System Programming: an introduction

Raymond Namyst

Dept. of Computer Science

University of Bordeaux, France

https://gforgeron.gitlab.io/progsys/

Goals

- Understand how to use the Operating System API efficiently
  - Get insights about how Operating Systems work
- In-depth cover of the following topics
  - File operations
  - Process management
  - Pipes
  - Signals

+ Introduction to parallel programming

Organization

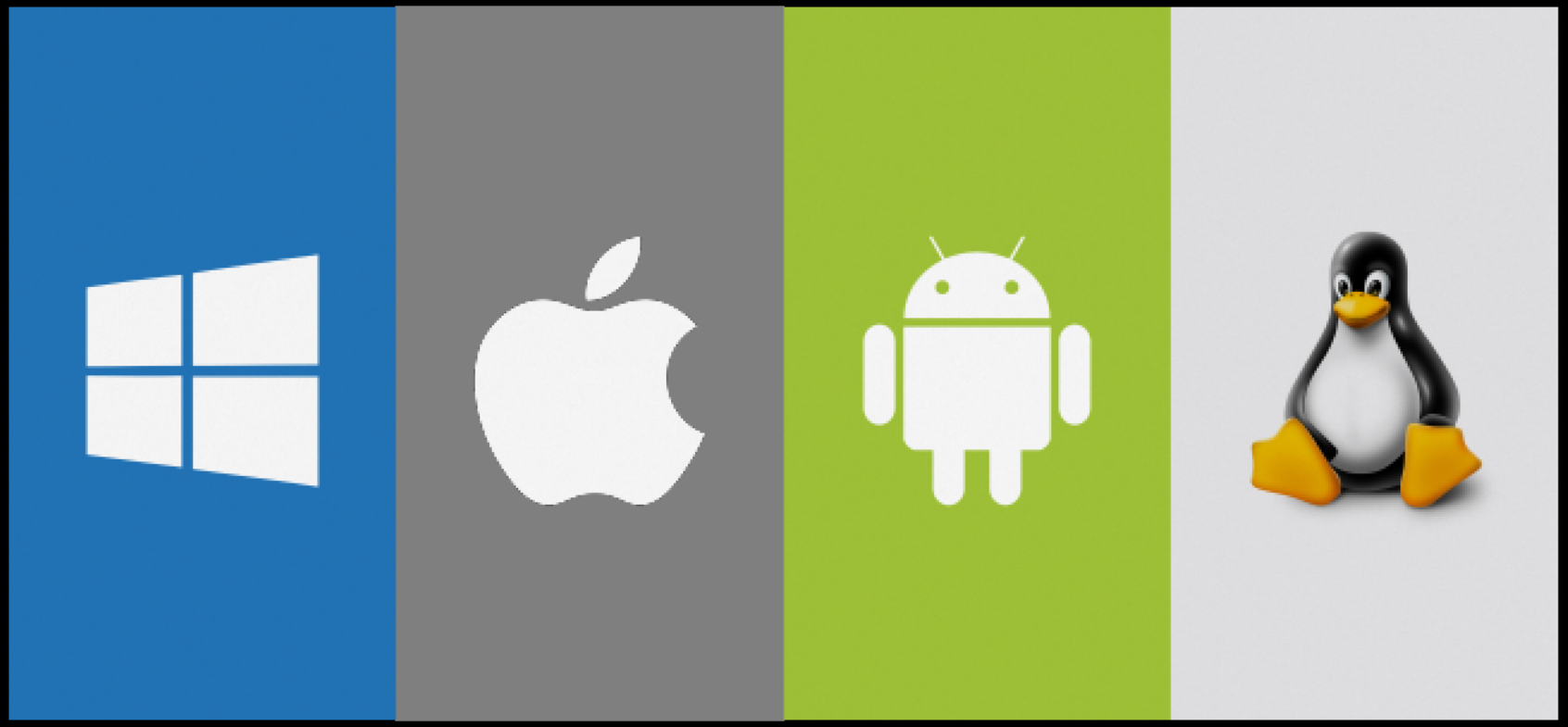
- System Programming strongly relies on practice work
  - 1h20 lecture a week
  - 2h40 lab session week
- Evaluation
  - One mini-project + periodic Moodle polls (Moodle)
  - Two mid-course tests (DS1, DS2)
  - Final grade = 30% DS1 + 40% DS2 + 30% Moodle

Bibliography

- UNIX: Programmation et Communication
  J.M. Rifflet, J.B. Yunes
  Dunod

- Upcoming lecture slides
  - https://gforgeron.gitlab.io/progsys/cours/

What is an Operating System?



What is an Operating System?

- A set of cool applications?
- A window manager providing a consistent “Look and Feel” experience across applications?
- A set of device drivers to abstract hardware capabilities?
- Someone, dressed all black, who is watching you…

What's the purpose of an Operating System?

- Do I need one?
  - Well, not every personal computer does have one… But most of them do!
- Why do we use Operating Systems?
  - Hardware abstraction and code factorization
    - Device drivers: better portability and programmability
  - High-level abstractions
    - Files, Windows (Graphical Interface)
  - Resource virtualization
    - Memory, CPU, disk: seamlessly shared by applications and users
    - A faulty process causes no damage to others, neither to the "system"

An OS is a kind of *abstract machine*

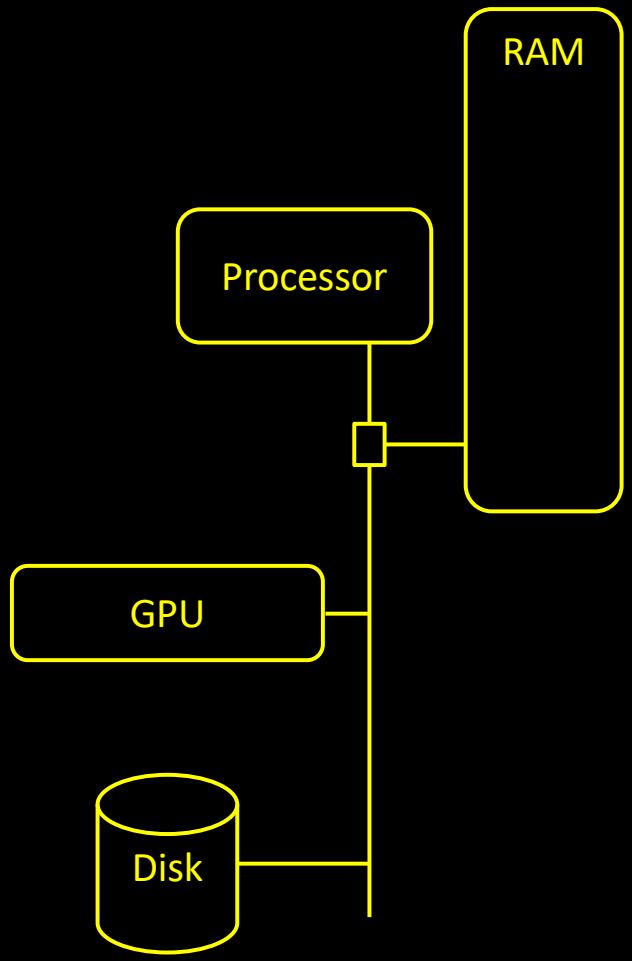
- It is composed of several important parts
  1. A set of device drivers (= code)
  2. A set of programs (= code)
     - Some of these programs are running in the form of background processes
       - So-called daemons: inetd, cupsd, sshd, syslogd, etc.
     - Some others are executed on demand
       - Internet navigator
       - File explorer
       - Email client
       - Etc.
  3. A set of libraries (= code)
  4. A mysterious authority which rules the world

Dr Jekyll and Mr Hyde

- Operating Systems provide us with great high-level features
  - Graphical Interfaces
  - Multi-tasking

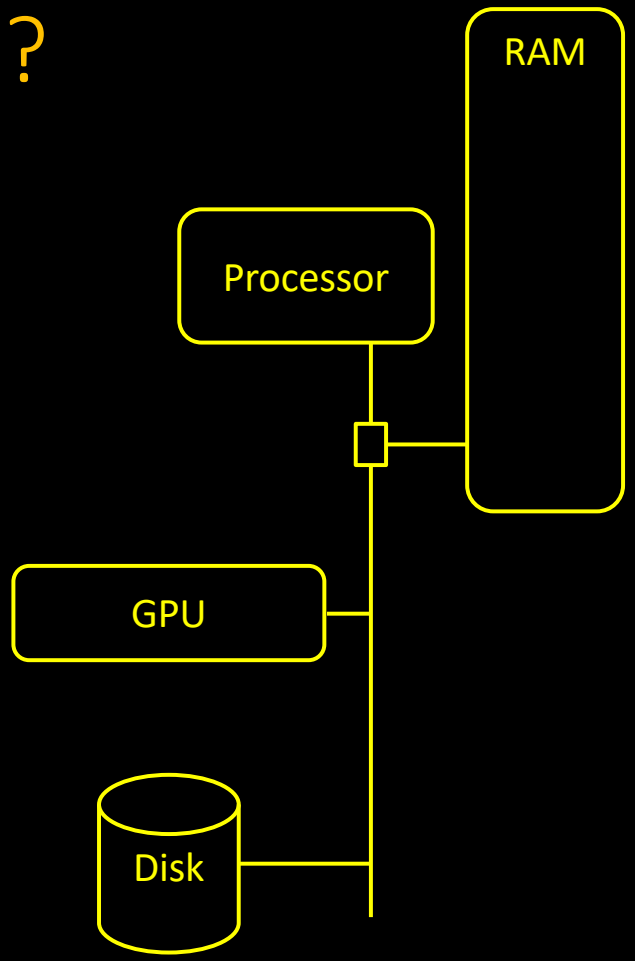
- To do so, they stand in between applications and the hardware
  - On good old single-user Operating Systems (e.g. MS DOS)
    - Programs were executed one at a time… and could enjoy direct access to the hardware
    - They could corrupt the OS memory, freeze the machine, etc.
    - Great times!
  - On nowadays' systems
    - The OS hinders direct access to the hardware
      - How can that be?

Typical Computer Architecture



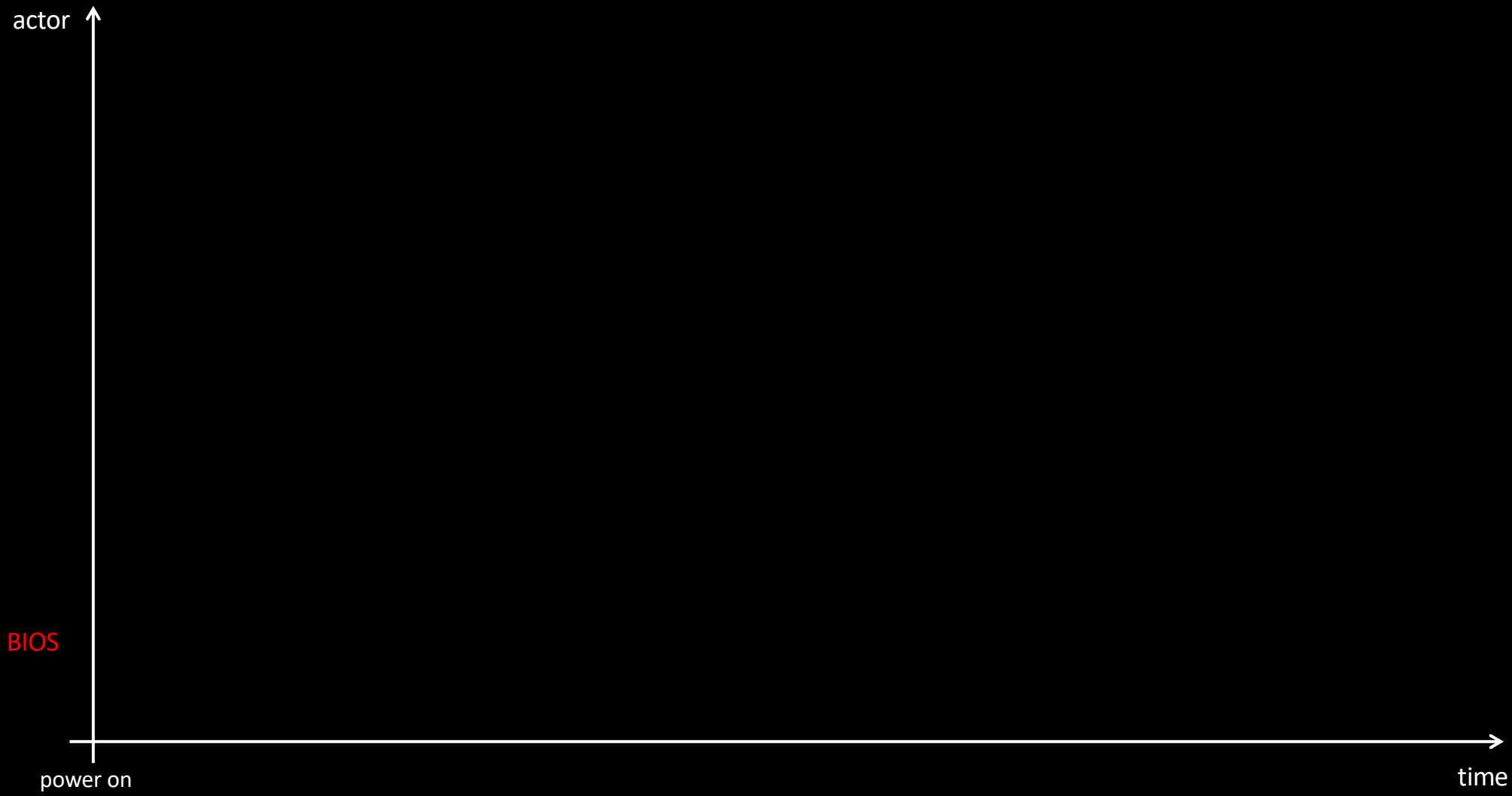
Where is the first instruction?

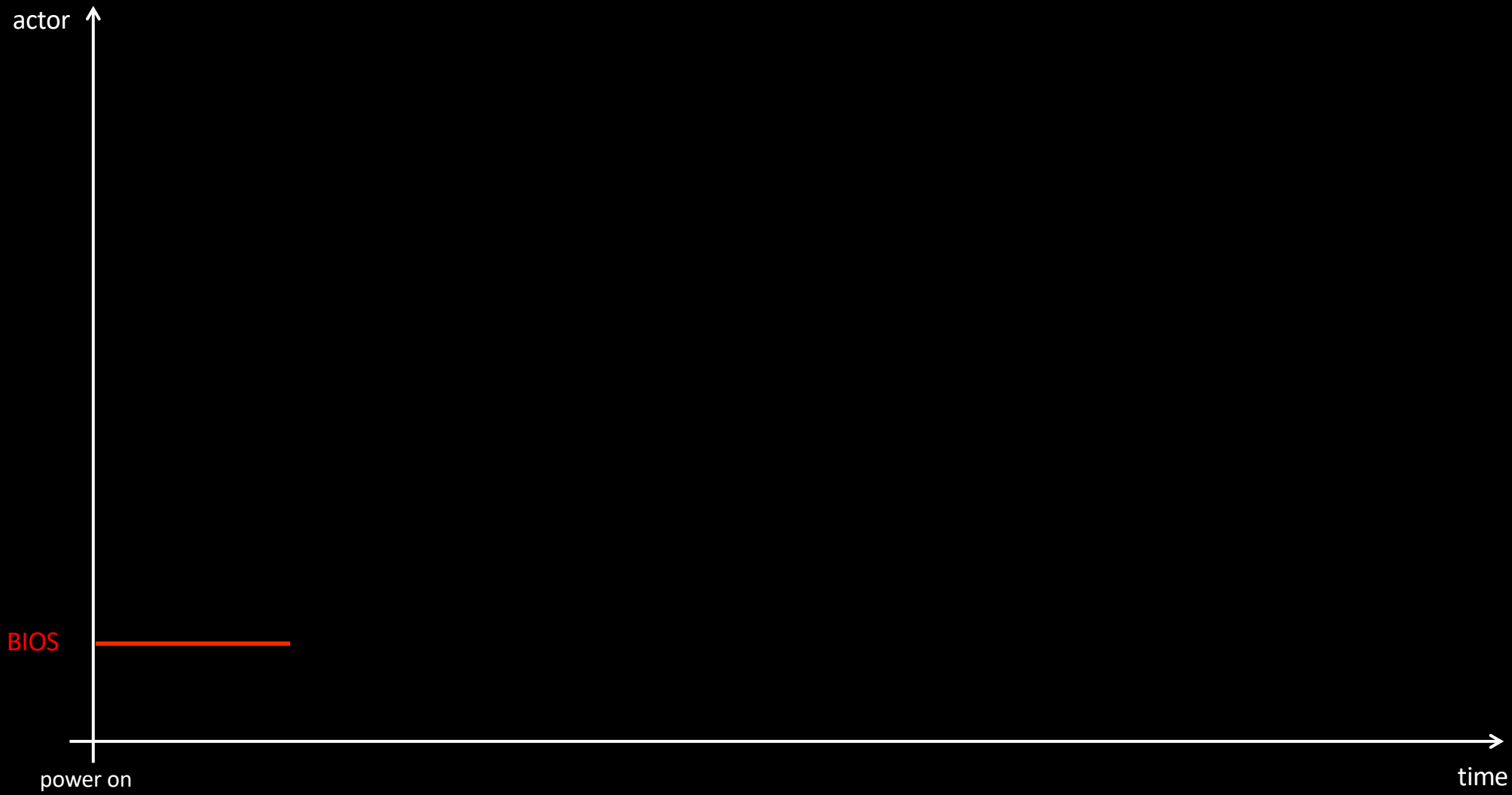
1. The CPU needs instructions!
2. The RAM is empty
3. OS bootstrap is probably on disk
4. To fetch theses instructions to RAM…
5. …CPU needs instructions!
   goto 1



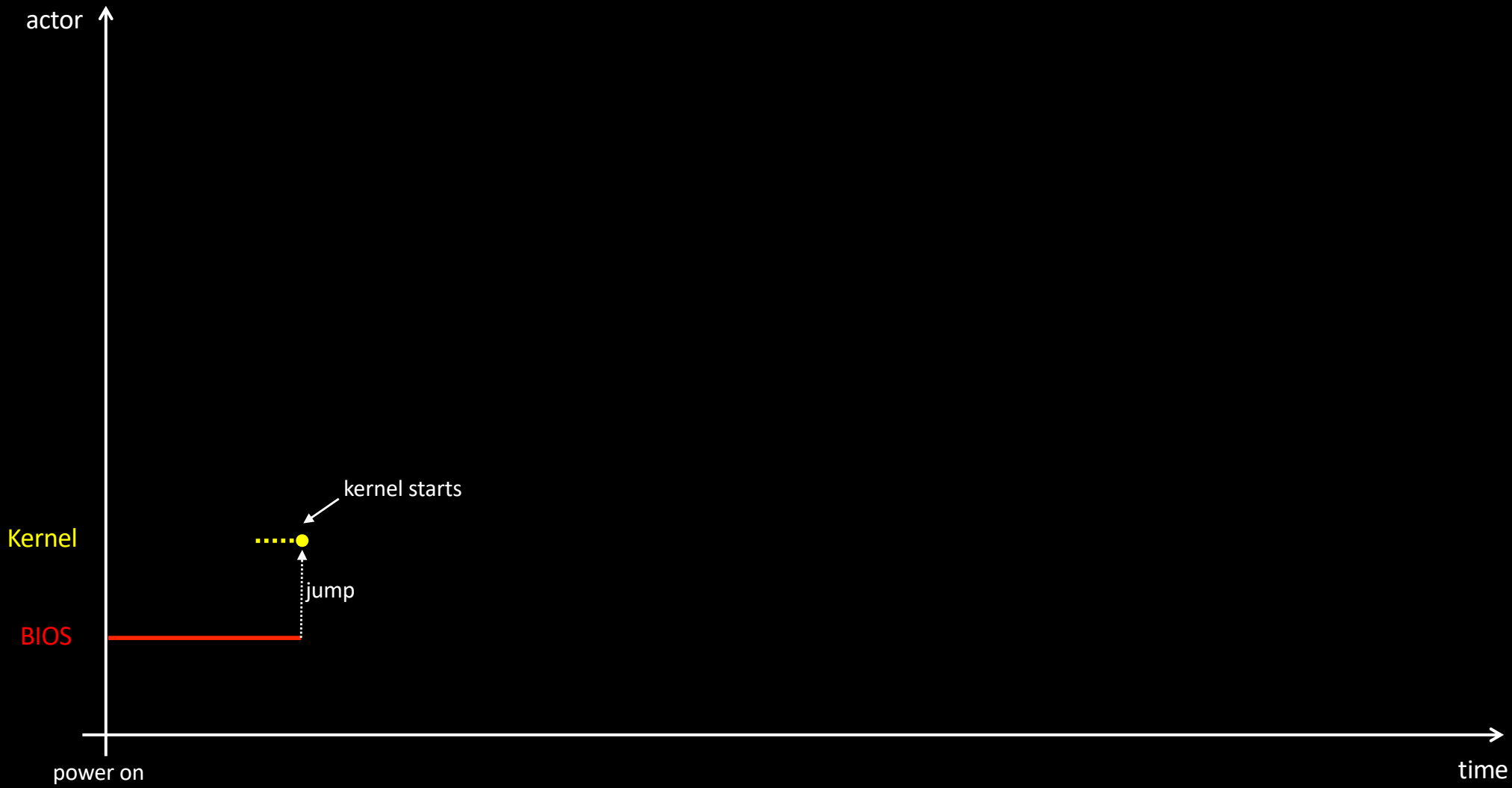
The BIOS (aka ROM BIOS or System BIOS)

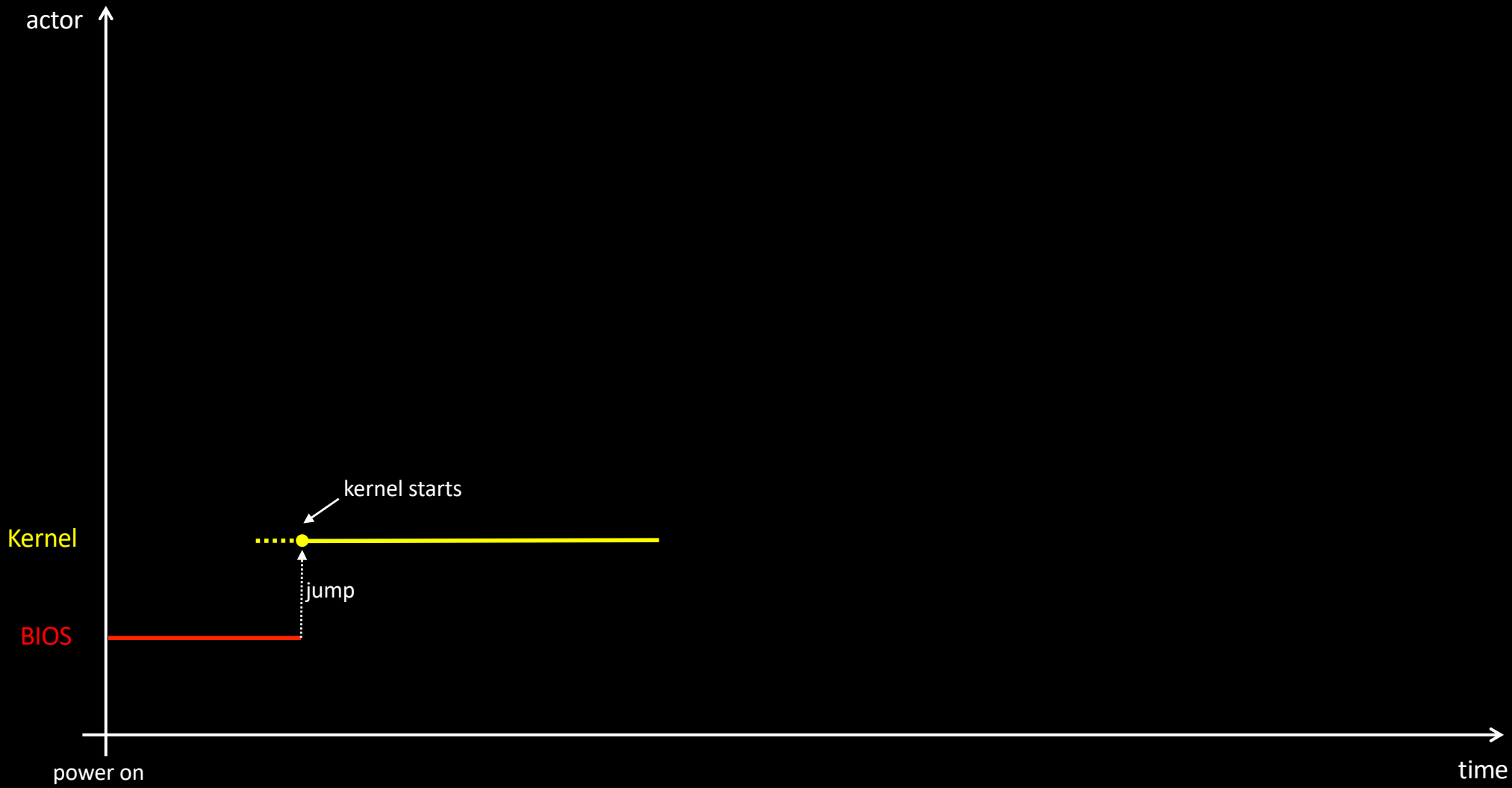
- Firmware stored in ROM chip / flashable memory
  - Contains the very first instructions executed by the processor
    - No BIOS = No Boot
- The BIOS is responsible for
  - Hardware discovery and initialization
    - CPUs, memory, I/O controllers, devices, etc.
  - Hardware configuration
  - OS boot
- In the PC World, legacy BIOS has been replaced by the more powerful UEFI
  - Unified Extensible Firmware Interface (2005)
  - But we still call it BIOS ☺

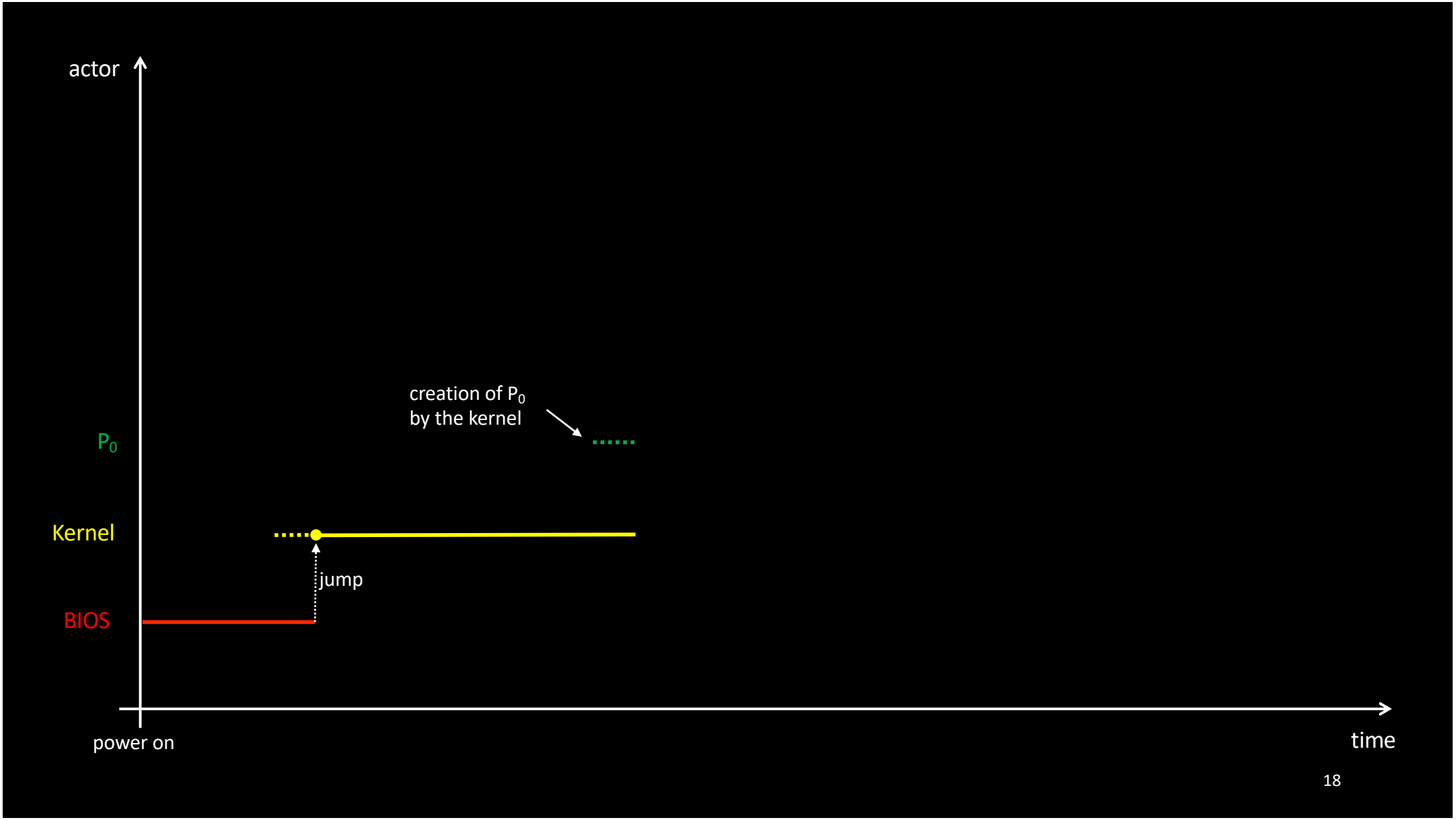

actor
BIOS
power on
time

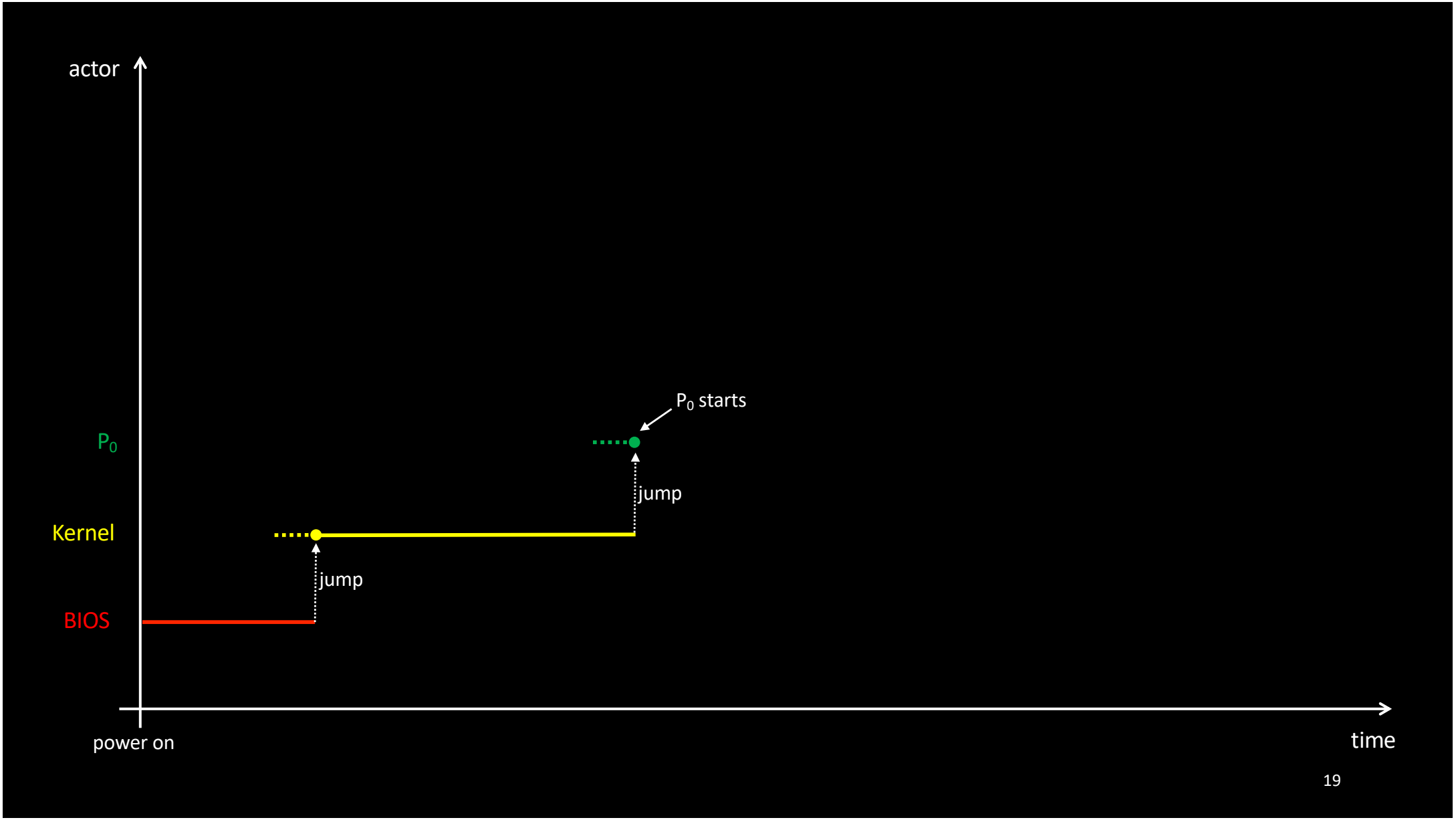

actor
BIOS
power on
time

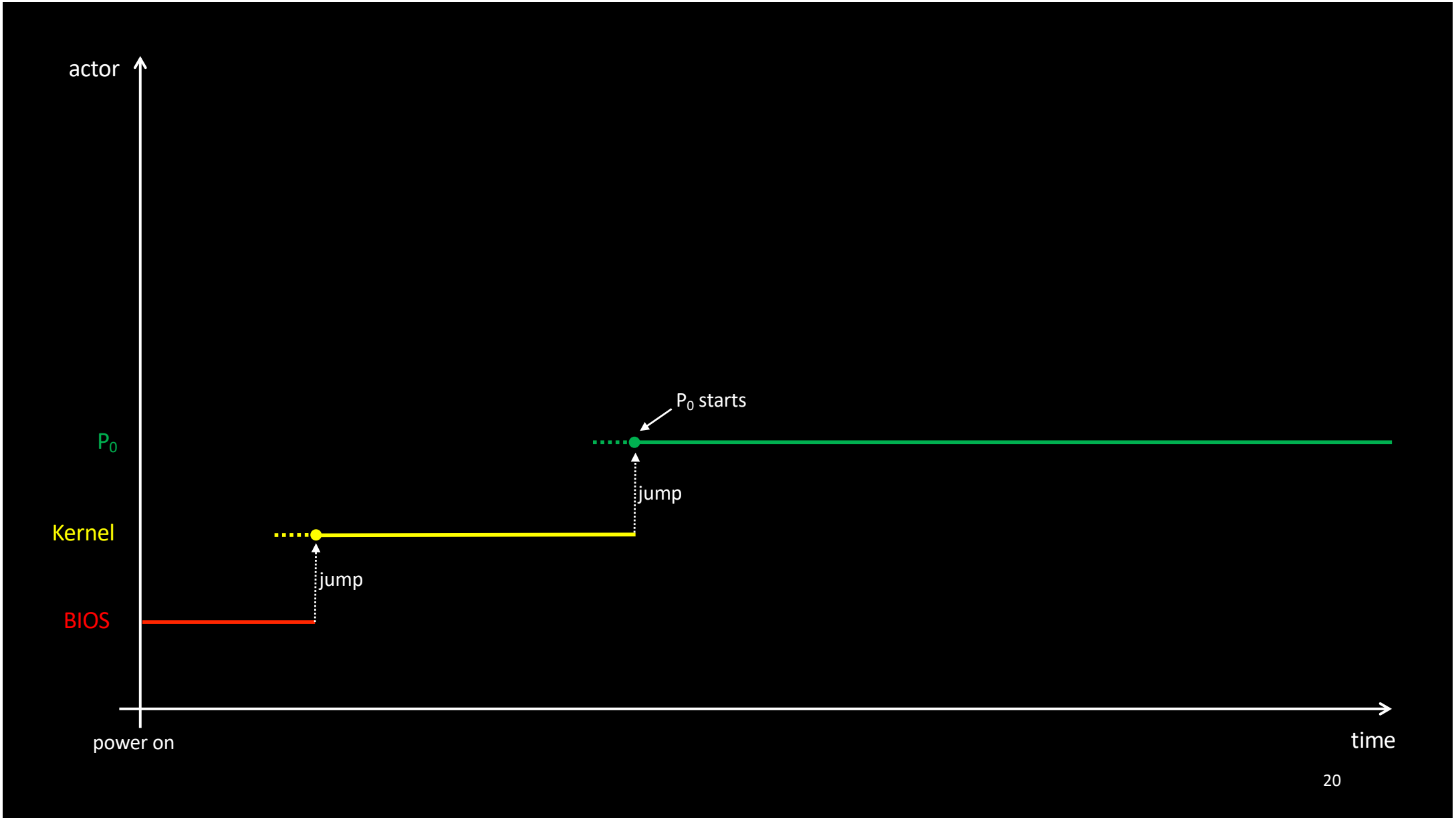

actor
kernel is loaded
from disk
Kernel
BIOS
power on
time

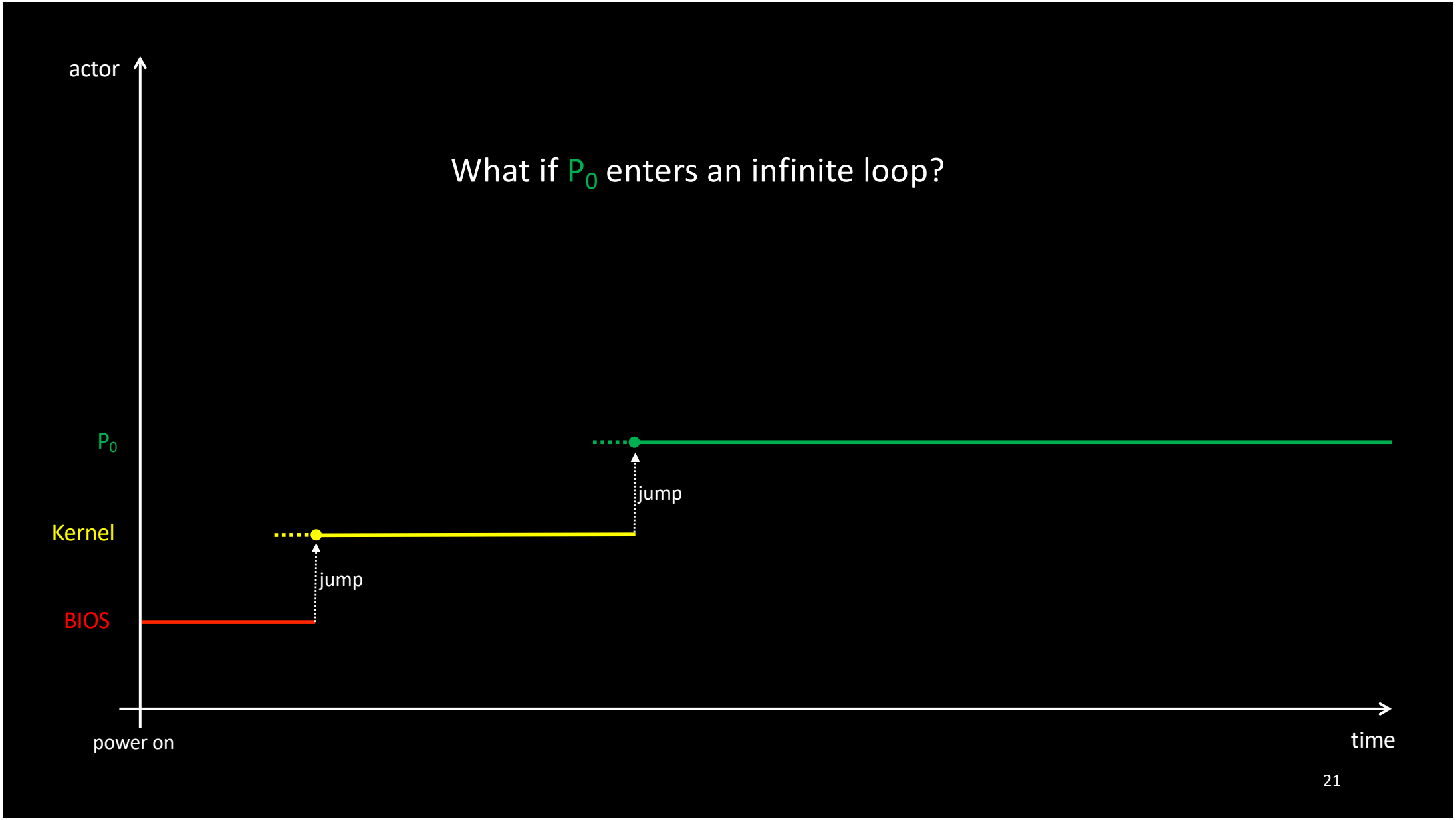

actor
kernel starts
Kernel
jump
BIOS
power on
time


actor
kernel starts
Kernel
jump
BIOS
power on
time


actor
creation of P_0
by the kernel
P_0
Kernel
jump
BIOS
power on
time


actor
P_0
Kernel
BIOS
power on
time
P_0 starts
jump
jump


actor
P_0
Kernel
BIOS
power on
time
P_0 starts
jump
jump


What if P_0 enters an infinite loop?
actor
P_0
Kernel
BIOS
jump
jump
power on
time

Interrupts

- An interrupt is a (rude) signal sent to a CPU
  - Can be sent by external hardware
    - Keyboard, mouse, timers, etc.
  - Or raised by the CPU itself
- No information attached, except interrupt number
- Most of the time, the CPU is forced to handle interrupts with no delay
  - Jump to a predefined routine address (interrupt handler)
  - Each interrupt can have its own interrupt handler
    - An interrupt vector table must be setup in RAM (one entry per interrupt)
    - Done by the kernel!
  - The interrupt handler calls “iret” to resume previous execution

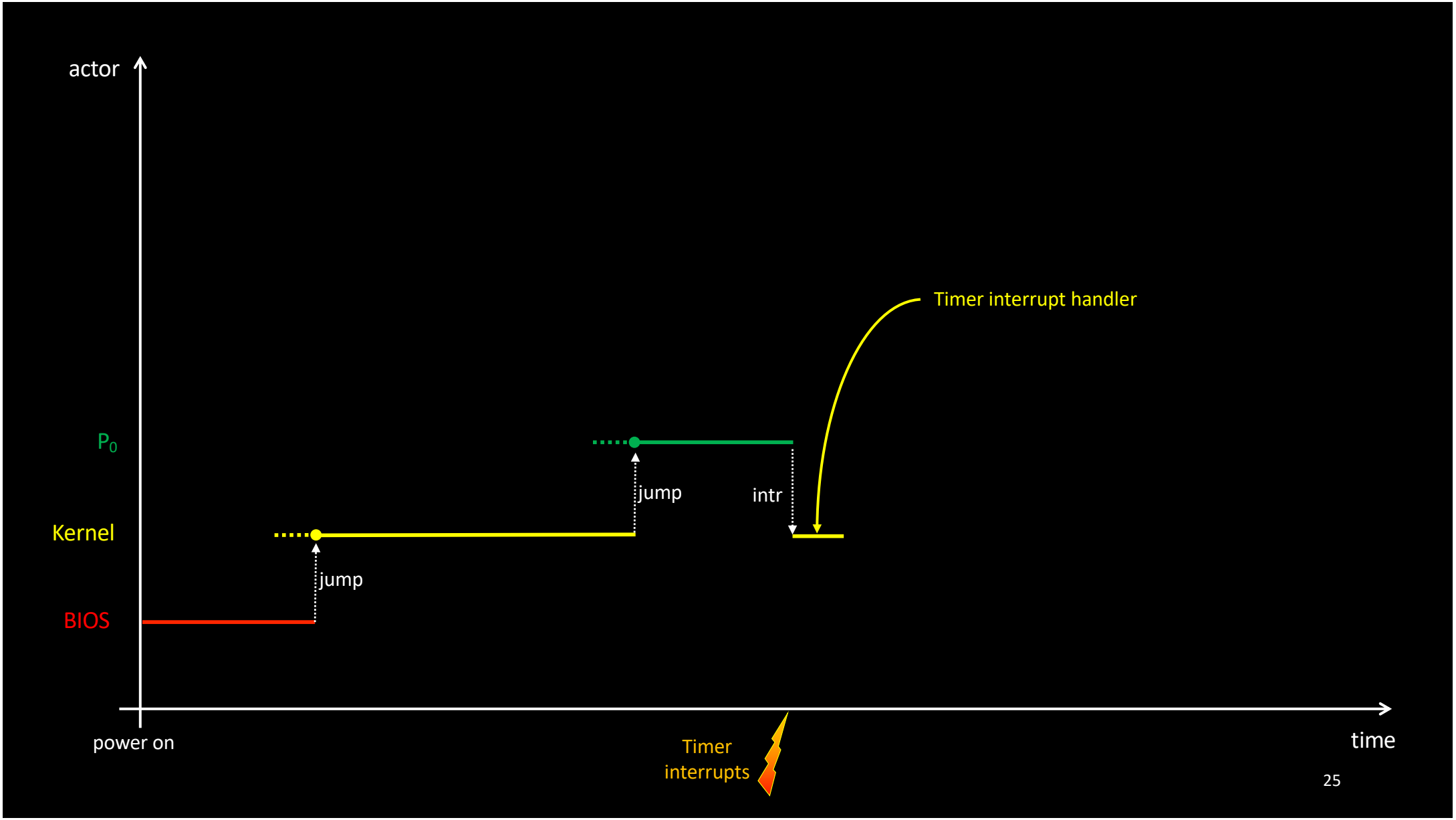
Interrupts

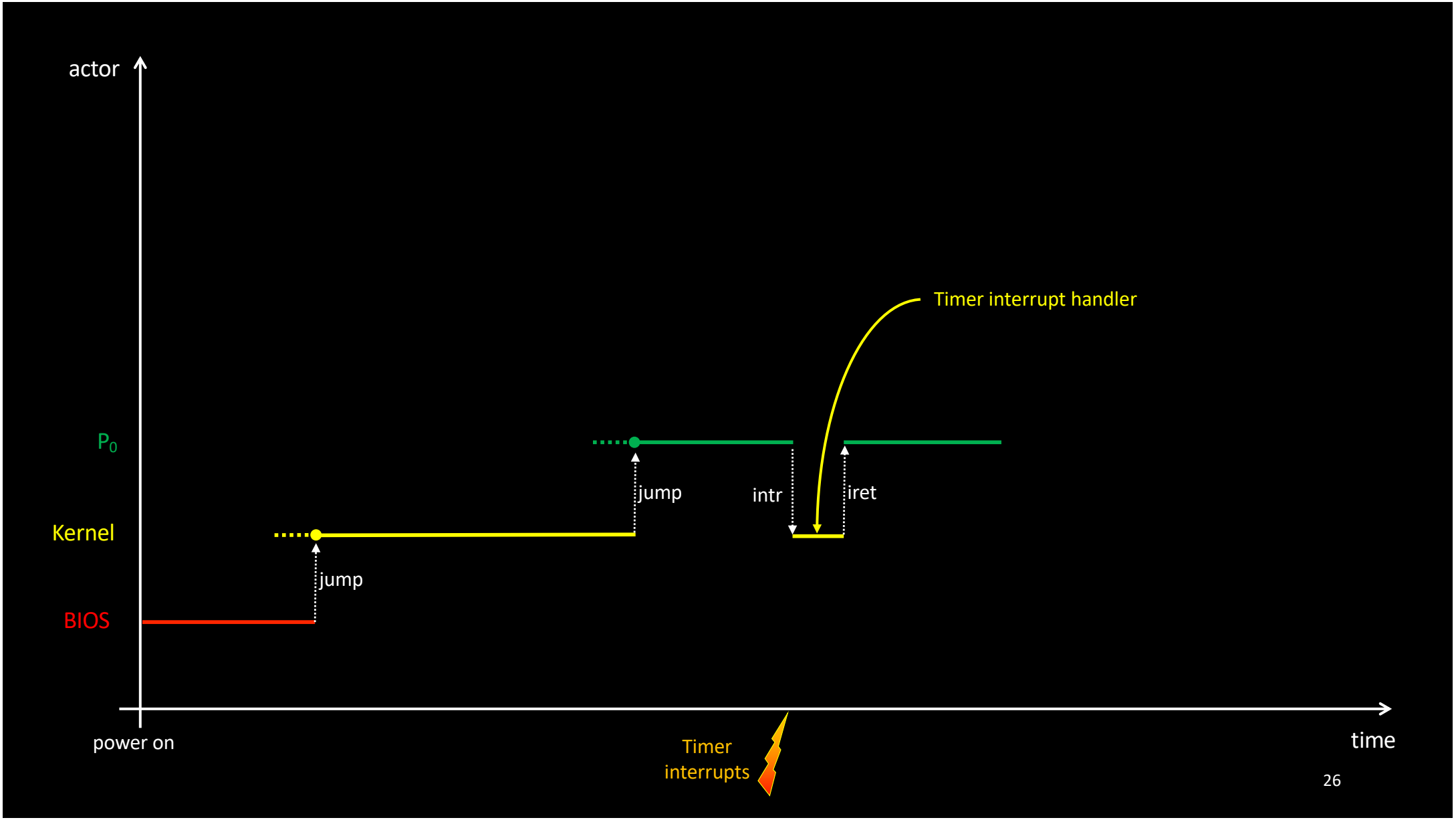
- Moving the Mouse generates interrupts
  - Don’t move your mouse erratically when playing a demanding 3D game!
- Pressing (and releasing) the shift key on the keyboard generates 2 interrupts
- The Network Interface Card (NIC) generates an interrupt each time a packet is received
- Etc.
  - Try `xosview` under Linux

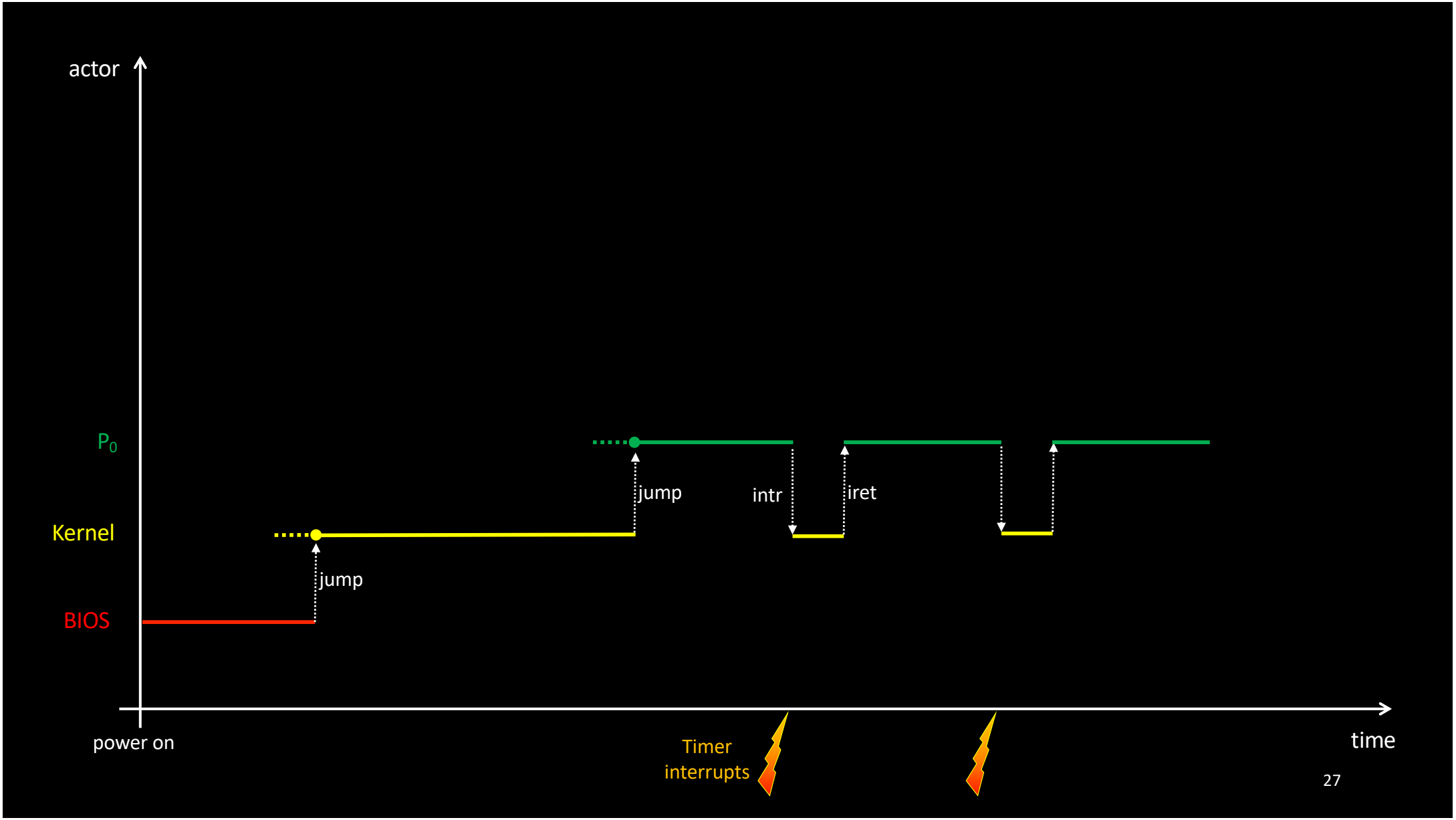
Implementing Time Sharing

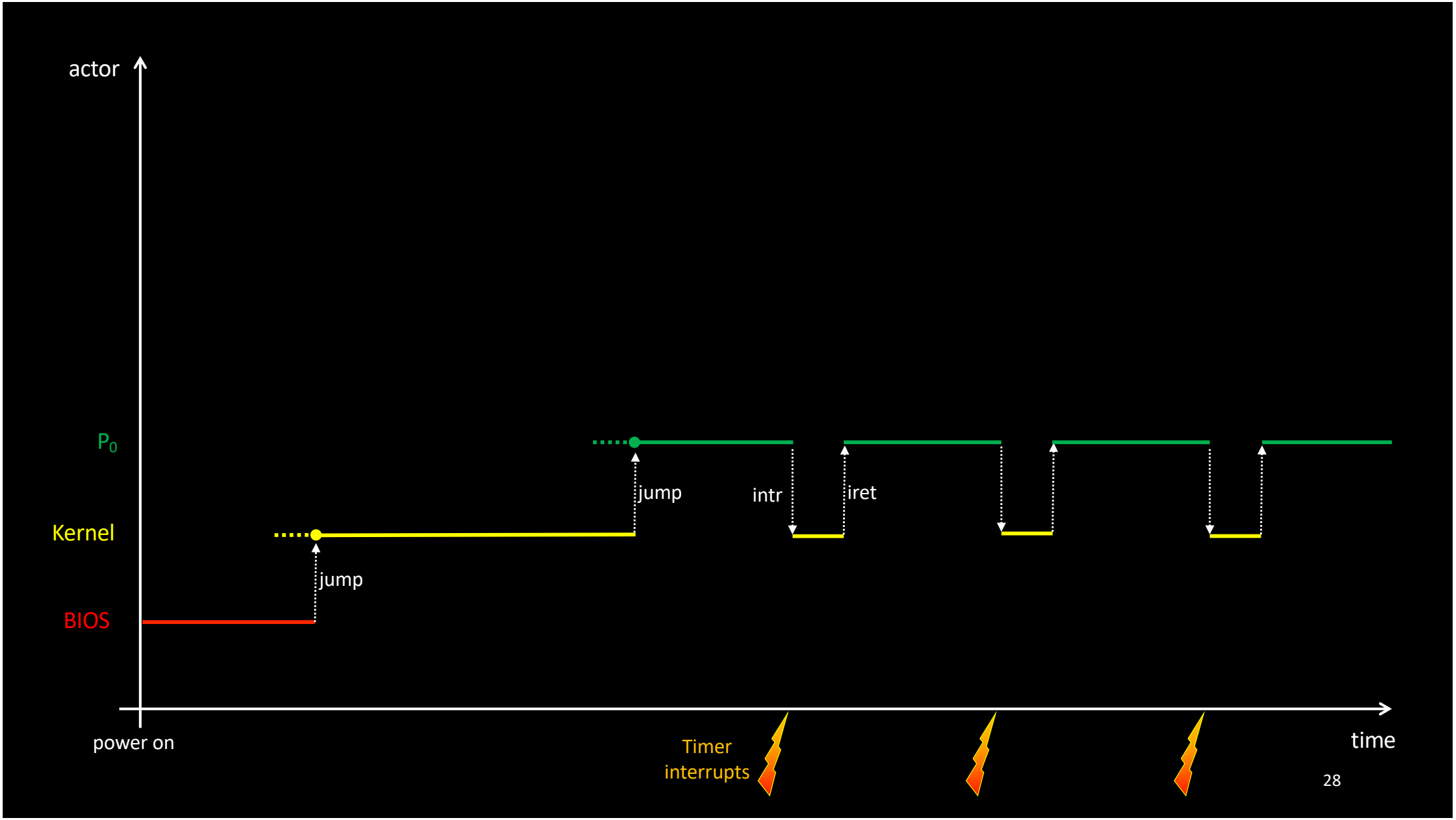
- To prevent processes running during unbounded periods, the kernel sets up a timer
  - A timer interrupt will be periodically triggered (~ 10ms)
  - This ensures that the associated kernel routine will be executed on a regular basis

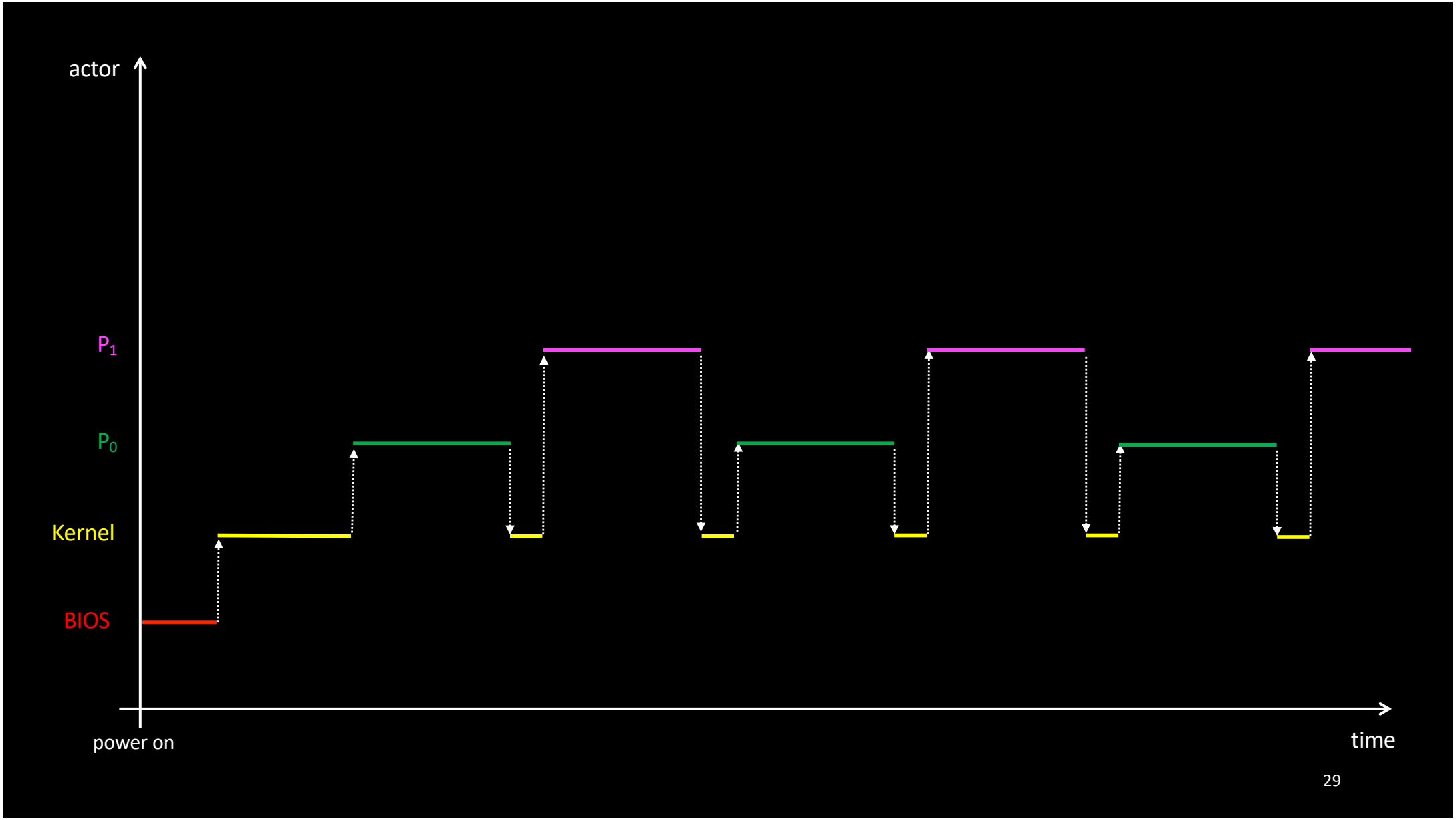
  - Of course, the *Interrupt Vector Table* must be initialized beforehand!


actor
P_0
Kernel
BIOS
jump
jump
intr
Timer interrupt handler
power on
Timer
interrupts
time


actor
Timer interrupt handler
P0
jump
intr
iret
Kernel
jump
BIOS
power on
Timer
interrupts
time


actor
P0
Kernel
BIOS
jump
jump
intr
iret
power on
Timer
interrupts
time


actor
P0
Kernel
BIOS
jump
jump
intr
iret
power on
Timer
interrupts
time


actor
P_1
P_0
Kernel
BIOS
power on
time