History

The programs and OS all live in the same computer memory: we need some way of protecting programs and the OS from each other

History

The programs and OS all live in the same computer memory: we need some way of protecting programs and the OS from each other

This has to be done by hardware support as it needs to be fast and unobtrusive: potentially every memory access needs to be checked

History

The programs and OS all live in the same computer memory: we need some way of protecting programs and the OS from each other

This has to be done by hardware support as it needs to be fast and unobtrusive: potentially every memory access needs to be checked

We shall start by looking at general hardware protection mechanisms

History

Certain operations, like accessing tape or a printer, must be reserved for use by the OS and not be accessible by a random user program

History

Certain operations, like accessing tape or a printer, must be reserved for use by the OS and not be accessible by a random user program

So in the hardware (CPU) machine instructions are divided into two (or more) classes

History

Certain operations, like accessing tape or a printer, must be reserved for use by the OS and not be accessible by a random user program

So in the hardware (CPU) machine instructions are divided into two (or more) classes

- Unprivileged operations. Like addition, jumps. Any program can execute these

History

Certain operations, like accessing tape or a printer, must be reserved for use by the OS and not be accessible by a random user program

So in the hardware (CPU) machine instructions are divided into two (or more) classes

- Unprivileged operations. Like addition, jumps. Any program can execute these
- Privileged operations. Like access peripherals, reboot the machine. Only certain privileged programs can run these

History

Certain operations, like accessing tape or a printer, must be reserved for use by the OS and not be accessible by a random user program

So in the hardware (CPU) machine instructions are divided into two (or more) classes

- Unprivileged operations. Like addition, jumps. Any program can execute these
- Privileged operations. Like access peripherals, reboot the machine. Only certain privileged programs can run these

And the processor can run in two (or more) modes

History

Certain operations, like accessing tape or a printer, must be reserved for use by the OS and not be accessible by a random user program

So in the hardware (CPU) machine instructions are divided into two (or more) classes

- Unprivileged operations. Like addition, jumps. Any program can execute these
- Privileged operations. Like access peripherals, reboot the machine. Only certain privileged programs can run these

And the processor can run in two (or more) modes

- Unprivileged. Normal computation, called *user mode*

History

Certain operations, like accessing tape or a printer, must be reserved for use by the OS and not be accessible by a random user program

So in the hardware (CPU) machine instructions are divided into two (or more) classes

- Unprivileged operations. Like addition, jumps. Any program can execute these
- Privileged operations. Like access peripherals, reboot the machine. Only certain privileged programs can run these

And the processor can run in two (or more) modes

- Unprivileged. Normal computation, called *user mode*
- Privileged. For systems operation, called *kernel mode*

History

Modern processor architectures can have four or more levels of privilege, but for the most part it is rare that more than two levels are used in commodity computers

History

Modern processor architectures can have four or more levels of privilege, but for the most part it is rare that more than two levels are used in commodity computers

For example, the Intel x86 architecture has four *rings*. Ring 0 can execute any instruction, while Ring 3 is for user mode. Rings 1 and 2 are rarely used these days

History

Modern processor architectures can have four or more levels of privilege, but for the most part it is rare that more than two levels are used in commodity computers

For example, the Intel x86 architecture has four *rings*. Ring 0 can execute any instruction, while Ring 3 is for user mode. Rings 1 and 2 are rarely used these days

OS/2 used Ring 2

History

Modern processor architectures can have four or more levels of privilege, but for the most part it is rare that more than two levels are used in commodity computers

For example, the Intel x86 architecture has four *rings*. Ring 0 can execute any instruction, while Ring 3 is for user mode. Rings 1 and 2 are rarely used these days

OS/2 used Ring 2

The latest Intel and AMD architectures added a Ring -1 (for OS virtualisation)

History

Note that privilege is a state of the *processor*, not the program, but we tend to say “a privileged program” rather than “a program running with the CPU in privileged mode”

History

Note that privilege is a state of the *processor*, not the program, but we tend to say “a privileged program” rather than “a program running with the CPU in privileged mode”

If an unprivileged program (i.e., a program running in an unprivileged mode) tries to execute a privileged operation the hardware causes an interrupt (also called a system *trap*) and sets the processor to privileged mode. The interrupt service routine then jumps to the OS

History

Note that privilege is a state of the *processor*, not the program, but we tend to say “a privileged program” rather than “a program running with the CPU in privileged mode”

If an unprivileged program (i.e., a program running in an unprivileged mode) tries to execute a privileged operation the hardware causes an interrupt (also called a system *trap*) and sets the processor to privileged mode. The interrupt service routine then jumps to the OS

The OS can then decide what to do

History

Note that privilege is a state of the *processor*, not the program, but we tend to say “a privileged program” rather than “a program running with the CPU in privileged mode”

If an unprivileged program (i.e., a program running in an unprivileged mode) tries to execute a privileged operation the hardware causes an interrupt (also called a system *trap*) and sets the processor to privileged mode. The interrupt service routine then jumps to the OS

The OS can then decide what to do

For example, the OS may decide to disallow the operation, and kill the program (i.e., not run it any more)

History

The system starts in kernel (privileged) mode

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program
3. The program runs user mode (unprivileged)

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program
3. The program runs user mode (unprivileged)
4. The program finishes or decides it needs a system resource

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program
3. The program runs user mode (unprivileged)
4. The program finishes or decides it needs a system resource
5. The program executes a special “call OS” (or *syscall*) instruction that jumps to the OS

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program
3. The program runs user mode (unprivileged)
4. The program finishes or decides it needs a system resource
5. The program executes a special “call OS” (or *syscall*) instruction that jumps to the OS
6. This enables privileged mode, so the OS regains control, with privilege

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program
3. The program runs user mode (unprivileged)
4. The program finishes or decides it needs a system resource
5. The program executes a special “call OS” (or *syscall*) instruction that jumps to the OS
6. This enables privileged mode, so the OS regains control, with privilege
7. The OS decides what to do next

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program
3. The program runs user mode (unprivileged)
4. The program finishes or decides it needs a system resource
5. The program executes a special “call OS” (or *syscall*) instruction that jumps to the OS
6. This enables privileged mode, so the OS regains control, with privilege
7. The OS decides what to do next

Of course, even if the program does not do a syscall, a timer interrupt will come along at some point, anyway

History

The syscall instruction always jumps to the same place in the OS. So the program cannot use it to gain privilege for itself and run its own code privileged

History

The syscall instruction always jumps to the same place in the OS. So the program cannot use it to gain privilege for itself and run its own code privileged

This to-ing and fro-ing between modes ensures that the OS is running in privileged mode and the user program is running in unprivileged mode

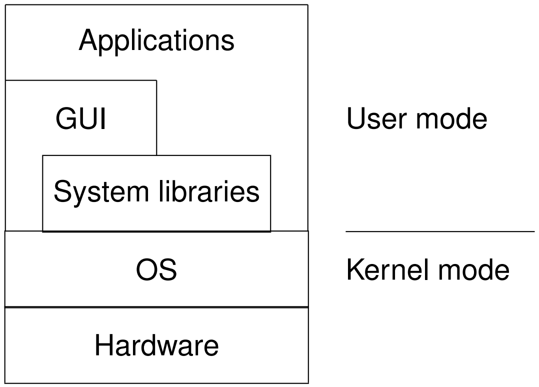
History

The syscall instruction always jumps to the same place in the OS. So the program cannot use it to gain privilege for itself and run its own code privileged

This to-ing and fro-ing between modes ensures that the OS is running in privileged mode and the user program is running in unprivileged mode

And the user program can never manage to get into privileged mode as every transition to privileged mode is tied by the hardware to a jump to the OS

History



There is a strict divide between kernel (OS) code and user code, controlled by the hardware

History

Unless there are bugs in the kernel code…

□

History

Unless there are bugs in the kernel code…

Incidentally, the system libraries usually include a bunch of “nice” interfaces to the syscalls: wrapping them to make using them easier

□

History

Unless there are bugs in the kernel code…

Incidentally, the system libraries usually include a bunch of “nice” interfaces to the syscalls: wrapping them to make using them easier

E.g., the “open file” syscall might need certain values (file name, etc.) to be placed in certain CPU registers; and the “open file” opcode to be placed in a register before the syscall

□

History

Unless there are bugs in the kernel code…

Incidentally, the system libraries usually include a bunch of “nice” interfaces to the syscalls: wrapping them to make using them easier

E.g., the “open file” syscall might need certain values (file name, etc.) to be placed in certain CPU registers; and the “open file” opcode to be placed in a register before the syscall

The `open` system library function simply hides these details from the programmer

□