

Lecture 3

CPU User vs. Kernel mode

We have seen that programs runs not in the real machines, but instead in virtual machines that is controlled by the OS.

This week we look at how these virtual machines work efficiently.

References:

- [OSC] Sections 2.5 (Hardware Protection), 3.5 (System structure)
- [ULK] pp. 19–23 (Process/Kernel Mode, Process Implementation, Reentrant Kernels, Process Address Space).

POS(0230A)

POS(0230A)-3.1

But...

The approach in the last page: the OS “interprets” the program we write.

But that approach has a serious drawback: it is **very slow!**

Although Java is essentially taking the approach of interpreting.

- To execute an instruction of the virtual machine, the OS has to spend dozens of instructions for fetch, check, execute and write back.
- Note that each iteration does exactly what is done by a **machine cycle** of the computer for a normal CPU instruction.
- The only thing extra: **check** the permission and **translate** the addresses from virtual addresses to real addresses.

Is there a faster solution, then?

POS(0230A)-3.2

POS(0230A)-3.3

Regaining control

The CPU switches to the kernel mode when an **interrupt** occurs.

At that time, the CPU will execute an **interrupt handler**, storing the original CPU mode and other registers in the stack.

An **interrupt return** (IRET) instruction restores all the original status.

This happens in the following situations...

- The process perform a **disallowed instruction**.
Perhaps it tries to do I/O directly, or it writes to invalid memory, etc.
- The process **asks for service** by the OS (makes a **system call**).
- A **hardware device** requests attention by sending an **interrupt**.
E.g., the hard disk has finished writing. It sends an interrupt to the CPU requesting for immediate attention, so that the CPU can send the next chunk of data.

POS(0230A)-3.4

Which instructions are “privileged”?

It is indeed quite easy for an OS to implement a virtual machine...

- For each virtual machine, the OS loads the programs of the virtual machine to memory **as data**, and allocate memory for program data.
- The OS reads repeatedly does the following...
 1. **Select** a virtual machine to execute.
 2. Get the **next instruction** to run for that virtual machine.
 3. Translate the **virtual addresses** in the instruction.
 4. If the instruction is not **permitted**, sets up the next instruction for the virtual machine so that it runs the **error handler**.
If none is given, terminate the virtual machine.
 5. Execute whatever requested by the instruction. **Put the result back** to the virtual machine.

A better alternative to interpreting

Most CPU has two **modes** of operations:

- Kernel (or “monitor”) mode: all instructions are allowed.
- User mode: only selected “non-privileged” instructions are allowed.

So instead of interpreting the program by following 5 steps, the OS can run a virtual machine by...

1. **Setup** the computer in the **user mode**.
2. **Jump** to the code of the virtual machine.
Actually, it is done in a single step as we will soon see.

There are two problems remaining: how the OS regain control and select another virtual machine to run? And, how to translate addresses?

POS(0230A)-3.3

POS(0230A)-3.5

Implementing virtual machines via CPU modes

- The OS **sets up the interrupt handlers** and do other initializations.
- The OS **prepares** the context for the first virtual machine, and load the first program to its memory.
- The OS **IRET** into the virtual machine and thus give up control. The **CPU now runs the virtual machine**.
- If an interrupt occurs, the OS receives control again.
- The OS then do whatever things needed: modify the virtual machine to indicate error, perform service for the process (perhaps creating more process), wake up processes waiting for an event that occurred, etc.
- At the end the OS selects a virtual machine to run, and IRET into it. It will again give up control waiting for another interrupt.
So the OS is interrupt driven: it remains dormant until an interrupt comes.

POS(0230A)-3.6

Implement time sharing

- In the first lecture: CPU can produce an **illusion** that it executes many virtual machines at the same time, by switching among them quickly.
- How to implement it in the OS? How to make sure the program in a virtual machine will stop and give control back to the OS once a while?
- Easy... just have a device which **periodically generates interrupts!**
- The OS kernel thus receives control (via the interrupt handler), no matter what is being done by the user process.
- If at the end of an OS operation, it is found that the virtual machine has already executed for too long, then the OS will do a **context switch**, i.e., switch to run another process.

POS(0230A)-3.7

More concrete example of mode switching: system call

Linux: *getpid()* in the standard C library looks like this:

```

0x400b65e0 <getpid+0>: mov    $0x14,%eax
0x400b65e5 <getpid+5>: int   $0x80
0x400b65e7 <getpid+7>: cmp   $0xfffff001,%eax
0x400b65ec <getpid+12>: jae   0x400b65ef <getpid+15>
0x400b65ee <getpid+14>: ret
0x400b65ef <getpid+15>: ...   ; Error handling
0x400b660c <getpid+44>: jmp   0x400b65ee <getpid+14>

```

0x80 (128) is the system call interrupt number, 0x14 (20) means getpid.

Return value of the system call is stored in the `%eax` register.

In Linux: an unsigned number larger than 0xfffff001 means error. In 2's complement, it means a negative number in the range -4095 - -1.

The code above is called the C-library glue to system call. So that the users don't need to know how to make INTs themselves.

POS(0230A)-3.8

Within kernel

After `int $0x80` instruction, the CPU runs the kernel program in kernel mode. The code (in `<kernel>/arch/i386/kernel/entry.S`):

```

ENTRY(system_call)
    pushl %eax
    SAVE_ALL          # Save all registers
    GET_CURRENT(%ebx) # Get the PCB of the process
    cmpl $(NR_syscalls),%eax
    jae badsys
    testb $0x02,tsk_ptrace(%ebx)
    jne tracesys
    call *sys_call_table(,%eax,4)
    ...               # restore registers, choose new process
    iret              # IRET back to user mode

```

Note that the kernel compares the system call number against `NR_syscalls` to make sure the process uses a reasonable number.

It must not assume that the user process is a "good" person who always pass the right argument. Otherwise, if something unexpected happens, the whole OS crashes.

POS(0230A)-3.9

System call table

The `system_call` looks up a table called `sys_call_table`...

```

ENTRY(sys_call_table)
    .long sys_ni_syscall
    .long sys_exit
    ...
    .long sys_getpid # 20

```

It then calls the corresponding function `sys_getpid()`, which fetches the PID of the currently running process and put it to `%eax` for return.

The code in `sys_getpid()` is simply "return `current->tgid`".

In Linux, `return` values are put into `%eax`, so the above return to `system_call`, with `%eax` filled by the "Task Group ID" (the Linux way of saying PID) of the current process.

POS(0230A)-3.10

Summary: Linux system calls

- A program calls the C-library function like `getpid()`.
- `getpid()` puts the service number (20) into the register `%eax`, and generates a software interrupt using `int $0x80`.
- The interrupt vector specifies that the kernel function `system_call` should be executed in kernel mode.
- `system_call` checks that the service number is valid, and looks up a table to find and execute the implementation function `sys_getpid()`.
- `sys_getpid()` finds the PID and puts the result to `%eax` and returns to `system_call`.
- `system_call` uses IRET to get back to the user program, and the CPU falls back to user mode.
- `getpid()` checks for error and returns to the user program.

POS(0230A)-3.11

Is it really safe for the kernel?

Once a system call occurs...

- The CPU goes into kernel mode. [Unsafe?]
- The CPU executes the interrupt handler, which was arranged by the kernel and can't be modified by the user. [Okay, **code** segment is safe.]
- The interrupt handler may access data as instructed by the virtual machine, but the kernel would check them to make sure the user should be able to access it. [Okay, **data** segment is safe.]
- But... what about the **stack**? What if a bad virtual machine changes the stack pointer to a kernel address before making system call?
- The CPU automatically **switches to a fixed kernel stack**, so that won't harm the kernel.
Linux give each process (actually, thread) a small (2048 byte) kernel stack.

POS(0230A)-3.12

Characteristics of system calls

- Entry points of the kernel is completely fixed in ISR table. We can **add a system call** only by **modifying the kernel code**.
- Mode switches, and thus kernel calls are **slow**, since every register needs to be saved, and the machine must be setup for full privileges during the system call.
Most systems implement such calls to run in around 50–1000 CPU cycles.
- Outside the kernel, a process making mistakes only results in its own dismay. In the **kernel, bugs** easily result in **computer lock-up**.
- Kernel code are **responsible for maintaining the security barrier**. For example, if the kernel needs to write to a buffer provided by the user, it must check explicitly whether the user process has the permission.

POS(0230A)-3.13

How much to run in kernel mode?

In **monolithic kernels**, all services provided by the virtual machine runs in kernel mode.

- Example: Most Unix, OS/2
- Kernel has **process management** and **memory management** and **interprocess communication (IPC)**, **device drivers**, **filesystem**, **cache management**, **terminal control**, and perhaps **windowing system**.

In **micro-kernels**, only a **minimal set of services** is provided. Everything else is done using user-mode server processes.

- Examples: Mach, Windows NT (?)
- Kernel only retain **essential** features like **process management**, **memory management**, and **IPC**.
E.g., if a process want to read/write a file, it communicates to the file-system server using IPC. This is hidden in library, so not inconvenient.

POS(0230A)-3.14

Good and bad

- Micro-kernels is **more flexible**, as administrators can replace system services by **writing their own servers**.
E.g., if you want your own filesystem, you can write a filesystem server on top of anything you can access.
- Micro-kernels are small, so it is **easier to be correct**.
This is important, since kernel runs in the powerful kernel mode.
- There can be **multiple API** implemented on top of a micro-kernel.
If you want different API, just run different servers.

But micro-kernels are typically **slower**. E.g., for most simple services:

- Monolithic kernel: get into kernel mode (1 mode switch), the kernel returns the result (1 more mode switch).
- Micro-kernel: switch to another process (1 **context** switch), it returns the result via **IPC**, and your process continues (1 more context switch).
But most micro-kernels can switch mode much quicker than monolithic kernels.

POS(0230A)-3.15

Organization of kernel: Modules

To improve flexibility, many monolithic kernels implement kernel modules.

- Some **functionality can be loaded as modules** when required.
- When unneeded, kernel modules **can be unloaded from memory**.

Advantages

- Administrators can change system components without rebooting.
- The OS remains fast.

Disadvantages

- Modules bugs are kernel mode bugs, crashing the whole system.
- Impossible to allow normal (non-root) users to modify or replace system components.

POS(0230A)-3.16

Organization of kernel features: layered approach

Just like other complex projects, the implementation of most operating systems are structured as **layers to control complexity**.

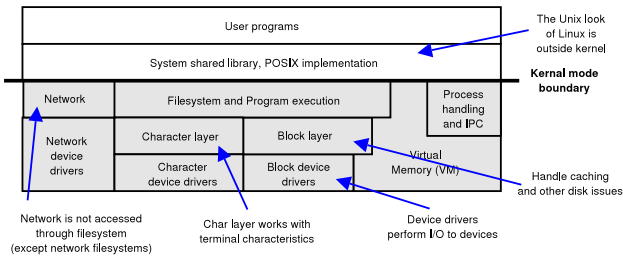
So that inter-dependencies between features are reduced.

- **Bottom** layer is the **hardware**. The **top** layer is the **application interface**.
- Each layer other than the bottom provides some **additional functionalities** for upper layers. Each layer normally **only use layers below it**.
- This **ease debugging and feature additions**: lower layers provide guarantees for the upper layers.

POS(0230A)-3.17

Example—Linux: monolithic kernel

Everything except library and applications are in a big kernel.
 Many features can be loaded by a module, so it is not as bad as it sounds.



Indeed, most Unix systems are similar, except that many implement POSIX within the kernel itself.

Example—GNU/Hurd: a real micro-kernel

Hurd uses a micro-kernel (Mach) to support processes and message passing. A Unix-like OS is built on top of it with a "translator".
 The translator automates some IPC calls required by the micro-kernel approach.

