Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
The Journal of Systems and Software 79 (2006) 163–175
The Journal of Systems and Software
www.elsevier.com/locate/jss

Hotswapping Linux kernel modules

Yueh-Feng Lee *, Ruei-Chuan Chang

Department of Computer and Information Science, National Chaio Tung University, Hsinchu 30050, Taiwan, ROC

Received 15 March 2004; received in revised form 17 May 2005; accepted 21 May 2005
Available online 12 July 2005

Abstract

Contemporary operating system kernels are able to improve their functionality by installing kernel extensions at runtime. However, when an existing kernel extension needs to be upgraded, it must be completely removed before the new kernel extension is installed. Consequently, the new kernel extension needs to be run from the beginning, which also influences the applications using this kernel extension.

This work describes the design and implementation of a Linux module system that supports hotswapping, in which a module can be replaced while it is in use. Rather than completely removing the old module, the new module can inherit the state held by the old module so the dependent applications are not affected. For example, a Linux file system module can be hotswapped without unmounting the corresponding partitions and terminating the applications that use these partitions.

The proposed system is implemented on Linux kernel 2.6.11. Existing modules can be loaded into the hotswap system without change and can be hotswapped by changing only a few lines. Additionally, the hotswap system does not impose any runtime overhead on module invocations.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Dynamic software update; Hotswapping; Linux kernel; Module; Operating system

1. Introduction

Modern operating system kernels are normally extensible, allowing kernel extensions to be installed into the system after booting. Kernel extensions can be applied in several applications, such as hardware drivers, file system drivers, and networking protocols. By installing various kernel extensions, an operating system can provide various services without rebuilding the kernel.

However, the traditional approach to upgrading, an existing kernel extension involves shutting down the applications that depend on this kernel extension, removing the old kernel extension, installing the new one, and restarting the applications. Since the old kernel extension's state is completely lost, removing it has severe side effects. For example, a web server program typically depends on the file system drivers. Consider the case of upgrading a file system driver for a web server running banking applications. First, the administrator needs to shut down the web server program, since web pages are stored in the partition managed by this file system driver. Next, the administrator needs to unmount this partition because the file system driver cannot be removed while it is in use. Then, the administrator needs to uninstall the old driver, install the new driver, mount the partition, and finally restart the web server program. Consequently, upgrading a file system driver is similar to a reboot. Furthermore, when the web server is shut down, all existing web connections are broken, causing ongoing banking transactions to fail.

This problem can be solved by providing a dynamic update facility in the operating system kernel. In such a system, old kernel extensions can be replaced by new

* Corresponding author. Tel.: +886 3 5712121 56656; fax: +886 3 5721490.

E-mail addresses: yflee@os.nctu.edu.tw (Y.-F. Lee), rc@cc.nctu.edu.tw (R.-C. Chang).

0164-1212/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2005.05.011

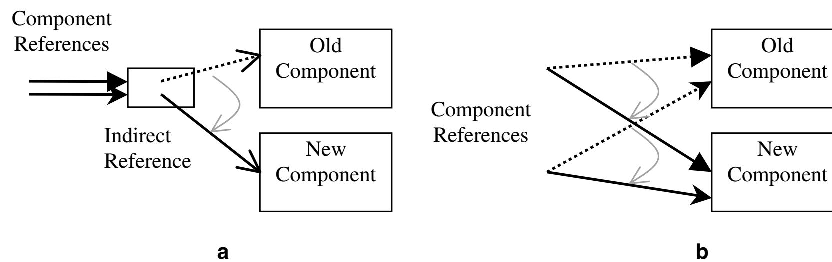

Fig. 1. (a) Indirect swap and (b) direct swap.

extensions without affecting the dependent applications. Dynamic update not only improves system availability, but also eliminates application shutdown and restart times, cuts the cost of backup servers, and lowers the system maintenance time.

Dynamic update can be implemented in two ways, by interposition or hotswapping (Soules et al., 2003). Interposition alternates the component behavior by attaching code that is executed before or after the original code, but cannot fix software faults since it does not change the original code. Hotswapping replaces the original code with a new implementation and can manage the components in two ways, by indirect swapping or direct swapping (Fig. 1). Indirect swapping associates a component with an indirect reference. A component can be swapped by switching the indirect reference rather than changing all the component references. Indirect swapping is simple but has extra runtime overheads due to the indirection layer. Since direct swapping has no additional layer between the component and component references, each component reference has to be updated during hotswapping. Although more complex than indirect swapping, direct swapping it is appropriate for performance-sensitive systems since there is no extra runtime overhead.

K42 is a research-oriented operating system with a new kernel design supporting indirect swapping (Soules et al., 2003). However, neither indirect swapping nor direct swapping has been achieved in existing operating systems because hotswapping generally needs special software architectures, such as object-oriented and component-based environments. To support hotswapping, both the operating system kernel and existing kernel extensions must be modified. Additionally, hotswapping may degrade the system performance, for example by using indirect swapping.

Linux kernel extensions are called modules. Introducing hotswapping to Linux modules is difficult because the kernel and the modules are neither object-oriented nor component-based. This work proposes a direct hotswapping system for Linux modules. The fundamental infrastructure of the original module system is not changed. Additionally, the system imposes no runtime overhead on module invocations. An existing module can be installed into the system without modification, and can be hotswapped by changing only a few lines. The hotswap system is based on kernel version 2.6.11 and is currently implemented on a uniprocessor system.

The rest of this work is organized as follows. Section 2 describes the general requirements of hotswapping and specific challenges involved in hotswapping Linux modules. Section 3 presents the design and implementation of the hotswap module system. Section 4 describes an example and performance evaluation of hotswapping. Section 5 introduces related work. Finally, conclusions are drawn in Section 6.

2. Hotswapping and Linux modules

2.1. General requirements of hotswapping

A system that supports hotswapping should meet several requirements. Below, four proposed requirements for hotswapping are introduced (Moazami-Goudarzi, 1999; Soules et al., 2003): component boundaries, mutually consistent states, component state transfer and external references.

Component boundaries: Component boundaries help the system identify the target to be hotswapped. In object-oriented or component-based programming environments, an object or a component itself already defines the component boundary, because both the required data and functions are encapsulated into the object or the component. However, a situation in which a component is not written using such environments makes it difficult to identify the component boundary.

Mutually consistent states: A system must ensure that all components are in a mutually consistent state after hotswapping. Two mechanisms are present to preserve mutually consistent states: consistency through recovery and consistency through avoidance (Moazami-Goudarzi, 1999). The recovery mechanism permits hotswapping to start in an unsafe state but the resulting inconsistent state can be recovered later. The avoidance mechanism ensures that hotswapping starts only in a safe state.

Component state transfer: During hotswapping, the state held by the old component must be properly transferred to the new component. For instance, in object-

oriented programming languages such as Java, object states can be transparently transferred with the help of the runtime system (Arnold et al., 2000). However, non-object-oriented programming languages do not have mechanisms for transferring states transparently.

External references: A component can be used by other components through external references. An external reference may refer to a component, a data structure or a function defined by a component. When a component is hotswapped, each external reference must be redirected to the appropriate part of the new component.

2.2. The challenges on designing a hotswap module system for Linux

As mentioned earlier, making the Linux module system meet these requirements is difficult because the Linux kernel and the module system are neither object-oriented nor component-based. This section describes five specific challenges and possible solutions on hotswapping Linux modules.

For ease of discussion, this work uses *kernel* to refer to the monolithic part of the Linux kernel, excluding the modules. The term *module system* is used to refer to the part of the kernel that manages the modules. This discussion is based on the kernel version 2.6 unless otherwise indicated. The introduction and programming of Linux modules can be found elsewhere (Bovet and Cesati, 2002; Rubini and Corbet, 2001).

2.2.1. Mutual consistency preserving scheme

Consistency through recovery is unsuitable for operating system kernels because it requires more system support and runtime overhead during recovery. Instead, consistency through avoidance can be used. At least three approaches for achieving consistency through avoidance exist: synchronization, contention, and quiescence.

The synchronization solution was first proposed by Kramer and Magee (1990) and then improved by Moazami-Goudarzi (1999). In this method, a component must differentiate between active and passive states. A component enters the passive state after receiving the passive signal. A component in the passive state cannot serve any requests and hotswapping can start only when the component is in this state. However, all Linux modules must be modified, or even redesigned, to support this synchronization protocol.

In contrast, the contention and quiescence methods do not differentiate between the active and passive states. The contention method uses synchronization techniques such as semaphores to reach a safe state for hotswapping. However, this method also requires modifying the modules because a synchronization technique must be used before invoking a module function.

The quiescence method does not use any synchronization technique to reach the safe state. This method continuously monitors the component behavior and discovers a system state in which the target component is not in use. Hotswapping can start in this state, called the quiescent state. This method does not need to modify any module function, although the module system needs to implement a mechanism to detect the quiescent state.

2.2.2. No concrete module states

In object-oriented programming languages such as C++, object states can be mapped directly to component states. However, Linux modules are written in C, which does not provide a general method to map language constructs to the component state. Therefore, the module state must be defined either implicitly by the module system or explicitly by the module programmer. When explicitly defining the module state, the programmer should specify which elements belong to the module state. Explicit definition is more flexible than implicit definition but is also a programming burden. By contrast, if the module system defines the module states implicitly, programmers do not have to be concerned with them.

2.2.3. Undefined symbols can be resolved only once

Symbol exporting is the standard mechanism for export services within the Linux kernel. The kernel can provide services to modules by exporting its symbols. Modules can also provide services to other modules in the same manner. A symbol may be a function or a variable defined in the source. When the kernel is loaded, the symbols it exports are stored in the kernel symbol table. Similarly, each module has a module symbol table to store its exported symbols.

If a module uses a symbol defined by another module, the symbol is left undefined after compilation. During module loading, the undefined symbol address is linked to the actual implementation of that symbol. First, the kernel symbol table is searched. If the symbol is not found, each module symbol table is searched in turn. This process is called *resolution*. However, the current Linux module system only allows module symbols to be resolved once, during module loading. Suppose that module A depends on the symbols defined by module B. If module B is hotswapped by module B′, which is not loaded into the same address, then module A cannot obtain the correct addresses of the new symbols.

Three solutions are available to handle this problem: indirect swapping, dynamic linking, and dynamic resolution and relocation. Indirect swapping, as described above, can tolerate the changing of symbol addresses because of the indirection layer between the symbol implementation and its caller, but it suffers from a performance penalty. Dynamic linking is widely used in UNIX shared libraries. With the assistance of a dynamic linker, executables compiled with dynamic

linking support can access dynamic shared libraries in an address-independent manner. However, the Linux kernel lacks a built-in dynamic linker. Without needing the dynamic linker and the dynamic linking format, dynamic resolution and relocation can links a symbol statically many times. For example, when module B is hotswapped by module B′ which is loaded into another address, then the undefined symbols of module A can be resolved and relocated so that module A can refer to symbols defined by module B′.

2.2.4. Variable address passing

Variable address passing usually occurs when a module exports its services to the kernel. For example, a file system module should pass its variable address to the kernel when invoking the *register_filesystem* function. However, the address kept by the kernel would become invalid when the module is replaced by a new one.

The problem of variable address passing can be solved in three ways: adaptive kernel functions, external variable allocation, and static address sections. Adaptive kernel functions are kernel functions that are aware of hotswapping. For example, an adaptive *register_filesystem* function can be invoked by the same file system module multiple times. When invoked again by the same module, it drops the old variable address and keeps the address passed by the new module. External variable allocation is a programming technique in which the memory space required by a variable is not allocated inside the module. For instance, a variable can be allocated outside the module by invoking the *kmalloc* function. Since the memory space is allocated independently, hotswapping does not change the address of this variable. A static address section is a region in which the variables do not change their addresses during hotswapping. Thus, the variables contained in the static address section are still valid after hotswapping. Although the effects of external variable allocation and static address section resemble each other, static address sections need fewer modifications to the module source.

2.2.5. Module descriptor management

A module descriptor is a unique variable defined inside the module, but managed by the kernel. The module descriptor is contained in a special section called *.gnu.linkonce.this_module* and the kernel maintains all of the active module descriptor *s* in the *modules* list. Since both the old and the new modules have their own module descriptors, they must be properly managed during hotswapping.

Two elements of the module descriptor need special treatment during hotswapping: the reference count and the use list. The reference count prevents a module from being removed while still needed. Reference counts are maintained by each module's callers rather than by the module system. A reference count is incremented by the *try_module_get* function and decremented by the *module_put* function. Note that reference counts are not appropriate for determining whether a module can be hotswapped because usually they are updated lazily.

A use list prevents a module from being removed when other modules still depend on its symbols. For instance, when module B needs a symbol defined by module A, module B is appended to the use list of module A. Simultaneously, the reference count of module B is also incremented. Module B is not removed from the use list of module A until module B is removed from the system.

A successful module descriptor management should meet two requirements. First, the reference count and use list of the old module descriptor should be correctly transferred to the new module descriptor. Second, the kernel or other modules that already use the old module descriptor must be able to use the same descriptor after hotswapping. Module descriptors can be managed in two ways: single descriptor and multiple descriptors. In a single-descriptor system, only one module descriptor is valid at one time. In a multiple-descriptor system, when a new module replaces an old module, both the old and the new descriptors remain valid.

3. Design and implementation

This section, first describes the design principles, and then introduces the selected solution in response to each problem described earlier. Next, important steps involved in hotswapping are described. Additionally, some related issues are discussed at the end of this section.

3.1. Design principles

Preserve backward compatibility as much as possible: Backward compatibility is important for two aspects, loading and hotswapping. All existing modules must be successfully loaded into the new system, and should be hotswapped in the new system with few or no modifications.

Change module system source only: Only the source of the module system is modified, that is, the architecture-dependent and architecture-independent parts of module.c and module.h. Other parts of the kernel remain unmodified.

No modification to existing data structures: Modifying existing data structures may lead to compatibility problems. Therefore, the system adds new data structures instead of modifying existing ones.

No new system call: No new system calls are introduced. Additionally, the arguments of an existing system call are not modified. Therefore, the new module system is compatible with existing module utilities, such as *insmod*, *rmmod*, and *lsmod*.

No overhead during normal execution: Since the kernel is sensitive to performance, only mechanisms that do not impose extra runtime overhead are selected.

3.2. Detailed implementation

3.2.1. Preserve mutual consistency using stack trace

The quiescence approach is chosen here to preserve mutual consistent states, where the quiescent state is detected using stack tracing. A module is in the quiescent state when none of its functions are in any process stack. The stack tracing code examines each frame of each process's kernel stack. If any stack frame contains a return address that falls into the address range of the target module, then the target module is not quiescent and therefore cannot be hotswapped immediately. Additionally, a busy was implemented to reduce stack tracing time when a module has been detected as not quiescent. When a process's stack frame contains a return address belonging to the target module, the process PID is added to the busy list. When the stack is traced again, the processes contained in the busy list are checked first. If the non-quiescent condition is still found, other processes in the system do not have to be checked. In other words, the system only needs to examine all of the processes if the system is quiescent. The busy list is developed because not every process in the system invokes the module being hotswapped. If a process ever invokes the target module, it is likely to invoke the same module in the future. Notably, the hotswap system only checks the kernel mode stacks since module functions are invoked only in kernel mode.

3.2.2. Replace handler and module state

Generally, programmers can treat global variables as the module state. A replace handler function transfers the values of global variables to the new module. Programmers can transfer global variables by prefixing their names with *_old_*, allowing the new module to access two versions of the same variable without conflict. For example, if the old module defines a variable *i*, then the new module's replace handler can access this variable using the variable name *_old_i*.

Since global variables are generally defined either in the *.bss* section or the *.data* section, the module system could automatically copy these two sections from the old version to the new version. However, this feature is not implemented for two reasons. First, the new module does not need all the old variables because some variables may need new initial values. Second, a variable may be declared in the new module as a different type from that in the original code, and thus may have a different size. Such a variable cannot be correctly and automatically transferred.

Fig. 2 shows how to handle an evolving variable using the *_old_* prefix. In this example, the old module

```
struct struct_a
{
  int i;
  int j;
}
static struct struct_a a;
```

a

```
struct old_struct_a
{
  int i;
  int j;
}

struct struct_a
{
  int x;
  int i;
  int j;
}
static struct struct_a a;
extern struct old_struct_a _old_a;
```

b

```
static void replace_handler
{
  a.x = 100;
  a.i = _old_a.i;
  a.j = _old_a.j;
}
```

c

Fig. 2. Dealing with an evolving variable: (a) definition and declaration used in the old module; (b) definitions and declaration used in the new module; (c) corresponding replace handler.

and the new module use different definitions of the variable *a*, both of which are called *struct_a*. The old *struct_a* defines only two members, while the new *struct_a* adds a member to the beginning of the structure. The old *struct_a* is also defined in the new module but has to be renamed as *old_struct_a*. The old variable *a* is declared in the new module as an external variable named *_old_a*. Before invoking the replace handler, the module system automatically attaches the symbol *_old_a* to the address of the old variable *a* so that the old variable *a* can be correctly retrieved.

3.2.3. Dynamic resolution and internal data structures

Dynamic resolution and relocation are used since it does not impose overhead during module invocations. To support dynamic resolution and relocation, each module object file is backed up in a region of memory allocated by the module system. The module system preserves old object files because dynamic resolution and relocation requires symbol table and relocation sections, which appear only in module object files. A new data structure, the *replace_handle* structure, is added to hold the addresses of the backup object file and its important sections.

Dynamic resolution and relocation is performed on modules that refer to the symbols defined by the target module. To save resolution time, dynamic resolution does not resolve kernel and module symbols not defined by the target module, because these symbols do not change addresses during hotswapping.

3.2.4. Static address sections

Static address sections are used to resolve the variable address passing problem. The module system defines two static address sections: *static_new and static_old*. If a variable is placed in the *static_new* section, then its content is overwritten by the new module. If the variable is placed in the *static_old* section, then its content is not changed. The programmer can put a module variable in a section by adding a tag to the variable declaration. For instance, in the GCC environment, the tag *attribute(section(static_old))* can put the declared variable to the *static_old* section. Notably, the *static_old* section can also be used to transfer module states. If a variable is placed in the *static_old* section, then it is automatically transferred to the new module because it will not be overwritten by the new module. The *static_old* section can also transfer the static variables declared in a module function. When a static function variable is placed in the *static_old* section, the new version can automatically obtain the correct value without additional treatment.

3.2.5. Module descriptor management

The hotswap system only allows a single module descriptor. However, the old module descriptor must be preserved until the end of the hotswap procedure since a hotswap procedure may fail at any stage, and the old module must be still usable if hotswapping fails. The old descriptor is backed up to a safe memory region when hotswapping begins. At the end of hotswapping, important information of the old descriptor, such as the reference count and the use list, are transferred to the new descriptor. Additionally, module descriptors are handled similar to a static address section because the addresses of module descriptors are often passed to the kernel. If the new descriptor does not have the same address as the old descriptor, then the kernel would lose track of the module descriptor after hotswapping.

3.3. Module loading and hotswapping

The implementation described earlier is reflected in the module loading and hotswapping procedures. In kernel version 2.6, a module loading procedure is started by the *insmod* utility and is processed by the *sys_init_module* system call. The module loading procedure of kernel 2.4 differs significantly from that of kernel 2.6 and can be found elsewhere (Bovet and Cesati, 2002).

3.3.1. Module loading and hotswapping in the hotswap system

The hotswap system slightly alters the semantics of the *sys_init_module* system call. If *sys_init_module* is invoked with an unloaded module, then the module system loads the module. If *sys_init_module* is invoked with a module already in the system, then the module system performs hotswapping.

Figs. 3 and 4 show the memory layout of module loading and module hotswapping, respectively. The module loading and hotswapping procedures are described together as follows:

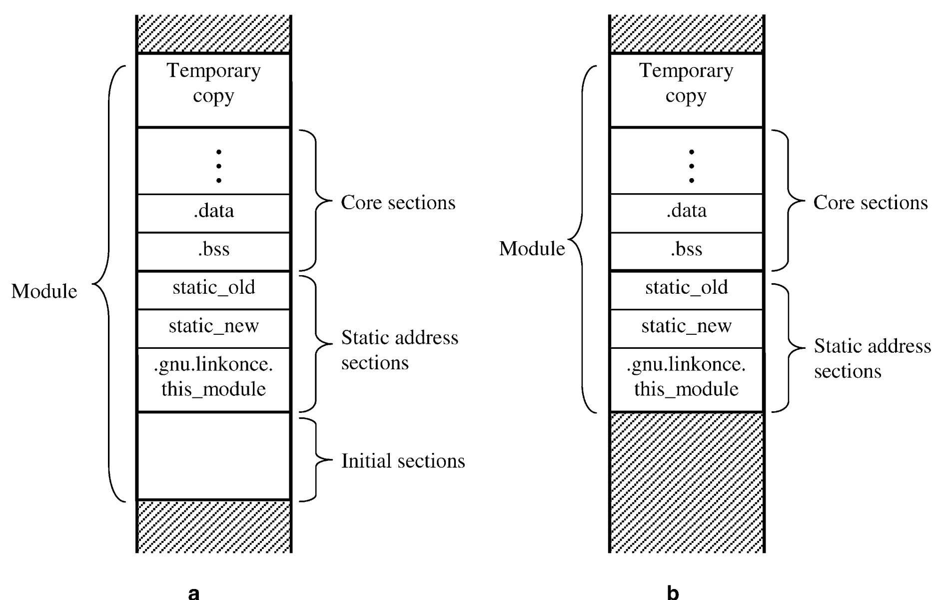

Fig. 3. Memory layout of module loading in the hotswap system: (a) during loading, (b) after loading.

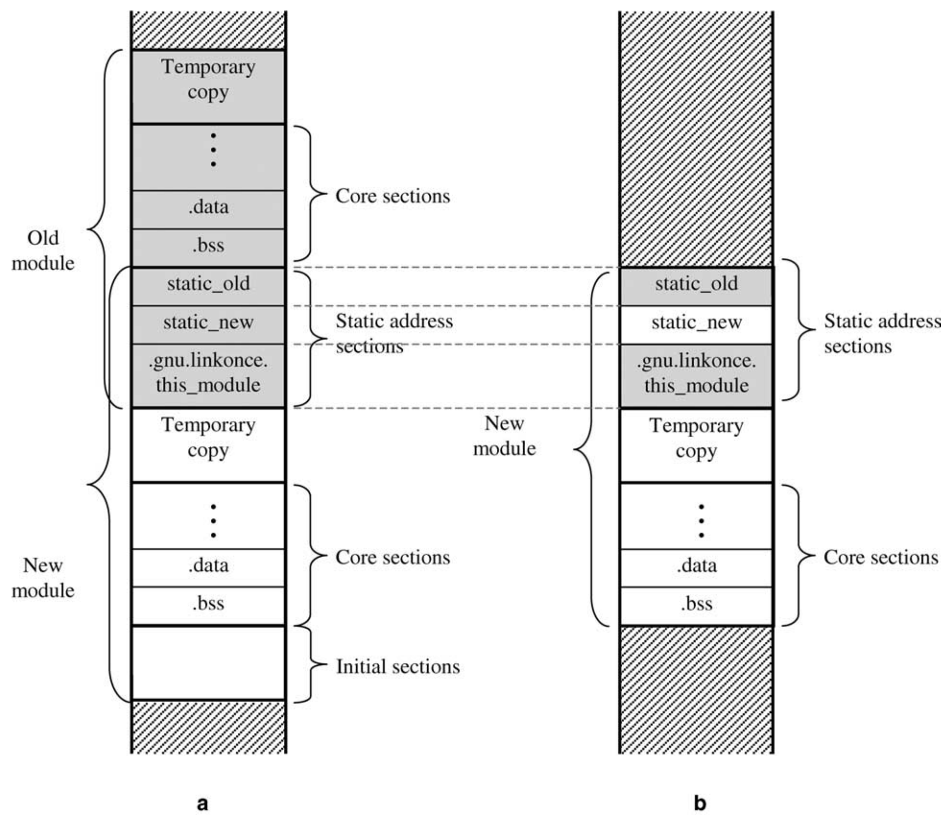

Fig. 4. Memory layout of module hotswapping in the hotswap system: (a) during hotswapping, (b) after hotswapping.

1. The *insmod* utility reads the module object file into the memory, maps the object file's memory region, and passes the memory region's handle to the *sys_init_module* system call.
2. The *sys_init_module* system call invokes the *load_module* function, which performs the following steps:
   (a) Allocate a temporary memory region and copy the content indicated by the memory handle to the temporary region. Then, checks whether the previous version of the module is present, and sets the hotswap flag if the previous version exists.
   (b) If the hotswap flag is set, then track the kernel stack of each process in the system. If any function of this module is found in the process stack, retry until a threshold is reached.
   (c) Classify the sections of the module object file as core sections, initial sections, or static address sections, then calculate the total size of each section type. Notably, the module descriptor, contained in the *.gnu.linkonce.this_module* section, is classified as a static address section.
   (d) If the hotswap flag is not set, then allocate three memory regions, one each for the core sections, the initial sections, and the static address sections. Otherwise, only allocate core sections and initial sections because the memory of static address sections is inherited from the previous version.
   (e) Copy each section into the appropriate region of memory according to the previous classification.
   (f) Resolve each undefined module symbol. First, look up the kernel symbol table, then the module symbol tables. If the undefined symbol name begins with *_old_*, then it is attached to the corresponding symbol defined in the previous version.
   (g) Perform relocation after all the symbols are successfully resolved. Relocation calculates the addresses of the code and data.
   (h) Save the information needed by hotswapping into the *replace_handle* structure, and add the structure to a list named *module_handle*. The *replace_handle* structure keeps important information needed by the static address sections, dynamic resolution and relocation, and module descriptor management.
   (i) Free the temporary memory region if the hotswap flag is not set and return the module descriptor to its caller.

3. If the hotswap flag is not set, then *sys_init_module* appends the returned module descriptor to the *modules* list.
4. If the hotswap flag is set, then *sys_init_module* performs dynamic resolution and relocation for each module that depends on this module.
5. If the hotswap flag is not set, then *sys_init_module* invokes the *init_module* function provided by the module. Otherwise, the *replace_module* function is invoked. Then, the memory used by initial sections is freed.
6. If the hotswap flag is set, then the two module descriptors are merged by copying important data from the old descriptor to the new descriptor. Finally, the *sys_init_module* function returns.

The memory usage of the hotswap system is different from that of the original system in two ways. First, the original system does not have static address sections and thus the module descriptor is classified as a core section. Second, the original system frees the temporary copy after the loading procedure while the hotswap system keeps this copy for hotswapping.

3.4. Discussion

3.4.1. Race condition prevention

A hotswap procedure must be free from any race condition, which the current hotswap system can prevent in uniprocessor systems. Two race conditions may occur during hotswapping. The first is that the hotswap system replaces the module while the module is still in execution, and the second is that some kernel threads access the module while this module is under hotswapping. The first one is avoided by stack tracing, and the second one is avoided by disabling kernel preemption and hardware interrupts. Notably, some asynchronous invocation mechanisms, such as software interrupts, bottom halves, tasklets, and software timers, do not lead to race conditions because they are actually implemented by kernel threads, which are scheduled by the kernel. These kernel threads cannot preempt the hotswap procedure if kernel preemption is disabled before hotswapping.

3.4.2. A general procedure to make a module hotswappable

Generally, a module can be made hotswappable by the following steps. First, the module programmer needs to know which variables must be transferred to the new version. Then, the programmer should determine how the variable is used. If used to communicate with the kernel, the variable should be placed into a static address section, since the kernel has already kept its address. Otherwise, the programmer can add code to the replace handler to transfer this variable or put it into the *static_old* section. If a variable is placed into the *static_old* section, then it is not changed during hotswapping, but its definition and size cannot change. By contrast, a normal variable can change its definition and size, but needs the replace handler to transfer its value. Other problems, such as quiescence detection, dynamic resolution and module descriptor management, as described earlier, can be directly handled by the hotswap system and thus need not concern the programmer.

3.4.3. Limitations

This section describes some limitations of the hotswap system. The hotswap system can hotswap most modules except in the following cases. First, the system cannot handle a module that starts a persistent kernel thread. In this case, the module is always in execution, so can never be hotswapped. Second, a variable placed into the static address section cannot grow in size, because each variable address is fixed and no extra space is reserved between any two variables.

Finally, the hotswap system is not suitable for a module in which the functions invoke the scheduler frequently. Since the functions are usually in execution, the system may find the quiescent state very slowly since the quiescence detection aborts repeatedly. However, this situation seldom appears because most module functions return immediately, rather than invoking the scheduler. If module functions invoke the scheduler too often, then the overhead is small because of the busy list mentioned earlier.

If the hotswap system uses reference counting rather than stack tracing, then it is free from repeated aborts. Although reference counting is much more efficient for detecting the quiescent state, it suffers from two drawbacks. First, in order to increment or decrement the reference counter, the module or the module caller must be modified. Second, since hotswapping is not very frequent, the reference counting overhead accumulated during runtime is much greater than the overhead of stack tracing, which only arises during hotswapping. By contrast, the stack tracing technique does not modify the module source or the module callers, and does not introduce extra overheads during normal execution.

4. Evaluation

This section demonstrates the working of the hotswap system and then evaluates its performance using the vfat and fat modules, which are real file system modules taken from the Linux kernel 2.6.11 source tree.

4.1. A vfat file system example

This example involves two modules, vfat and fat. The vfat module supports the FAT32 file system, whereas

the fat module implements basic functions needed by the vfat module. The symbols defined by the fat module are also used by the vfat module. Before hotswapping, a FAT32 partition was created. The workload files of WebBench 5.0 (Lionbridge Technologies, 2004) were then installed onto this partition. Next, we execute the following test sequence was executed:

1. Insert the fat module followed by the vfat module.
2. Mount the FAT32 partition.
3. Start the Apache server (limiting the number of concurrent processes to 200).
4. Start the clients of WebBench on other machines, and then start the benchmark.
5. Hotswap the fat module and the vfat module during the benchmark.

Without either shutting down the Apache server or unmounting the FAT32 partition, the vfat and fat modules were successfully hotswapped during the benchmark. Fig. 5 shows all the modifications to the vfat module, with the modified parts shown in italics. Fig. 6 shows the definition of the *file_system_type* structure and the replace handler of the vfat module. The *vfat_fs_type* variable is a global variable whose address would be passed to the kernel, and which has to be placed into the *static_old* section because the *file_system_type* structure defines both data and function pointer members. The data members need to be transferred from the old version, but the function pointer members must be overwritten by the new version. Since the *static_old* attribute causes the function pointer members to be overwritten by the old version, they must be reassigned in the replace handler. In the example, the function pointer members *vfat_get_sb* and *kill_block_super* were reassigned. The member *name* was also reassigned because the original name string was defined in the old module rather than in the new module. Two other variables, *vfat_dentry_ops* and *vfat_dir_inode_operations*, are placed into the *static_new* section since they define

```
static struct dentry_operations vfat_dentry_ops[4]
__attribute__((section(".static_new"))) = {
  {
    .d_hash   = vfat_hashi,
    .d_compare = vfat_cmpi,
  },
  {
    .d_revalidate = vfat_revalidate,
    .d_hash   = vfat_hashi,
    .d_compare = vfat_cmpi,
  },
  {
    .d_hash   = vfat_hash,
    .d_compare = vfat_cmp,
  },
  {
    .d_revalidate = vfat_revalidate,
    .d_hash   = vfat_hash,
    .d_compare = vfat_cmp,
  }
};

struct inode_operations vfat_dir_inode_operations
__attribute__((section(".static_new"))) = {
  .create   = vfat_create,
  .lookup   = vfat_lookup,
  .unlink   = vfat_unlink,
  .mkdir  = vfat_mkdir,
  .rmdir  = vfat_rmdir,
  .rename   = vfat_rename,
  .setattr  = fat_notify_change,
};

static struct file_system_type vfat_fs_type
__attribute__((section(".static_old"))) = {
  .owner  = THIS_MODULE,
  .name   = "vfat",
  .get_sb   = vfat_get_sb,
  .kill_sb  = kill_block_super,
  .fs_flags= FS_REQUIRES_DEV,
};
```

Fig. 5. Modifications to the vfat module.

```
struct file_system_type {
  const char *name;
  int fs_flags;
  struct super_block *(*get_sb) (struct file_system_type *, int,
            const char *, void *);
  void (*kill_sb) (struct super_block *);
  struct module *owner;
  struct file_system_type * next;
  struct list_head fs_supers;
};
```

a

```
static int replace_module(void)
{
  vfat_fs_type.name="vfat";
  vfat_fs_type.get_sb=vfat_get_sb;
  vfat_fs_type.kill_sb=kill_block_super;
  return 0;
}
```

b

Fig. 6. (a) Definition of the file_system_type struct; (b) replace handler of the vfat module.

only function pointers, allowing all the members to be overwritten by new function addresses. The fat module is similarly modified; its code is not shown here.

4.2. Other hotswapping examples

In addition to the vfat and fat modules, the BFS (UnixWare Boot Filesystem) and e100 modules were also made hotswappable. Unlike the FAT32 file system, which needs two modules, the BFS file system is supported by a single module. The modifications to the BFS module are also similar to those of the vfat and fat modules. The e100 module is the driver for PCI-based Ethernet cards using the Intel 8255 chipset. The e100 module can be hotswapped when the Linux protocol stack has active TCP connections. Critically, the e100 module implements an interrupt handling function. During hotswapping, the replace handler needs to install a new interrupt handling function to prevent the old handling function from being invoked. Although the e100 module demonstrates that the hotswap system can handle device drivers implementing interrupt handling functions, it does not effectively show the benefit of hotswapping. If the e100 module is removed and the new version is then installed, the TCP connections can also be preserved since TCP supports retransmission. Although a temporary loss of the Ethernet state lead to the loss of some TCP segments, these lost segments can be retransmitted after restarting the e100 module.

4.3. Performance

This section shows the performance of loading and hotswapping the vfat module. The experimental platform was a PC equipped with a Celeron 1.13 GHz processor and 384 MB main memory. The Linux kernel version was 2.6.11.

Table 1 shows the loading times of the vfat module. The original system needed 2119 μs, whereas the hotswap system needed 2192 μs. The additional overhead of the hotswap system was only 3.45%.

Table 2 compares the vfat loading and hotswapping times in the hotswap system. Although a hotswap procedure is much longer than the loading procedure, their difference is difficult to notice from the user perspective because they last only a few milliseconds.

Table 3 lists the two most time-consuming steps in hotswapping the vfat module, of which the most time-consuming step is resolution, which resolves the undefined symbols of the vfat module, and which takes

Table 1
Loading times of the vfat module

Original system (μs)	Hotswap system (μs)	Ratio (%)
2119	2192	103.45

Table 2
Loading and hotswapping times of the vfat module

	Hotswap system (μs)
Module loading	2192
Hotswapping	3042
Ratio	138.77%

Table 3
The most time-consuming steps on hotswapping the vfat module

	Hotswap system (μs)	Percentage (%)
Resolution	1476	48.52
Quiescence detection	933	30.67
Others	633	20.81
Total	3042	100

almost half of the hotswap time. The length of this step depends on the system's symbol number and the module's undefined symbol number. Table 4 lists the module and symbol status of the experimental system. The vfat module contains 36 undefined symbols.

The next most time-consuming step is quiescence detection. The length of this step depends on the number of processes in the system, at this point 247. Of these, 200 were created by the Apache web server, as described earlier. The other 47 processes belong to the desktop configuration of the Fedora 3 installation.

Table 5 shows the execution times of the *init_module* function and the *replace_module* functions. The *init_module* function is much longer than the *replace_module* function because *init_module* has to invoke the *register_filesystem* function, whereas *replace_module* only alters some members of the *vfat_fs_type* variable. However, both of the functions have little impact on module loading and hotswapping because they are relatively shorter than the entire module loading or hotswapping procedures.

Table 6 compares the normal resolution time and dynamic resolution time of the vfat module. Since the vfat module depends on the fat module, the dynamic resolution of vfat is triggered by hotswapping the fat module. The length of the dynamic resolution time depends on the symbol number defined by fat and used by vfat. In this work, the symbol number is 12. The analytical result shows that the dynamic resolution takes about 35% of the normal resolution time due to the search space reduction technique described earlier, which also demonstrates that the dynamic resolution is not an expensive step in the hotswapping procedure.

The memory overhead of hotswapping is easy to estimate. The maximum memory overhead is twice the module binary size because a clean module binary is kept inside the module system. For instance, the vfat binary size is 113 KB, and the memory occupied by the hotswappable vfat module is not larger than 226 KB.

Table 4
Module and symbol status of the experimental system

Module number	36
Module symbols	584
Module GPL symbols	1
Kernel symbols	2166
Kernel GPL symbols	240

Table 5
Execution times of the init_module and replace_module functions

	Module loading (μs)	Hotswapping (μs)
init_module or replace_module function	6.637	0.583
Total	2192	3042
Ratio	0.30%	0.02%

Table 6
Normal resolution and dynamic resolution times of the vfat module

	Time (μs)
Dynamic resolution	517
Normal resolution	1476
Ratio	35.03%

5. Related work

5.1. Dynamic reconfiguration, dynamic software update, and hotswapping

A system is dynamically reconfigurable if its configuration can be changed while the system is running. Dynamic reconfiguration was originally implemented in distributed programming languages (Bloom, 1983; Kramer and Magee, 1985; Hofmeister et al., 1992) and has been studied throughout two decades. Dynamic software updating (Hicks et al., 2001) is also a form of dynamic reconfiguration, which focuses on changing the code of the underlying system. Hotswapping is also a form of dynamic reconfiguration, which generally involves a dynamic software update and implies that the component states should be preserved after reconfiguration. The requirements of dynamic reconfiguration have been identified by Moazami-Goudarzi (1999) and Soules et al. (2003). Kramer and Magee (1990) and Moazami-Goudarzi (1999) have thoroughly discussed mutual consistency states.

5.2. Dynamic reconfiguration and operating systems

Extensible kernels are also closely related to dynamic reconfiguration. Kernel extensions can be added to an extensible kernel during runtime, just as components can be dynamically installed into an dynamically reconfigurable system. Some well-known extensible kernels are SPIN (Bershad et al., 1995), Exokernel (Engler et al., 1995) and VINO (Seltzer et al., 1994). The Linux kernel is also extensible because it employs modules, but only for some kernel services.

An extensible kernel is said to be dynamically reconfigurable if a running kernel extension can be replaced by another kernel extension. Senert et al. (2002) utilized the THINK component framework (Fassino et al., 2002) to build a dynamically reconfigurable kernel, although did not provide the detailed implementation. K42 (Soules et al., 2003) is an object-oriented operating system kernel that can be reconfigured dynamically using interposition and indirect swapping.

Transaction processing systems (Gray and Reuter, 1993), which are generally built on top of the operating

systems, can also facilitate hotswapping. Software modules can be hotswapped in transaction processing systems in two ways. In the first method, the system blocks incoming transactions, tracks ongoing transactions, rolls them back, hotswaps the software modules, redoes the transactions and finally processes incoming transactions. In the second method, the system blocks incoming transactions, waits until all the ongoing transactions finish, hotswaps the software modules and finally processes new transactions. However, since most operating system kernels do not support transaction processing, the transaction-based kernel hotswapping is not easy to achieve.

5.3. Other application areas of dynamic reconfiguration

Dynamic reconfiguration is also applicable to object-oriented programming languages. Dynamic reconfiguration enables a new class to replace an instantiated class during runtime. Several dynamically reconfigurable language systems have been developed, including CLOS (Keene, 1989), C++ (Hjalmtysson and Gray, 1998), Java (Malabarba et al., 2000; Dmitriev, 2001), and a Java-based language (Costanza, 2001). Dynamic reconfiguration is also supported in CORBA (Almeida, 2001), which is a distributed object-oriented environment.

Hicks et al. (2001) adopted the Typed Assembly Language (TAL) (Morrisett et al., 1999) and dynamic linking feature of the ELF format (Lu, 1995) to implement general-purpose dynamically reconfigurable applications. However, operating system kernels cannot utilize this approach because most kernels do not have a built-in dynamic linker. Furthermore, Levine (2000) indicated that dynamic linking suffers from a performance penalty.

6. Conclusions

This work proposes a Linux-based hotswap module system which solves several problems involved in hotswapping, relating to mutually consistent states, module states, module symbol exporting, variable address passing and module descriptor management.

Each problem was solved by a design that best fits the system's design goals. For mutually consistent states, the quiescent detection with stack tracing is used. A module can be hotswapped only when it is detected as quiescent, that is, when each process stack contains no functions of the module being hotswapped. The system also implements a busy list to lower the stack tracing penalty. For module states, a programmer can write a replace handler using the variable prefix *_old_* to transfer module variables. Module symbols can be dynamically resolved and relocated. All the modules that depend on the hotswapped module can be efficiently resolved and relocated again. Module variables whose addresses are passed to the kernel can be placed into static address sections. Since variables in these two sections are not moved, the kernel can continue to access them after hotswapping. For module descriptor management, only a single module descriptor is valid, and the old descriptor and the new descriptor are merged at the last stage of the hotswap procedure.

The work most closely related to the proposed system is the K42 operating system (Soules et al., 2003), in which the kernel components are written in C++. This proposed system differs from K42 in several aspects. First, K42 supports hotswapping by using a new kernel design, while the proposed system extends the Linux module system. Since the proposed system is designed to be compatible with existing modules, the fundamental structure of the Linux module system is not changed. Thus, the system has more design constraints than a system designed from scratch. Second, K42 components are written in C++ while Linux modules are written in C. The component states are easy to identify for C++ objects because a C++ object has more runtime information than a C program does. Third, K42 supports indirect swapping while the proposed system supports direct swapping. An indirect swapping system simplifies the design, but a direct swapping system performs better in normal execution. Finally, K42 uses lock-free data structures (McKenney and Slingwine, 1998) combined with thread generation mechanism (Soules et al., 2003) to identify the quiescent state, whereas the proposed system detects the quiescent state through stack tracing.

Although the current hotswap system is implemented only for uniprocessor systems, future work will be to support symmetric multiprocessor (SMP) systems. When extending the hotswap system to SMP systems, preserving the quiescent state is the most difficult task because when one processor is replacing a module, other processors may also access it. Moreover, interrupts are more difficult to manage in SMP systems since the global interrupt disabling function has been removed in kernel version 2.6. To preserve the quiescent state, the hotswap system should utilize the scheduler and interprocessor interrupts. When the hotswap procedure starts on one processor, it sets a hotswap flag and sends interprocessor interrupts to all other processors. The Linux kernel always invokes the scheduler after running the interprocessor interrupt handler. Therefore, when the scheduler on each processor notices the hotswap flag, it executes a piece of code that implements a barrier (Mellor-Crummey and Scott, 1991). When all the processors reach the barrier, each processor disables its interrupts locally, then the hotswapping processor detects quiescence. If the number of processes is large, then the process stacks can be checked more effectively by distributing them to all the processors. If the system is quiescent, then the hotswapping processor replaces the

module and then clears the hotswap flag so that the schedulers on other processors can return to normal execution.

References

Almeida, J.P.A., 2001. Online reconfiguration of object-middleware-based distributed systems. Master's thesis, University of Twente, The Netherlands, June 2001.

Arnold, B. et al., 2000. The Java Programming Language. Addison-Wesley, Boston, MA.

Bershad, B.N. et al., 1995. Extensibility, safety and performance in the SPIN operating system. In: Proceedings of 15th ACM Symposium on Operating System Principles, pp. 267–284..

Bloom, T., 1983. Dynamic module replacement in a distributed system. PhD thesis, MIT Laboratory for Computer Science, March 1983.

Bovet, D.P., Cesati, M., 2002. Understanding the Linux Kernel. O'Reilly & Associates, Cambridge, MA.

Costanza, P., 2001. The programming language Gilgul. In: Proceedings of Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE) at OOPSLA 2001. Available from: <http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml/>.

Dmitriev, M., 2001. Towards flexible and safe technology for runtime evolution of Java language applications. In: Proceedings of Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE) at OOPSLA 2001. Available from: <http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml/>.

Engler, D.R. et al., 1995. Exokernel: an operating system architecture for application-level resource management. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles, pp. 251–266.

Fassino, J.P. et al., 2002. THINK: a software framework for component-based operating system kernels. In: Proceedings of 2002 USENIX Annual Technical Conference, pp. 73–86.

Gray, J., Reuter, A., 1993. Transaction Processing: Concepts and Techniques. Morgan Kaufmann.

Hicks, M. et al., 2001. Dynamic software updating. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 13–23.

Hjalmtysson, G., Gray, R., 1998. Dynamic C++ classes: a lightweight mechanism to update code in a running program. In: Proceedings of 1998 USENIX Annual Technical Conference, pp. 65–76.

Hofmeister, C. et al., 1992. Surgeon: a package for dynamically reconfigurable distributed applications. In: Proceedings of the IEEE International Conference on Configurable Distributed Systems. IEEE Computer Society Press.

Keene, S.E., 1989. Object-oriented Programming in Common LISP: a Programmer's Guide to CLOS. Addition-Wesley, Reading, MA.

Kramer, J., Magee, J., 1985. Dynamic configuration for distributed systems. IEEE Transactions on Software Engineering 11 (4), 424–436.

Kramer, J., Magee, J., 1990. The evolving philosophers problem: dynamic change management. IEEE Transactions on Software Engineering 16 (11), 1293–1306.

Levine, J.R., 2000. Linkers and Loaders. Morgan Kaufmann, San Francisco, CA.

Lionbridge Technologies, 2004. WebBench 5.0 Overview. Available from: <http://www.veritest.com/benchmarks/webbench/home.asp>.

Lu, H., 1995. ELF: from the Programmer's Perspective. NYNEX Science & Technology Inc.

Malabarba, S. et al., 2000. Runtime support for type-safe dynamic Java classes. In: Proceedings of the Fourteenth European Conference on Object-Oriented Programming.

McKenney, P.E., Slingwine, J.D., 1998. Read-copy update: using execution history to solve concurrency problems. In: Proceedings of 10th International Conference on Parallel and Distributed Computing and Systems.

Mellor-Crummey, J.M., Scott, M.L., 1991. Algorithms for scalable syncrhonization on shared-memory multiprocessors. ACM Transactions on Computer Systems 9 (1), 21–65.

Moazami-Goudarzi, K., 1999. Consistency preserving online reconfiguration of distributed systems. PhD thesis, Imperial College, London, March 1999.

Morrisett, G. et al., 1999. From system F to typed assembly language. ACM Transactions on Programming Languages and Systems 21 (3), 527–568.

Rubini, A., Corbet, J., 2001. Linux Device Drivers. O'Reilly & Associates, Cambridge, MA.

Seltzer, M. et al., 1994. An introduction to the architecture of the VINO kernel. Technical Report 34–94. Harvard University Center for Research in Computing Technology.

Senert, A. et al., 2002. Developing dynamically reconfigurable operating system kernels with the think component architecture. In: Proceedings of 2002 OOPSLA Workshop on Engineering Context-Aware Object-Oriented Systems and Environments (ECOOSE 2002).

Soules, C. et al., 2003. System support for online reconfiguration. In: Proceedings of 2003 USENIX Annual Technical Conference, pp. 141–154.

Yueh-Feng Lee received the B.S. and M.S. degrees in computer science from National Tsing Hwa University in 1997 and 1999, respectively. He is now pursuing a Ph.D. degree at the Department of Computer and Information Science, National Chiao Tung University. His research interests include operating systems, mobile communications, and Java.

Ruei-Chuan Chang received the B.S. degree in 1979, the M.S. degree in 1981, and his Ph.D. degree in 1984, all in computer science from National Chiao Tung University. In August 1983, he joined the Department of Computer and Information Science at National Chiao Tung University as a Lecturer. Now he is a Professor of the Department of Computer and Information Science. He is also an Associate Research Fellow at the Institute of Information Science, Academia Sinica, Taipei.