SYSTEMS ENGINEERING · WEEK 12 · STATION III

Universality for Systems

The universal machine, the halting problem, Rice on schedulers, and deadlock detection as the approximation a kernel can afford.

THE UNIVERSAL MACHINE

Mipster is one program that becomes every program. That is why a kernel exists.

Turing, 1936: one machine, handed the description of any other, does what that one does. The emulator of week 3 is that machine, and the program it is handed is data. A universal machine cannot be specialised to safe programs only, because it must accept every description.

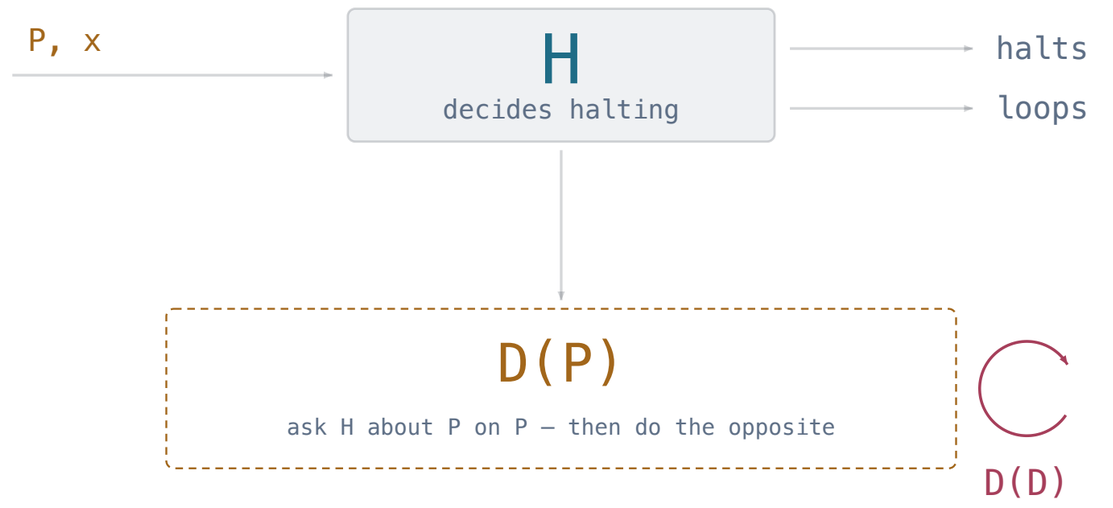
So the machine will run programs that loop, that address memory they do not own, that never yield. Nothing can screen them out in advance, and something has to deal with them at run time. That something is the kernel. Operating systems are what universality makes necessary.

THE POINT OF THE WEEK

The kernel is not a collection of workarounds for badly written programs. It is the run-time consequence of three theorems about what cannot be known at design time. Each theorem, each consequence, on the next slides.

HALTING

No program decides whether another program stops. The proof is a process that asks about itself.



the halting construction: a program that halts iff it does not

Suppose *H* decides halting. Build *D*: run *H* on its own input, and loop if *H* says halt, halt if it says loop. Run *D* on *D*. Either answer contradicts itself. Cantor's diagonal, on programs; Gödel's sentence, on machines. Self-reference, the guiding principle of the book, does the work.

The kernel's answer, week 5: do not decide, bound. The timer interrupt is a proof-free guarantee that every context yields, bought by preempting the ones that would not. What a scheduler knows about a program's future is exactly nothing, and the design is consistent with that.

Fork, week 7, is the construction made literal: a process that runs a copy of itself and asks the copy a question. The kernel copes because the copy is isolated, and isolation is what stops self-reference from becoming a contradiction.

RICE

Every non-trivial property of what a program does is undecidable. Here is the kernel's list.

THE KERNEL WOULD LIKE TO KNOW	A PROPERTY OF	WHAT IT DOES INSTEAD
will this process yield?	behaviour	timer interrupt · week 5
how long will this job run?	behaviour	a policy, round robin · week 5
will this process touch that page?	behaviour	page faults on demand · week 4
will this block be used again?	behaviour	reachability · week 9
will these threads deadlock?	behaviour	detect the cycle when it exists · today
is this program safe to run?	behaviour	isolate it, and bound it · every week
has this process called exit?	the door	decided, exactly · week 7

The last row is the pattern: what happens at the interface is syntax, recorded and decidable. What happens inside is semantics, and Rice says the kernel may not know it. Every mechanism in the class sits in the right-hand column.

DEADLOCK

Prevention is undecidable. Detection is a cycle in a graph.

Four conditions, all necessary: mutual exclusion, hold and wait, no preemption of held resources, and a circular wait. Break any one by design and deadlock is impossible. Ordering all locks breaks the fourth; that is the discipline most kernels adopt, and it is a proof obligation on the programmer, not the machine.

Whether an arbitrary program *will* deadlock is a behaviour property: Rice. Whether the system *is* deadlocked now is a fact about the current state: a cycle in the waits-for graph, blocked contexts as nodes, held locks as edges. Linear time. The kernel checks the present because it cannot see the future.

THEN WHAT?

Detection tells you a cycle exists. Recovery is killing a process in it, which is the halting problem's answer once more: preempt what cannot be reasoned with. Or do what most systems do, and let the user notice. Detection is an approximation the kernel can afford; prevention is a proof it cannot make.

GÖDEL, ONCE MORE

A kernel cannot verify its own isolation. Neither can anything else that is expressive enough.

Week 6 said it for hypster: isolation comes from a ground the kernel does not control, hardware privilege or a smaller kernel beneath. Gödel's second theorem, run as a system. The trusted computing base is the axiom set, and the verified kernels of the world are proofs relative to it.

The same theorem applies to any system that is universal, including the ones that next week will run as processes under your kernel. A system cannot certify itself; the certificate has to come from outside, from something smaller, that somebody understands.

NEXT WEEK

Agents as processes. A generator produces programs; a kernel runs them under isolation and bounds. Sandboxing is virtualization, and the gate is everything this class built.