The performance is measured using the *netperf* tool [22] which is a benchmarking tool for network applications. The tool performs various networking tests by changing the input data size and provides results in terms of transaction rate (a transaction being one send and receive).

As expected, there is a degradation of performance in the C++ driver. The degradation is as low as 2%. Given the advantages of writing the device driver in C++, the overhead of 2% in terms of performance is minimal.

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

*Prepared using* speauth.cls

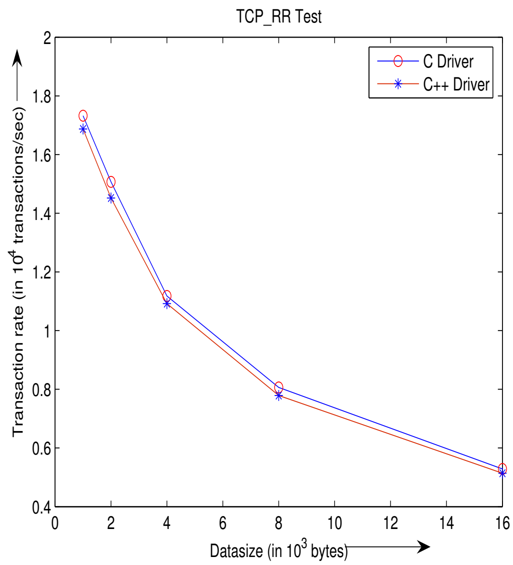


Figure 9. Driver performance for TCP

The graphs corresponding to TCP and UDP tests performed by *netperf* tool are shown in Figure 9 and Figure 10.

Further experimentation was carried out using *Hbench* micro benchmarking [19] tool to compare the performance of procedural and object oriented network driver on the modified Linux kernel. *Hbench* is a collection of benchmarking tools which measure various aspects of OS such as memory read, write, network bandwidths and latencies. The readings are taken over 10 iterations and the median of the values is presented. We have limited our experimentation to the tests corresponding to the network as the main objective was to check the performance of network driver. The tests performed using *Hbench* tool are:

1. TCP bandwidth test (bw_tcp)
   This test measures the bandwidth attainable when transferring data through a TCP connection between two processes. This test is performed using both C and C++ drivers, varying the buffer size being transmitted between the processes. The test outputs the attainable bandwidth in mbps and the more the value the better the performance. A slight degradation in the performance is observed with C++ driver as shown in figure 11.
2. TCP connection latency test (lat_connect)
   This test measures the latency in establishing a TCP connection between the two hosts. It outputs the latency in micro seconds. The less the value, the better the performance.

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

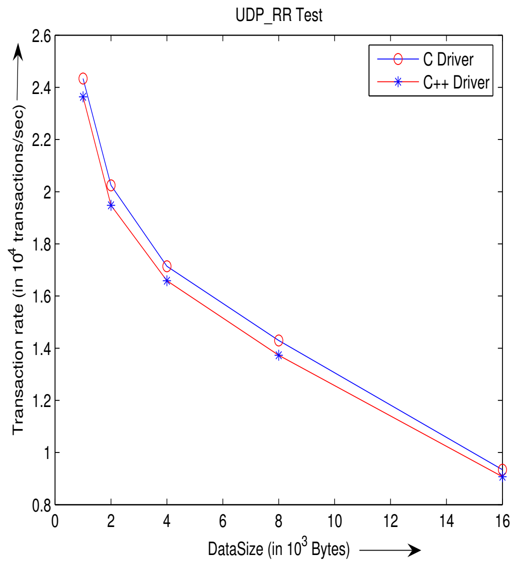


Figure 10. Driver performance for UDP

3. TCP transaction latency test (lat_tcp)
   This test measures the latency of a 1-byte ping-pong between two processes connected via a TCP connection.
4. UDP latency test (lat_udp)
   This test measures the latency of a 1-byte ping-pong between two processes using UDP datagrams to transfer the data.
5. RPC latency (lat_rpc)
   This test measures the latency of RPC calls via UDP and TCP transports.
   The performance results collected using *Hbench* tool are shown in figures 11,12.
   In addition to slight degradation in performance, we have observed a slight increase in the size of the wrapper based C++ kernel when compared to that of the original kernel. An increase of 118KB is observed in modified kernel image size. This change is attributed to the addition of C++ libraries to the kernel.

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

*Prepared using* speauth.cls

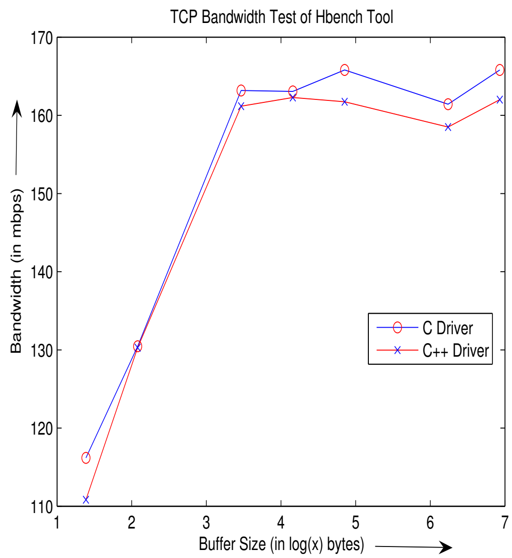


Figure 11. TCP Bandwidth tests of Hbench tool

## 8. Related Work

### 8.1. Object Oriented Operating Systems

Choices is one of the most popular object oriented operating system developed in C++ [1]. Choices encourages reuse by including mechanisms for inheritance and polymorphism. It provides a framework for reusing the design with in the OS.
Spring is an another object oriented operating system which uses interface definition language (IDL) for specification of system interfaces [2]. It separates the interface from implementation so that both can vary independently.

### 8.2. OO Wrappers For Operating Systems

Wrappers have been provided for Mach operating systems [3]. Mach is a micro kernel based operating system in which only minimal components are kept in kernel space. All the remaining subsystems run in the user space. So this includes a lot of communication between the subsystems that run in the user space and the kernel. The main objective of providing OO wrappers for Mach kernel is to hide state management issues involved in communicating with micro kernel, and to make the communication with kernel easier [3].

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

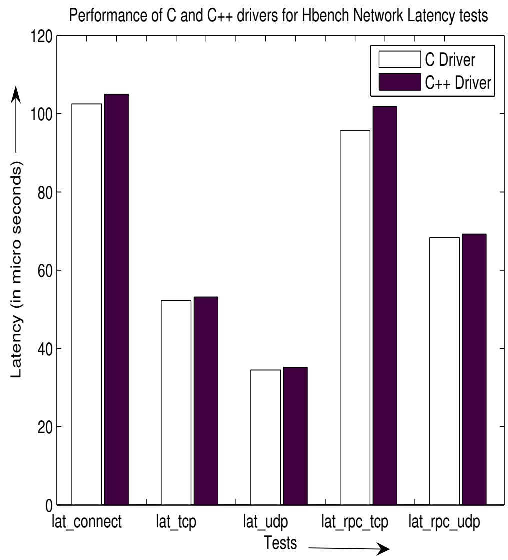


Figure 12. Network latency tests of Hbench tool

Our work is different from that of Mach work in the sense that we have provided wrappers in the kernel space for in-kernel API for the benefit of kernel modules like device drivers. Kernel modules can access kernel either directly through kernel API or through wrapper interface. In this way the existing modules need not be changed and the new modules can be developed using the wrapper interface using C++.The wrapper functions have been made inline wherever possible, so as to reduce the performance overhead.

### 8.3. I/O Kit

I/O Kit is a device driver model specially built to support the MACH kernel of Mac OS X. It is a completely object oriented framework built in C++ with a layered architecture. The main architectural features of I/O Kit include hardware modeling and layering of driver objects, driver matching, I/O kit registry and I/O catalog, and I/O class hierarchy [24].

While I/O kit is designed specifically for device drivers, the wrappers we provide can be used to build any of the external kernel modules for Linux. I/O Kit handles the requests to a driver by processing it at different layers (which is developed in C++), where as in our wrappers we delegate it to the Linux kernel (which is developed in C). So we expect the performance of device drivers to be better in our case.

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

## 9. Conclusion And Future Work

C++ wrappers have been built for the core kernel of Linux operating system and it can be used by kernel module programmers for developing their programs in C++, which enhances maintainability and reuse for their programs. A number of device drivers have been re-engineered to C++ and a detailed performance study has been presented.
The wrapper support for Linux kernel opens up new options of intercepting the messages that go to the kernel. We are exploiting this option, to provide a new security mechanism based on biometric devices. We are also studying the usability of filter objects [13] in Linux kernel to enhance the security of the kernel. Using wrappers and filter object model, it is possible to build interception of messages to kernel. This approach can provide device driver isolation as provided by NOOKS [12]. The wrappers for Linux kernel can be downloaded at http://dos.iitm.ac.in/MOOL/

**ACKNOWLEDGEMENTS**

The authors acknowledge the Department of Information Technology, Government of India, for providing financial support for the Linux redesign project. The project review team has provided critical reviews to the project.

**REFERENCES**

1. Roy H. Campbell, Nayeem Islam, David Raila, Peter Madany. *Designing and implementing* Choices*: an object-oriented system in C++.* Communications of the ACM 36, 9, 117-126, 1993.
2. Islam N, Campbell R.H. *Latest developments in operating systems.* Communications of the ACM 39, 9, 38–40, 1996.
3. Kurtzman S, Dattatri K. *Design goals of object-oriented wrappers for the Mach micro kernel.* Compcon'95, 367–372
4. Wheeler D.A. *More Than a Gigabuck: Estimating GNU/Linuxs Size.* Sixth International Workshop on Program Comprehension, 1998.
5. Yokote Y. *The Apertos reflective operating system: the concept and its implementation.* ACM SIGPLAN Notices, 27, 10, 414–434, 1992
6. Reddy V.K, Janakiram D. *Cohesion Analysis in Linux Kernel.* Proceedings of the XIII Asia Pacific Software Engineering Conference, 461–466, 2006.
7. DC Schmidt, H Rohnert, M Stal, D Schultz *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* John Wiley & Sons, Inc. New York,USA 2000
8. Nagaraju Pendyala. *Implementation of message filters in Minimalistic Object Oriented Linux,* MTech Thesis, http://dos.iitm.ac.in/MOOL
9. Yu L, Schach S.R, Chen K, Offutt J. *Categorization of common coupling and its application to the maintainability of the Linux kernel.* IEEE Transactions on Software Engineering, 30, 10, 694–706, 2004
10. Yu L, Schach S.R, Chen K, Heller G.Z, Offutt J. *Maintainability of the kernels of open-source operating systems: A comparison of Linux with FreeBSD, NetBSD, and OpenBSD.* The Journal of Systems & Software, 79, 6, 807–815, 2006.
11. Stroustrup B. *The design and evolution of C++.* ACM Press/Addison-Wesley Publishing Co, 1995.
12. Swift M.M, Annamalai M, Bershad B.N, Levy H.M. *Recovering device drivers* Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation 2004
13. Joshi R.K, Vivekananda, N, Ram D.J. *Message filters for object-oriented systems.* Software-Practice and Experience, 27, 6, 677–699, 1997

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.

14. Gamma E, Helm R, Johnson R, Vlissides J. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co, 1995.
15. D Schmidt, *Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes*, C++ Report, 11, 2, 40–50, 1999.
16. G. Canfora, A. Cimitile, M. Munro. *An Improved Algorithm for Identifying Objects in Code.* Software-Practice and Experience, 26, 1, 25-48, 1996.
17. Harald Gall, R Klösch. *Finding Objects in Procedural Programs: An Alternative Approach.* Proceedings of the Second Working Conference on Reverse Engineering , 208–216, 1995.
18. Liu S.S, Wilde N. *Identifying objects in a conventional procedural language: an example of data design recovery.* Conference on Software Maintenance, 266–271, 1990.
19. Brown A, Seltzer M. *Operating system benchmarking in the wake of lmbench: a case study of the performance of NetBSD on the Intel x86 architecture.* Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, 214-224.
20. Minimalistic Object Oriented Linux Homepage http://dos.iitm.ac.in/MOOL [03 August 2007]
21. C++ in Linux kernel. http://netlab.ru.is/exception/linuxcxx.shtml [03 August 2007]
22. Netperf Home page. http://www.netperf.org [03 August 2007]
23. The Linux-kernel mailing list FAQ. http://www.kernel.org/pub/linux/docs/lkml/ [03 August 2007]
24. I/O Kit Fundamentals. http://developer.apple.com/DOCUMENTATION/DeviceDrivers/Conceptual/IOKitFundamentals/ [03 August 2007]

This is a preprint of an article accepted in Software—Practice And Experience Copyright © 2007 John Wiley & Sons, Ltd.