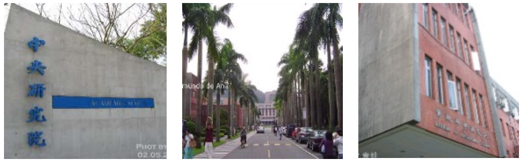
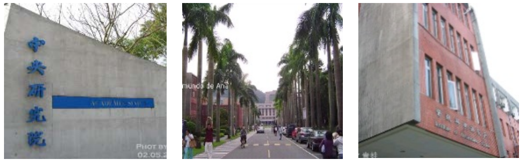


10 System Calls

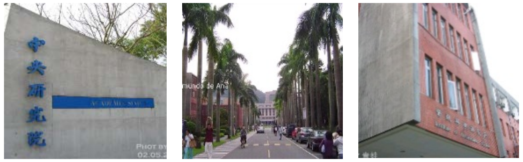
Outline

- **POSIX APIs and System Calls**
- System Call Handler and Service Routines
- Entering and Exiting a System Call
- Parameter Passing
- Kernel Wrapper Routines



System Call’s Advantages

- 1. It makes programming easier by freeing users from studying low-level programming characteristics of hardware devices.
- 2. It greatly increases system security, because the kernel can check the accuracy of the request at the interface level before attempting to satisfy it.
- 3. These interfaces make programs more portable, because they can be compiled and executed correctly on every kernel that offers the same set of interfaces.

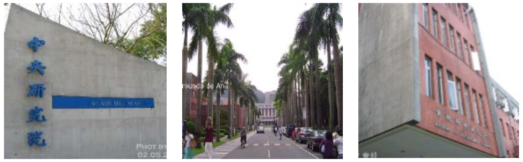


POSIX APIs and System Calls

- Unix systems include several libraries of functions that provide APIs to programmers.
- Some of the APIs defined by the *libc* standard C library refer to *wrapper routines*.
  - The API could offer its services directly in User Mode.
  - A single API function could make several system calls.
- The POSIX standard refers to APIs and not to system calls. A system can be certified as POSIX-compliant if it offers the proper set of APIs to the application programs.
- System calls belong to the kernel, while User Mode libraries don’t.
- Most wrapper routines return an integer value, whose meaning depends on the corresponding system call.
  - A return value of –1 usually indicates that the kernel was unable to satisfy the process request.
  - The POSIX standard specifies the macro names of several error codes. Defined in *include/asm-i386/errno.h*

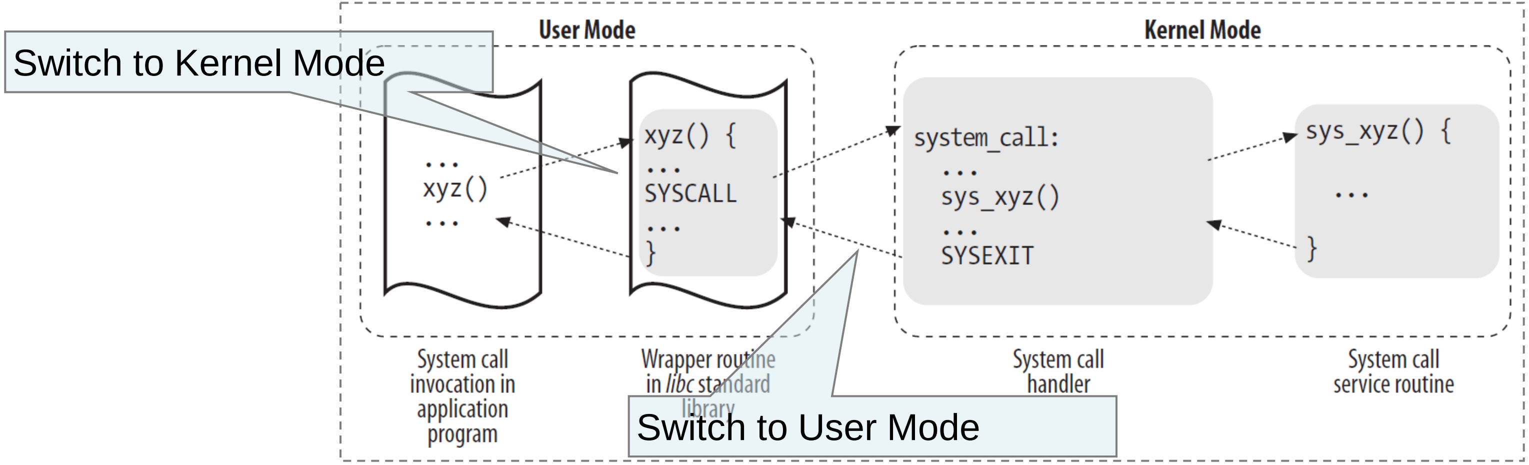
Outline

- POSIX APIs and System Calls
- **System Call Handler and Service Routines**
- Entering and Exiting a System Call
- Parameter Passing
- Kernel Wrapper Routines



System Call Handler and Service Routines

- The system call handler, which has a structure similar to that of the other exception handlers:
  - Saves the contents of most registers in the Kernel Mode stack.
  - Handles the system call by invoking a corresponding C function called the *system call service routine*.
  - Exits from the handler: the registers are loaded with the values saved in the Kernel Mode stack, and the CPU is switched back to User Mode.
- To associate each *system call number* with its corresponding service routine, the kernel uses a *system call dispatch table*, which is stored in the sys_call_table array and has NR_syscalls entries.



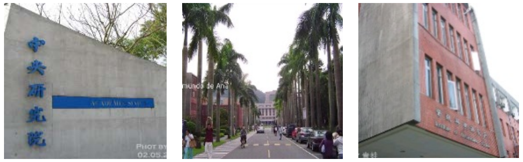
```
...
xyz()
...
```

```
xyz() {
...
SYSCALL
...
}
```

```
system_call:
  ...
  sys_xyz()
  ...
  SYSEXIT
```

```
sys_xyz() {
  ...
}
```

User Mode
Kernel Mode
Switch to Kernel Mode
System call invocation in application program
Wrapper routine in libc standard library
System call handler
System call service routine
Switch to User Mode



Outline

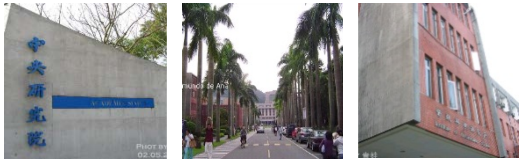
- POSIX APIs and System Calls
- System Call Handler and Service Routines
- **Entering and Exiting a System Call**
- Parameter Passing
- Kernel Wrapper Routines

Entering and Exiting a System Call

- Native applications can invoke a system call in two different ways:
  - 1. By executing the int $0x80 assembly language instruction
  - 2. By executing the *sysenter* assembly language instruction
- The kernel can exit from a system call:
  - By executing the iret assembly language instruction.
  - By executing the sysexit assembly language instruction
- Supporting two different ways to enter the kernel is not as simple as it might look:
  - The kernel must support both libraries.
  - A standard library that makes use of the *sysenter* instruction must be able to cope with older kernels that support only the int $0x80 instruction.
  - The kernel and the standard library must be able to run both on the older and the new processors.

System Call with int 0x80

- The vector 128 is associated with the kernel entry point.
- The trap_init() function sets up the Interrupt Descriptor Table entry corresponding to vector 128 as follows:
- The call loads the following values into the gate descriptor fields:
  - *Segment* Selector: _ _KERNEL_CS
  - *Offset*: The pointer to the *system_call( )*
  - *Type*: 15 (a Trap)
  - *DPL*: set to 3



The system_call() Function

- Start by saving the system call number and all the CPU registers that may be used by the exception handler on the stack—except for eflags, cs, eip, ss, and esp, which have already been saved automatically by the control unit.

```
system_call:
  pushl %eax
  SAVE_ALL
  movl $0xffffe000, %ebx /* or 0xfffff000 for 4-KB stacks */
  andl %esp, %ebx
```

Get the address of *thread_info*

- If this is the case, system_call( ) invokes the *do_syscall_trace( )* function twice if either one of the TIF_SYSCALL_TRACE and TIF_SYSCALL_AUDIT flags included in the *flags* field of the *thread_info* structure is set.

- Then a validity check is performed:

```
   cmpl $NR_syscalls, %eax
   jb nobadsys
   movl $(-ENOSYS), 24(%esp)
   jmp resume_userspace
nobadsys:
   call *sys_call_table(0, %eax, 4)
```

stores the -ENOSYS value in the stack location where the eax register has been saved

Invoke the specific service routine

Exiting from the System Call

- The system_call( ) function gets its return code from eax and stores it in the stack location where the User Mode value of the eax register is saved:

```
movl %eax, 24(%esp)
```

- Then, the *system_call()* function disables the local interrupts and checks the *flags* in the *thread_info* structure of current:

```
cli
movl 8(%ebp), %ecx
testw $0xffff, %cx
je restore_all
```

The flags field is at offset 8 in the thread_info structure

If none of these flags is set, the function jumps

System Call via *sysenter*

- The int assembly language instruction is inherently slow because it performs several consistency and security checks.
- The sysenter instruction provides a faster way to switch from User Mode to Kernel Mode.
- Three special registers that must be loaded with the following information:
  - SYSENTER_CS_MSR
    - The Segment Selector of the kernel code segment
  - SYSENTER_EIP_MSR
    - The linear address of the kernel entry point
  - SYSENTER_ESP_MSR
    - The kernel stack pointer

System Call via *sysenter* (Cont.)

- When the *sysenter* instruction is executed, the CPU control unit:
  - 1. Copies the content of SYSENTER_CS_MSR into cs.
  - 2. Copies the content of SYSENTER_EIP_MSR into eip.
  - 3. Copies the content of SYSENTER_ESP_MSR into esp.
  - 4. Adds 8 to the value of SYSENTER_CS_MSR, and loads this value into *ss*. (because the descriptor of the kernel stack segment follows the descriptor of the kernel code segment).

System Call via *sysenter* (Cont.)

- The enable_sep_cpu() function is executed once by every CPU at system initialization for:
  - 1. Writes the Segment Selector of the kernel code (_ _KERNEL_CS) in the SYSENTER_CS_MSR register.
  - 2. Writes in the SYSENTER_CS_EIP register the linear address of the *sysenter_entry()*
  - 3. Computes the linear address of the end of the local TSS, and writes this value in the SYSENTER_CS_ESP register.
    - The User Mode wrapper routine cannot properly set this register, because it does not know the address of this memory area;
    - On the other hand, the value of the register must be set before switching to Kernel Mode.
    - Therefore, the kernel initializes the register so as to encode the address of the *Task State Segment* of the local CPU.

The vsyscall Page

- A wrapper function in the *libc* standard library can make use of the *sysenter* instruction only if both the CPU and the Linux kernel support it.
  - In the initialization phase the sysenter_setup() function builds a page frame called *vsyscall page* containing a small ELF shared object.
  - When a process issues an ***execve()*** system call to start executing an ELF program, the code in the *vsyscall page* is dynamically linked to the process address space.

The sysenter_setup() Function

- Allocates a new page frame for the vsyscall page and associates its physical address with the FIX_VSYSCALL fix-mapped linear address.
- Then, the function copies in the page either one of two predefined ELF shared objects.
  - If the CPU does not support sysenter:

```
__kernel_vsyscall:
   int $0x80
   ret
```

  - Otherwise,

```
__kernel_vsyscall:
   pushl %ecx
   pushl %edx
   pushl %ebp
   movl %esp, %ebp
   sysenter
```

When a wrapper routine in the standard library must invoke a system call, it calls the *__kernel_vsyscall()* function

Entering the System Call via *sysenter*

- 1. The wrapper routine in the standard library loads the system call number into the *eax* register and calls the *_ _kernel_vsyscall()* function.
- 2. The *_ _kernel_vsyscall()* function saves on the User Mode stack the contents of ebp, edx, and ecx, copies the user stack pointer in *ebp*, and then executes the *sysenter* instruction.
- 3. The CPU switches from User Mode to Kernel Mode, and starts executing the *sysenter_entry()* function. (pointed by the SYSENTER_EIP_MSR register)
- 4. The sysenter_entry() assembly language:
  - a. Sets up the kernel stack pointer:
  - b. Enable local interrupts:
  - c. Save the related Segment Selector, regs, and the return address
  - d. Restore the epb value passed by the wrapper func.
  - e. Invoke the system call handler.

Load from the 512B local TSS

```
movl -508(%esp), %esp
```

```
pushl $(__USER_DS)
pushl %ebp
pushfl
pushl $(__USER_CS)
pushl $SYSENTER_RETURN
```

Existing from the System Call

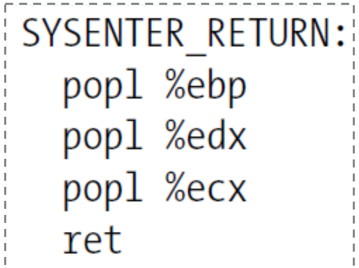
- First, sysenter_entry() gets the return code of the system call service routine from *eax* and stores it in the kernel stack location where the User Mode value of the eax register is saved.
- Then, the function disables the local interrupts and checks the *flags* in the *thread_info* structure of current and handles them.
- If sysenter_entry() determines the flags are cleared, it return to User Mode:

```
movl 40(%esp), %edx
movl 52(%esp), %ecx
xorl %ebp, %ebp
sti
sysexit
```

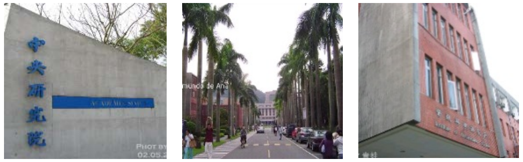
The edx and ecx registers are loaded with a couple of the stack values saved by sysenter_entry() in step 4c in the previos section: edx gets the address of the SYSENTER_RETURN label, while ecx gets the current user data stack pointer.

The sysexit Instruction

- Allow a fast switch from Kernel Mode to User Mode:
  - 1. Adds 16 to the value in the SYSENTER_CS_MSR register, and loads the result in the *cs* register.
  - 2. Copies the content of the *edx* register into the *eip* register.
  - 3. Adds 24 to the value in the SYSENTER_CS_MSR register, and loads the result in the *ss* register.
  - 4. Copies the content of the ecx register into the esp register.
- The SYSENTER_RETURN code
  - The code at the *SYSENTER_RETURN* label is stored in the *vsyscall page*, and it is executed when a system call is being terminated



```
SYSENTER_RETURN:
    popl %ebp
    popl %edx
    popl %ecx
    ret
```



Outline

- POSIX APIs and System Calls
- System Call Handler and Service Routines
- Entering and Exiting a System Call
- **Parameter Passing**
- Kernel Wrapper Routines