**6: Mechanism: Limited Direct execution**

2da clase sobre esta unidad:

rebootear: vuelve a poner a la maquina en un estado previsible

-el proceso a por mas que este ejecutando en el kernel sigue ejecutando p el proceso b

mechanismo de cambio de contexto

**6 Mechanism: Limited Direct Execution**

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?

-The second is control: how can we run processes efficiently while retaining control over

the CPU?

6.1 Basic Technique: Limited Direct Execution

limited direct execution:

direct execution:

run the program directly on the CPU. Thus, 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 (from disk), locates its entry point (i.e., the main() routine or something similar), jumps to it, and starts running the user’s code

OS Program

Create entry for process list

Allocate memory for program

Load program into memory

Set up stack with argc/argv

Clear registers

Execute call main()

Run main()

Execute return from main

Free memory of process

Remove from process list

Figure 6.1: Direct Execution Protocol (Without Limits)

-“limited” part of the name arises from; without limits on running programs, the OS wouldn’t be in control of anything and thus would be “just a library”

6.2: Problem #1: Restricted Operations

A SIDE : W HY S YSTEM C ALLS L OOK L IKE P ROCEDURE C ALLS

You may wonder why a call to a system call, such as open() or read(), looks exactly like a typical procedure call in C; that is, if it looks just like a procedure call, how does the system know it’s a system call, and do all the right stuff? The simple reason: it is a procedure call, but hidden inside that procedure call is the famous trap instruction. More specifically, when you call open() (for example), you are executing a procedure call into the C library. Therein, whether for open() or any of the other system calls provided, the library uses an agreed-upon calling convention with the kernel to put the arguments to open() in well-known locations (e.g., on the stack, or in specific registers), puts the system-call number into a well-known location as well (again, onto the stack or a register), 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. Thus, the parts of the C library that make system calls are hand-coded in assembly, as they need to carefully

follow convention in order to process arguments and return values correctly, as well as execute the hardware-specific trap instruction. And now you know why you personally don’t have to write assembly code to trap into an OS; somebody has already written that assembly for you.

-user mode; code that runs in user mode is restricted in what it can do.

T IP : U SE P ROTECTED C ONTROL T RANSFER

The hardware assists the OS by providing different modes of execution. In user mode, applications do not have full access to hardware resources.

In kernel mode, the OS has access to the full resources of the machine. Special instructions to trap into the kernel and return-from-trap back to user-mode programs are also provided, as well as instructions that allow the OS to tell the hardware where the trap table resides in memory.

-**a system-call number is usually assigned to each system call.**

**6.3 Problem #2: Switching Between Processes**

The OS should just decide to stop one process and start another. But it actually is a little bit tricky: specifically, 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) it is a real problem: there is clearly no way for the OS to take an action if it is not running on the CPU. Thus we arrive at the crux of the problem.

A Cooperative Approach: Wait For System Calls

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.

A Non-Cooperative Approach: The OS Takes Control

a timer interrupt: 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. 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.

-Saving and Restoring Context

Now that the OS has regained control, whether cooperatively via a system call, or more forcefully via a timer interrupt, 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

6.4 Worried About Concurrency?

# void swtch(struct context \*\*old, struct context \*new);

# Save current register context in old

# and then load register context from new.

.globl swtch

swtch:

# Save old registers

movl 4(%esp), %eax # put old ptr into eax

popl 0(%eax)

# save the old IP

movl %esp, 4(%eax) # and stack

movl %ebx, 8(%eax) # and other registers

movl %ecx, 12(%eax)

movl %edx, 16(%eax)

movl %esi, 20(%eax)

movl %edi, 24(%eax)

movl %ebp, 28(%eax)

# Load new registers

movl 4(%esp), %eax # put new ptr into eax

movl 28(%eax), %ebp #restore other registers

movl 24(%eax), %edi

movl 20(%eax), %esi

movl 16(%eax), %edx

movl 12(%eax), %ecx

movl 8(%eax), %ebx#return addr put in place

movl 4(%eax), %esp #stack is switched here

pushl 0(%eax) # finally return into new ctxt

ret

Figure 6.4: The xv6 Context Switch Code

A SIDE : H OW L ONG C ONTEXT S WITCHES T AKE

A natural question you might have is: how long does something like a

context switch take? Or even a system call? For those of you that are cu-

rious, there is a tool called lmbench [MS96] that measures exactly those

things, as well as a few other performance measures that might be rele-

vant