Mechanism: Limited Direct Execution - SistOp

Lautaro Bachmann

Contents

Mechanism: Limited Direct Execution

Introduction

Overview

In order to virtualize the CPU, the operating system needs to somehow share the physical CPU among many jobs running seemingly at the same time. run one process for a little while, then run another one, and so forth. By time sharing the CPU in this manner, virtualization is achieved.

challenges,

performance:
how can we implement virtualization without adding excessive overhead to the system?

control:
how can we run processes efficiently while retaining control over the CPU? without control, a process could simply run forever and take over the machine, or access information that it should not be allowed to access. **Obtaining high performance while maintaining control is thus one of the central challenges in building an operating system.**

Basic Technique: Limited Direct Execution

Description

just run the program directly on the CPU. when the OS wishes to start a program running, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory, locates its entry point, jumps to it, and starts running the user's code.

a few problems

The first
if we just run a program, how can the OS make sure the program doesn't do anything that we don't want it to do, while still running it efficiently?

The second:
when we are running a process, how does the operating system stop it from running and switch to another process,

the "limited" part of the name

without limits on running programs, the OS wouldn't be in control of anything and thus would be "just a library" — a very sad state of affairs for an aspiring operating system!

Problem #1: Restricted Operations

Direct execution

has the obvious advantage of being fast; the program runs natively on the hardware CPU and thus executes as quickly as one would expect.

THE CRUX: HOW TO PERFORM RESTRICTED OPERATIONS

A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system. How can the OS and hardware work together to do so?

ASIDE: WHY SYSTEM CALLS LOOK LIKE PROCEDURE CALLS

You may wonder why a call to a system call, looks exactly like a typical procedure call in C; The simple reason: it is a procedure call, but hidden inside that procedure call is the famous trap instruction. the library uses an agreed-upon calling convention with the kernel to put the arguments in well-known locations puts the system-call number into a well-known location as well and then executes the aforementioned trap instruction. The code in the library after the trap unpacks return values and returns control to the program that issued the system call. the parts of the C library that make system calls are hand-coded in assembly,

processor mode,

user mode;
code that runs in user mode is restricted in what it can do. For example, when running in user mode, a process can't issue I/O requests;

kernel mode,
The mode that the operating system runs in. In this mode, code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions.

perform privileged operation as a user

To enable this, virtually all modern hardware provides the ability for user programs to perform a **system call.** system calls allow the kernel to carefully expose certain key pieces of functionality to user programs,

How to execute a system call

a program must execute a special trap instruction. This instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode; once in the kernel, the system can now perform whatever privileged operations are needed (if allowed), When finished, the OS calls a special **return-from-trap** instruction, which, as you might expect, returns into the calling user program while simultaneously reducing the privilege level back to user mode.

what code executes upon a trap.

The kernel does so by setting up a **trap table** at boot time. When the machine boots up, it does so in privileged (kernel) mode, and thus is free to configure machine hardware as need be. One of the first things the OS thus does is to tell the hardware what code to run when certain exceptional events occur.

The OS informs the hardware of the locations of these **trap handlers,** Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted, and thus the hardware knows what to do when system calls and other exceptional events take place.

system-call number

is usually assigned to each system call. The user code is thus responsible for placing the desired system-call number in a register or at a specified location on the stack;

the OS, when handling the system call inside the trap handler, examines this number, ensures it is valid, and, if it is, executes the corre- sponding code.

protection;
user code cannot specify an exact address to jump to, but rather must request a particular service via number. being able to execute the instruction to tell the hardware where the trap tables are is a very powerful capability. Thus, it is a **privileged operation.**

Kernel Stack
We assume each process has a kernel stack where registers are saved to and restored from when transitioning into and out of the kernel.

limited direct execution protocol phases

Boot time
The kernel initializes the trap table, and the CPU remembers its location for subsequent use. The kernel does so via a privileged instruction

Running a process
the kernel sets up a few things before using a return-from-trap instruction to start the execution of the process; this switches the CPU to user mode and begins running the process. When the process wishes to issue a system call, it traps back into the OS, which handles it and once again returns control via a return-from-trap to the process.

Problem #2: Switching Between Processes

Overview

if a process is running on the CPU, this by definition means the OS is not running. If the OS is not running, how can it do anything at all? (hint: it can't)

THE CRUX: HOW TO REGAIN CONTROL OF THE CPU

How can the operating system regain control of the CPU so that it can switch between processes?

A Cooperative Approach: Wait For System Calls

the cooperative approach.
In this style, the OS trusts the processes of the system to behave reasonably. Processes that run for too long are assumed to periodically give up the CPU so that the OS can decide to run some other task.

in a cooperative scheduling system, the OS regains control of the CPU by waiting for a system call or an illegal operation of some kind to take place.

A Non-Cooperative Approach: The OS Takes Control

Without some additional help from the hardware, it turns out the OS can't do much at all when a process refuses to make system calls (or mistakes) and thus return control to the OS.

THE CRUX: HOW TO GAIN CONTROL WITHOUT COOPERATION

How can the OS gain control of the CPU even if processes are not being cooperative? What can the OS do to ensure a rogue process does not take over the machine?

The answer
A timer device can be programmed to raise an interrupt every so many milliseconds; when the interrupt is raised, the currently running process is halted, and a pre-configured interrupt handler in the OS runs. At this point, the OS has regained control of the CPU, and thus can do what it pleases:

TIP: DEALING WITH APPLICATION MISBEHAVIOR

Operating systems often have to deal with misbehaving processes, In modern systems, the way the OS tries to handle such malfeasance is to simply terminate the offender.

the hardware has some responsibility

when an interrupt occurs, in particular to save enough of the state of the program that was running when the interrupt occurred such that a subsequent return-fromtrap instruction will be able to resume the running program correctly.

Saving and Restoring Context

Now that the OS has regained control, a decision has to be made: whether to continue running the currently-running process, or switch to a different one. This decision is made by a part of the operating system known as the **scheduler;**

If the decision is made to switch,

the OS then executes a low-level piece of code which we refer to as a **context switch.**

Context Switch

A context switch is conceptually simple: all the OS has to do is save a few register values for the currently-executing process and restore a few for the soon-to-be-executing process By doing so, the OS thus ensures that when the return-from-trap instruction is finally executed, instead of returning to the process that was running, the system resumes execution of another process.

save the context of the currently-running process,

the OS will execute some low-level assembly code to save the general purpose registers, and then restore said registers, and switch to the kernel stack for the soon-to-be-executing process. By switching stacks, the kernel enters the call to the switch code in the context of one process and returns in the context of another

two types of register saves/restores

Timer interrupts

when the timer interrupt occurs; in this case, the user registers of the running process are implicitly saved by the hardware, using the kernel stack of that process.

Switch

when the OS decides to switch from A to B; in this case, the kernel registers are explicitly saved by the software but this time into memory in the process structure of the process.

Worried About Concurrency?

the OS does indeed need to be concerned as to what happens if, during interrupt or trap handling, another interrupt occurs. One simple thing an OS might do is **disable interrupts** during interrupt processing;

the OS has to be careful disabling interrupts for too long could lead to lost interrupts, which is (in technical terms) bad. Operating systems also have developed a number of sophisticated **locking schemes** to protect concurrent access to internal data structures.

Summary

limited direct execution.

The basic idea is straightforward: just run the program you want to run on the CPU, but first make sure to set up the hardware so as to limit what the process can do without OS assistance.

the concept of baby proofing a room:
locking cabinets containing dangerous stuff and covering electrical sockets. When the room is thus readied, you can let your baby roam freely, secure in the knowledge that the most dangerous aspects of the room have been restricted.

the OS "baby proofs" the CPU,
by first (during boot time) setting up the trap handlers and starting an interrupt timer, and then by only running processes in a restricted mode.

ASIDE: KEY CPU VIRTUALIZATION TERMS (MECHANISMS)

modes of execution:

a restricted user mode and a privileged (non-restricted) kernel mode.

system call

Typical user applications run in user mode, and use a system call to trap into the kernel to request operating system services.

The trap instruction

saves register state carefully, changes the hardware status to kernel mode, and jumps into the OS to a pre-specified destination: the **trap table.**

return-from-trap instruction,

When the OS finishes servicing a system call, it returns to the user program via another special return-from-trap instruction, which reduces privilege and returns control to the instruction after the trap that jumped into the OS.

trap tables

must be set up by the OS at boot time, and make sure that they cannot be readily modified by user programs.

timer interrupt.

Once a program is running, the OS must use hardware mechanisms to ensure the user program does not run forever, namely the **timer interrupt**

This approach is a non-cooperative approach to CPU scheduling.

context switch.

Sometimes the OS, during a timer interrupt or system call, might wish to switch from running the current process to a different one, a low-level technique known as a **context switch.**