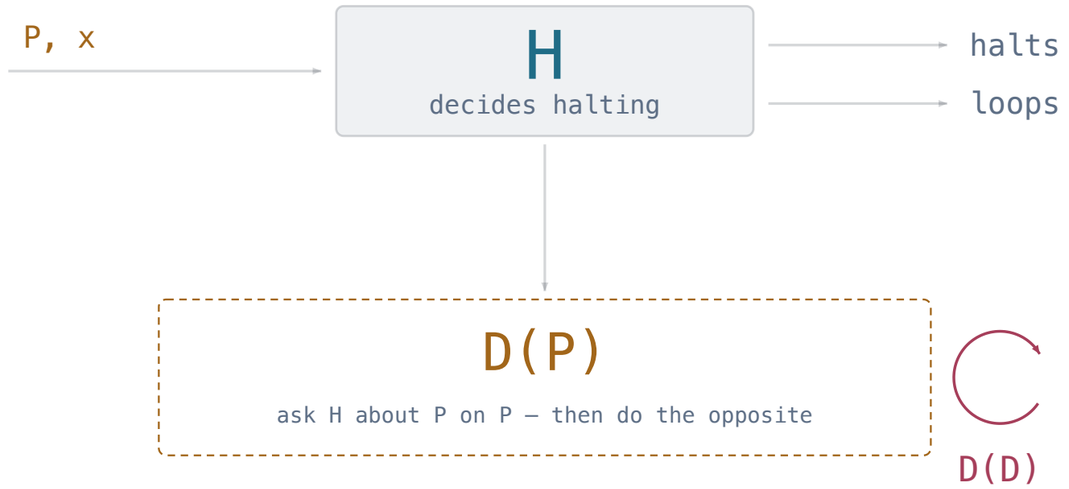
SYSTEMS ENGINEERING · WEEK 5 · STATION III

Time-Sharing

The timer interrupt is the kernel's answer to the halting problem. Scheduling is what it cannot answer.

THE THEOREM IN THE KERNEL

Will this program give the processor back? Undecidable. So take it back.



the halting construction, from the introduction class

Cooperative multitasking asks each program to yield. A program that never does takes the machine with it, and no kernel can screen programs for that in advance: Turing, 1936, on the scheduler.

Preemptive multitasking does not ask. A hardware timer raises an exception after a fixed number of cycles, whatever the program is doing; the kernel saves the context and runs another. The halting problem is not solved; it is made irrelevant by a bound.

Every kernel principle in this class has this shape: what cannot be decided is bounded. The timer is the purest instance.

IN SELFIE

A timeout is a number of instructions. Mipster counts down and returns.

```
// selfie.c, the shape of it
uint64_t* mipster(uint64_t* to_context, uint64_t timeout) {
restore_context(to_context);
run_until_exception(timeout); // TIMER is an exception
save_context(to_context);
return to_context; // whoever called decides
}
```

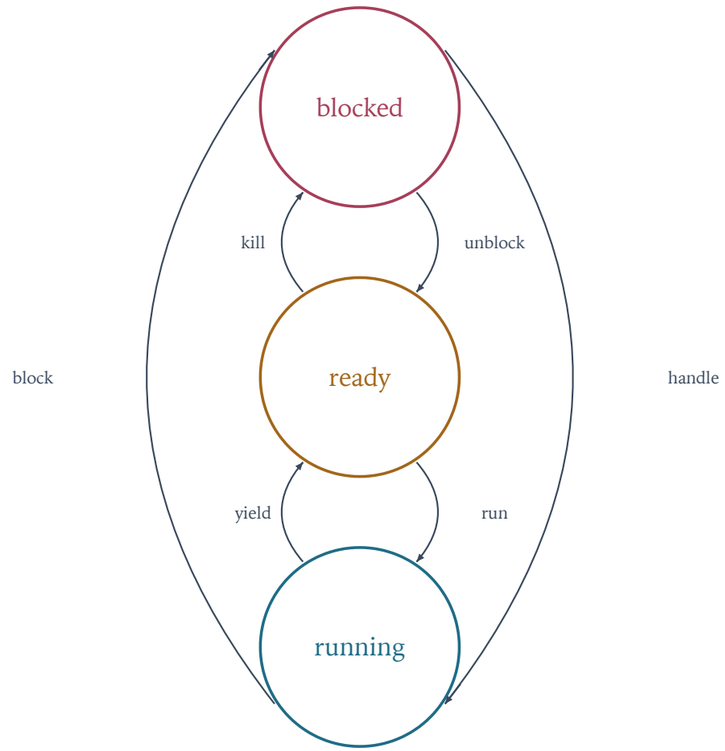
The emulator's loop from week 3 with one more exit: after `timeout` instructions it raises a timer exception and returns the context, exactly as it would on a system call or a page fault. The caller, the kernel, looks at why the machine stopped and decides.

The kernel in selfie is a loop around this call: pick a context, run it for a slice, handle what stopped it, pick again. That loop is the operating system by emulation. Every process, every thread you write this semester is an entry in its list.

```
$ make os-emu
./selfie: 166 syscalls, 529 page faults, 1898 timer interrupts
```

PROCESS STATES

Running, ready, blocked. A machine context is always in exactly one.



the three states and the events that move a context between them

Running: its registers are in the processor. Ready: saved, waiting for a slice. Blocked: saved, waiting for something else — input, a child, a lock. The timer moves running to ready; a system call that must wait moves running to blocked; the awaited event moves blocked to ready.

A finite state machine, per context, from the notation chapter. The kernel's whole knowledge of a program is which of three states it is in and a few kilobytes of saved registers. It knows nothing about what the program is doing, by Rice, and needs to know nothing.

SCHEDULING

Which ready context runs next? Fair is easy. Optimal is NP-hard.

ROUND ROBIN

A queue; the timer moves the running context to the back. Every context gets a slice every *n* slices. Fair, simple, and what selfie does. It optimises nothing.

PRIORITIES, DEADLINES

Real schedulers weigh interactivity, deadlines, fairness across users, energy. Each is a policy; each can be gamed; each is a bound on something that cannot be decided.

THE LIMIT

Given jobs with lengths and deadlines and several processors, finding a schedule that meets every deadline is NP-hard: the Cost chapter's station, in the kernel. Even if the kernel knew every job's length, which by Rice it cannot, it could not afford to schedule them optimally.

Two limits, one design: the scheduler does not know and could not compute. So it is a policy, chosen by people, adjusted by measurement. Week 11 measures.

ASSIGNMENT

`processes`: run several instances of the same program, time-shared.

1. Extend selfie so that `-m` can run *n* instances of the loaded program as separate machine contexts with separate memory, round-robin scheduled by the kernel loop with a timeout you choose.
2. Each instance prints something that identifies it, so that the interleaving is visible on the terminal. Make the slice small enough that the interleaving shows.
3. Check that no instance can see another's memory, and that the total instruction count is the sum of the instances'.
4. `./grader/self.py processes`.

REUSE

`create_context`, `load`, the kernel loop, and mipster with a timeout are all there. The assignment is to arrange them into a list of contexts and a scheduler. Look at how the existing code handles a context that exits, and keep the case of one instance behaving exactly as before.