Implementation of Flexible System Call Interface System for Multi-platform Operating Systems

Ian Juha Cho[1]

[1]Hankuk Academy of Foreign Studies

E-mail : ianisnumber2027@gmail.com

ABSTRACT

This study delves into developing a multi-platform operating system that can execute various programs from different operating systems. For an operating system to execute different-platformed programs, it must detect the program's platform and provide the kernel API corresponding with the program. From the myriads of APIs in operating systems, this study specifically focuses on providing a flexible system call interface system that can handle the different platforms' system call requests with the corresponding interface. An implementation of the interface system is presented and implemented in Linux Kernel Version 6.7.4. in C language. A brief test was conducted to determine whether the system correctly detects the difference in the platform and provides the proper dedicated interface. To simulate the difference in the platform of two programs, two ELF programs – a program with a modified file signature and a program without any modification – are tested to observe whether the provided system call interface differs from the file signature. The test result showed that the system can adequately recognize the platforms of different programs and provide the proper corresponding interfaces.

Keywords

Opreating System, System Call, Linux Kernel, Kernel API

I. Introduction

Operating systems provide an API(Application Programming Interface) layer for applications to use kernel's services. Because each operating system has a different kernel API system, different programs from different operating systems can not be executed on other operating systems, even if they share the same ISA(Instruction Set Architecture) and ABI(Application Binary Interface). To enable an operating system to execute programs of multiple programs, the operating system must provide corresponding kernel API to each platform or simulate the environment of the platform.

Several programs provide the flexible API layer, allowing the execution of programs of other platforms. Virtual Machines allow an emulation of a physical computer environment on an operating system such that another operating system can run on the emulated space. The 'guest' machine – the operating systems that run within the virtual environment – are emulated on the 'host' machine, the physical computer running the virtualization software. The problem, however, with virtual machines is that they demand large amounts of host resources. Windows 11, for instance, requires a minimum of 4GBs of RAM and 64GBs of storage for the proper emulation, which is highly demanding for average PCs and would significantly decrease the host system's performance. [1]

High-resource-demanding Virtual Machines can be replaced with Wine, a Linux-based application, which offers a promising solution for executing Windows programs on Linux-based operating systems. Wine interprets the binary codes of EXE files and translates Windows-specific binary codes to Linux-compatible codes. Wine translates Windows' system calls and Windows API run-time to be compatible with Linux operating systems. Unlike virtual machines constructing the environment for an operating system to run on, Wine does not emulate the program; instead, it translates the Windows-dependent codes of the program to Linux-compatible codes. Furthermore, as it does not require any hardware-based virtualization of PC environment, the resource consumption is significantly lower than using virtual machines. [2] The downside of Wine is that it is limited to application-lev-

el services; Wine still is an application which can not use direct OS-level services. For the complete compatibility of the application with the operating system, OS-level services are required to provide services such as a flexible GUI, file system, or usage of device drivers.

This study specifically focuses on the methods for individually providing the System Call Interfaces to programs. System call provides the kernel services to the applications. Applications use the system call by using low-level CPU instructions. When a system call is evoked by the application, the kernel takes the CPU resource and processes the system call request on the kernel-level. Examples of system call requests consist of file I/O operations, task-related operations, or an access to the kernel drivers.

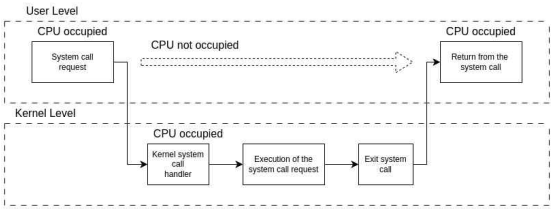


Figure 1. System call routine

Each operating system has its own distinct system call convention that the program must follow to use the interface. Windows, for instance, has different conventions for calling the system call with Linux's; Linux-based programs use the "RAX" registers as the system call number and "syscall" instruction as the request for the system call in Linux (in Intel x86 architecture), whereas Windows-based programs use Windows-provided DLL(Dynamic-Link Library) to request the system call, as the system call convention of Windows is hidden from users(possibly for security reasons.) [3]

Although Windows mostly uses Win32 API to request the kernel services, note that the lowest layer of the kernel-application interface is still based on the system call layer.

This study suggests the design of the system that can integrate the multiple conventions of different operating systems, so that more than one platform of the program can be executed on one operating system. The primary goal of the system is to serve to the program the interface corresponding to the program's platform. Functions will be implemented on the system that can detect the program's platform and correctly link with the matching platform.

(Note that implementing this system only will not result in being able to fluently execute multiple platforms of programs, as there exists other types of interfaces to be considered other than system call interface.)

II. The Overview of the System

The overall components of the interface system comprises the Interface, the Platform Detector, and the Task. Upon loading the program into the task, the platform of the program is detected, and the corresponding interface will be provided to the task. The system call interface works as the layer between the application and the operating system, performing the application's system call requests for the kernel. The platform detector determines what platform the application is on and loads the matching interface to TCB(Task Control Block.) The kernel provides the interface stored on the task, allowing multiple system call interfaces in one operating system.

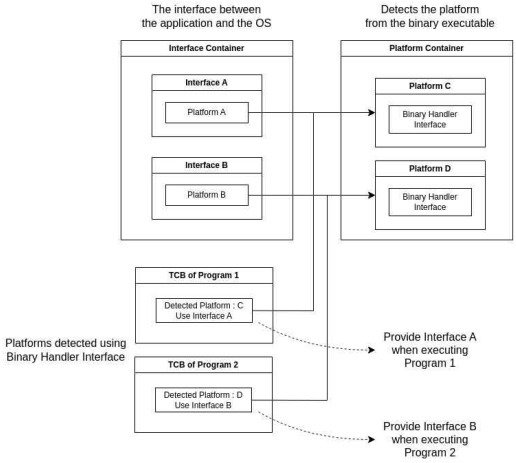


Figure 2. The Overview of the Interface System

The platform detector must be able to correctly detect what the platform matches with the task. From the executable binary file of the task the platform detector can determine the format of the file to match the platform. For providing the corresponding interface, the interface must contain information about the platform, such as a pointer to the binary handler or any identifiable data that can be used to match the platform.

Once the application's platform is detected, the corresponding system call interface is stored in the TCB of the task created from the application. The operating system uses the stored interface when a system call request is made from the program.

1. System Call Interface

The system call interface consists of the System

Call Table, the System Call Method, and the System Call Argument.

1. ***System Call Table*** The system call table is an array of pointers of system call handler functions. The handlers are stored with their corresponding system call number as the array index.
2. ***System Call Method*** Different platforms have different methods of making system calls. The operating system must register each platform's methods. The interface must contain the function that automatically registers each platform's method of calling the system.
3. ***System Call Argument*** Different platforms also have their way of passing the arguments for the system call. (Argument here indicates the arguments passed on by assembly registers.) By differing the entry point of the system call for each program, the system call argument can be passed using a different method.

Later on the implementation section, these three parts of the interface will be stored in an integrated structure for a more cohesive system design.

2. The Platform Detection

Binary handlers detect the application's platform. The operating system circulates the binary interfaces from the container to find the platform that matches the program. The platform is detected whenever an executable file is loaded onto a program. When the program's platform is detected, the system call interface is stored on the program TCB to provide the designated system call to the program later.

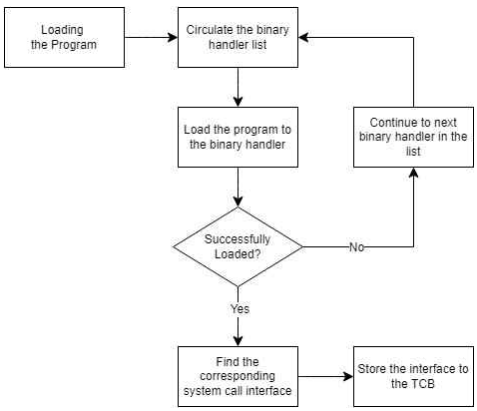


Figure 2. Platform Detection Process

3. Providing the Interface to Applications

The operating system uses the platform information to provide the interface to the application. The currently running processor's system call interface is used whenever the system call service request is made from user mode. Thus, the operating system must switch the interface provided to the applications. If an operating system has a centralized system call interface, the central system call interface is changed to the application's system call interface whenever the system call request is made.

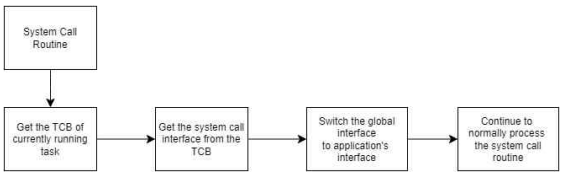


Figure 4. The System Call Interface Switching Process

As previously mentioned, the system call entry point should also be switched into the system call entry point of the platform. Each platform contains the pointer to the system call entry point and the additional pointer to the function that switches the system call entry point to the program's system call entry point in the interface structure. Individually storing the pointer to the entry-point switching function allows more flexibility on the system calling method, as different platforms have different methods of changing the system call entry point because of their difference in system call method.

III. Implementation

Although the implementation is centered on Linux, the system can theoretically be implemented on any operating system that contains a task management system, binary loading mechanism, and the system call routine. Note that the essential features such as task management system that are missing from the operating system must be implemented before the adaptation of this system. Also note that the implementation presented in the study is one example of myriads of possible implementation. The purpose of this implementation is to give the direction of implementing the system by providing one example that successfully resulted in a flexible interface

system.

(All the codes used in the implementation section are from the Linux kernel version 6.7.4.) [4]

The implementation is divided into five sections: system call interface, system call interface manager, system initialization, platform detection, and providing system call to the application.

1. Implementation of System Call Interface

The system call interface structure consists of the interface identifier, system call table array, pointer to the binary handler, and the system call interface functions. An interface identifier helps search the interface from the lists of interfaces or links the interface with binary handlers or system call tables. The binary handler pointer allows the system to identify what interface should be used for what binary handler. System call interface functions process the platform-specific operations such as registering the system call handler, initializing the system call routine, or determining the system call number from registers.

```
struct sys_call_interface {
   char platform[30];
   // binary handler
   struct linux_binfmt *bin_handler;
   // system call table
   sys_call_ptr_t *tbl_ptr;
   // system call interface handlers
   struct sys_call_interface_func_container interface_functions;
};
```

The platform string is the interface's identifier used whenever the interface needs identification. The binary handler is the pointer to the Linux binary handler. The “interface_functions” field contains the interface functions necessary for compensating the discrepancy between platforms and kernels. The primary task of those interface functions is interpreting the platform's interface conventions to the kernel's conventions.

```
struct sys_call_interface_func_container {
   bool (*sys_call_wrapper)(struct pt_regs *regs);
   void (*register_system_call_method)(void);
};
```

(Details of the implementation of “sys_call_interface_func_container” will later be discussed in other sections.)

In the operating system's initialization phase, system call interfaces are registered by an architecture-dependent function. As system call interfaces differ by architecture, each must contain its system call initializer function.

The system call interface structure and related functions are declared in a newly created source file, and the header files are named “dynamic_sys_call_interface.c” and “dynamic_sys_call_interface.h.”

Further modification is required in the Kernel's TCB and binary handler structures to store the interface. A system call interface pointer is added to the TCB structure to store the interface that will be used in context switching. The same as TCB, a system call interface pointer is added to the binary handler structure(linux_binprm) to set the system call interface of the program's TCB to the platform's interface.

Binary handler(linux_binprm) :

```
/* In binfmts.h */

/*
* This structure is used to hold the arguments that are
used when loading binaries.
*/
struct linux_binprm {
   ...

   // system call interface
   uint64_t system_call_interface_ptr;
} __randomize_lay
```

TCB(task_struct) :

```
/* In sched.h */
struct task_struct {
   ...

  // system call interface
   uint64_t system_call_interface_ptr;
   ...
   randomized_struct_fields_end
   /* CPU-specific state of this task: */
   struct thread_struct thread ;
   ...
};
```

2. Implementation of System Call Interface Manager

The system call interface is managed by containing the interfaces on the list. Functions that initialize, register, and discard the interfaces are additionally implemented for list operations.

```
/* In dynamic_sys_call_interface.h */

// Add system call table to list
void syscall_interface_init_list (void);
void syscall_interface_set_default_interface
     (struct sys_call_interface *interface);

struct sys_call_interface
```

```
*syscall_interface_get_default_interface (void);

struct sys_call_interface *syscall_interface_add
      (sys_call_ptr_t *tbl_ptr , const char *platform ,
       struct sys_call_interface_func_container interfaces);

bool syscall_interface_set_binary_handler
     (struct linux_binfmt *bin_handler , const char *platform);

struct sys_call_interface
*syscall_interface_search_by_bin_handler(struct linux_binfmt
*bin_handler);
struct sys_call_interface *syscall_interface_search_by_platform
      (const char *platform);

void syscall_interface_set_global_interface
     (struct sys_call_interface *interface);

// architecture-dependent
void syscall_interface_register_arch_dependent(void);
```

The interface list is declared in the file "dynamic_sys_call_interface.c". The list is implemented using the default Linux linked list :

```
/* In dynamic_sys_call_interface.c */
...
// declare the list
static LIST_HEAD (sys_call_list);
...
```

syscall_interface_init_list() Initializes the system call list. Executed during the kernel initializing process.

```
/* In dynamic_sys_call_interface.c */

void __init sys_tbl_init_list (void ) {
   INIT_LIST_HEAD(&sys_call_list);

   // register system call tables (architecture-specific
function)
   sys_tbl_register_systbls();
}
```

The function is executed during the kernel's initialization process. In Linux, this is done in the "start_kernel()" function in the "main.c" file.

syscall_interface_set_default_interface(), syscall_interface_get_default_interface() The mutator and accessor of the default system call interface, respectively.

```
/* In dynamic_sys_call_interface.h */
...
static struct sys_call_interface *default_interface;

void syscall_interface_set_default_interface
   (struct sys_call_interface *interface)
{ default_interface = interface; }

struct sys_call_interface
*syscall_interface_get_default_interface(void)
{ return default_interface; }
...
```

When a task is created without a designated platform, its system call interface is set to default. The static variable that stores the default system call table is located globally in the file "dynamic_sys_call_interface.c."

syscall_interface_add() Adds the new interface to the list. Create a new "sys_call_interface" object and return it for later usage. The system call method is registered by calling the interface function.

```
struct sys_call_interface *syscall_interface_add
      (sys_call_ptr_t *tbl_ptr , const char *platform ,
       const struct sys_call_interface_func_container interfaces)
{
   struct sys_call_interface *new_interface
       = kmalloc (sizeof(struct sys_call_interface)
         , GFP_KERNEL);
   new_interface->tbl_ptr = tbl_ptr;
   new_interface->bin_handler = 0x00;
   memcpy(&new_interface->interface_functions , &interfaces
         , sizeof (struct sys_call_interface));
   strcpy(new_interface->platform , platform);
   // add to the list
   list_add(&new_interface->lst , &sys_call_list);
   // register the system call method to system
   interfaces.register_sys_call_method();
   return new_interface;
}
```

syscall_interface_set_binary_handler() Searches the interface by the platform name and set the binary handler of the interface to the binary handler in the argument. Later, in the initialization of the binary handler, the initializer function sets the binary handler by the designated platform name.

```
bool syscall_interface_set_binary_handler
    (struct linux_binfmt *bin_handler , const char *platform) {
   struct sys_call_interface *cur;
   list_for_each_entry(cur , &sys_call_list , lst) {
      if(strcmp(cur->platform , platform) == 0) {
      if(cur->bin_handler != 0x00 ) return false;
         cur->bin_handler = bin_handler;
         return true;
      }
   }
   return false;
}
```

syscall_interface_search_by_bin_handler() Get the interface pointer by the binary handler.

```
struct sys_call_interface
*syscall_interface_search_by_bin_handler(struct linux_binfmt
*bin_handler) {
   struct sys_call_interface *cur;
   list_for_each_entry(cur , &sys_call_list , lst) {
      if(cur->bin_handler == bin_handler) {
         return cur;
      }
   }
   return 0x00 ;
}
```

The function circulates the list and returns the interface that matches the binary handler.

syscall_interface_search_by_platform() Return the interface pointer using the platform name.

```
struct sys_call_interface *syscall_interface_search_by_platform
(const char *platform) {
   struct sys_call_interface *cur;
   list_for_each_entry(cur , &sys_call_list , lst) {
      if(strcmp(cur->platform , platform) == 0) {
         return cur;
```

```
    }
  }
  return 0x00;
}
```

syscall_interface_set_global_interface() Set the global system call interface to the system call interface from the argument. The global system call table is declared in the architecture-dependent source files in Linux. Initially, the table is declared in an array format; however, since the table should be able to be modified, it is changed to the pointer format. In Intel x86 architecture, the system call table is declared in the "arch/x86/entry/syscall_64.c" file :

```
/* In syscall_64.c */
...
sys_call_ptr_t sys_call_table_linux[] = {
#include <asm /syscalls_64.h>
// ../include/generated/asm/syscalls_64.h
};

asmlinkage sys_call_ptr_t *sys_call_table = sys_call_table_linux;
...
```

"syscall_interface_set_global_interface()" imports the global table and sets it to the table of the new interface. The global table can be imported using the "extern" keyword :

```
/* In dynamic_sys_call_interface.c */
...
extern asmlinkage sys_call_ptr_t *sys_call_table;

void syscall_interface_set_global_interface
    (struct sys_call_interface *interface) {
  if(interface == 0x00 ) { return; }
  sys_call_table = interface->tbl_ptr;
}
...
```

3. Implementation of System Initialization

The system initialization is done in the following order :

1. Initialization of System Interface List
2. Registration of system call interfaces
3. Registration of binary handlers
4. Linking the binary handlers with system call interfaces

The system call interface's initialization must transpire before the binary handler's initialization, as the binary handler uses the interface in the initialization.

3.1. Initialization and Registration of Interface

The system call list is initialized by calling the "syscall_interface_init_list()" function. The function executes the "syscall_interface_register_arch_ dependent()" function that registers all the necessary system call interfaces to the system. The "syscall_interface_register_arch_dependent()" function is architecture-dependent – the implementation differs by architecture. The function creates the interfaces for each platform and registers them to the system. ("syscall_interface_register_ arch_dependent()" function is declared in the "syscall_64.c" file, in directory "arch/x86/entry".)

```
/* In syscall_64.c */

sys_call_ptr_t sys_call_table_linux[] = {
  /* linux system call table */
};
sys_call_ptr_t sys_call_table_custom[] = {
  /* custom system call table */
};
// global system call table pointer
sys_call_ptr_t *sys_call_table = sys_call_table_linux;
bool linux_sys_call_wrapper(struct pt_regs *regs) {
  ...
}
bool custom_sys_call_wrapper(struct pt_regs *regs) {
  ...
}
void linux_register_system_call_metho (void) {
  ...
}
void custom_register_system_call_method(void) {
  ...
}
void syscall_interface_register_arch_dependent (void) {
  struct sys_call_interface_func_container linux_functions =
{linux_sys_call_wrapper , linux_register_system_call_method};
  struct sys_call_interface_func_container custom_functions =
{custom_sys_call_wrapper , custom_register_system_call_method};

  struct sys_call_interface *default_interface =
  syscall_interface_add (sys_call_table_linux , "linux " ,
linux_functions);
  syscall_interface_add(sys_call_table_custom , "custom" ,
custom_functions);

  // set linux's system call table as a default table
  syscall_interface_set_default_interface(default_interface);
}
```

3.2. Initialization and Registration of Binary Handler

The initializer function registers the binary handler structure named "linux_binfmt" to the system. The "linux_binfmt" structure consists of function pointers to the binary handler functions. These functions load the binary into the process.
Following is the example of the "linux_binfmt" structure in the elf binary handler :

```
static struct linux_binfmt elf_format = {
  .module = THIS_MODULE,
  .load_binary = load_elf_binary,
  .load_shlib = load_elf_library,
#ifdef CONFIG_COREDUMP
  .core_dump = elf_core_dump,
  .min_coredump = ELF_EXEC_PAGESIZE,
#endif
};
```

After registering the binary handler, the initializer function links the handler with the system call interface using the "syscall_interface_set_binary_handler()" function. The function will identify what system call interface should be linked using the platform string :

```
/* In binfmt_elf.c */
...
```

```
// module init
static int __init init_elf_binfmt (void)
{
   // register the binary handler to system
   register_binfmt(&elf_format);
   // register the binary handler to the interface
   syscall_interface_set_binary_handler(&elf_format , "linux");
   return 0;
}
// module exit
static void __exit exit_elf_binfmt (void)
{
   /* Remove the COFF and ELF loaders. */
   unregister_binfmt(&elf_format);
}
core_initcall(init_elf_binfmt);
module_exit(exit_elf_binfmt);

...
```

4. Platform Detection

In Linux kernel 6.7.4., the detection of the platform occurs when the binary handler system searches for a suitable handler for the program. The operating system goes around all the elements of the binary handler list and attempts each binary handler until the suitable handler is found. Detailed process of the execution of the program on Linux takes the following steps:

1. **“execve()”** function is called in a process for the program execution.
2. **“execve()”** calls **“sys_execve()”**, which calls **“do_execve().” “do_execve()”** calls **“do_execveat_common()”** function.
3. **“do_execveat_common()”** function allocates a new **“linux_binprm”** structure, which holds the arguments for loading a binary.
4. **“do_execveat_common()”** function fills out the necessary information in the newly created “binprm” structure.
5. **“do_execveat_common()”** function calls **“exec_binprm()”** function.
6. **“exec_binprm()”** calls the **“search_binary_handler()”** function, which finally searches for a suitable binary handler from the lists of handlers.
7. **“search_binary_handler()”** function calls the **“load_binary()”** function, finally creating the thread and executing the program.

The “search_binary_handler()” function cycles the binary handler list. It calls each handler's “load_binary()” function and checks whether it has successfully loaded the program.

```
/* In fs/exec.c */
...
static LIST_HEAD (formats);
...
/*
* cycle the list of binary formats handler, until one recognizes
the image
*/
static int search_binary_handler (struct linux_binprm *bprm) {
   ...
   list_for_each_entry(fmt, &formats, lh) {
      if(!try_module_get(fmt->module)) continue;

      read_unlock(&binfmt_lock);
      retval = fmt->load_binary(bprm);
      read_lock(&binfmt_lock);
      put_binfmt(fmt);
      if(bprm->point_of_no_return || (retval != -ENOEXEC)) {
         read_unlock(&binfmt_lock);
         // success
         return retval;
      }
   }
   ...
   return retval;
}
```

When the binary is successfully loaded, the newly created processor must contain the corresponding system call interface in the TCB. Following is the modification that loads the interface to the TCB of the current task
(task on which the program is loaded) :

```
static int search_binary_handler (struct linux_binprm *bprm)
{
   ...
   list_for_each_entry(fmt, &formats, lh) {
      ...
      if (bprm->point_of_no_return || (retval != -ENOEXEC)) {
      // success

      /** modified parts of code **/
      // set system call handler
      bprm->system_call_interface_ptr =
         (uint64_t )syscall_interface_search_by_bin_handler(fmt);

         /* if system call is not registered to binary handler,
            just use default interface */
         if (bprm->system_call_interface_ptr == 0x00 ) {
            bprm->system_call_interface_ptr =
             (uint64_t)syscall_interface_get_default_interface();
         }
         /* set the loaded program's system call interface to the
            corresponding one */
         current->system_call_interface_ptr =
               bprm->system_call_interface_ptr;
         read_unlock(&binfmt_lock);
         return retval;
      }
   }
   ...
   return retval;
}
```

The function executes the “syscall_interface_search_by_bin_handler()” function and receives the target interface from the binary handler. The default system call interface is used if no interface corresponds to the binary format(if the function returns a null pointer.) Finally, the TCB of the current task stores the target interface.

5. Providing the System Call Interface

As previously stated, the system call interface consists of three parts: System Call Table, System Call Method, and System Call Argument.

5-1. Providing the System Call Table

Individual system call tables are provided to programs by switching the global system call table into the program's system call table. The table switching should be done at the beginning of the system call routine. In the Linux Kernel, the "syscall_enter_from_user_mode()" function initializes the entry of the system call routine prior to calling the system call handler. The function is modified to set the global system call table to the current task's system call table :

```
/* In kernel/entry/common.c */

noinstr long syscall_enter_from_user_mode (struct pt_regs *regs,
long syscall)
{
   long ret;
   __enter_from_user_mode(regs);
   instrumentation_begin();
   local_irq_enable();
   ret = __syscall_enter_from_user_work(regs, syscall);
   instrumentation_end();

   /** modification **/
   // set the global interface to task's interface
   syscall_interface_set_global_interface
    ((struct sys_call_interface *)
    current->system_call_interface_ptr);

   return ret;
}
```

An alternative method of exchanging the table is switching the global table in context switching. During context switching, the global system call interface is switched to the interface of the next task. After the context switching, the next task is executed with the changed global interface. However, the method has failed to work as intended, as context switching could not provide a stable exchange of the global system call table.

5-2. Providing the System Call Argument

Individual system call interfaces require different entry points of system call routine, as each platform has a different convention of passing the argument to the kernel. These conventions can be integrated by placing an interface function that interprets the system call argument in the front of the system call routine. For instance, an interface function that passes the system call number by the RBX register is required for a platform that passes the system call number as a value of the RBX register.
The interface function is defined on the "sys_call_interface_func_container":

```
struct sys_call_interface_func_container {
   bool (*sys_call_wrapper)(struct pt_regs *regs);
   void (*register_system_call_method)(void);
};
```

The "sys_call_wrapper" function interprets the system call argument for the kernel and is implemented differently by platforms. Conventionally, in Linux Kernel, the "entry_ SYSCALL_64()" function is executed when the system call is invoked. "entry_SYSCALL_64()" prepares the registers for the system call argument and calls the "do_syscall_64()" function that is responsible for performing the system call request.
In the modified system, the "entry_SYSCALL_64()" function calls the wrapper caller function instead of the direct system call processor. The wrapper caller function calls the interface function(the wrapper function) from the current task's system call interface, which modifies the register according to the platform's convention. The wrapper function finally calls the "do_syscall_ 64()" function, and the process continues.

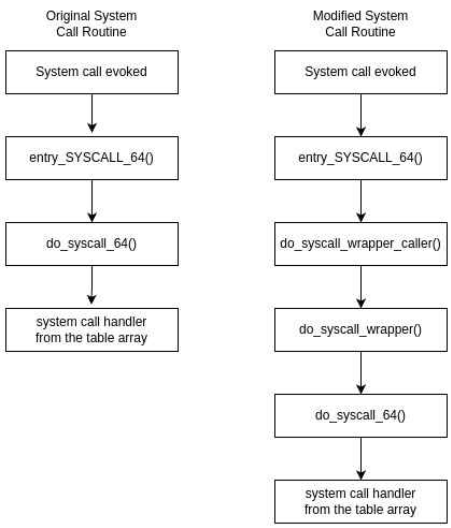


Figure 5. Diagram of overall system call process, left original and right modified.

In the original version of the kernel, the "entry_SYSCALL_64()" function passed the registers from the application and the target system call number (specifically in Intel x86 architecture, "entry_SYSCALL_64()" passed the value of RAX register as the system call number.) The modified version now only passes the registers from the application, leaving the "do_syscall_wrapper()" function to determine the system call number from the

registers :

```
/* In arch/x86/entry/entry_64.S */

SYM_CODE_START (entry_SYSCALL_64)
...
/* IRQs are off. */
movq %rsp, %rdi

/* Sign extend the lower 32bit as syscall numbers are treated as
int */

/* Modification : remove the system call number argument */
/* movslq %eax, %rsi */
/* clobbers %rax, make sure it is after saving the syscall nr */
IBRS_ENTER
UNTRAIN_RET
CLEAR_BRANCH_HISTORY
/* call do_syscall_64 */
/* Modification : call the wrapper caller function instead of
do_syscall_64() */
call do_syscall_64_wrapper_caller
...
SYM_CODE_END (entry_SYSCALL_64)
```

The "do_syscall_64_wrapper_caller()" function simply reads the current task's interface and calls the wrapper function :

```
/* In arch/x86/entry/common.c */
...
#include <linux/dynamic_sys_call_interface.h>

bool do_syscall_64_wrapper_caller (struct pt_regs *regs) {
   if(current->system_call_interface_ptr == 0x00 ) {
       // call default system call wrapper
     return
        syscall_interface_get_default_interface()
        ->interface_functions.sys_call_wrapper(regs);
   }
   return ((struct sys_call_interface *)
          current->system_call_interface_ptr)
          ->interface_functions.sys_call_wrapper(regs);
}
...
```

The wrapper function then determines what register should indicate the system call number. The implementation of the wrapper function exists on the same source file with the "syscall_interface_register_arch_dependent()" function :

```
/* In arch/x86/entry/syscall_64.c */
...
bool linux_sys_call_wrapper (struct pt_regs *regs) {
   /* linux default system call handler */
   /* For linux, RAX holds the system call number */
   return do_syscall_64(regs , regs->orig_ax);
}
bool custom_sys_call_wrapper (struct pt_regs *regs) {
   /* Implementation for the custom platform */
}
...
/* syscall_interface_register_arch_dependent() ... */
```

"do_syscall_64" function then continues the system call routine as usual.

The advantage of having a wrapper function is that the system call routine can be completely individual for each platform. The wrapper function, for example, may call its system call handler, which has completely different conventions from the original kernel's system. The disadvantage of having a wrapper is that the implementation of the wrapper function is architecture-dependent. The implementation suggested here does not apply globally to all the architectures; The implementation shown here is only possible for Intel x86 architecture. Although the overall structure of the system does not significantly change by architecture, the suggested implementation has to be implemented for all the architectures, degrading the architecture flexibility of the system.

An alternative method of exchanging the system call entry in context switching was tested; however, switching the system call table in context switching resulted in an unstable interface exchange.

System Call Method

For each platform, the corresponding system call methods are registered in the registration process of system call interfaces. In "syscall_interface_register_arch_dependent()", the functions tasked with registering the system call methods are added to the "sys_call_interface" structure. Upon adding the interface structure, the registered function is executed, resulting in the registration of system call methods.

```
/* In arch/x86/entry/syscall_64.c */

void linux_register_system_call_method (void ) {
   wrmsrl(MSR_LSTAR , (unsigned long )entry_SYSCALL_64);
}
void custom_register_system_call_method (void ) {
   ...
}

...

/* In dynamic_sys_call_interface.c */
// add the system call table to the list
struct sys_call_interface *syscall_interface_add(sys_call_ptr_t
*tbl_ptr , const char *platform , const struct
sys_call_interface_func_container interfaces) {
   ...
   // register the system call method to system
   interfaces.register_system_call_method();
   return new_interface;
}
```

(In Intel x86 architecture, the MSR_LSTAR register of the MSR(Model-Specific Registers) registers holds the entry point of the system call.)

IV. Testing the System

An ELF executable with the header's changed signature will be used to test the system. The replica of the ELF binary handler that only recognizes

the modified header signature of ELF will be registered to the system. The system call interface corresponding to the replica binary handler will also be registered.
(Processes of compiling or setting the environment for compiling are all omitted.)

The system will be tested in the following:
1. The binary handler should be able to load the modified version of the program.
2. The system should detect the platform correctly and insert the interface to the TCB using the binary handler.
3. The system should be able to provide the correct system call table to the application.
4. The platform of the system call should correspondingly change the system call routine.
5. The execution of two program versions (unmodified and modified) exhibits the difference in the system call interface.

A new custom platform will be created for execution of the modified ELF program(the newly made platform will now be called “custom.”) The “custom” platform will have a custom system call table and system call argument. The program will test whether the system call interface exchange transpires correctly. The Linux kernel will also be modified to print the debug information(such as pointers to system call tables) to observe whether the system works as intended.

1. Preparation

1.1. Registering the System Call Interface

The implementation of the system call table of the "custom" platform is following :

```
/* In arch/x86/entry/syscall_64.c */

sys_call_ptr_t sys_call_table_custom[] = {
__SYSCALL(0 , sys_ni_syscall)
__SYSCALL(1 , sys_ni_syscall)
__SYSCALL(2 , sys_ni_syscall)
... /* repeat */
__SYSCALL(70 , sys_exit)
__SYSCALL(71 , sys_ni_syscall)
__SYSCALL(72 , sys_ni_syscall)
... /* repeat */
__SYSCALL(314 , sys_write)
__SYSCALL(315 , sys_ni_syscall)
__SYSCALL(316 , sys_ni_syscall)
... /* repeat */
__SYSCALL(500 , sys_get_current_syscall_tbl)
__SYSCALL(501 , sys_ni_syscall)
__SYSCALL(502 , sys_ni_syscall)
... /* repeat */
__SYSCALL(543 , sys_ni_syscall)
__SYSCALL(544 , sys_ni_syscall)
__SYSCALL(545 , sys_ni_syscall)
__SYSCALL(546 , sys_ni_syscall)
__SYSCALL(547 , sys_ni_syscall)
};
```

(Following chapters will include further elaboration on “sys_get_current_syscall_tbl.”)

All handlers except the three system call handlers(sys_exit, sys_write, and sys_get_current_syscall_tbl) are ignored for debugging purposes. The “sys_exit” service is now system call number 70, the “sys_write” service is number 314, and the “sys_get_current_syscall_tbl” service is number 500.

The implementation of the system call argument is the following :

```
/* In arch/x86/entry/syscall_64.c */

bool custom_sys_call_wrapper (struct pt_regs *regs) {
   printk("custom_sys_call_wrapper() called\n");
   printk("system call number(rbx) = %ld\n" , regs->bx);
   return do_syscall_64(regs , regs->bx);
}
```

The value of the RBX register is used for the system call number and to test whether the argument of the system call is changeable by platform. The value of the RBX register is dumped to debug the system call number.

The implementation of the system call method of the platform is left identical to the architecture’s default of invoking by “syscall” instruction due to the architecture constraints in the instruction.

1.2. Adding the Semi-ELF Binary Handler

The original ELF binary handler is replicated and modified to recognize the modified ELF signature and load the binary. The modified ELF file has “0x4A 0x55 0x48 0x41” (“JUHA”) as the identifier(the first four bytes of the ELF.) From the original ELF handler, only the part that recognizes the ELF first identifier has been modified :

```
/* In fs/binfmt_custom_elf.c, the modified replica of
binfmt_elf.c */

static int load_elf_binary (struct linux_binprm *bprm)
{
   ...
   /* First of all, some simple consistency checks */
   if (memcmp (elf_ex->e_ident, "JUHA", SELFMAG) != 0 )
      goto out;
   ...
}
```

The handler is registered to the "custom" system call interface :

```
/* In fs/binfmt_custom_elf.c */
static int __init init_elf_binfmt(void)
{
   register_binfmt(&elf_format);
   printk("custom : elf_format : 0x%lx\n" ,
      (unsigned long)&elf_format);
   syscall_interface_set_binary_handler(&elf_format , "custom");
   return 0 ;
}
```

The address to the "elf_format" structure is printed to identify and debug the elf binary handler.

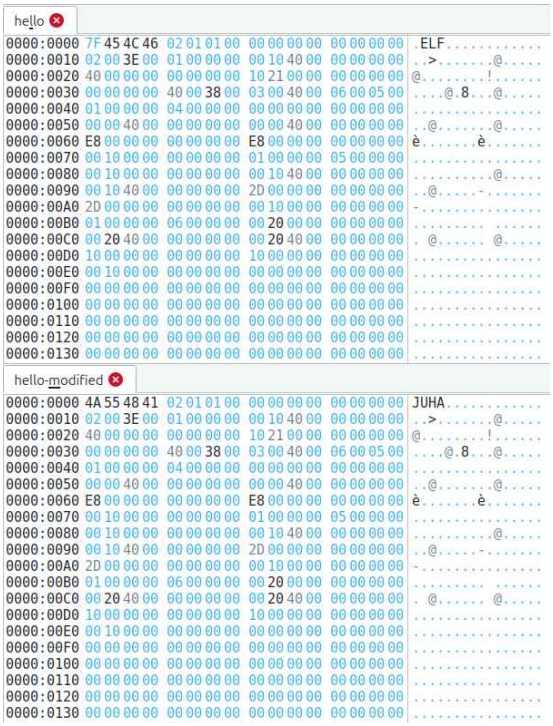
1.3. The ELF Program

The ELF program is compiled from assembly language to evoke the system call directly with customized register arguments. Following is the assembly source of the ELF program :

```
[BITS 64]

SECTION .text

global _start

_start:
   push rbp
   mov rbp , rsp
   ; For "custom" platform : The system call number is the value of
the RBX register.
   ; - System call #500 : sys_get_current_syscall_tbl
   ; - System call #314 : sys_write
   ; - System call #70  : sys_exit
   ; For original linux   : The system call number is the value of
the RAX register.
   ; - System call #548 : sys_get_current_syscall_tbl
   ; - System call #1  : sys_write
   ; - System call #60  : sys_exit
   mov rax , 548 ; sys_get_current_syscall_tbl for linux
   mov rbx , 500 ; sys_get_current_syscall_tbl for "custom"
   syscall

   mov rax , 547 ; sys_ni_syscall(do nothing) for linux
   mov rbx , 314 ; sys_write for "custom"

   mov rdi , 1  ; stdout
   mov rsi , string_hello
   mov rdx , 15
   syscall

   mov rax , 60 ; sys_exit for linux
   mov rbx , 70 ; sys_exit for "custom"
   syscall

SECTION .data
string_hello: db "Hello, world!" , 0x0D , 0x0A , 0x00
```

Because the interface of Linux and the "custom" platform have different table and register usage, the result of the program will differ depending on the platform on which it is executed. On the original Linux interface, the program's system call requests will use the RAX value as the call number and Linux's original call table as the target system call table. On the "custom" interface, the requests will use the RBX value as the number and the "custom" table as the target table.

The file signature has also been modified to fit the binary handler of the "custom" platform. The first four bytes of the header(The ELF Identifier) are modified to "0x4A 0x55 0x48 0x41" ("JUHA") from the original signature of "0x7F 0x45 0x4C 0x46." ("ELF".)



Picture 1. Unmodified ELF File(up), Modified ELF File(down)

The unmodified version will exhibit no resulting outputs(except for kernel debug messages) and exit normally, as the system call number 547(sys_ni_syscall) has been requested instead of the "sys_write" service. In contrast, the modified version will print "Hello, world!" and exit without any error, as the system call number 314(sys_write) has been requested on the exchanged system call table. "sys_get_current_syscall_tbl" service, a service that prints out the pointer of the current global call table, will also show the difference in the table pointers between the two versions.

2. Modifications for Debugging Purpose

A system call service called "sys_call_current_syscall_tbl" has been added to debug the current global system call table pointer. When the service is requested, the service prints out the pointer of the global table using the "printk()" function :

```
asmlinkage long __x64_sys_get_current_syscall_tbl(void ) {
   printk("(syscall) current system call table : 0x%lx\n" ,
      (unsigned long)sys_call_table);
   return 0x00 ;
}
```

In original Linux, the system call is registered by number 548 in the system call table file(that will be auto-generated into the C header file) :

```
/* arch/x86/entry/syscall/syscall_64.tbl */
...
544 x32 io_submit compat_sys_io_submit
545 x32 execveat compat_sys_execveat
546 x32 preadv2 compat_sys_preadv64v2
547 x32 pwritev2 compat_sys_pwritev64v2
548  common sys_get_current_syscall_tbl  sys_get_current_syscall_tbl
```

(As previously mentioned, the service is registered by number 500 in the "custom" platform.)

In order to identify which table has what pointer address, the pointers of the system call tables are printed on the registration of the interface(on sys-call_interface_register_arch_dependent() function) :

```
/* In arch/x86/entry/syscall_64.c */

void syscall_interface_register_arch_dependent (void) {
   printk("sys_call_table     : 0x%lx\n" ,
      (unsigned long )sys_call_table);
   printk("sys_call_table_linux  : 0x%lx\n" ,
      (unsigned long )sys_call_table_linux);
   printk("sys_call_table_custom : 0x%lx\n" ,
      (unsigned long )sys_call_table_custom);
   ...
}
```

To also identify which system call interface has what pointer address, the pointers of the system call interfaces are printed :

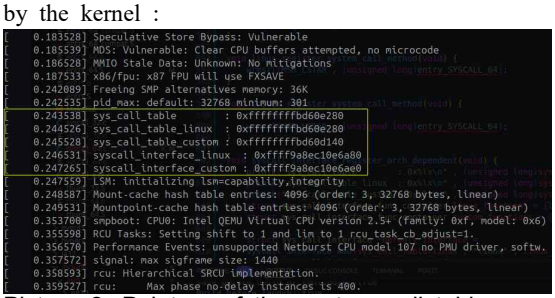
```
void syscall_interface_register_arch_dependent (void ) {
...
   struct sys_call_interface *default_interface =
   syscall_interface_add (sys_call_table_linux , "linux " ,
      linux_functions);
   struct sys_call_interface *custom_interface =
   syscall_interface_add (sys_call_table_custom , "custom " ,
      custom_functions);
   printk("syscall_interface_linux  : 0x%lx\n" ,
      (unsigned long)default_interface);
   printk("syscall_interface_custom : 0x%lx\n" ,
       (unsigned long)custom_interface);

   // set Linux's system call table as a default table
   syscall_interface_set_default_interface(default_interface);
}
```

To observe whether the system call interface is correctly stored in the TCB of the program, the pointer of the current task's interface and the pointer of the interface on the "bprm" structure are both printed :

```
/* In fs/exec.c */
static int search_binary_handler (struct linux_binprm *bprm)
{
   ...
   list_for_each_entry(fmt, &formats, lh) {
      ...
      if (bprm->point_of_no_return || (retval != -ENOEXEC)) {
         // success

         // set system call handler
         bprm->system_call_interface_ptr
        = (uint64_t)syscall_interface_search_by_bin_handler(fmt);

       // if binary format is not registered, just use default
table
         if (bprm->system_call_interface_ptr == 0x00 ) {
            printk("binary handler not registered to dynamic \
syscall list! using default table\n");
            bprm->system_call_interface_ptr
           = (uint64_t)syscall_interface_get_default_interface();
         }
         current->system_call_interface_ptr
          = bprm->system_call_interface_ptr;
         printk("bprm->system_call_interface_ptr   : 0x%llx\n" ,
            bprm->system_call_interface_ptr);
         printk("current->system_call_interface_ptr : 0x%llx\n" ,
            current->system_call_interface_ptr);
         read_unlock(&binfmt_lock);
         return retval;
      }
   }
   ...
}
```

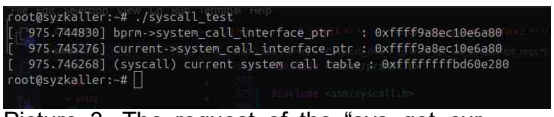
3. Execution and Analysis of the Result

Upon the booting of the modified kernel on the virtual machine(QEMU), the pointers of tables and interfaces can be identified by the messages dumped by the kernel :



Picture 2. Pointers of the system call tables and interfaces

To test whether the "sys_get_current_syscall_tbl()" service accurately dumps the program's system call table pointer, a simple program that calls system call number 548 is created :



Picture 3. The request of the "sys_get_current_syscall_tbl()" service

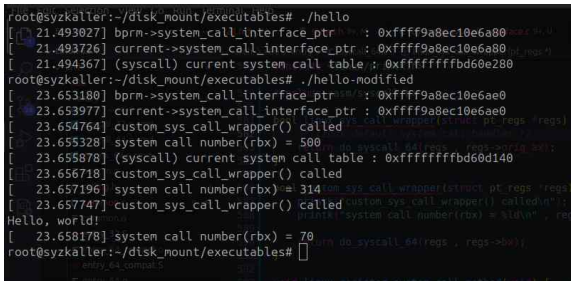
The source code of the test program :

```
#include <unistd.h>
#include <sys/syscall.h>
int main (void ) {
    syscall(548);
    return 0 ;
}
```

As shown in Figure 9, the interface loaded to the program is 0xfff9954010e6a80, which corresponds to Linux's default system call interface, as the binary is loaded with the default "binfmt_elf" handler. We can see that the system call service is successfully called, as the pointer to the system call table is printed, and the interface is accurately linked with the program. (The address 0xffffffffa440e280 corresponds to "sys_call_table_linux," as shown in Figure

8.)

The original version of the test program of the "custom" platform and the modified version of the program are both executed :



Picture 4. The execution of the test program. "hello"(upper) : the unmodified version and "hello-modified"(lower) : the modified version.

We can observe that the unmodified version exhibited no output other than direct kernel messages, which is the expected result since the requested system call was served with the Linux interface. Also, we can observe that the modified version printed "Hello, world!" and exited the program normally, suggesting that the program was served with the interface of the "custom" platform. The printed kernel messages show that different system call wrappers have been accurately executed for each platform, and thus, the system call number has been correctly altered. According to the kernel messages, the system call table of the unmodified program is located on the address 0xffffffffbd60e280, which is the pointer of "sys_call_tbl_linux," and the table of the modified version is located on the address 0xffffffffbd60d140, which is the pointer of "sys_call_tbl_custom." Furthermore, the interface of the unmodified is located at the address 0xffff9a8ec10e6a80, which is the pointer of Linux's default interface, and the interface of the modified is located at the address 0xffff9a8ec10e6ae0, which is the pointer of the "custom" platform's interface. (See Figure 8.) These results suggest that the global system call interface has been accurately switched according to the program's platform.

V. Conclusion

The system's implementation on Linux kernel 6.7.4 has successfully enabled program execution on more than one platform. Although the test result succeeded, the system requires heavy revisions and improvements.

The system lacks the complete architecture-dependent implementation; the architecture-dependent sections of the system call routine needs to be modified to allow the flexible implementation of the system call wrapper function. Switching the system call interface in the context switching would allow the system to be architecture-dependent, but attempts to implement the system failed due to the imprecise nature of the context switching. Further research is required to investigate the method of switching the interface on the context switching to provide the interface individually to different platforms.

The system needs more flexibility in the system call routine. The system can not fully provide the different system call routines for each platform, as the entry point of the system call remains the same throughout the different platforms. An exchange of the system call entry point in the context switching is required to provide the different routines for each platform. Attempts to implement it have also failed; thus, further research on the system is required.

In conclusion, the system presented in this study can be used to provide the system call layer of the kernel API in the operating system level, establishing the foundation for building the complete interface system that can adapt any operating system platform.

References

[1] Microsoft. Windows 11 Specs and System Requirements [Internet]. Available : https://www.microsoft.com/en-us/windows/windows-11-specifications

[2] WineHQ. WineHQ - About Wine [Internet]. Available : https://www.winehq.org/about

[3] Dollard, Kathleen. Walkthrough: Calling Windows APIs - Visual Basic [Internet]. Available : https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/com-interop/walkthrough-calling-windows-apis

[4] Torvalds, Linus. Release v6.7 · Torvalds/Linux · GitHub [Internet]. Available : https://github.com/torvalds/linux/releases/tag/v6.7. (User for core kernel sources)