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
  - Communication (pipes)
  - Signals

+ Introduction to parallel programming

Organization

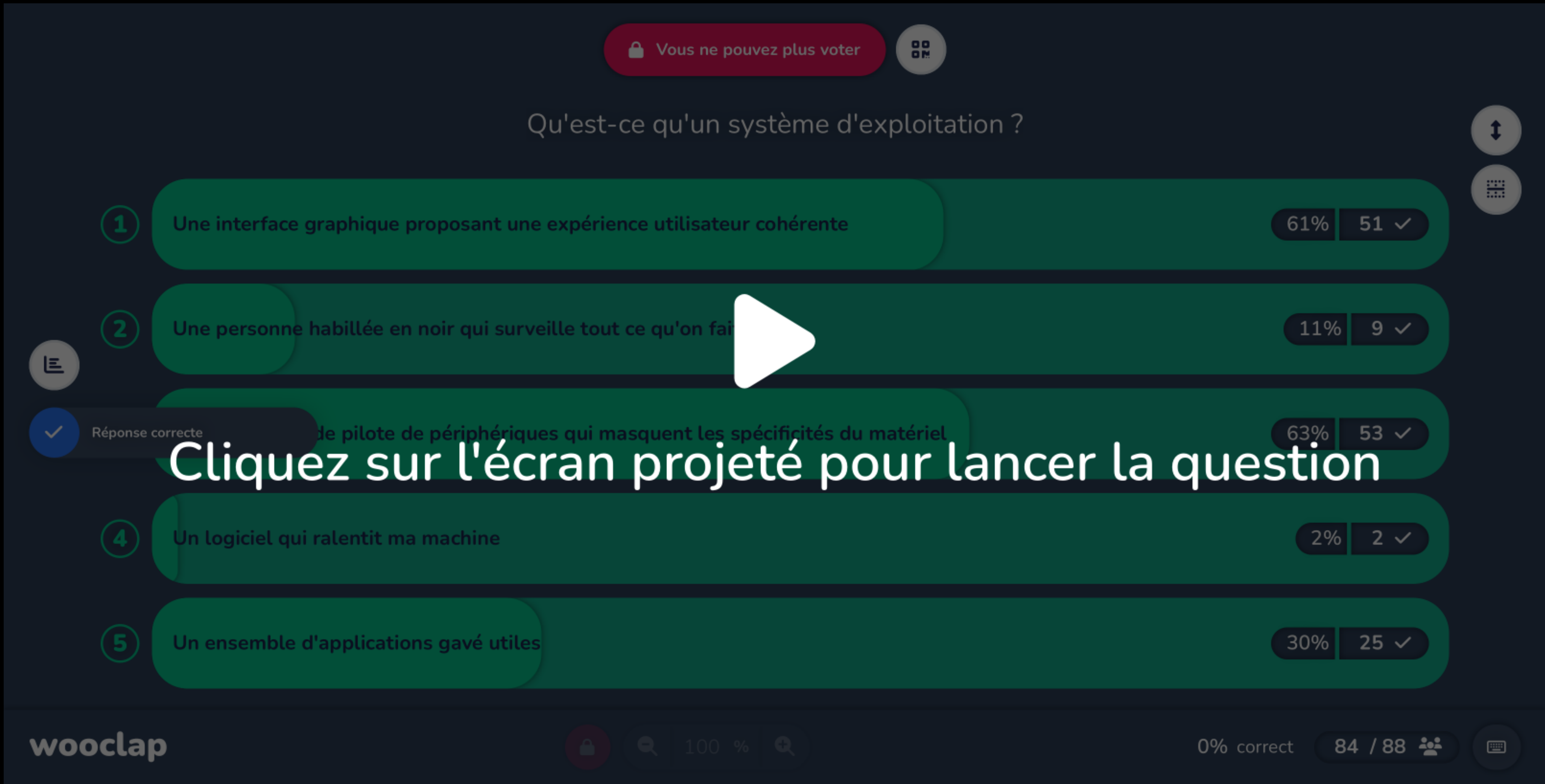
- System Programming strongly relies on practice work
  - 1h20 lecture a week
  - 2h40 lab session week
- Evaluation
  - Two mid-course tests (DS1, DS2)
  - One mini-project + periodic Moodle polls (Moodle)
  - Final grade = 30% DS1 + 40% DS2 + 30% Moodle

Bibliography

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

- Upcoming lecture slides (+ source code of examples)
  - https://gforgeron.gitlab.io/progsys/cours/

What is an Operating System?



Vous ne pouvez plus voter
Qu'est-ce qu'un système d'exploitation ?
1 Une interface graphique proposant une expérience utilisateur cohérente 61% 51
2 Une personne habillée en noir qui surveille tout ce qu'on fai 11% 9
Réponse correcte
de pilote de périphériques qui masquent les spécificités du matériel 63% 53
Cliquez sur l'écran projeté pour lancer la question
4 Un logiciel qui ralentit ma machine 2% 2
5 Un ensemble d'applications gavé utiles 30% 25
wooclap
100 %
0% correct
84 / 88

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 that rules the world

Dr Jekyll and Mr Hyde

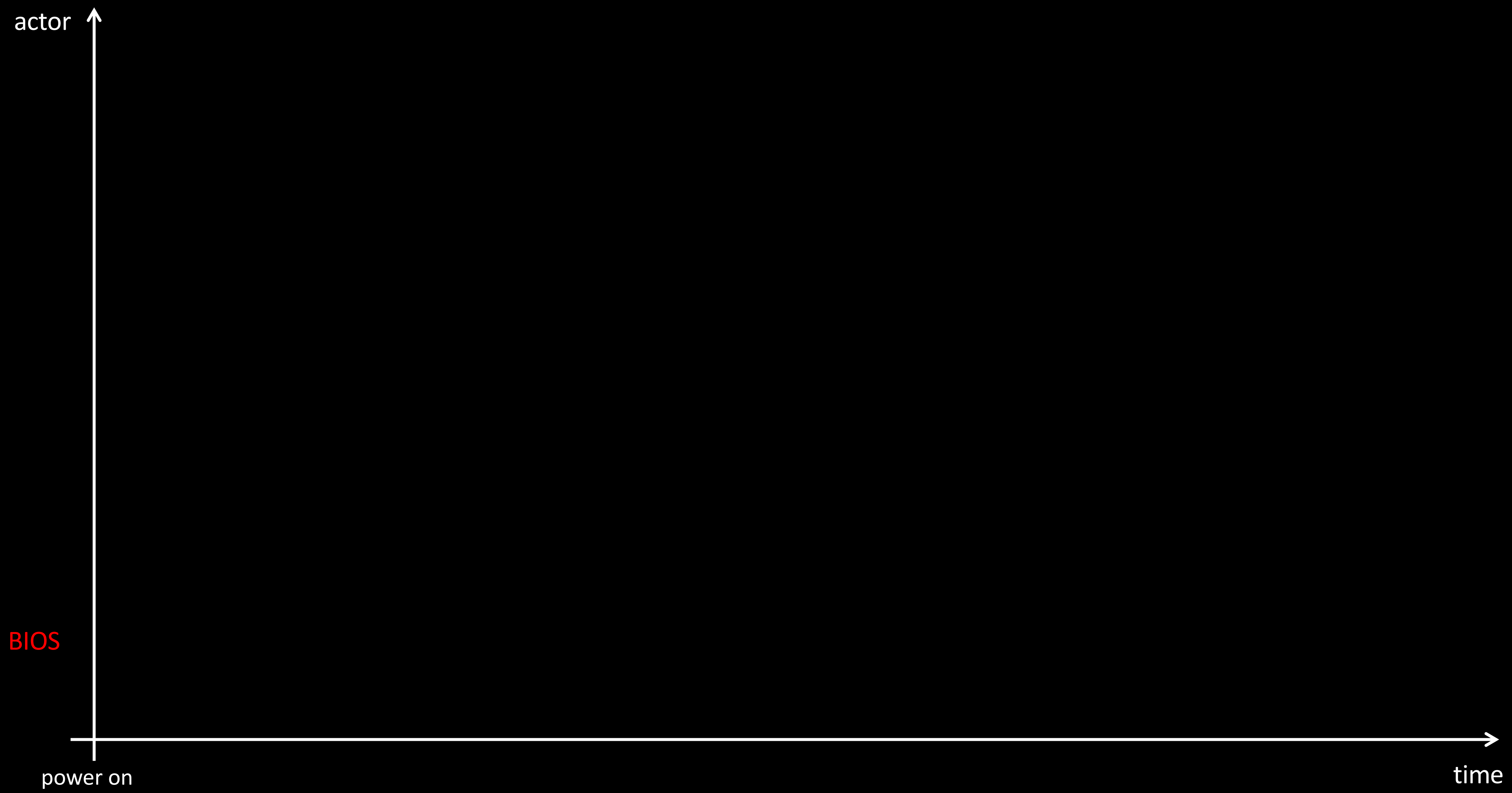
- Operating Systems provide us with great high-level features
  - Graphical Interfaces
  - Multi-tasking

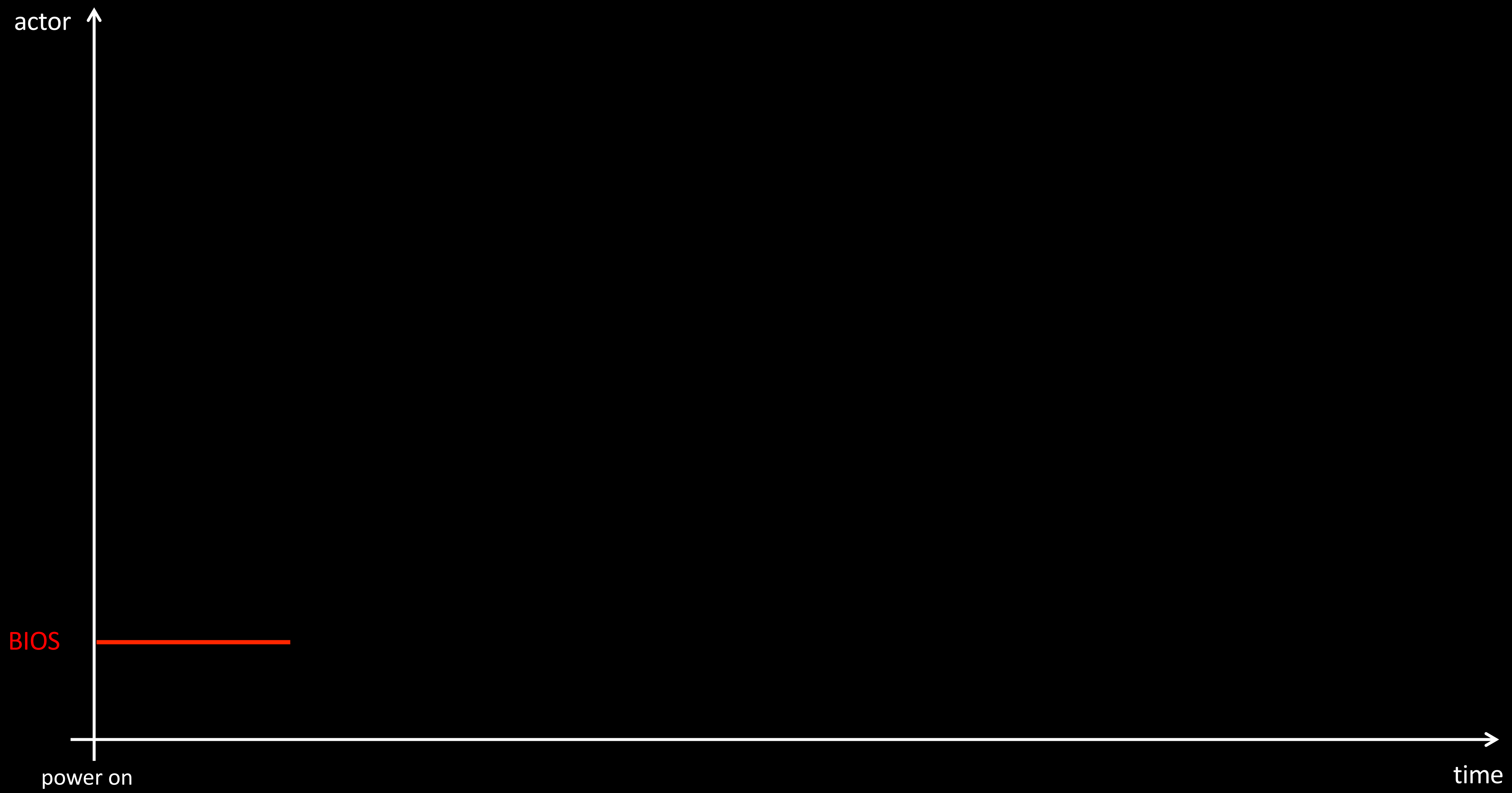
- To do so, they stand in between applications and the hardware
  - On good old single-user Operating Systems (e.g. MS DOS)
    - Programs were executed one at a time… and could enjoy direct access to the hardware
    - They could corrupt the OS memory, freeze the machine, etc.
    - Great times!

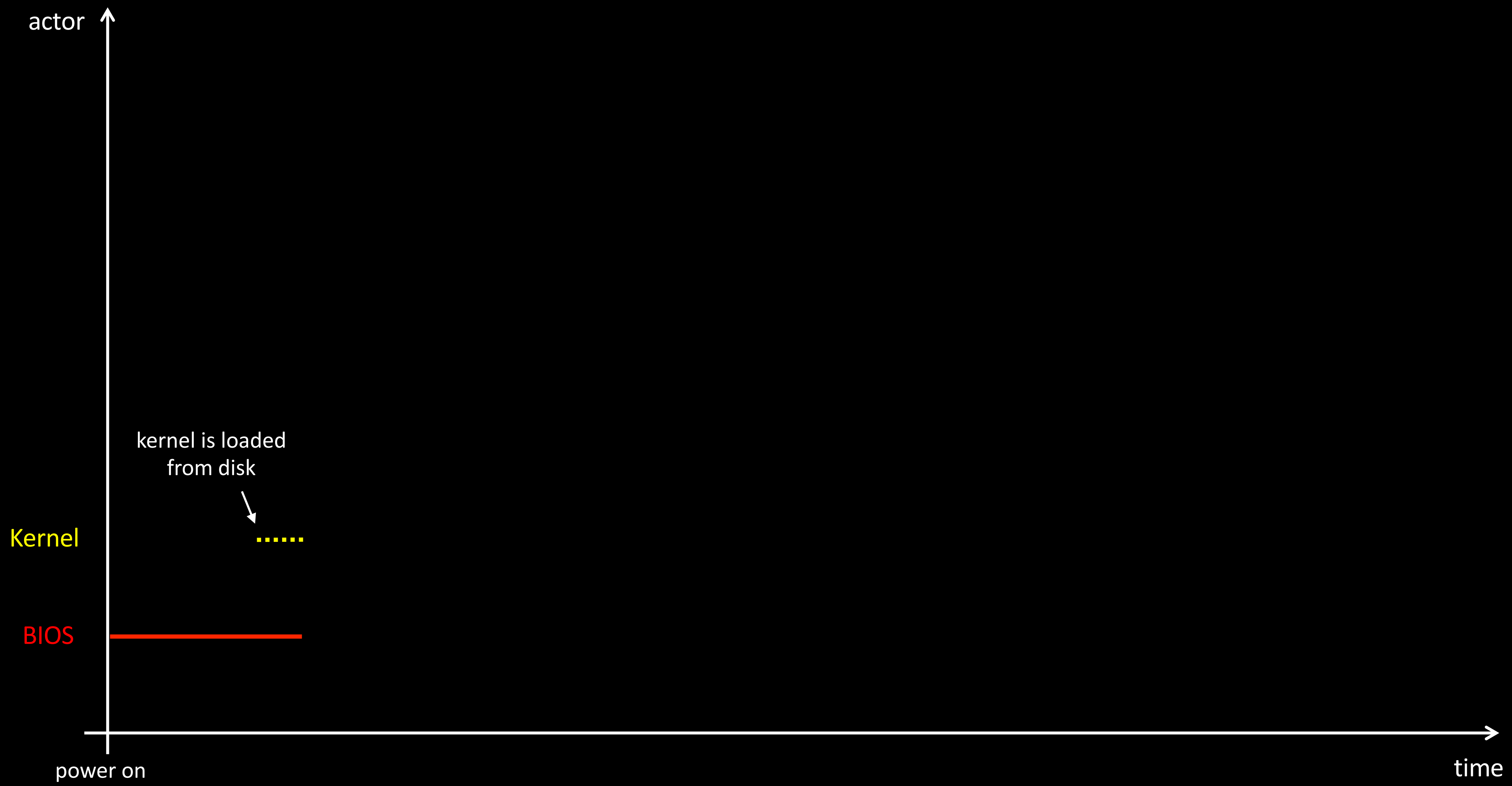
  - On nowadays' systems
    - The OS hinders direct access to the hardware
      - How can that be?

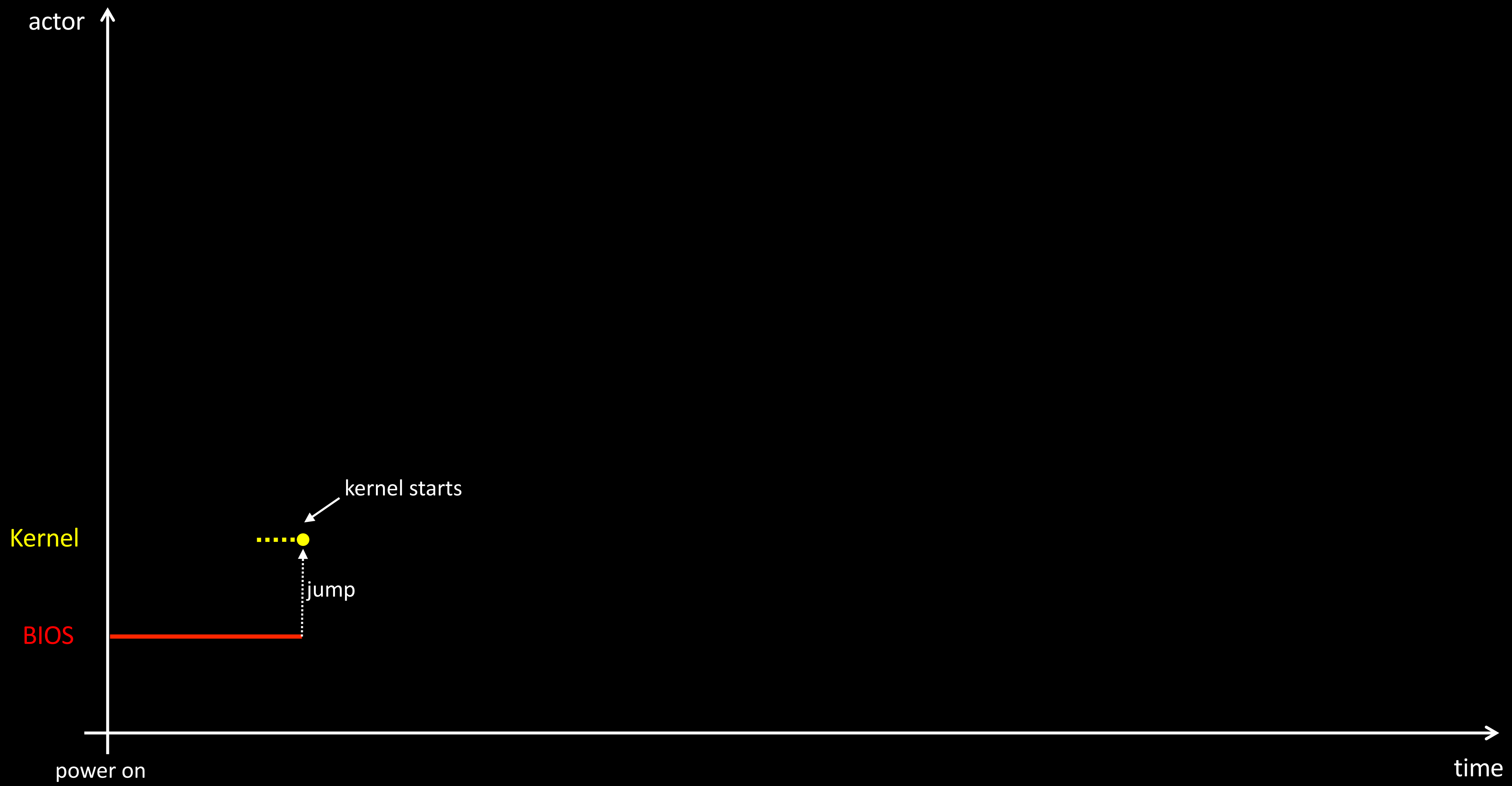
The BIOS (aka ROM BIOS or System BIOS)

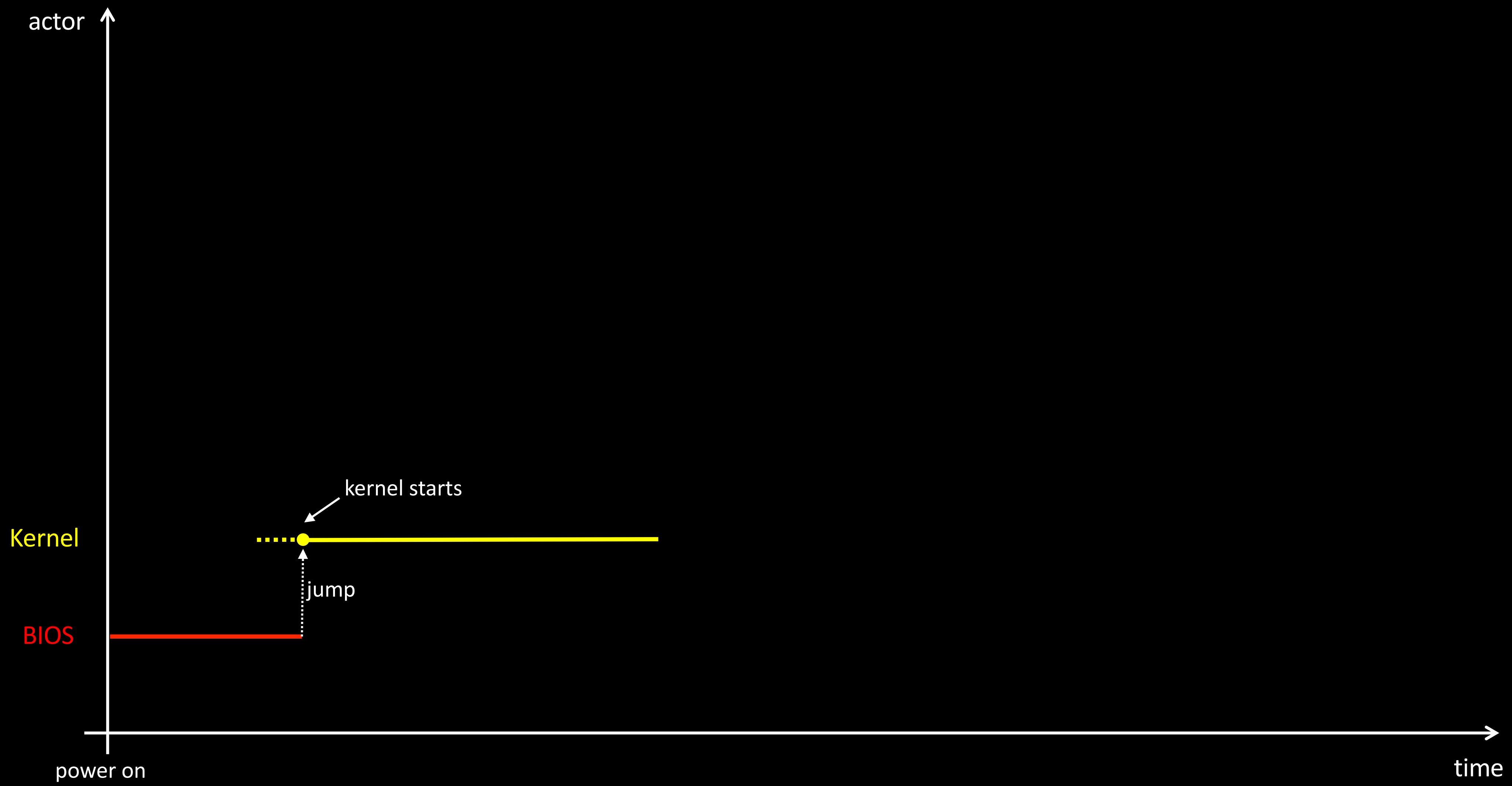
- Firmware stored in ROM chip / flashable memory
  - Contains the very first instructions executed by the processor
    - No BIOS = No Boot
- The BIOS is responsible for
  - Hardware discovery and initialization
    - CPUs, memory, I/O controllers, devices, etc.
  - Hardware configuration
  - OS boot
- 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 ☺

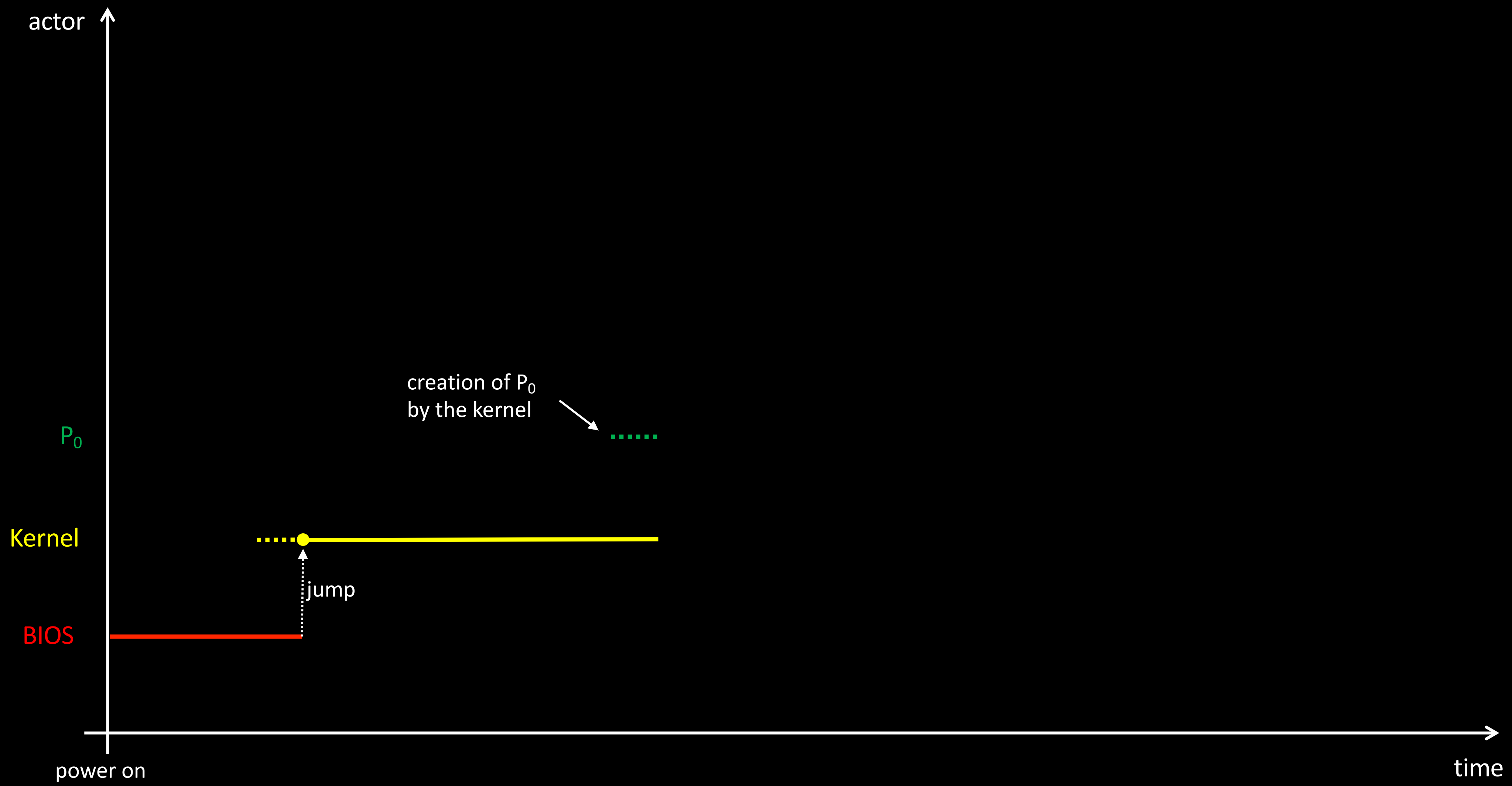

actor
BIOS
power on
time

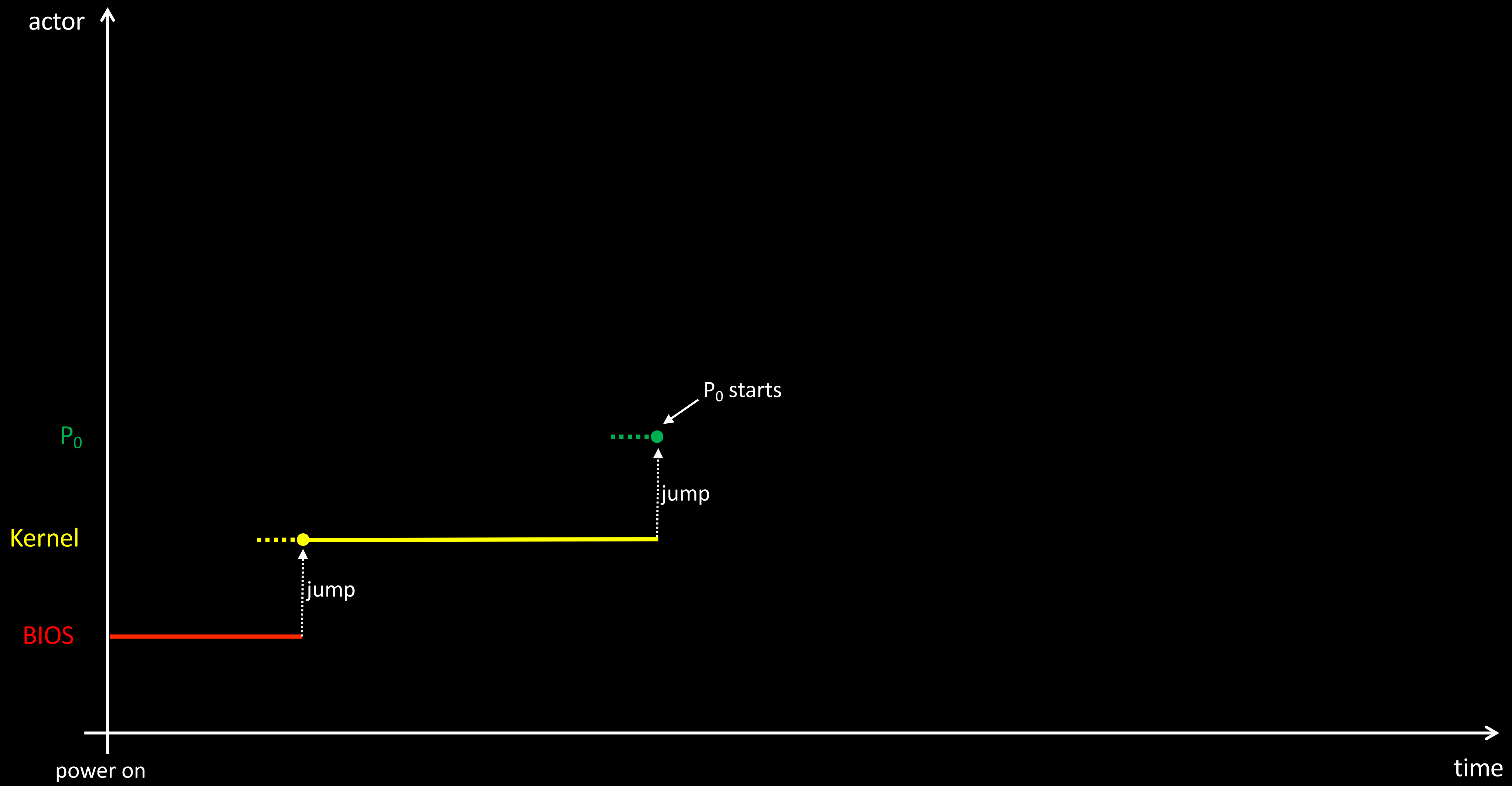

actor
BIOS
power on
time


actor
kernel is loaded
from disk
Kernel
BIOS
power on
time

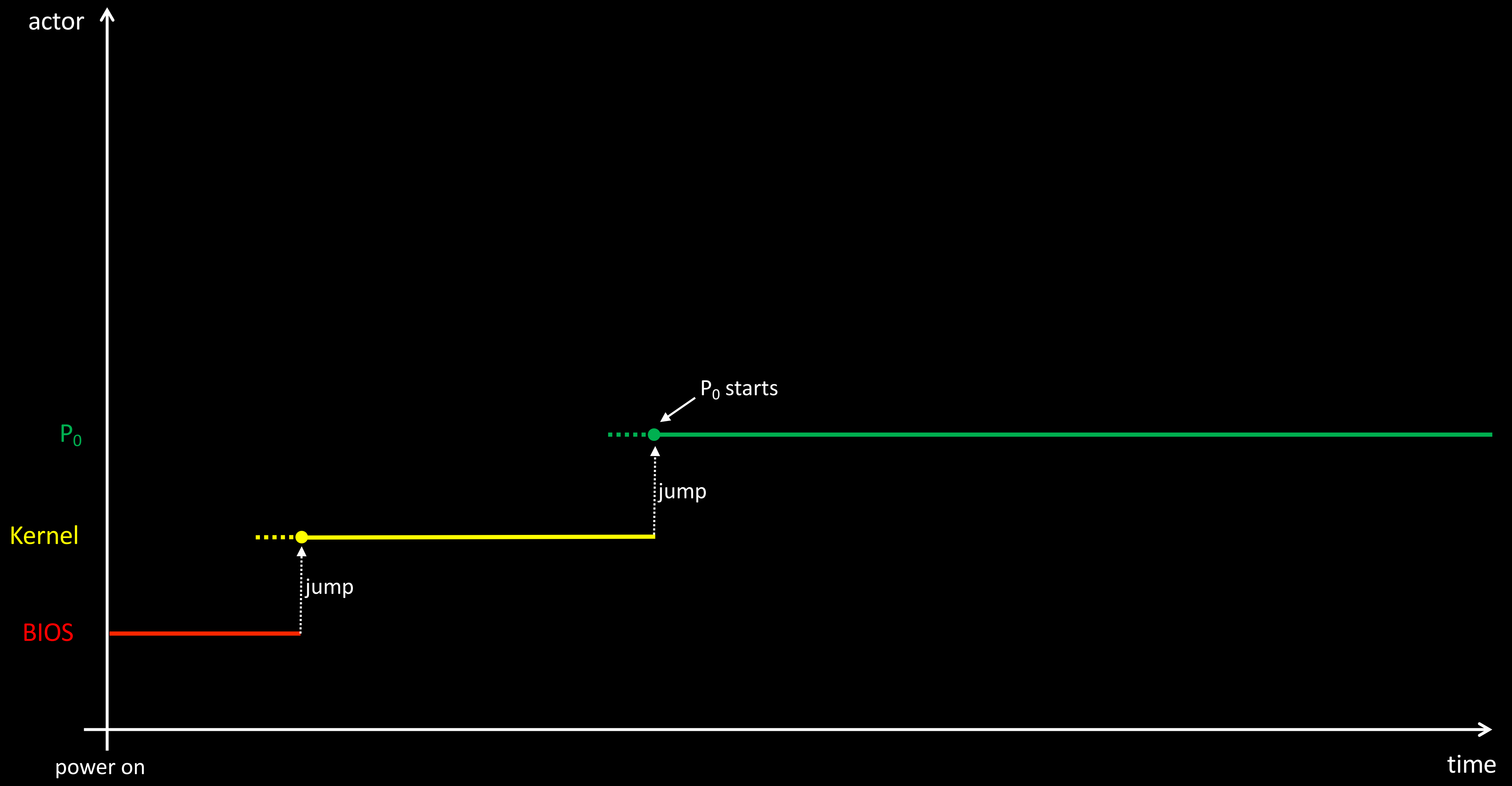

actor
kernel starts
Kernel
jump
BIOS
power on
time


actor
kernel starts
Kernel
jump
BIOS
power on
time

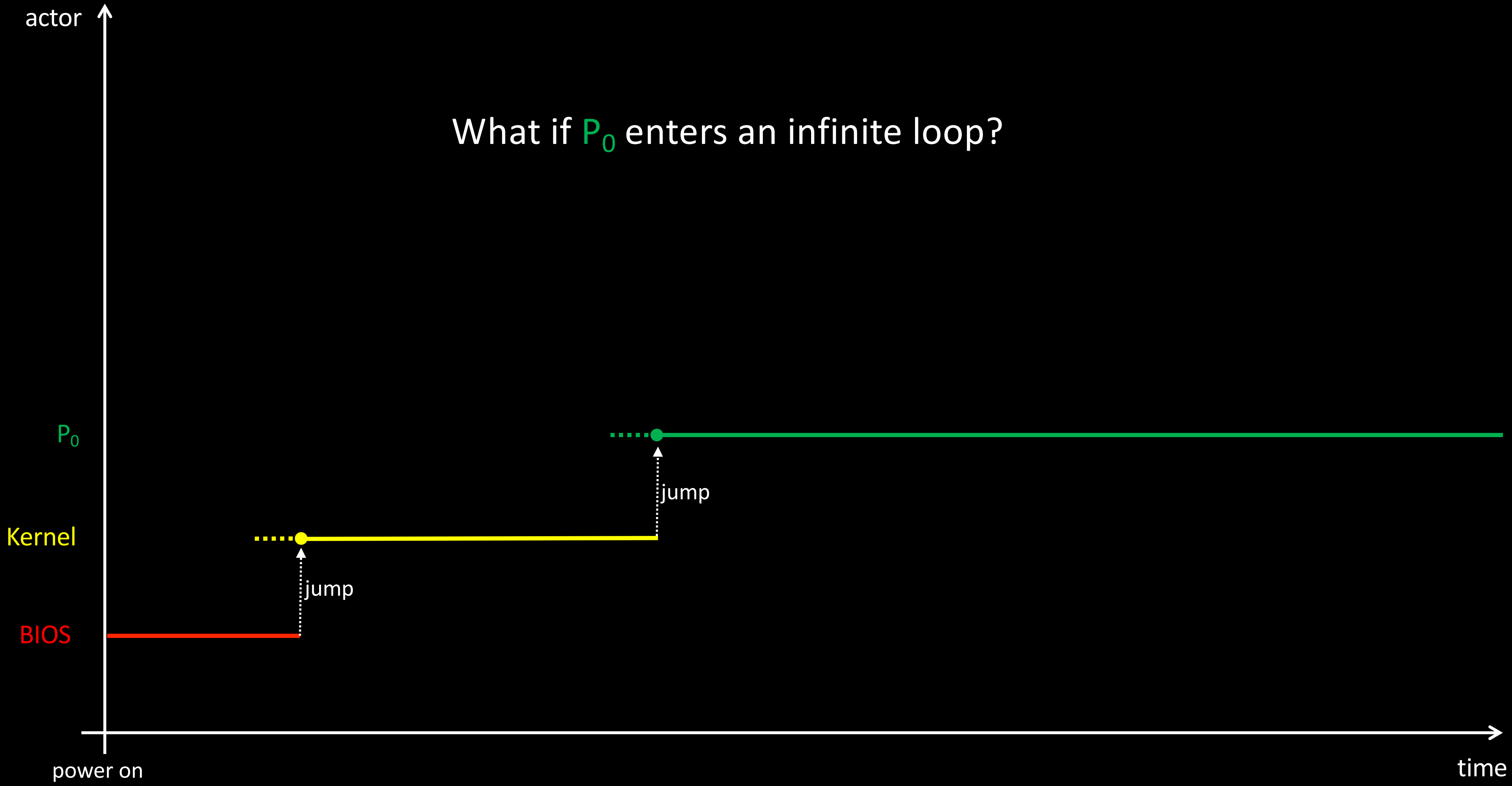

actor
creation of P_0
by the kernel
P_0
Kernel
jump
BIOS
power on
time



actor
P0 starts
P0
jump
Kernel
jump
BIOS
power on
time


actor
P_0 starts
P_0
jump
Kernel
jump
BIOS
power on
time

What if P_0 enters an infinite loop?



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

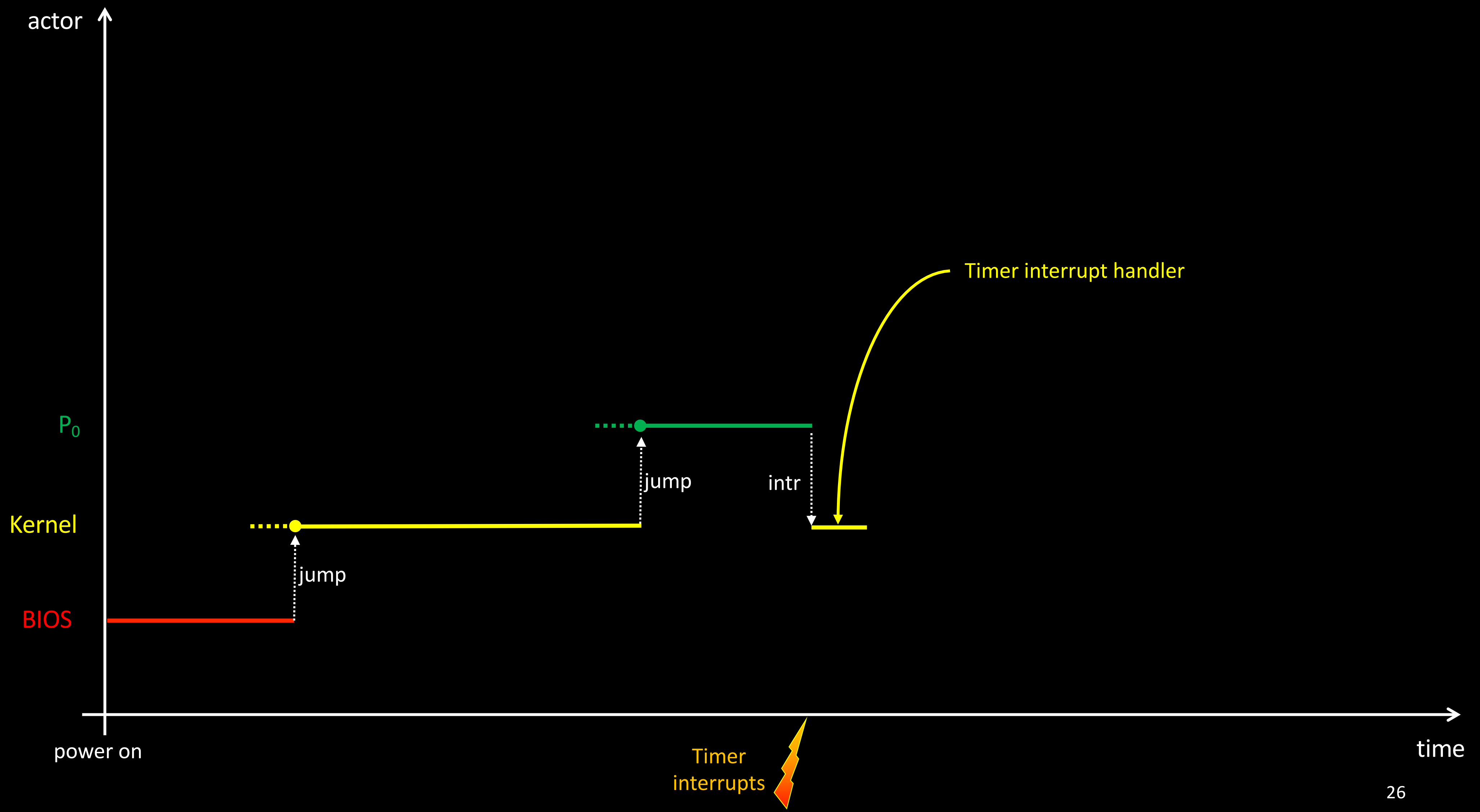
Interrupts

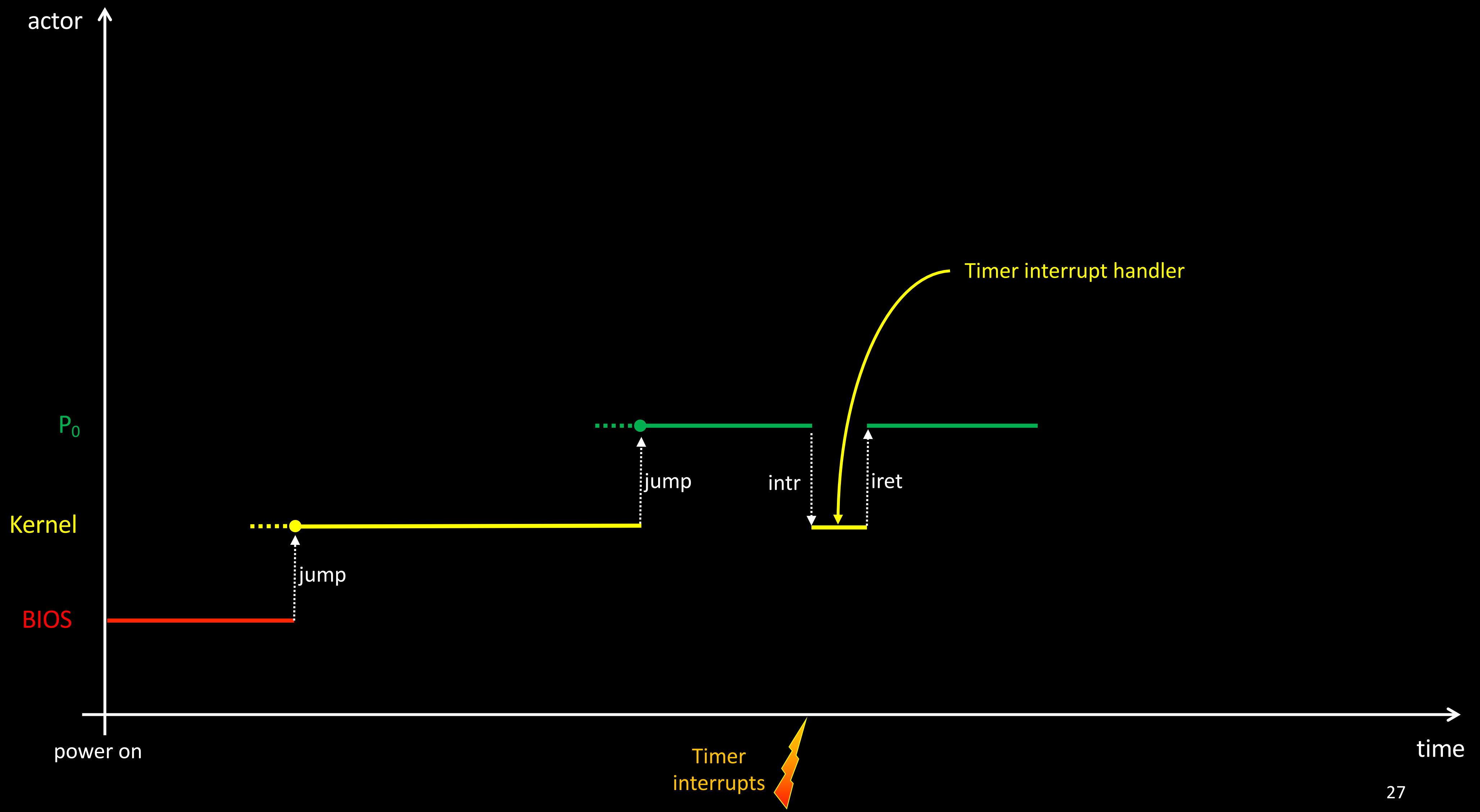
- Moving the Mouse generates interrupts
- Pressing (and releasing) the shift key on the keyboard generates 2 interrupts
- The Network Interface Card (NIC) generates an interrupt each time a packet is received
- Etc.
  - Try `xosview2` under Linux

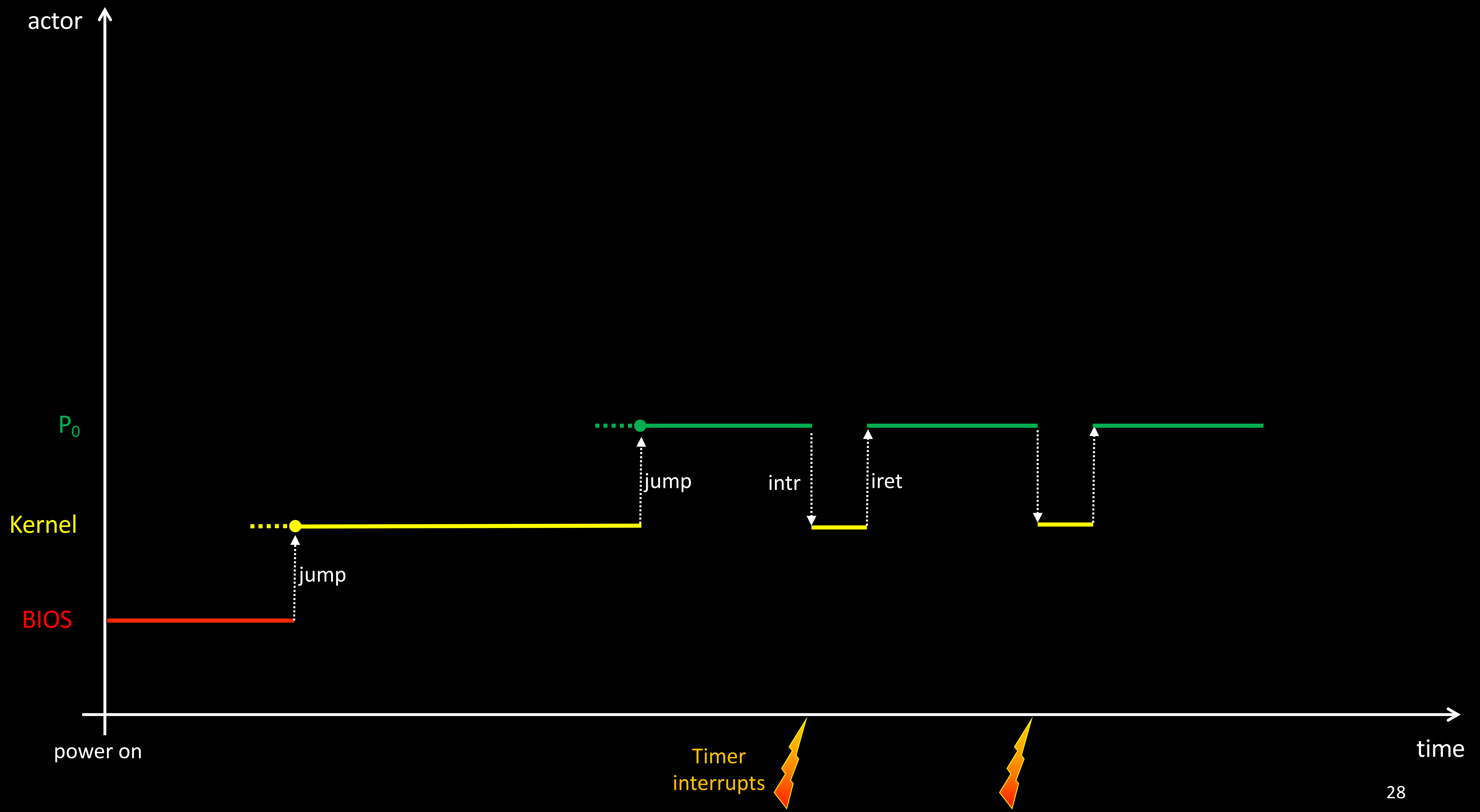
Implementing Time Sharing

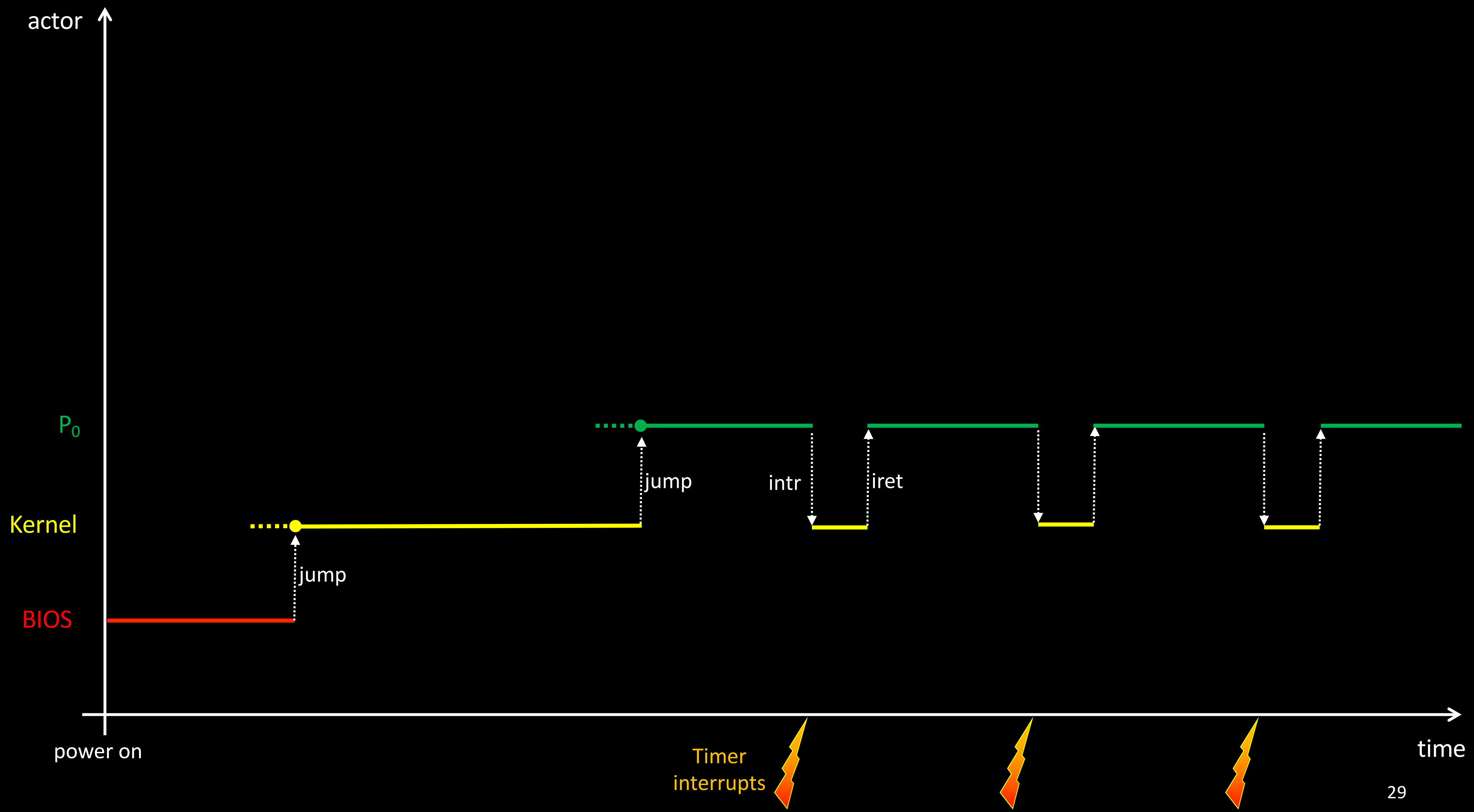
- To prevent processes running during unbounded periods, the kernel sets up a timer
  - A timer interrupt will be periodically triggered (~ 100 ms)
  - This ensures that the associated kernel routine will be executed on a regular basis

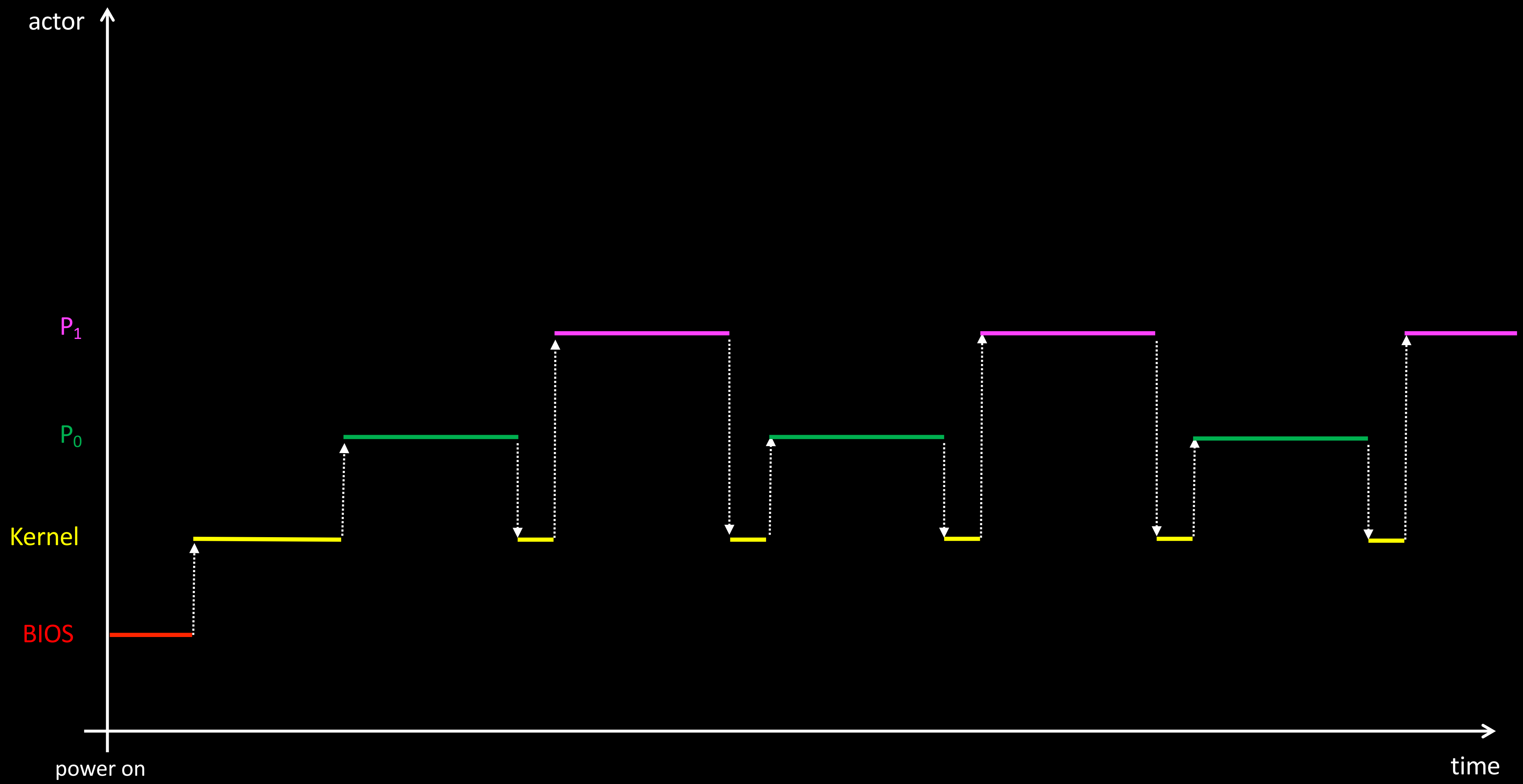
  - Of course, the Interrupt Vector Table must be initialized beforehand!


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


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


actor
P_0
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
P1
P0
Kernel
BIOS
power on
time

Restricting Processes' privileges

- Ok, great: processes can no longer run forever
  - timer interrupts allow the kernel to stop/kill a long running process
- Hmm wait… What if
  - P_0 changes the time interrupt handler routine address in the table?

Restricting Processes' privileges

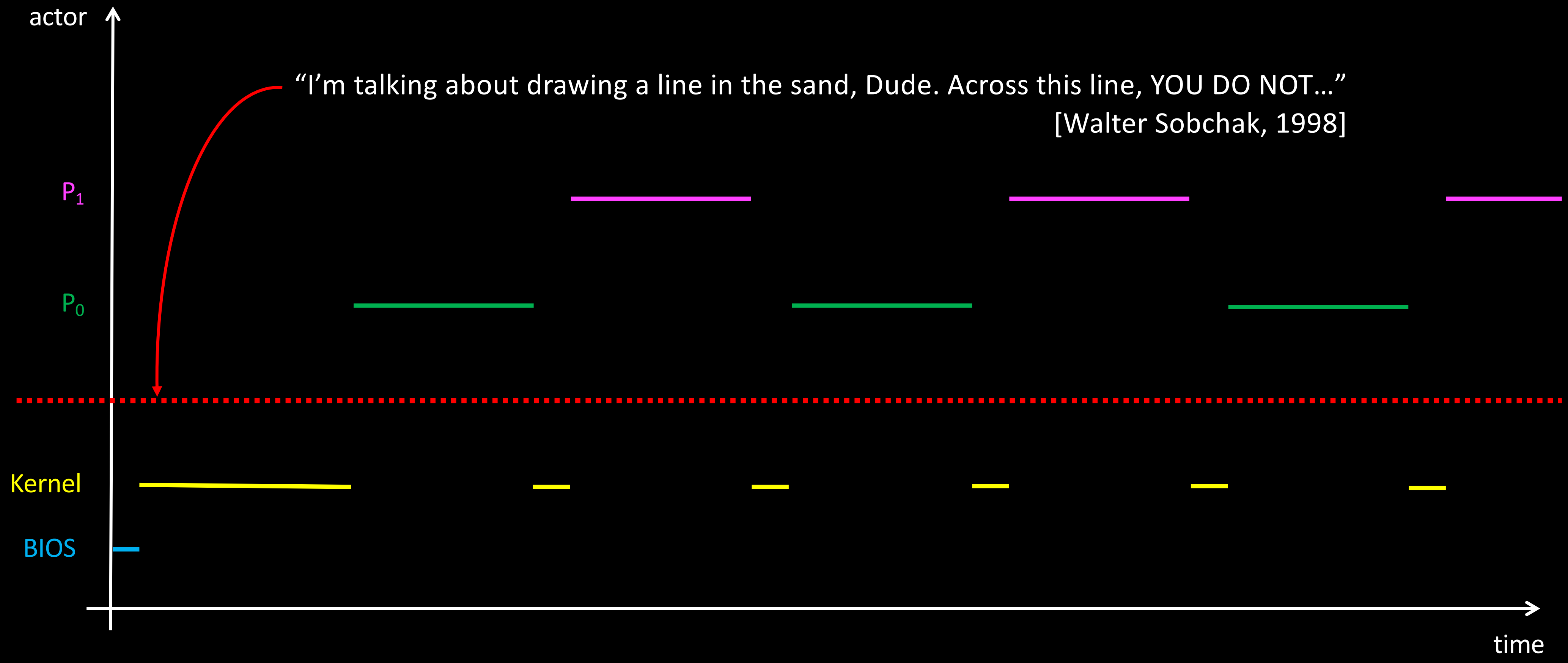
- Ok, great: processes can no longer run forever
  - timer interrupts allow the kernel to stop/kill a long running process

- Hmm wait… What if
  - P_0 changes the time interrupt handler routine address in the table?
  - P_0 reads the keyboard while a user is typing his session password?

Restricting Processes' privileges

- Ok, great: processes can no longer run forever
  - timer interrupts allow the kernel to stop/kill a long running process
- Hmm wait... What if
  - P_0 changes the time interrupt handler routine address in the table?
  - P_0 reads the keyboard while a user is typing his session password?
  - P_0 switches the machine off?
- We have a problem!

Restricting Processes' privileges



Restricting Processes' privileges

- We want to restrict what processes can do
  - Only the kernel should be almighty

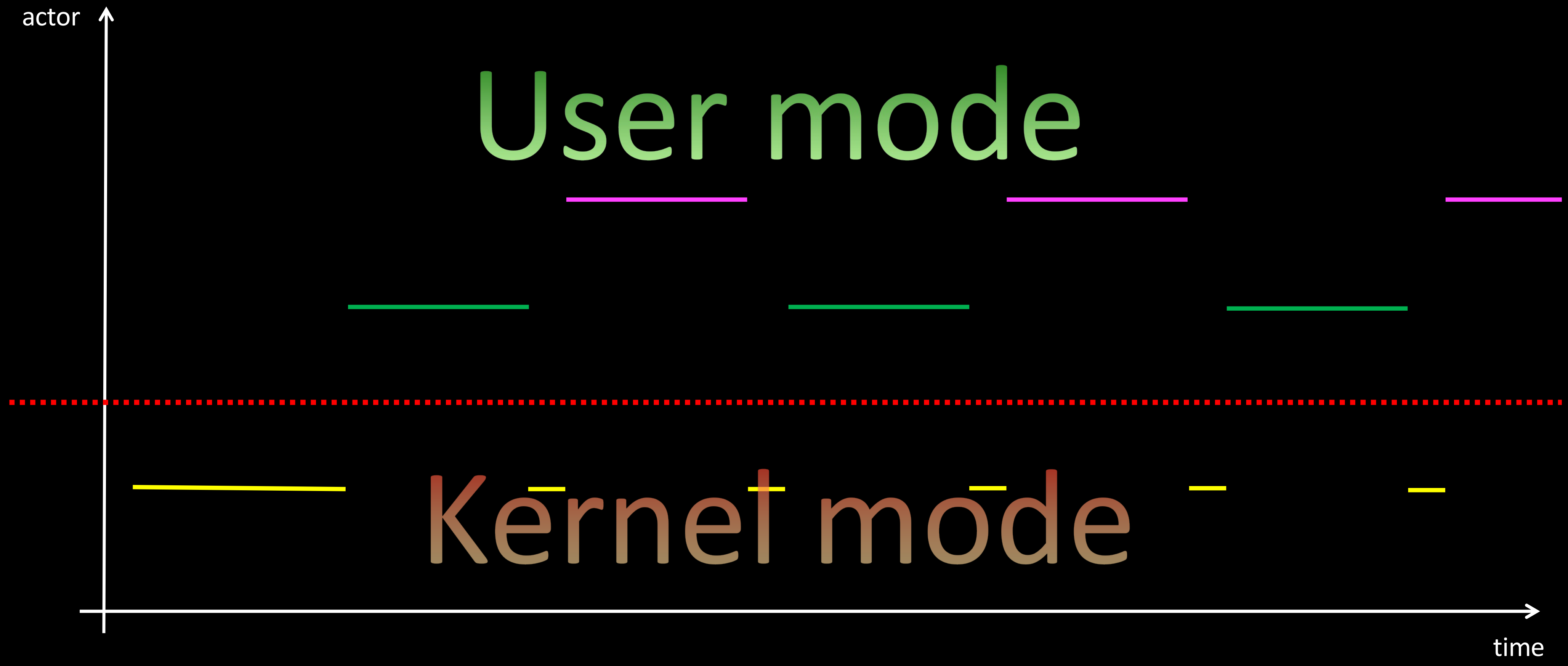
- Let's assume we can establish a list of forbidden CPU instructions

- How to prevent processes from calling specific instructions?
  - Clever compiler?
  - Real-time scan of the program by the kernel?

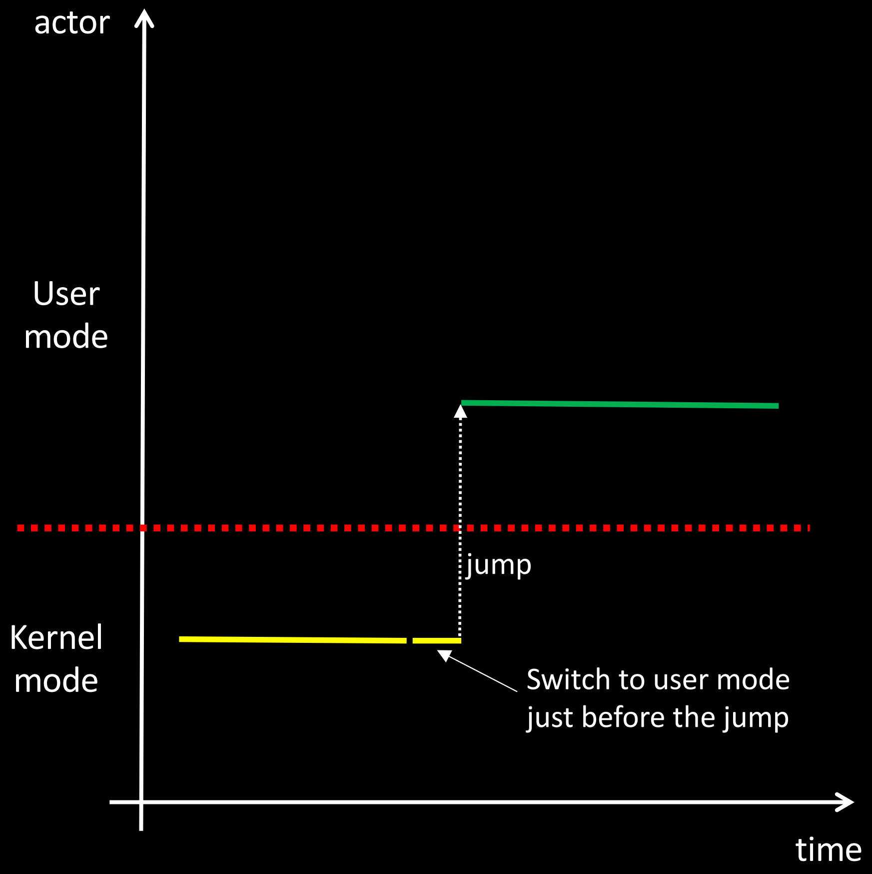
Restricting Processes' privileges

- This can be done only by the hardware, that is, the CPU
  - Privileged instructions are flagged
- The CPU can run in (at least) two different modes:
  - *User mode* (aka Protected mode) / *Kernel mode* (aka Real mode)
  - The current mode is stored in a control register
- In *user mode*, only a subset of the CPU instruction set is available
  - If the CPU is about to execute a privileged instruction in user mode…
    … an exception is raised (like an interrupt)

Restricting Processes' privileges

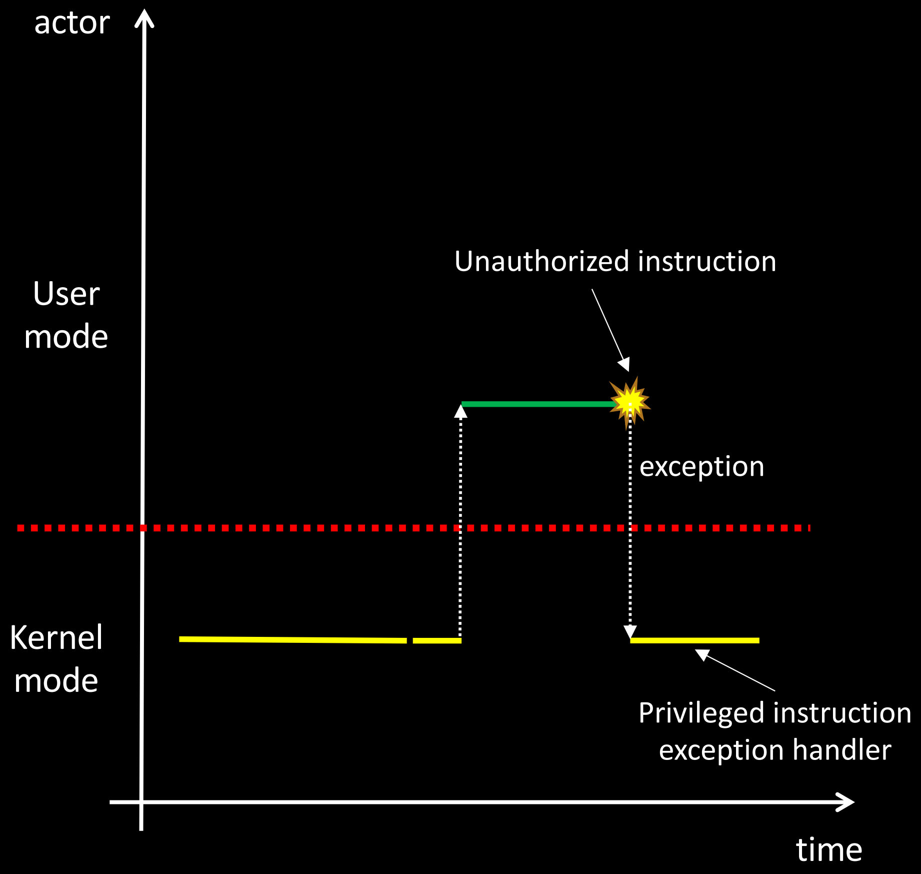


Restricting Processes' privileges



- The CPU wakes up (power on) in kernel mode
  - So BIOS and kernel initialize in kernel mode
- The kernel gives up its privileges by switching to user mode…
  - By changing the mode bits in the control register

Restricting Processes' privileges



- The CPU wakes up (power on) in kernel mode
  - BIOS and kernel both start in kernel mode
- At some point, the kernel gives up its privileges
  - Explicit switch to user mode

    = changing the mode number in the control register

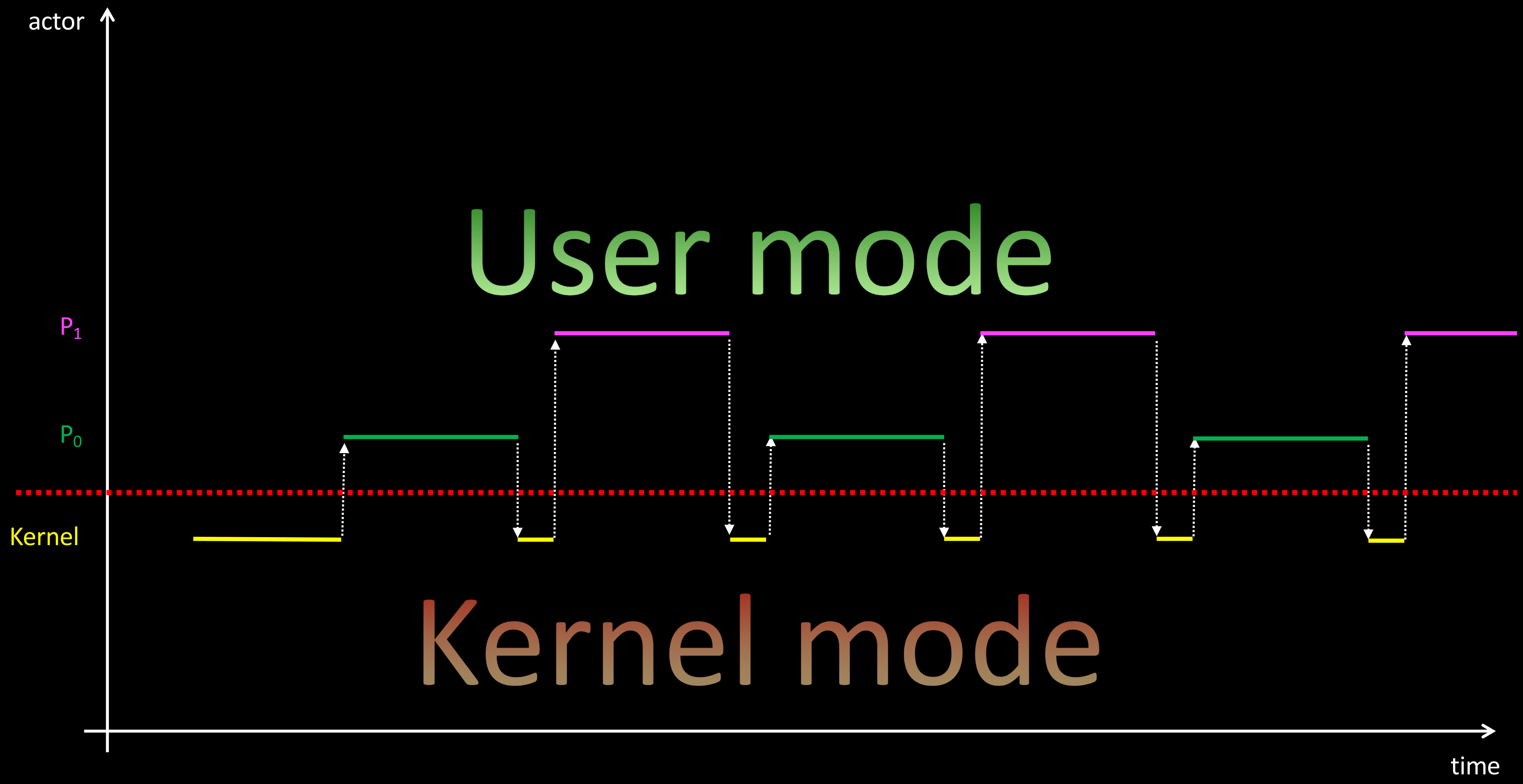
Restricting Processes' privileges

- Obviously, a process should not be able to easily go back to kernel mode

Restricting Processes' privileges

- Obviously, a process should not be able to easily go back to kernel mode
  - Explicit change to the control register is only possible in kernel mode

- Interrupts automatically enter kernel mode
  - And iret (Interrupt RETurn) automatically goes back to previous mode


actor
User mode
P1
P0
Kernel
Kernel mode
time

Requesting privileges

- Ok, the kernel is safe
  - Processes cannot directly access the hardware

- But this brings a new problem:
  - At some point, processes NEED to execute privileged instructions
    - Display a string in the terminal (e.g. `printf`)
    - Read a character from the keyboard (e.g. `getc`)
    - Create a new process (e.g. `fork`)

- How to allow processes to <u>temporarily</u> execute privileged instructions?
  - Ask kernel for permission + instructions check + signal when done?
  - Ask for privileges during a limited period?

Requesting privileges

- We need a safe way to do it
  - We already have a mechanism to switch to kernel mode…

Requesting privileges

- We need a safe way to do it
    - We already have a mechanism to switch to kernel mode: *interrupts*!

- Let's use a specific instruction to raise a software interrupt
    - `int 80h` (Linux x86 32bit kernels)
    - `syscall` (Linux x86 64bit kernels)

Requesting privileges

- We need a safe way to do it
  - We already have a mechanism to switch to kernel mode: *interrupts*!
- Let's use a specific instruction to raise a software interrupt
  - `int 80h` (Linux x86 32bit kernels)
  - `syscall` (Linux x86 64bit kernels)
- Idea
  - The kernel has a set of routines which can be useful to processes
  - To invoke one of these routines, a process performs a *system call*
    - How do we specify the desired routine?

Requesting privileges

- We need a safe way to do it
  - We already have a mechanism to switch to kernel mode: *interrupts*!
- Let's use a specific instruction to raise a software interrupt
    - `int 80h` (Linux x86 32bit kernels)
    - `syscall` (Linux x86 64bit kernels)
- Idea
  - The kernel has a set of routines which can be useful to processes
  - To invoke one of these routines, a process performs a *system call*
    - Put the *routine number* into a register (`%eax` on x86_84 architectures)
    - Raise the interrupt

System calls

- Example:
  - C implementation of the file "getpid" function in libc

```
pid_t getpid (void)
{
  mov __NR_getpid, %eax
  syscall
  ret
}
```

System calls

- On the kernel side, a table contains the addresses of routines implementing systems calls
  - `sys_getpid`, `sys_open`, `sys_write`, `sys_read`, etc.
  - The syscall interrupt handler uses the number found in %eax (on x86 processors) to call the requested routine
  - Kernel and libc need to be synchronized!
    - `unistd.h`, which assigns numbers to system call, is included on both sides

System calls

- Why is it a safe mechanism?
  - Because the process does not specify a routine address, but a *number*
    - The kernel has complete control on the code
    - Parameters checking is performed by the kernel and cannot be skipped
  - What parameters? Where are they?
    - They're pushed on the stack when calling the stub

```
pid_t getpid (void)
{
  mov __NR_getpid, %eax
  syscall
  ret
}
```

System calls & library calls

- Modern operating systems provide hundreds of system calls
  - ~330 in Linux, ~530 in Mac OS X

- The libc features a lot more routines
  - Is it easy to distinguish between system calls and regular routines?
    - No, but who cares?

  - If you care
    - You can run your program under the Linux `strace` utility
    - Or you can disassemble the very first instructions to check for the syscall CPU instruction

Structure of an OS