INTRODUCTION TO COMPUTER SCIENCE · WEEK 10 · STATION III

Systems

Two ways to build an operating system. Why they are the same. Why one is used anyway — and the loop at its centre.

THREE PROBLEMS

An operating system solves three problems. Two are ordinary. One is not.

SHARING

One processor, many programs. Give each a turn — a *context switch* every few milliseconds, driven by a timer. Ordinary.

MEMORY

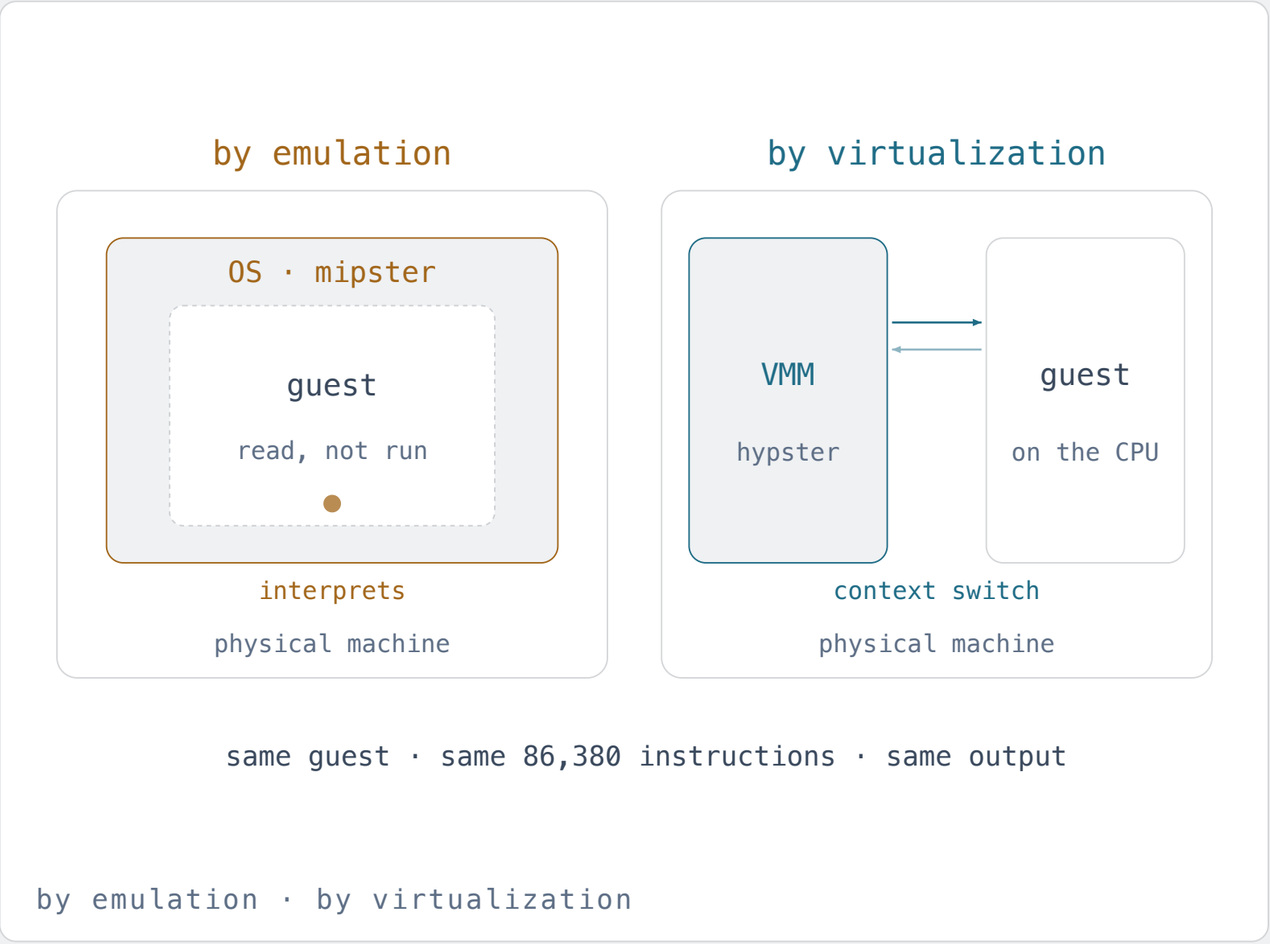
One memory, many programs. Give each its own addresses and translate — *paging*. Ordinary, and a lot of bookkeeping.

ISOLATION

None of them may touch the others, or the kernel. And the kernel must be isolated from the programs whose isolation it provides. Not ordinary. This is the loop.

Self-reference is not a party trick. It is the hard part of systems, and most courses step around it because in a production kernel it is buried under a million lines. Selfie makes it the subject and keeps the system small enough that the loop stays visible.

Two ways to build the same operating system.



By emulation. The operating system is an interpreter. The guest's code is data being read, one instruction at a time. Isolation is free: the guest never touches the processor. No self-reference anywhere.

By virtualization. The operating system puts the guest onto the real processor and takes it back when the timer fires or the guest asks for something. The guest runs at full speed — on the same hardware as the kernel.

Selfie has both: `mipster` is the first, `hypster` the second.

IN THE TERMINAL

They are semantically equivalent. Selfie can show it rather than say it.

```
$ make emu-emu
./selfie -c selfie.c -o selfie.m -m 3 -l selfie.m -m 2
…
> selfie.m: summary: 86380 executed instructions in tot
./selfie: summary: 222410917 executed instructions in t
```

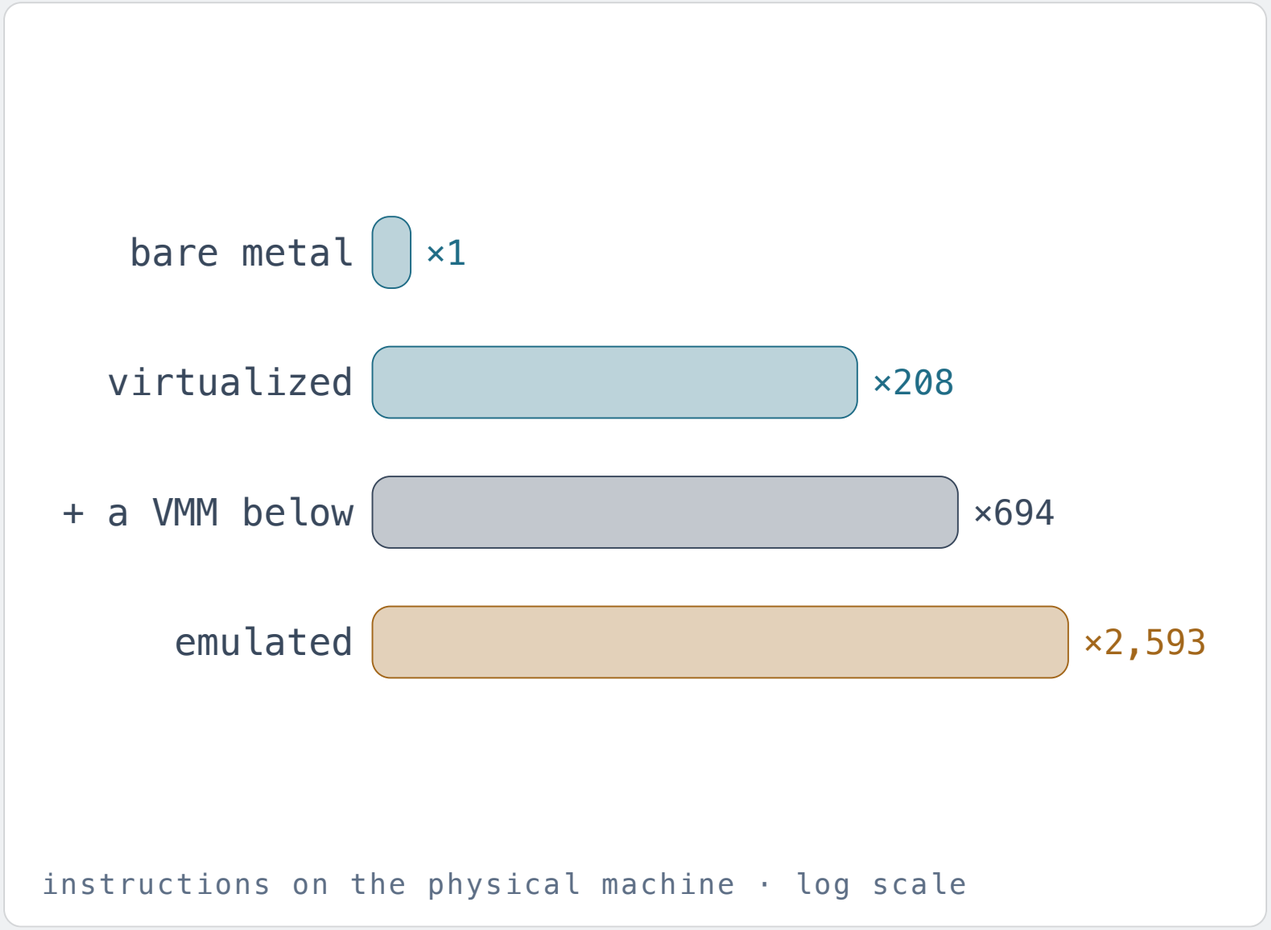
Same guest: selfie printing its synopsis. Hosted by an emulator on the left, by a hypervisor on the right. The guest executes 86,380 instructions both times — the same number, the same output. That is the evidence that the two designs are the same operating system.

```
$ make os-emu
./selfie -c selfie.c -o selfie.m -m 2 -l selfie.m -y 1
…
> selfie.m: summary: 86380 executed instructions in tot
./selfie: summary: 17860937 executed instructions in to
```

Only the bill for the machine underneath differs: 222 million instructions by emulation, 18 million by virtualization. Twelve times cheaper. That is the entire reason virtualization exists.

FOR EXACTLY ONE REASON

What the machine underneath is billed.



Bare metal: 85,754. Virtualized: ×208. With a hypervisor underneath as well: ×694. Emulated: ×2,593.

And the bar chart understates the case. On real work — selfie compiling selfie — the virtualized overhead collapses to ×1.72, because the guest runs on the processor and the kernel is only involved at the edges.

That collapse is the whole economic argument for virtualization, and for the cloud.

THE PRICE

Virtualization is emulation plus self-reference.

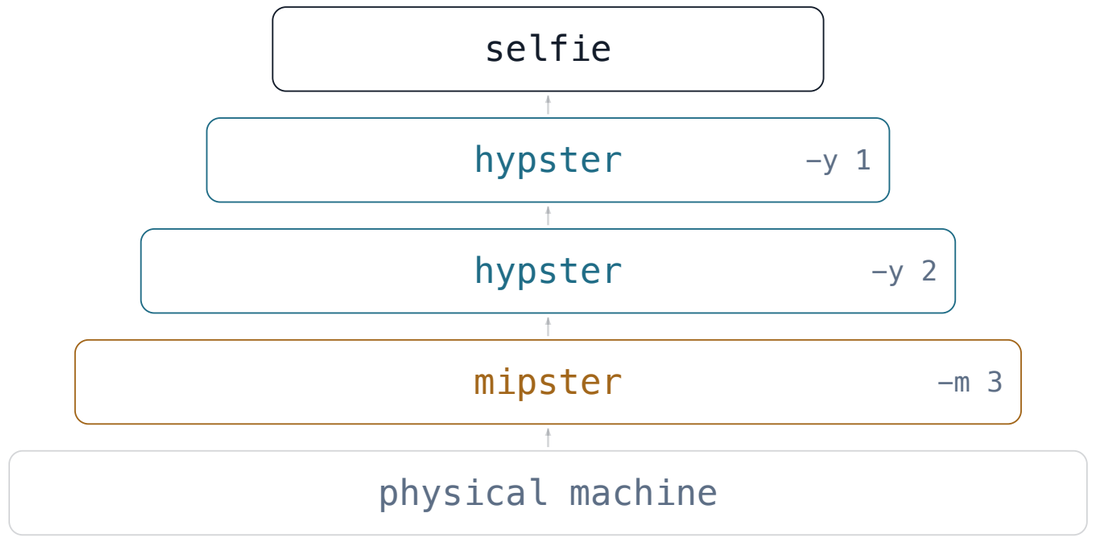
The kernel provides isolation. To provide it, the kernel runs on the same processor as the guests — so the kernel needs to be isolated too. From the guests. By whom?

By itself, would be the answer, and week 8 said what that is worth. Nothing certifies itself. So real kernels do it the other way: a small piece of code, isolated by construction, that must never fault, written and checked by hand — the trusted computing base. Keeping it small is the microkernel programme.

The certification comes from outside the loop. Same shape as Thompson's second compiler. Same theorem.

SELF-HOSTING

A hypervisor whose virtual machines can run the hypervisor.



hypster on hypster on mipster

```
$ ./selfie -c selfie.c -o selfie.m -m 3 -l selfie.m -y 2
…
```

The third command from week 1. An emulated machine hosts a hypervisor that hosts a hypervisor that hosts selfie, which prints its synopsis. Still reasonably fast, because there is only one emulator in the stack and each hypervisor adds little.

Understand the first command and you understand compilers; the second, machines; the third, operating systems. Only the third is hard to implement, and this deck said exactly where.

BOUNDING, NOT DECIDING

The timer interrupt is the engineering answer to the halting problem.

Last week: no kernel can decide whether the program it is running will ever stop. This week: it does not try. Every few million instructions a timer takes the processor away, whether or not the program was about to finish, and gives it to the next one.

Paging does the same for space: not *how much memory will it need*, which is undecidable, but *here is what it may have*, enforced.

THE PATTERN

What a system cannot decide, it bounds. Time by the timer, space by the page table, trust by the trusted computing base. That is Rice's theorem, answered — and it is the shape of every tool in week 12.

BEFORE NEXT WEEK

Recommended exercises.

1. Read the Computing chapter's opening, *Emulation*, *Virtualization*, and *Self-Reference*.
2. Run `make emu`, `make emu-emu`, `make os-emu`. Write down the guest's instruction count and the host's for each, and compute the factors.
3. Run `make self-emu` and `make self-os-emu` if your machine has the time, and compute the overhead on real work.
4. Say in one paragraph why an operating system by emulation has no self-reference and one by virtualization does.
5. Name the outside check in each of: Thompson's compiler, Gödel's second theorem, the trusted computing base.

NEXT WEEK

Cost: a question with a guaranteed answer you will never receive; hard to find, easy to check; a SAT solver in 400 lines; and every step that forgets costs energy.