Operating Systems: an introduction

Raymond Namyst

Dept. of Computer Science

University of Bordeaux, France

https://gforgeron.gitlab.io/se/

Goals

- Understand how an Operating System works
  - Most importantly: why does it work that way?
- In-depth cover of the following topics
  - Process scheduling
  - Synchronization
  - Memory management
  - Links between I/O and memory

Why should you care?

- Only a few of you will have to cope with OS internals in your professional life
  - Except security experts, needless to say ☺
- Being aware of OS internal mechanisms makes you a better programmer
  - Memory allocation
  - Multicore programming
  - Race conditions avoiding

Organization

- Operating Systems require a lot of practice work
  - Theoretical courses do not reflect the complexity of real implementation

Organization

- Operating Systems require a lot of practice work
  - Theoretical courses do not reflect the complexity of real implementation
  - Hacking a real OS requires more than 2h lab session per week
  - Mini-projects using an OS simulator (NACHOS)

- Evaluation
  - Students will achieve 3 mini-projects (pair-work)

    -> 3 practice grades: TP1, TP2, TP3
  - CC = (TP1 + TP2 + TP3 + mid-course test) / 4
  - Final grade = ½ CC + ½ Exam

NACHOS

- Not Another Completely Heuristic Operating System
  - Instructional software for teaching operating systems courses
    - Thomas Anderson, The University of California, Berkeley
  - We use a slightly modified version at University of Bordeaux

- NACHOS is deterministic
  - Pseudo randomization
    - Each seed leads to a same execution
  - Muuuuuuuuch more convenient to debug!

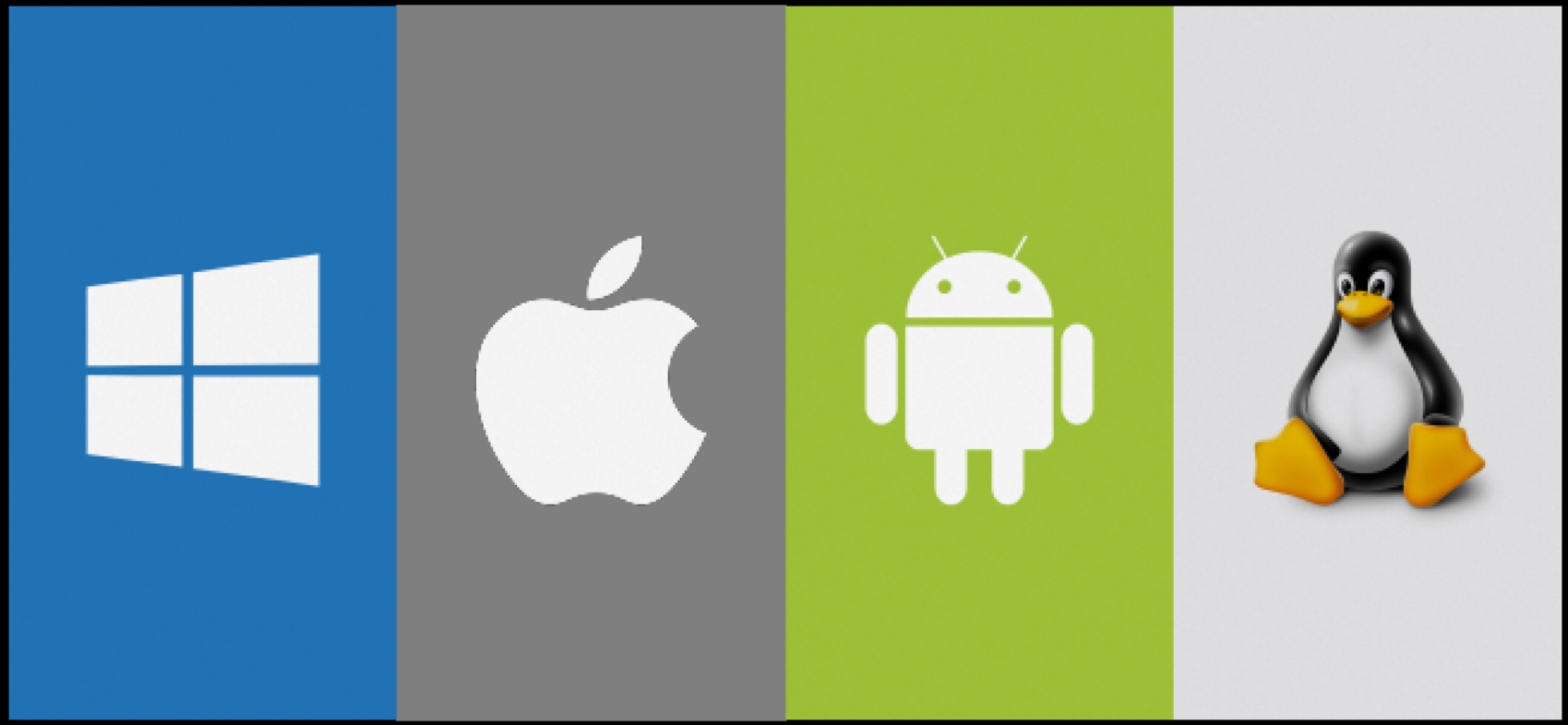
Bibliography

- Modern Operating Systems, Andrew S. Tanenbaum
- Operating System Concepts, A. Silberschatz, P. Galvin, G. Gagne
- Understanding the Linux Kernel, D. Bovet, M. Cesati

Outline

- Introduction
  - Kernel, interrupts, system calls
- Process management
  - Scheduling in interactive Operating Systems
- Synchronizations
  - Atomic instructions, locks, semaphores, Hoare monitors
- Memory Management
  - History, segmentation, pagination, swap
- Input/Ouput
  - Buffers and memory mapping

What is an Operating System?



www.wooclap.com/SEFOREVER

Qu'est-ce qu'un système d'exploitation ?

1	Une interface graphique proposant une expérience utilisateur cohérente	60%	55
2	Une personne habillée en noir qui surveille tout ce qu'on fait	4%	4
3	Une collection de pilote de périphériques qui masquent les spécificités du matériel	65%	59
4	Un logiciel qui ralentit ma machine	7%	6
5	Un ensemble d'applications gavé utiles	21%	19

What is 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*

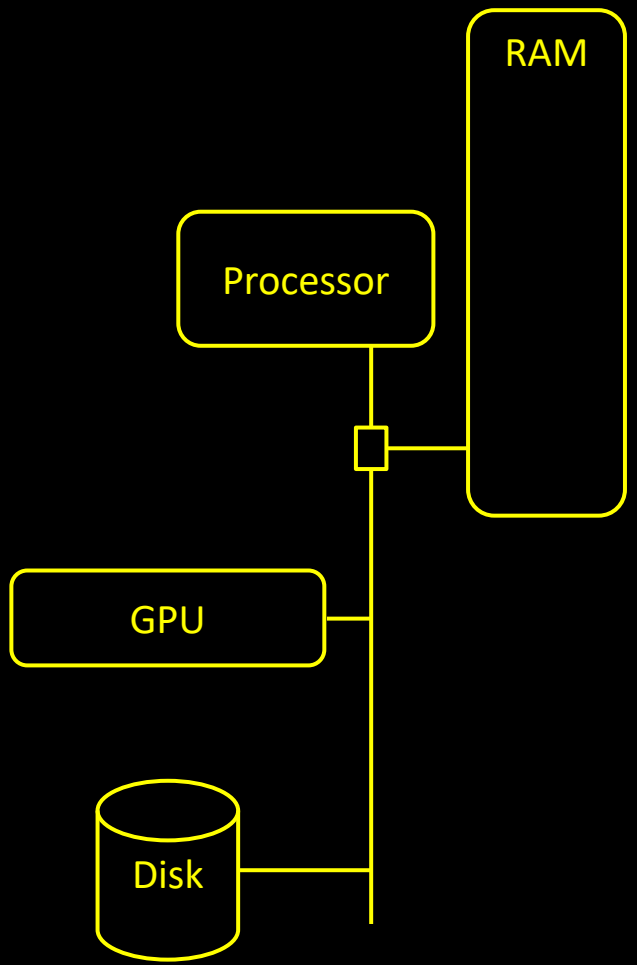
- It is composed of several important parts
  1. A set of device drivers (= code)
  2. A set of programs (= code)
     - Some of these programs are running in the form of background processes
       - So-called daemons: inetd, cupsd, sshd, syslogd, etc.
     - Some others are executed on demand
       - 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

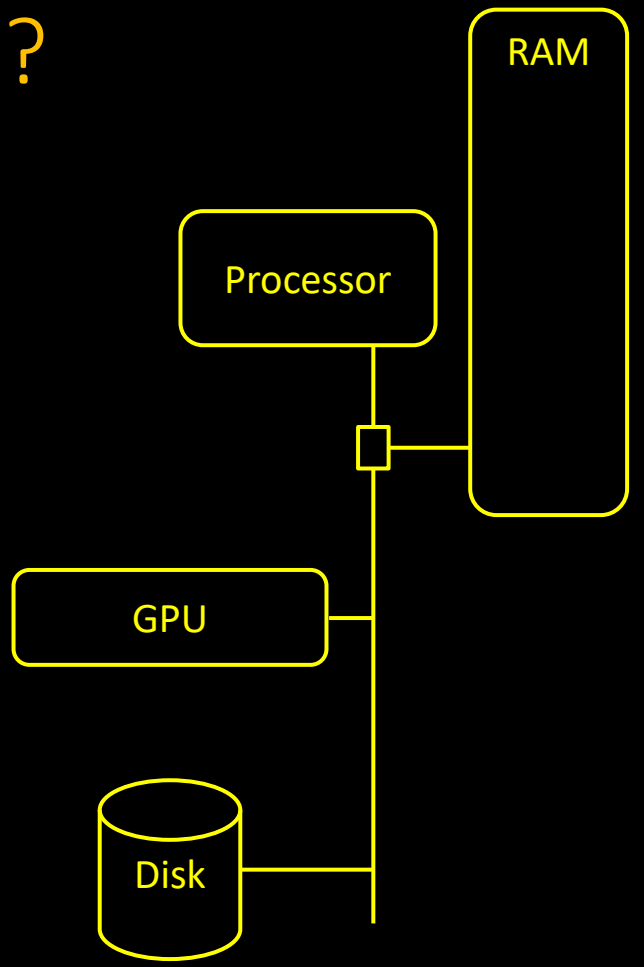
- To do so, they stand in between applications and the hardware
  - On good old single-user Operating Systems (e.g. MS DOS)
    - Programs were executed one at a time… and could enjoy direct access to the hardware
    - They could corrupt the OS memory, freeze the machine, etc.
    - Great times!
  - It’s no more the case on nowadays’ systems
    - The OS hinders direct access to the hardware
      - How can that be?

Typical Computer Architecture



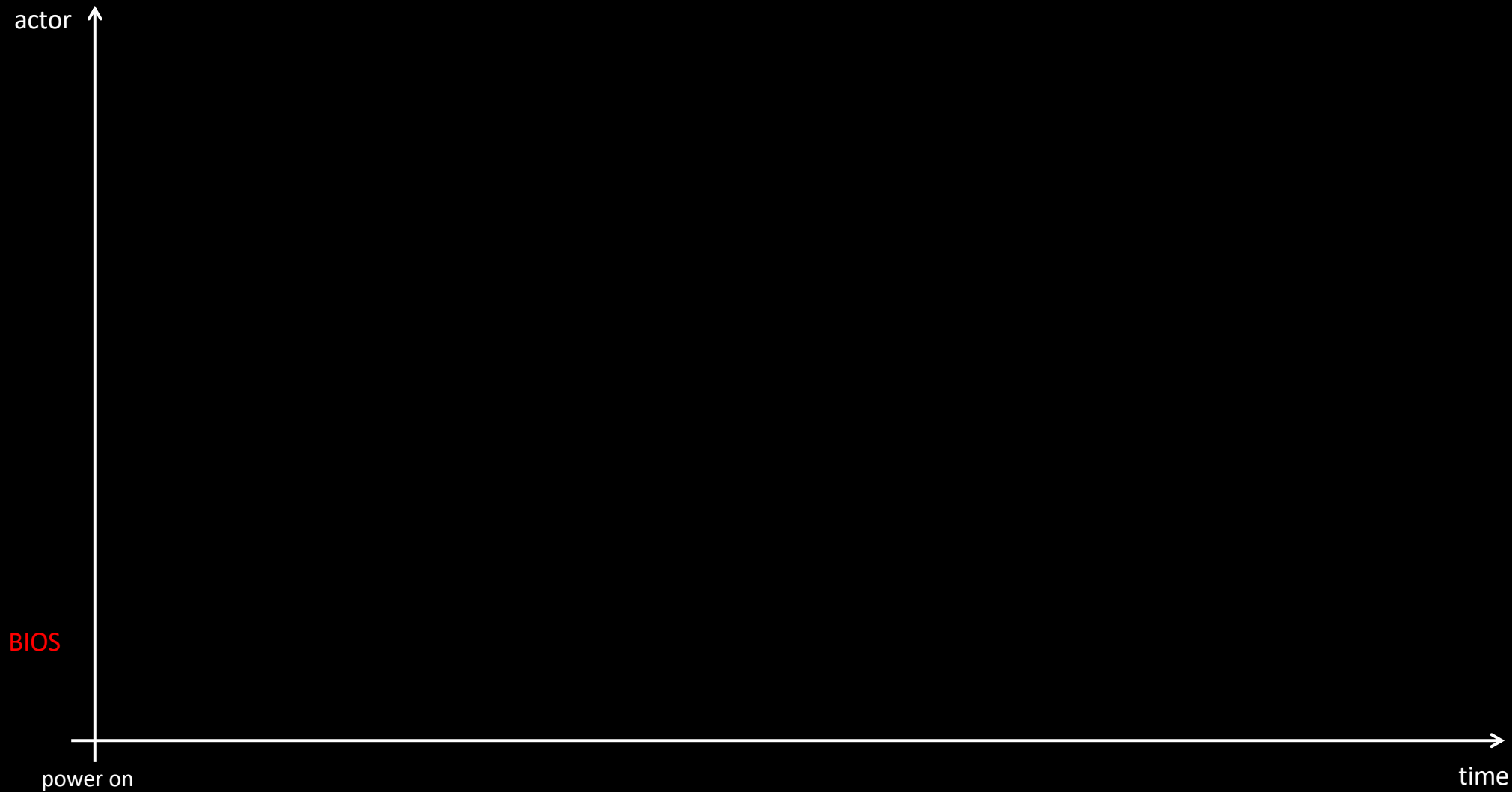
Where is the first instruction?

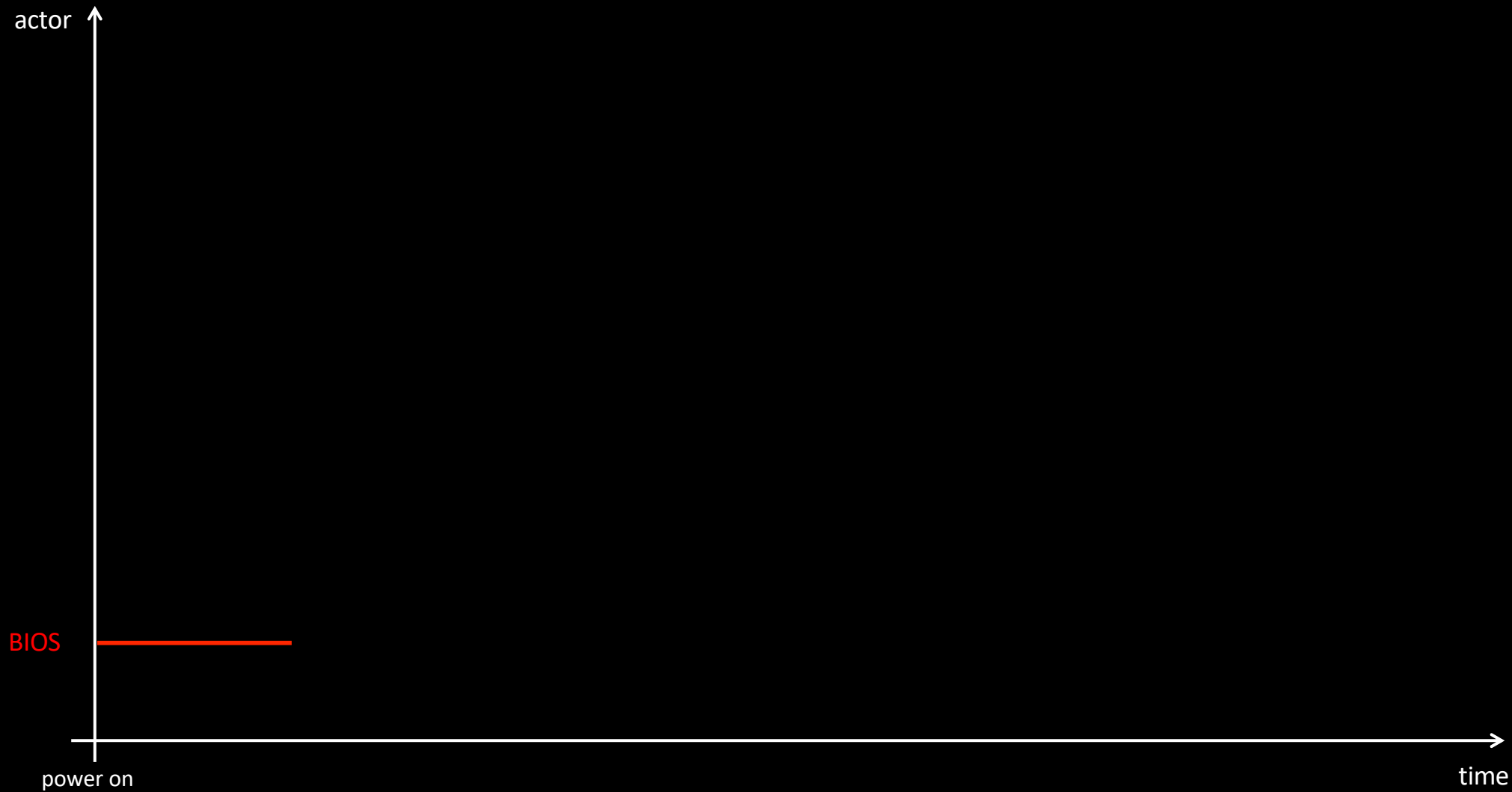
1. The CPU needs instructions!
2. The RAM is empty
3. OS bootstrap is probably on disk
4. To fetch theses instructions to RAM…
5. …CPU needs instructions!
   goto 1



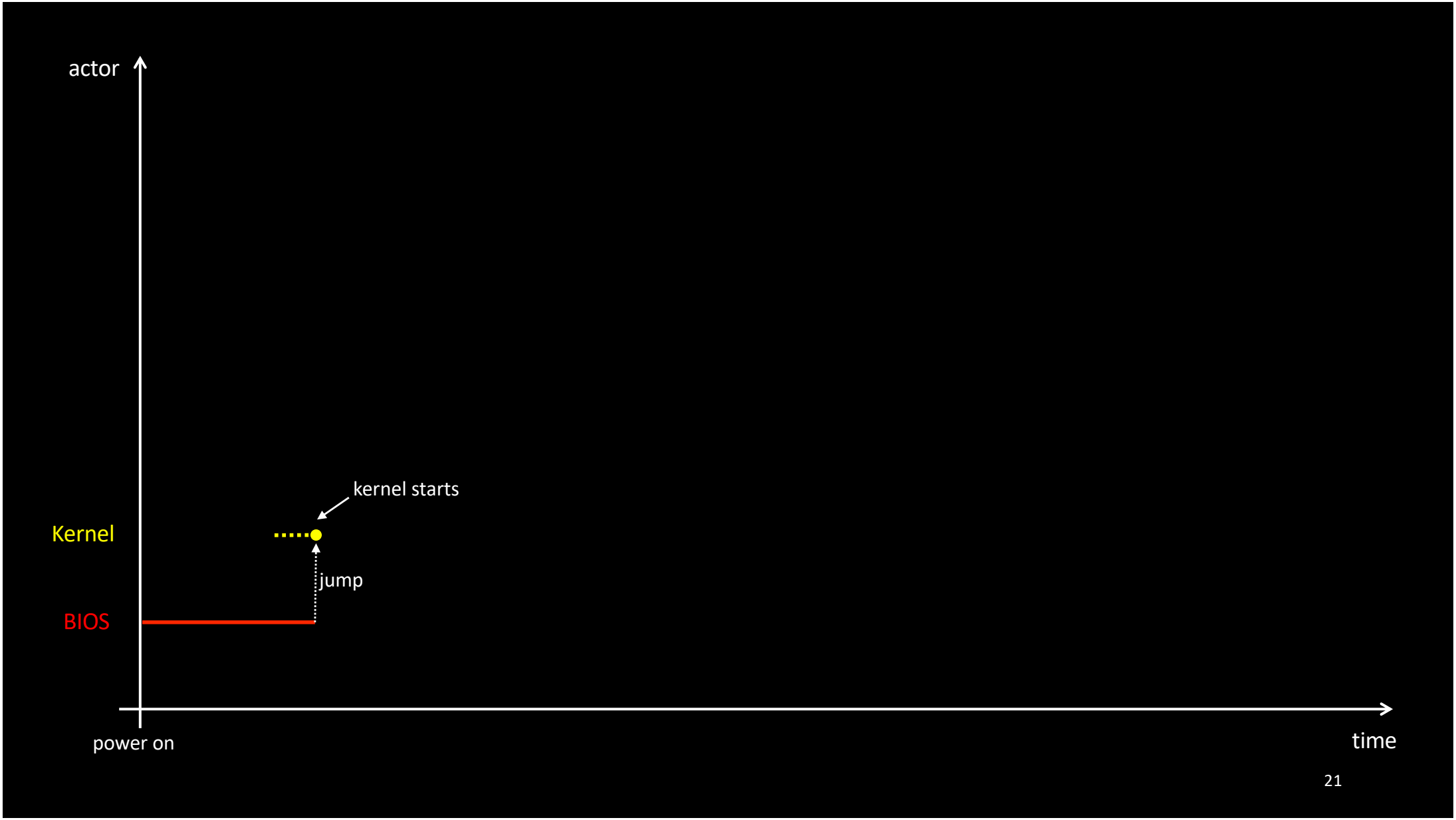
The BIOS (aka ROM BIOS or System BIOS)

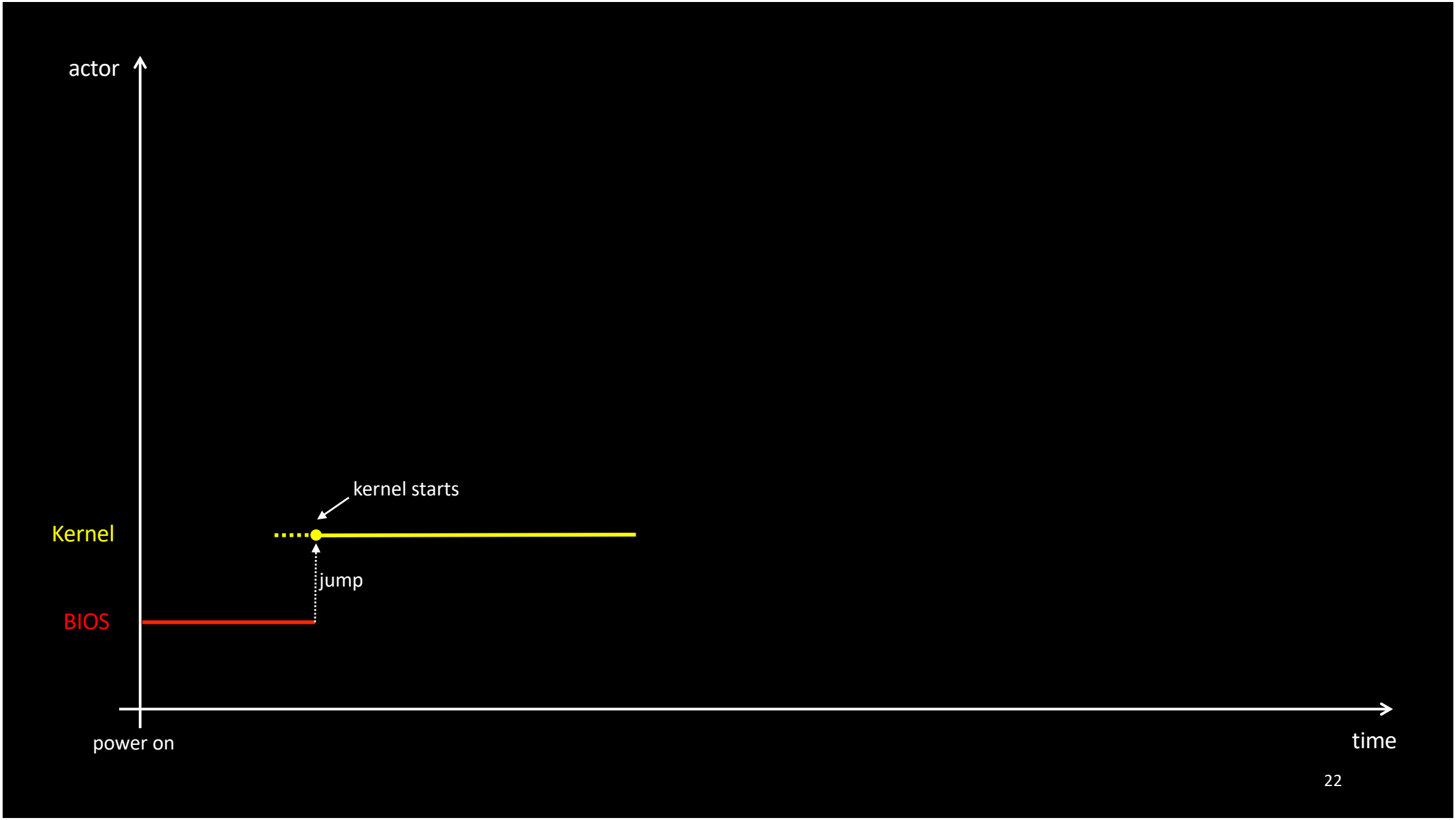
- Firmware stored in ROM chip / flashable memory
  - Contains the very first instructions executed by the processor
    - No BIOS = No Boot
- The BIOS is responsible for
  - Hardware discovery and initialization
    - CPUs, memory, I/O controllers, devices, etc.
  - Hardware configuration
  - OS boot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

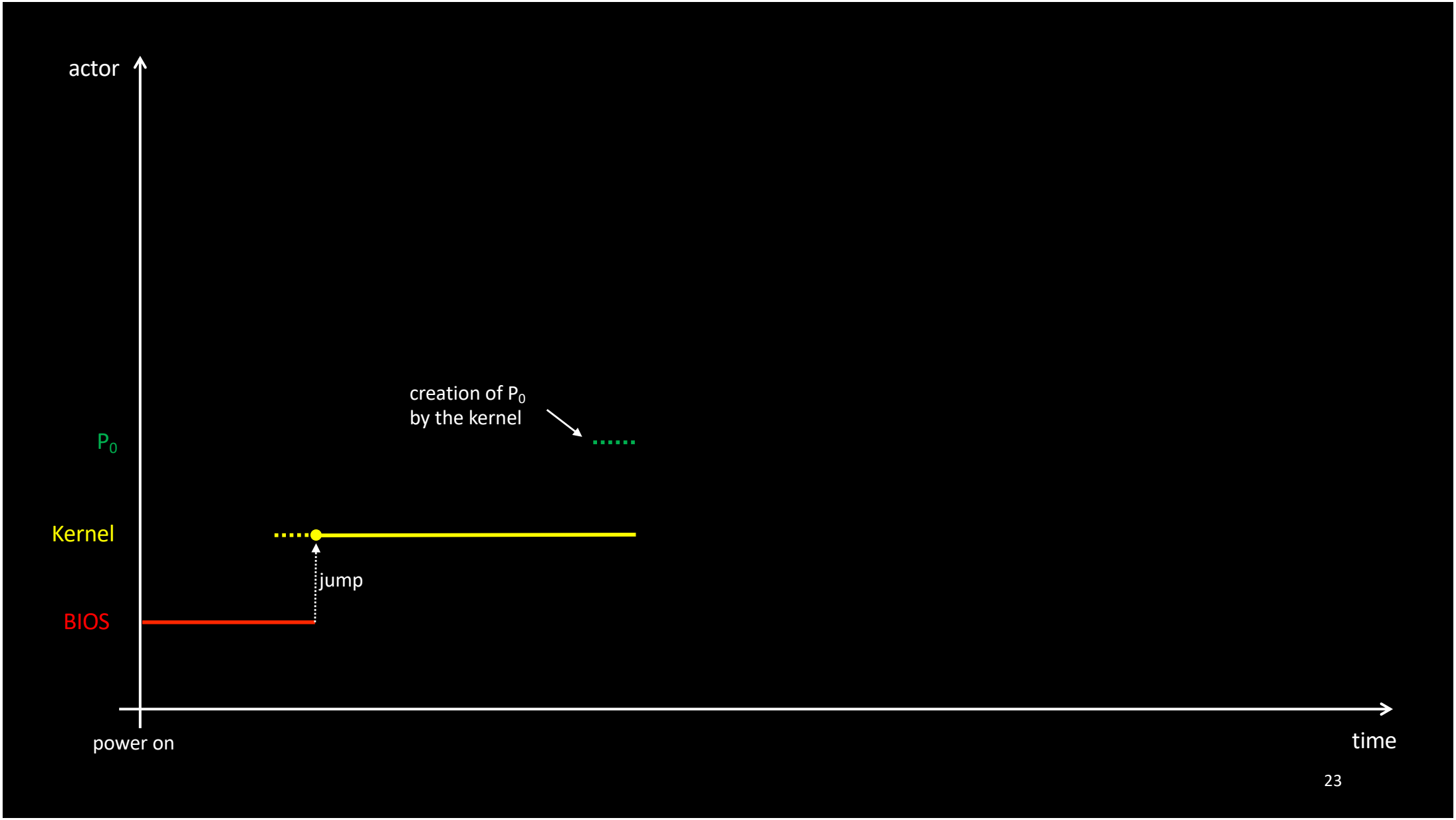

actor
BIOS
power on
time

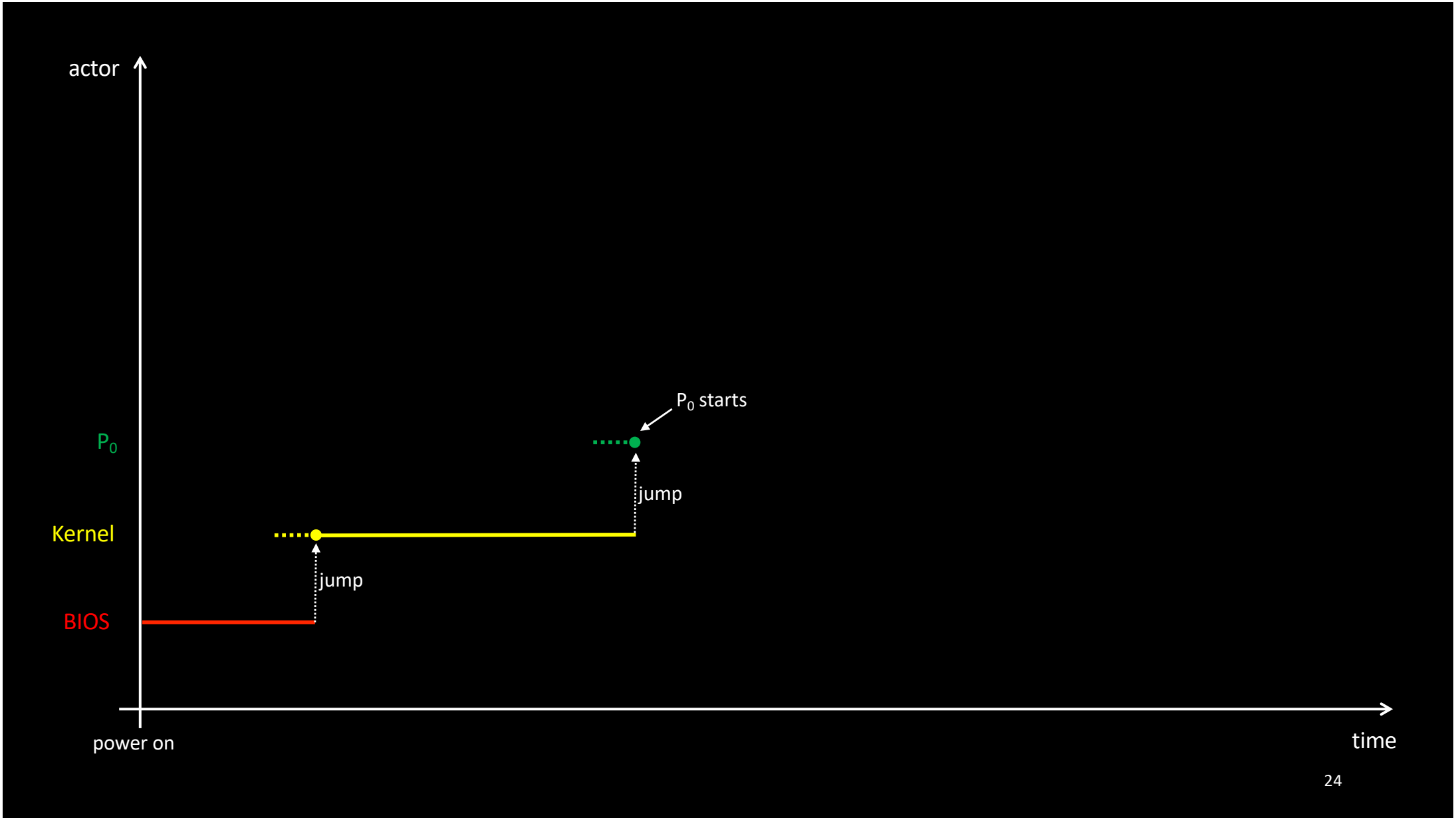

actor
BIOS
power on
time

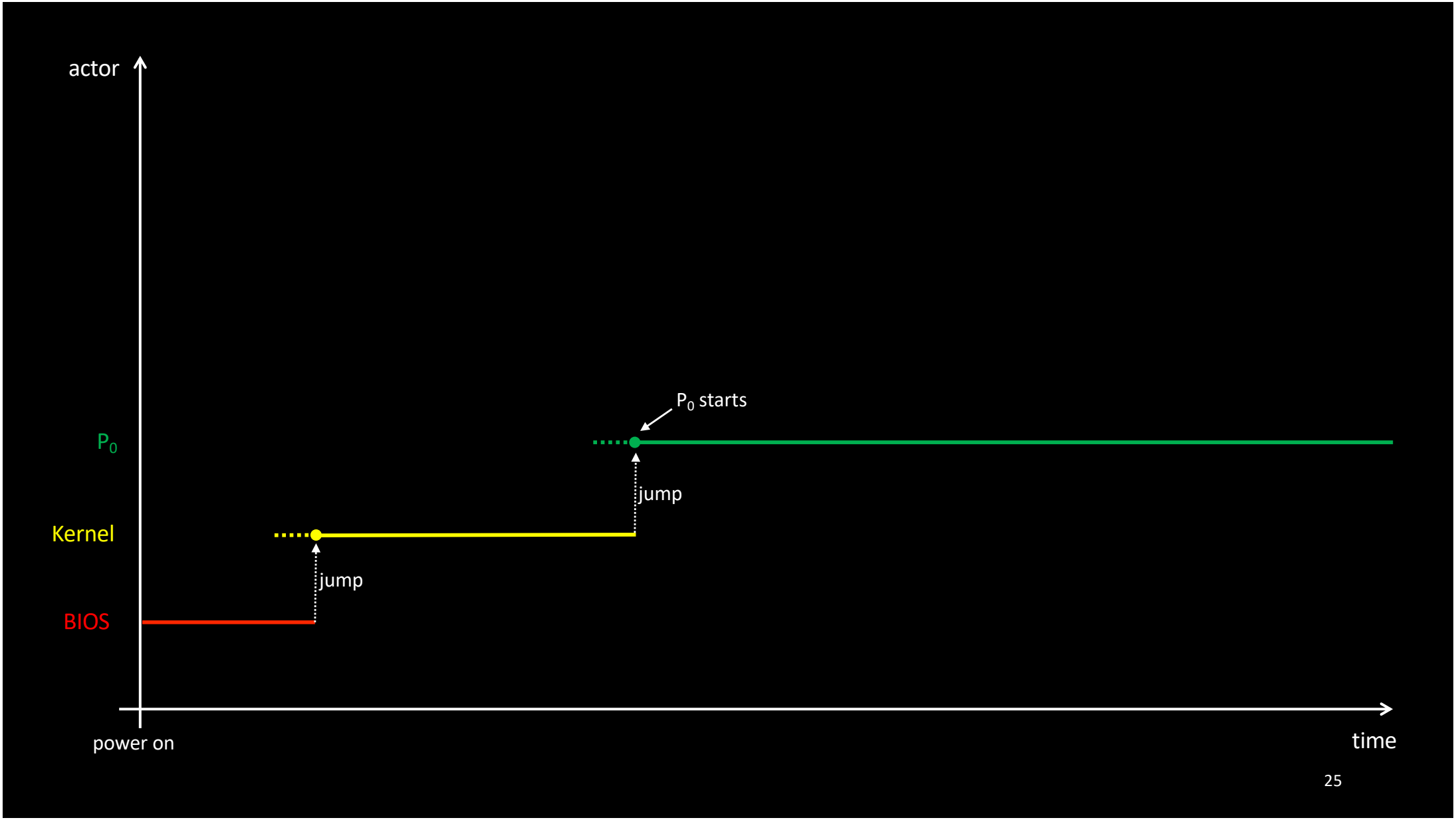

actor
kernel is loaded
from disk
Kernel
BIOS
power on
time

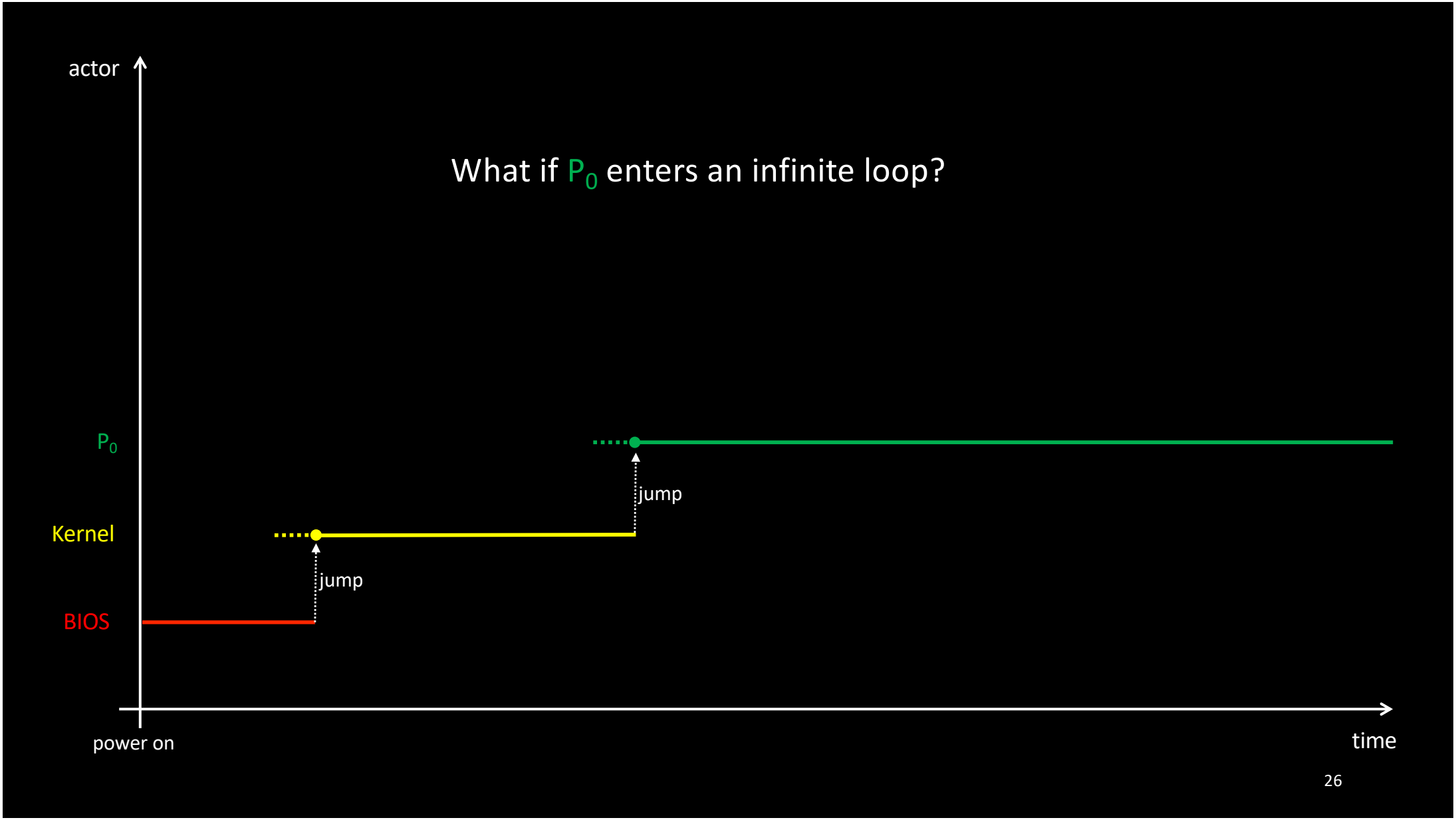

actor
kernel starts
Kernel
jump
BIOS
power on
time


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
  - Handling = 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

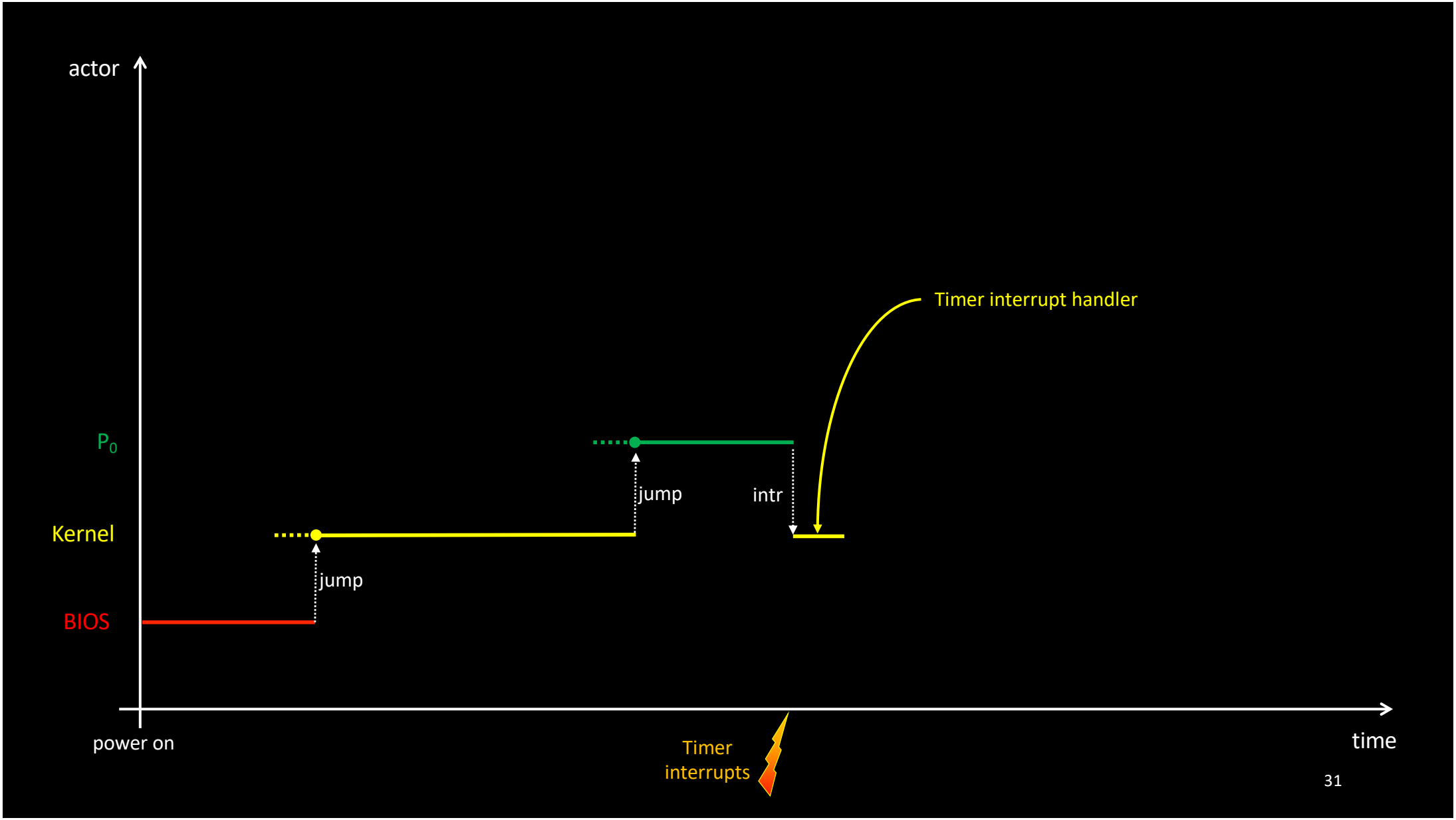
Interrupts

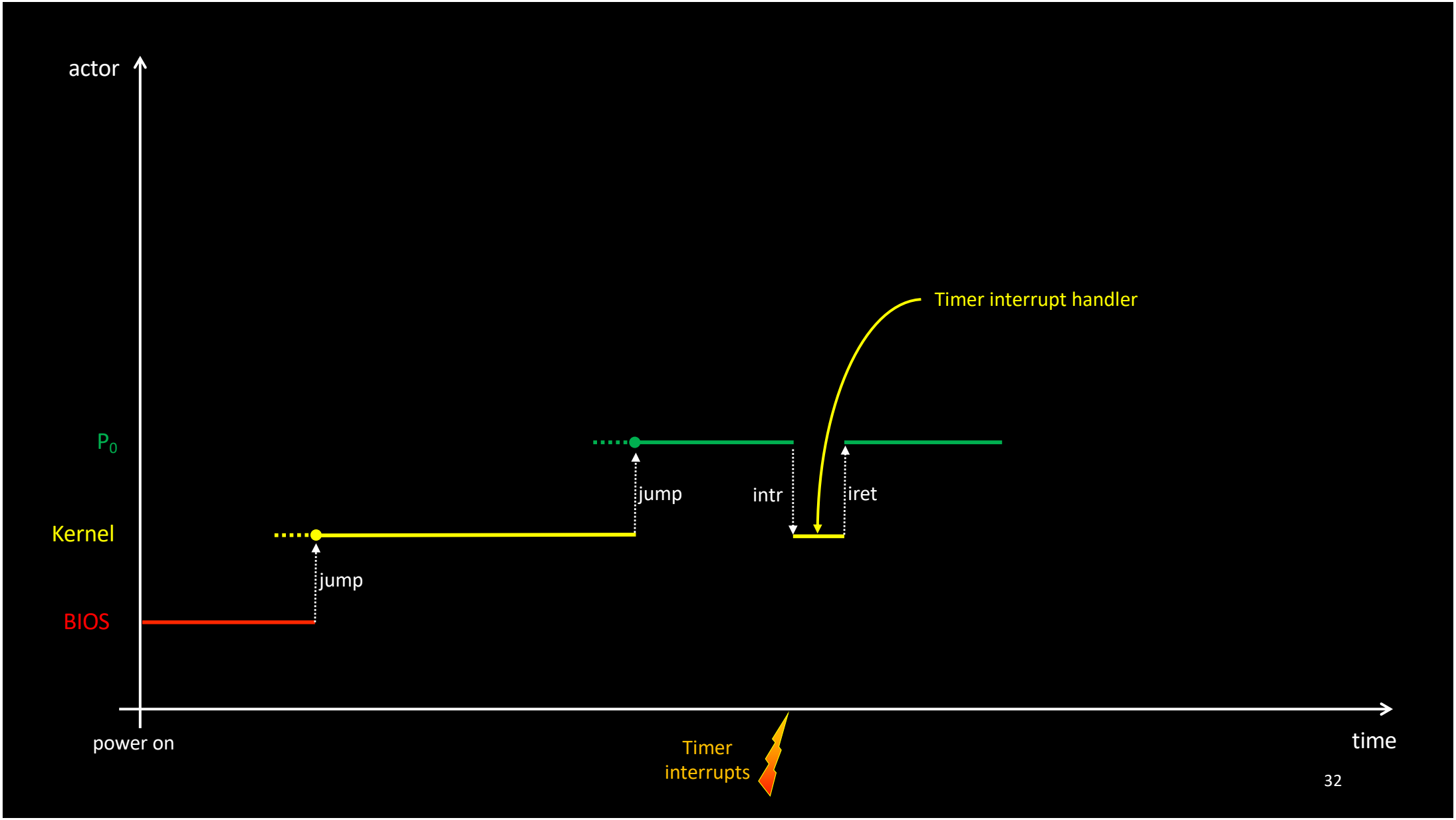
- Note: some interrupts are maskable
  - Timer interrupts, device-related interrupts (Mouse, Keyboard, Disk, Network Interface Card, etc.)
  - In some specific (and small) places, the kernel may need to temporarily postpone (≠ ignore) interrupts
    - `cli`: Clear Interrupts
    - `sti`: Set Interrupts
    - Used to avoid some "non-reentrant" code sections being interrupted
      - E.g. interrupting a code modifying a data structure and calling a handler reading the same data may result in faulty behavior
    - These instructions modify the *Interrupt Flag* only on the current CPU

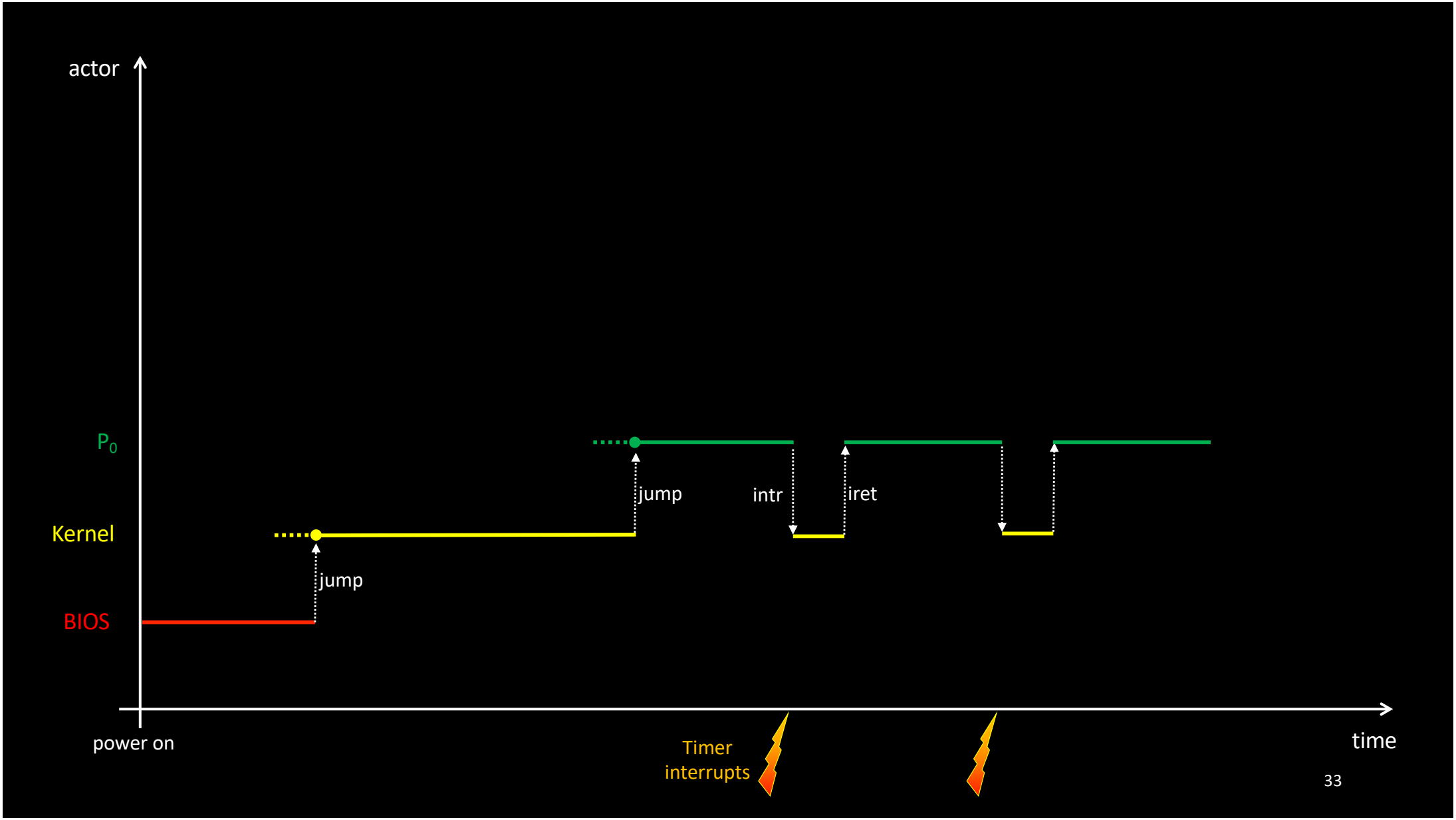
Implementing Time Sharing

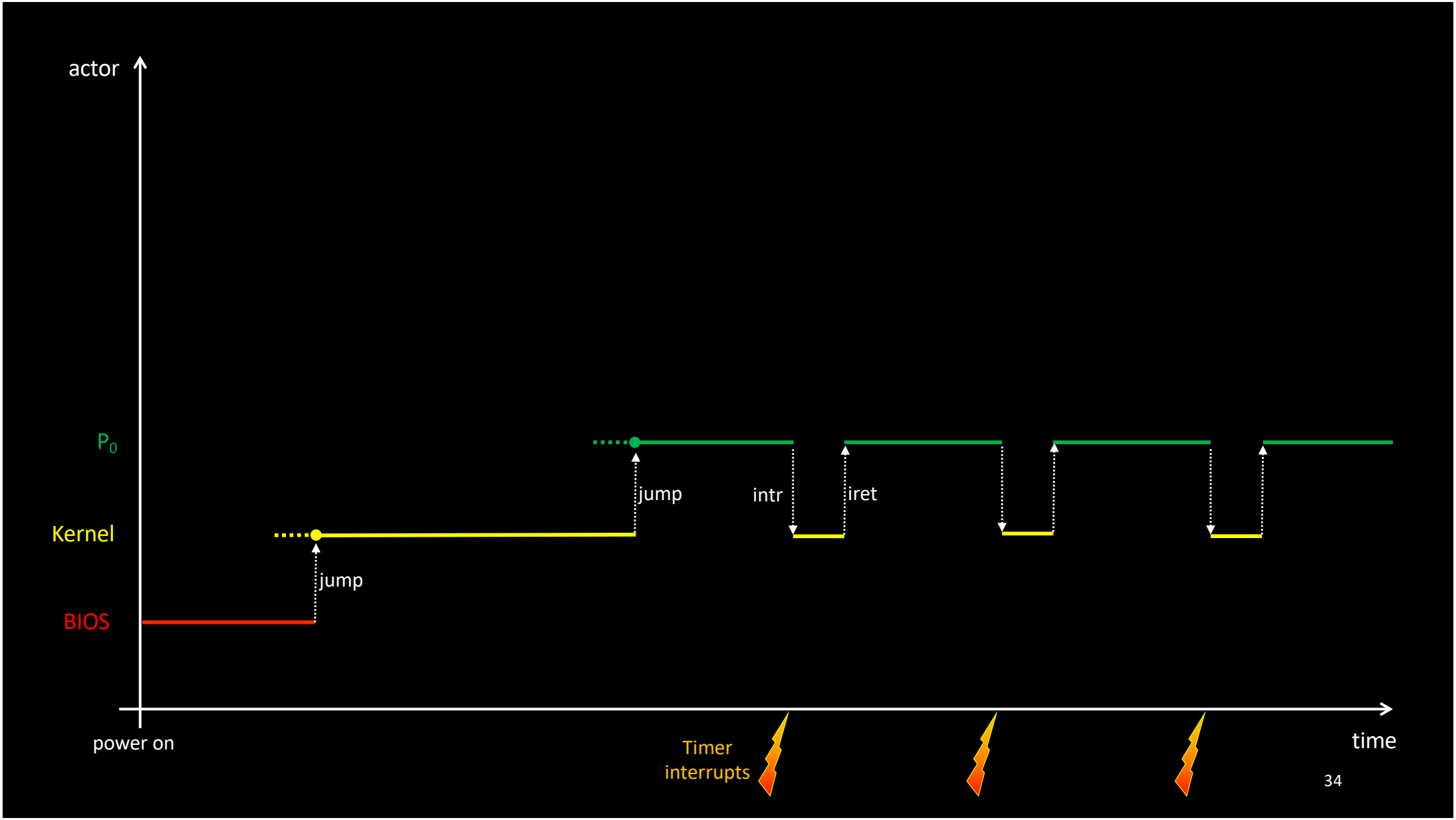
- To prevent processes running during unbounded periods, the kernel sets up a timer
  - A timer interrupt will be periodically triggered (~ 10ms)
  - This ensures that the associated kernel routine will be executed on a regular basis

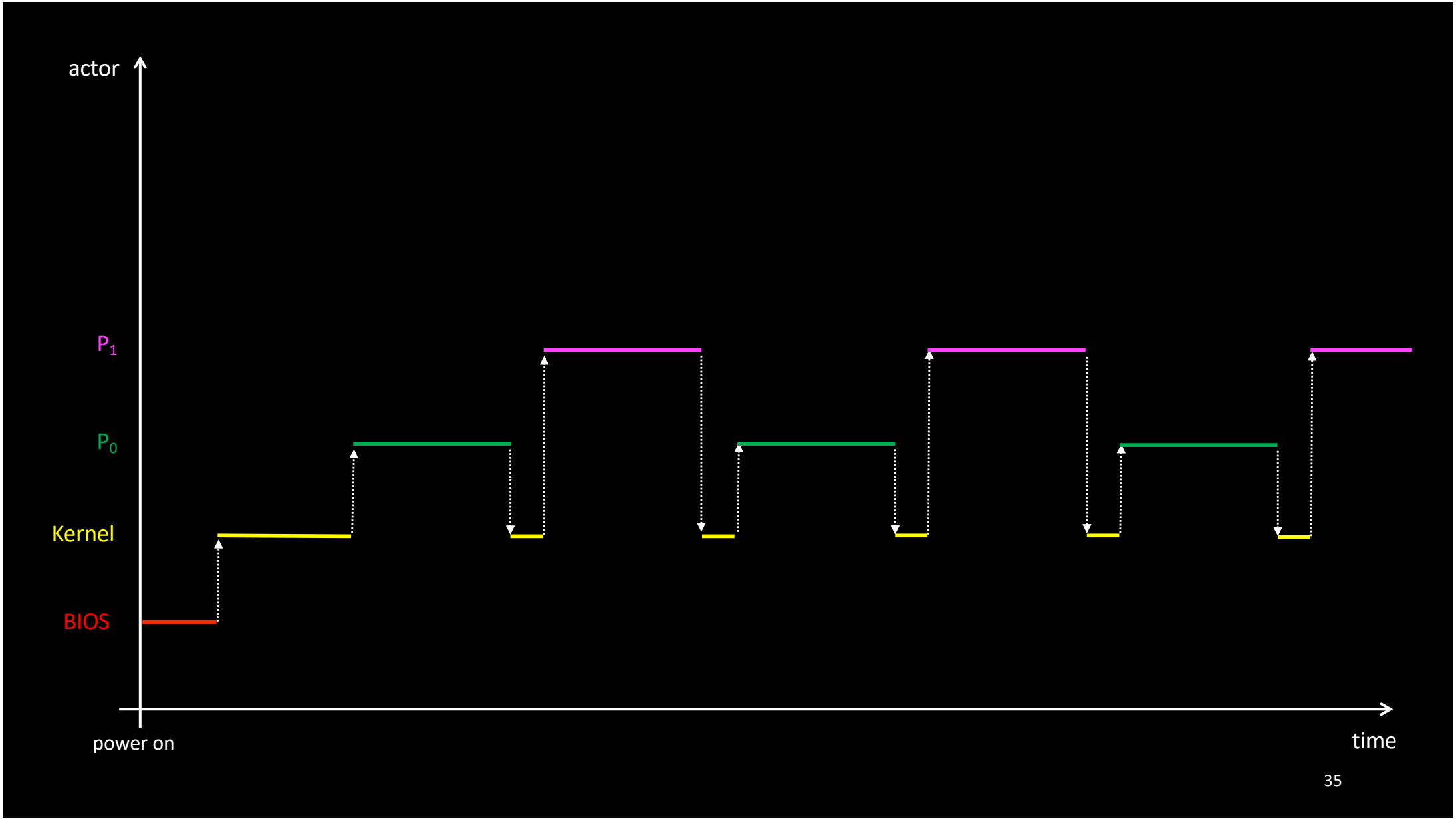
  - Of course, the *Interrupt Vector Table* must be initialized beforehand!


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


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


actor
P_0
Kernel
BIOS
jump
jump
intr
iret
power on
Timer
interrupts
time

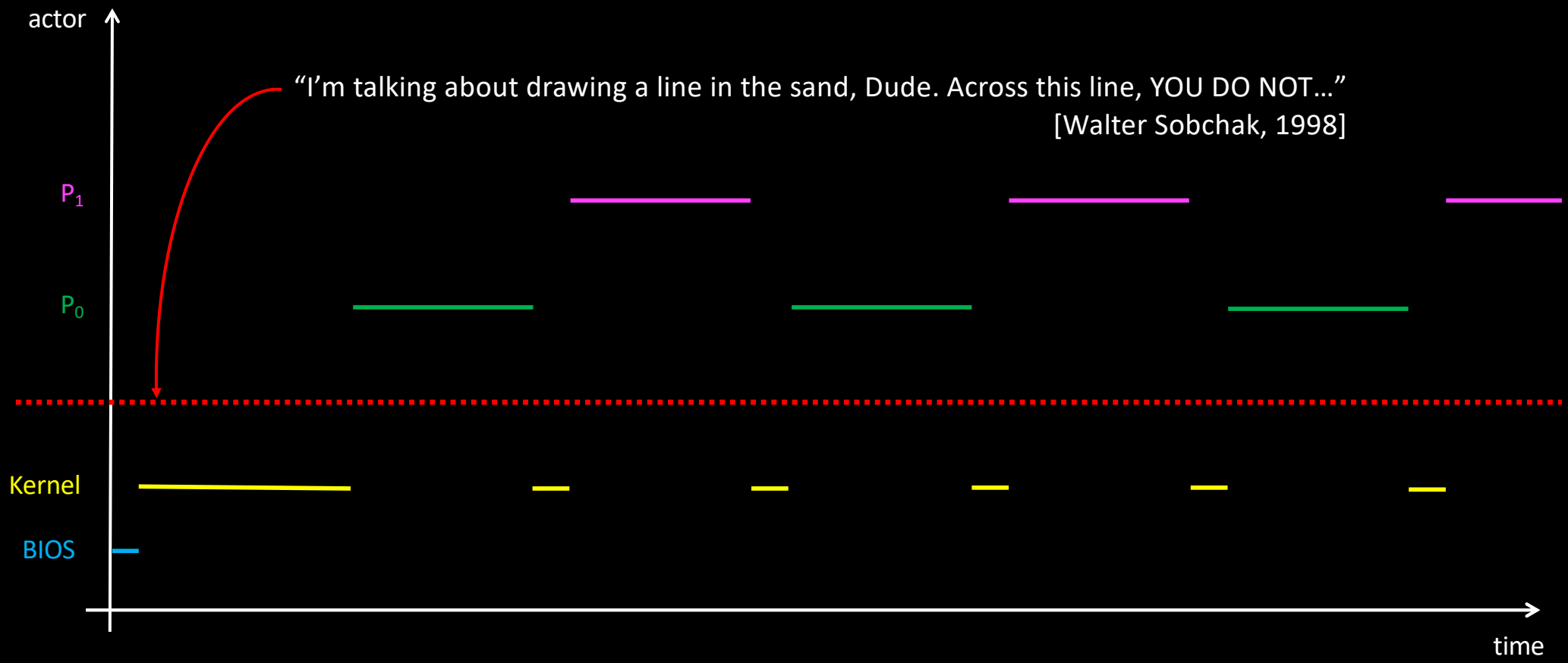

actor
P_0
Kernel
BIOS
jump
jump
intr
iret
power on
Timer
interrupts
time


actor
P_1
P_0
Kernel
BIOS
power on
time

Restricting Processes' privileges

- Ok : 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

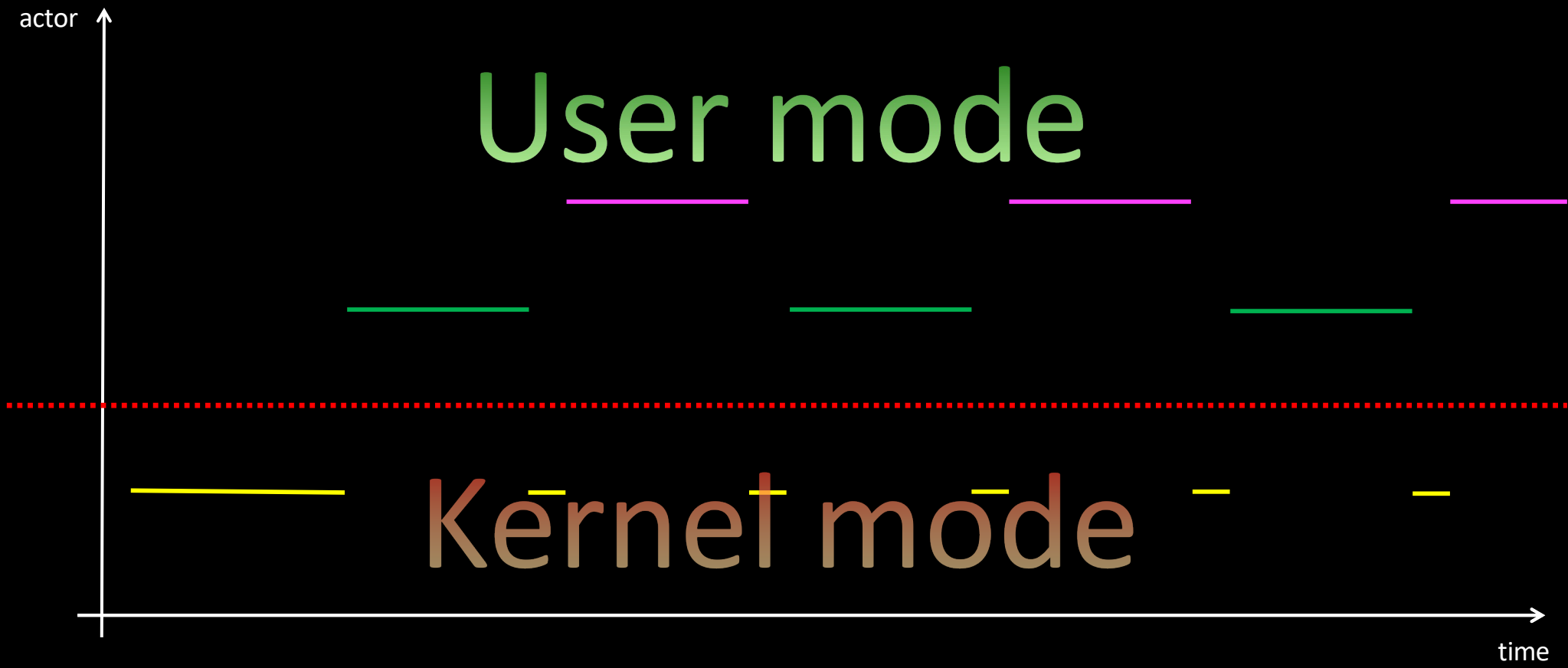
- We want to restrict what processes can do
  - Only the kernel should be almighty
- Let's assume we can establish a list of forbidden CPU instructions
  - (in Chapter "Memory Management", we'll see if this is sufficient to prevent arbitrary memory accesses)
- How to prevent processes from calling specific instructions?
  - Clever compiler?
  - Real-time scan of the program by the kernel?

Restricting Processes' privileges

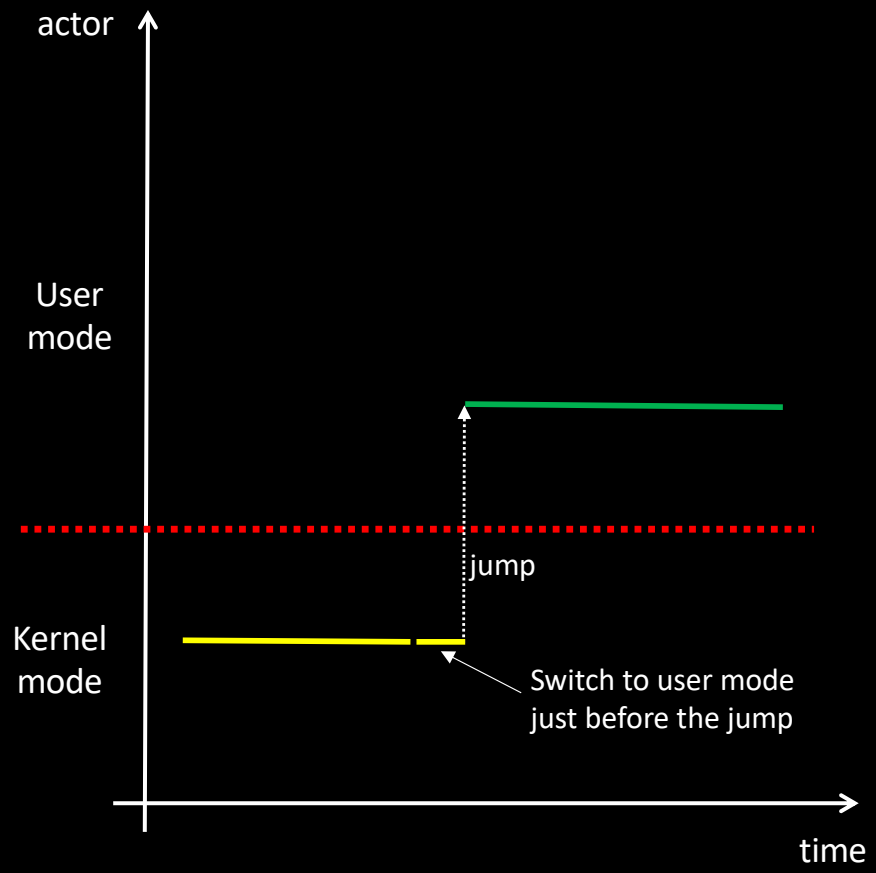
- This can be done only by the hardware, that is, the CPU
  - Privileged instructions are flagged
- The CPU can run in (at least) two different modes:
  - *User mode* (aka Protected mode) / *Kernel mode* (aka Real mode)
  - The current mode is stored in a control register
- In *user mode*, only a subset 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

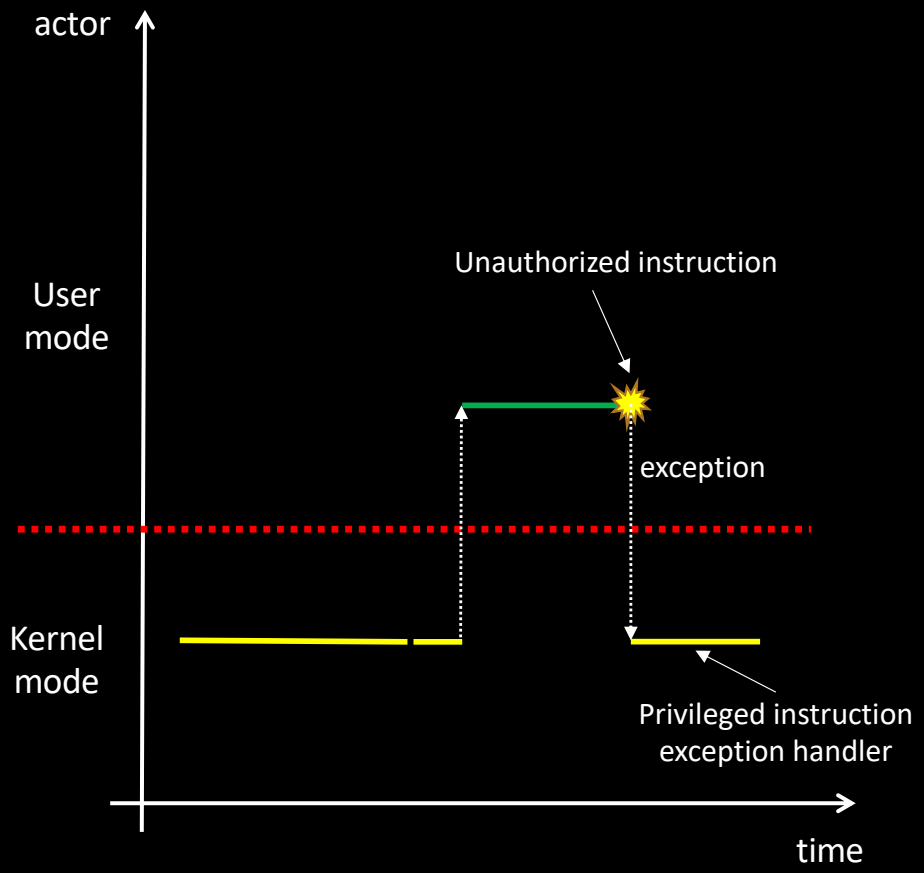


Restricting Processes' privileges



- 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

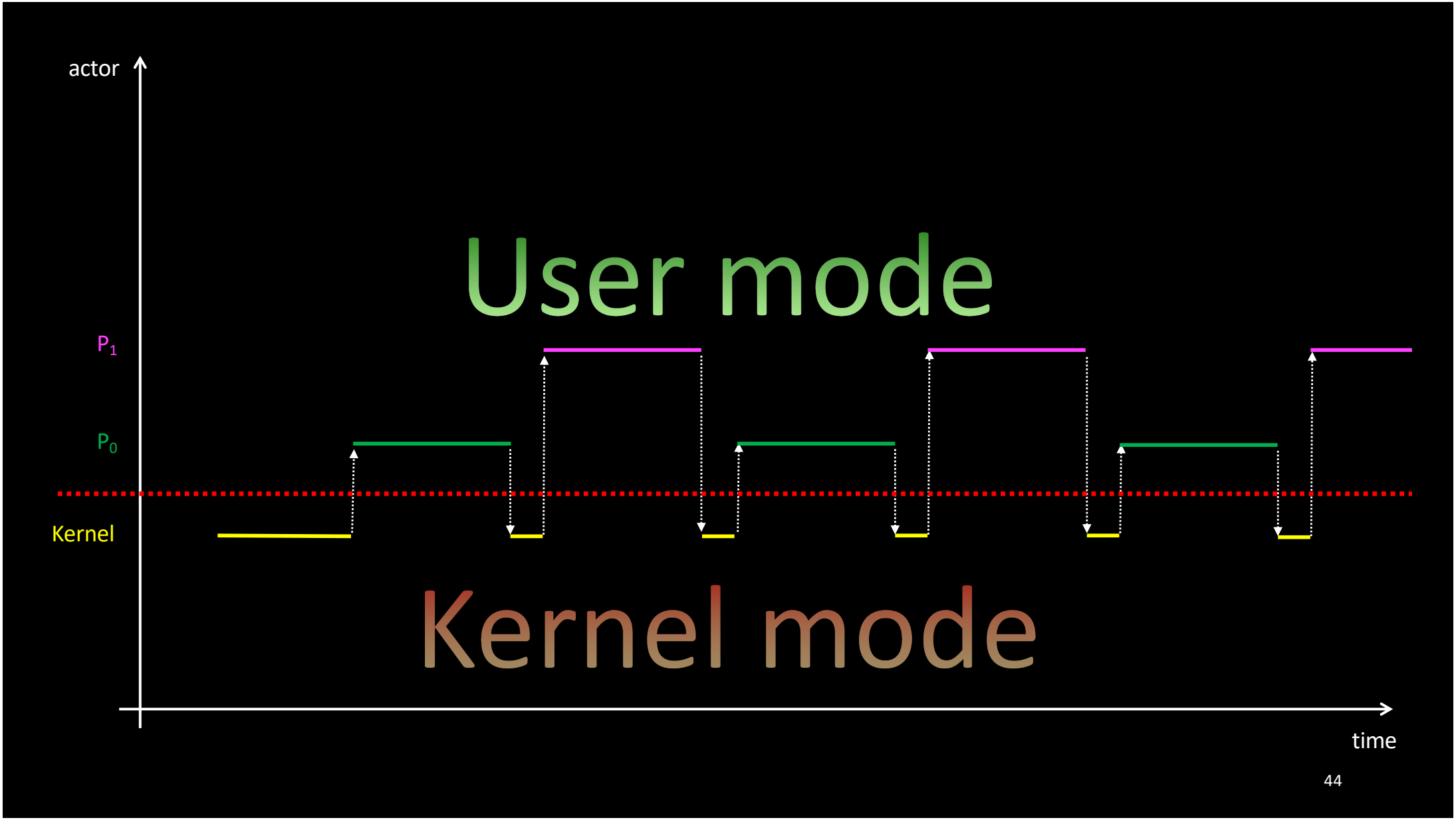
Restricting Processes' privileges



- The CPU wakes up (power on) in kernel mode
  - So 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
  - Explicit change to the control register is only possible in kernel mode
- However, interrupts automatically enter kernel mode
  - And iret (Interrupt RETurn) automatically goes back to previous mode


actor
User mode
P_1
P_0
Kernel
Kernel mode
time

Requesting privileges

- The kernel is safe
  - Processes cannot directly access the hardware
- 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 temporarily 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: *interrupts*!
- Let's use a specific instruction to raise a software interrupt
  - `int 80h` (old Linux 32bit kernels)
  - `syscall` (Linux 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 ”open” function in libc

```
int open(const char *pathname,
         int flags, mode_t mode)
{
  mov __NR_open, %eax
  syscall
  ret
}
```

System calls

- Example:
  - C implementation of the file ”write” function in libc

```
ssize_t write(int fd,
              const void *buf,
              size_t count)
{
  mov __NR_write, %eax
  syscall
  ret
}
```

System calls

- On the kernel side, a table contains the addresses of routines implementing systems calls
  - `sys_open`, `sys_write`, `sys_read`, `sys_fork`, etc.
  - The syscall interrupt handler uses the number found in %eax 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

```
ssize_t write(int fd,
              const void *buf,
              size_t count)

{
  mov __NR_write, %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?
  - OK… If you really care, then
    - You can run your program under the `strace` utility
    - Or you can disassemble the very first instructions to check for the syscall CPU instruction

System calls & library calls

- Example
  - `write` is a system call
  - `printf` is a pure user-level function, which relies on `write`

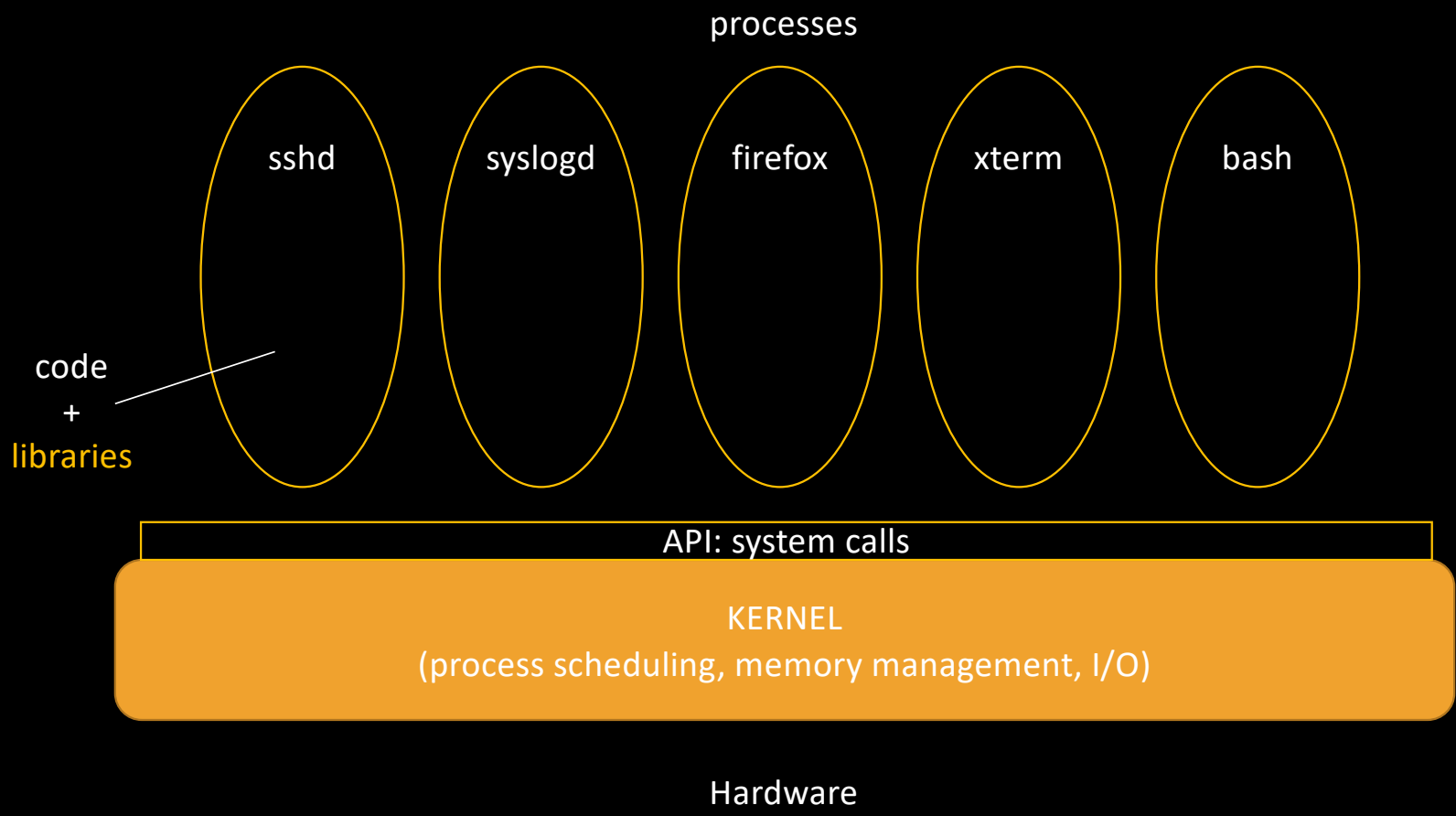
- printf (“Hello\n number %d\n”, 6*111) leads to:
  - write (STDOUT_FILENO, “Hello\n number 666\n”, 18);

System calls & library calls

- Example
  - `write` is a system call
  - `printf` is a pure user-level function, which relies on `write`

- printf (“Hello\n number %d\n”, 6*111) leads to:
  - ~~write (STDOUT_FILENO, “Hello\n number 666\n”, 18);~~
  - write (STDOUT_FILENO, “Hello\n”, 6);
  - write (STDOUT_FILENO, “ number 666\n”, 12);

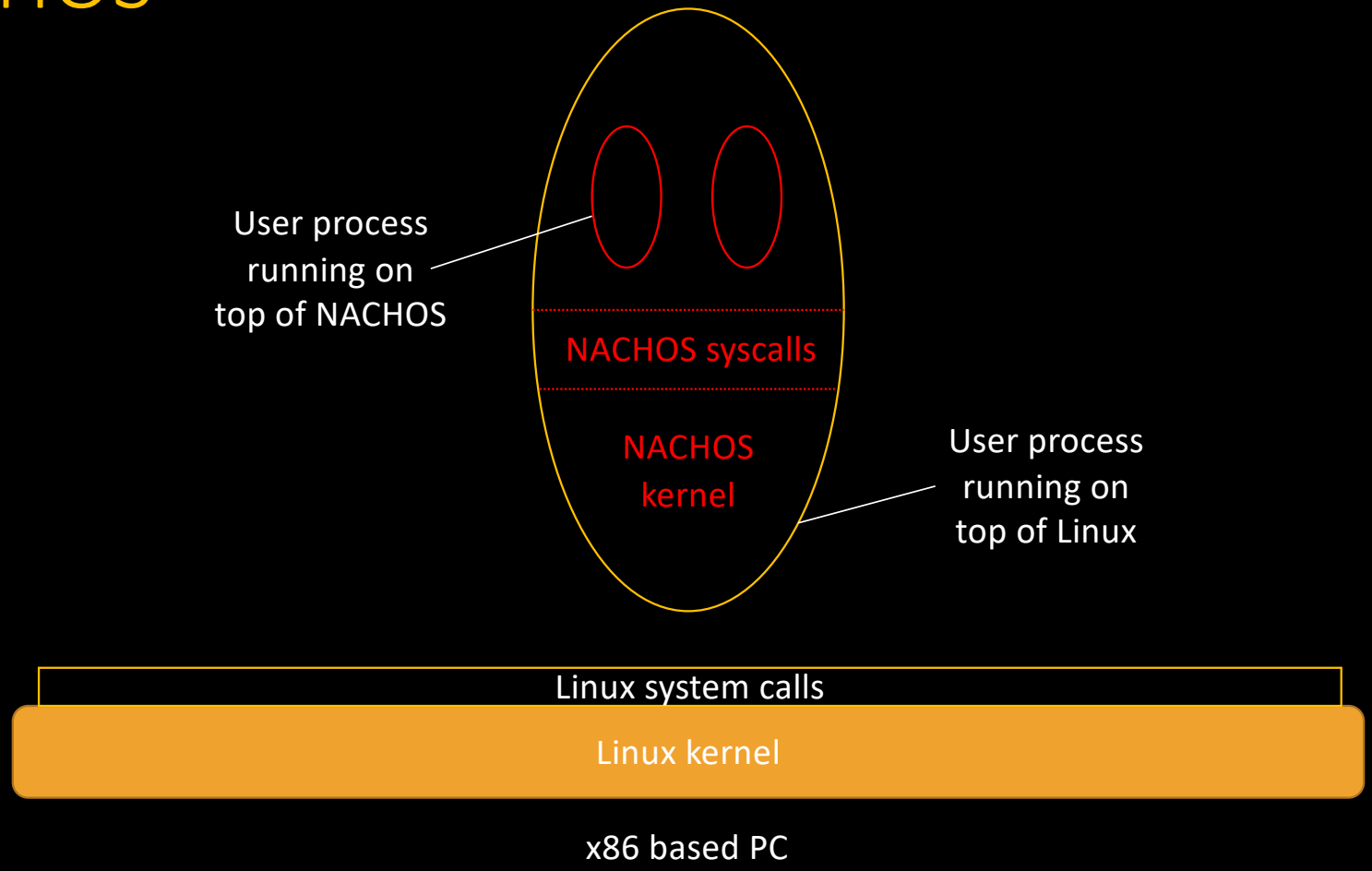
Structure of an OS



NACHOS

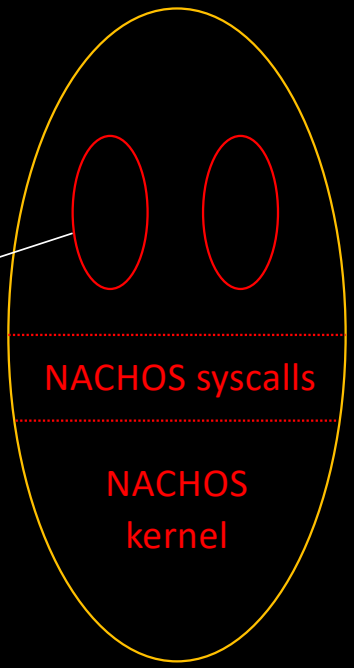
- The NACHOS simulator runs as a regular Linux process
  - Stopping NACHOS (if things go wrong) is harmless
  - Debugging NACHOS is as simple as using `gdb` ☺
- The NACHOS kernel is minimal
  - It basically features a Process Scheduler
- NACHOS can load and execute user programs
  - How can that be?

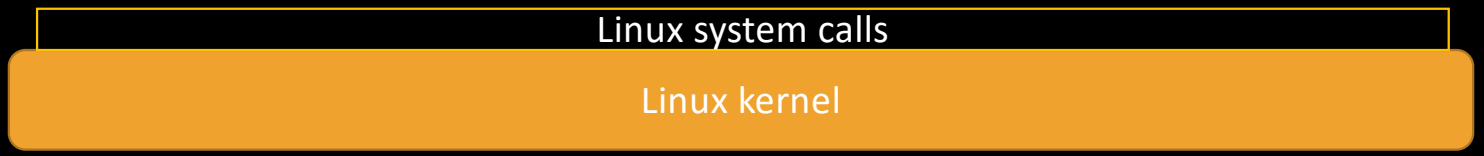
NACHOS


User process
running on
top of NACHOS
NACHOS syscalls
NACHOS
kernel
User process
running on
top of Linux
Linux system calls
Linux kernel
x86 based PC

NACHOS

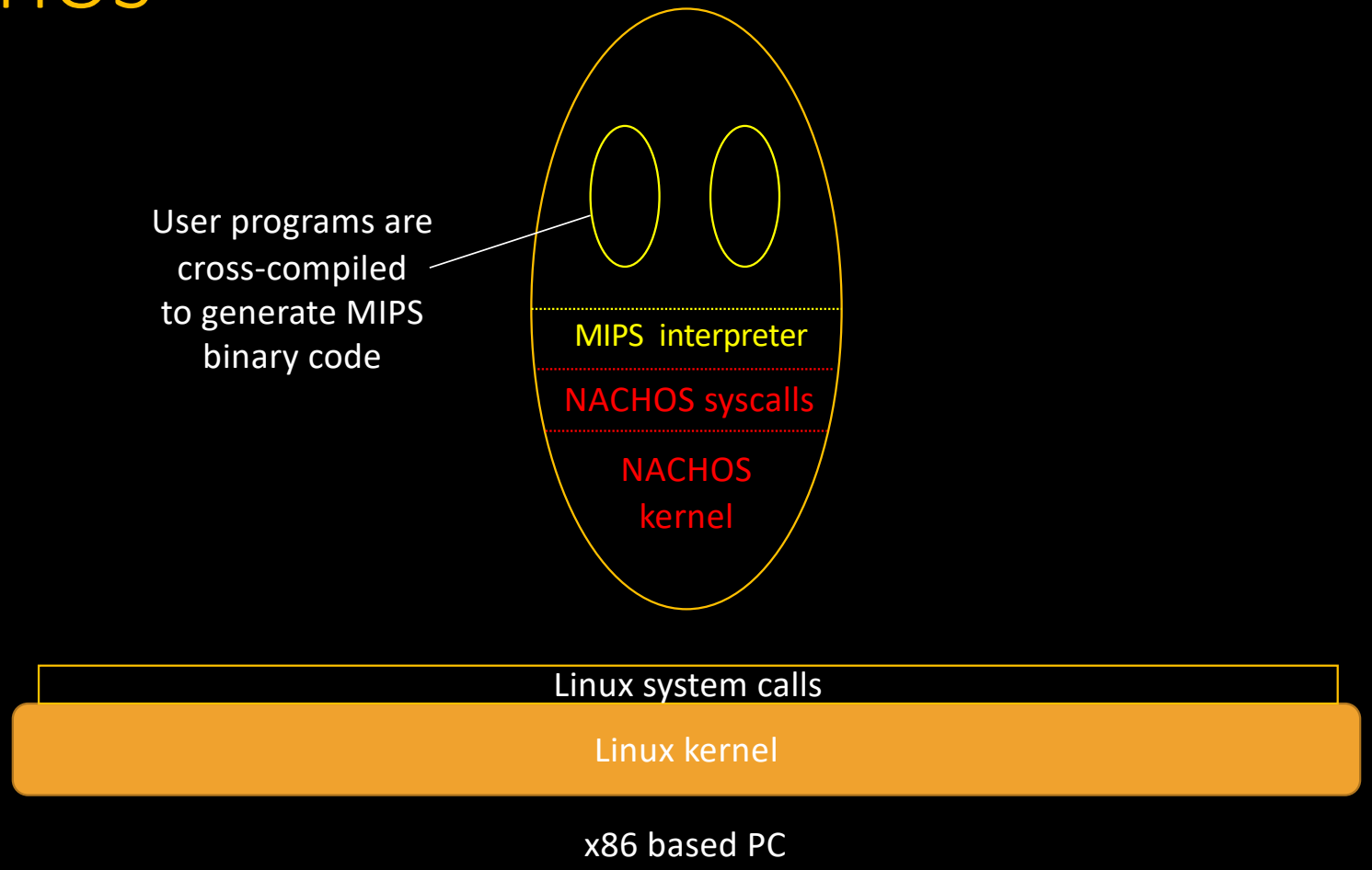
If this code raises an exception,
it will trap into the Linux kernel !
☹





x86 based PC

NACHOS


User programs are cross-compiled to generate MIPS binary code
MIPS interpreter
NACHOS syscalls
NACHOS kernel
Linux system calls
Linux kernel
x86 based PC

NACHOS

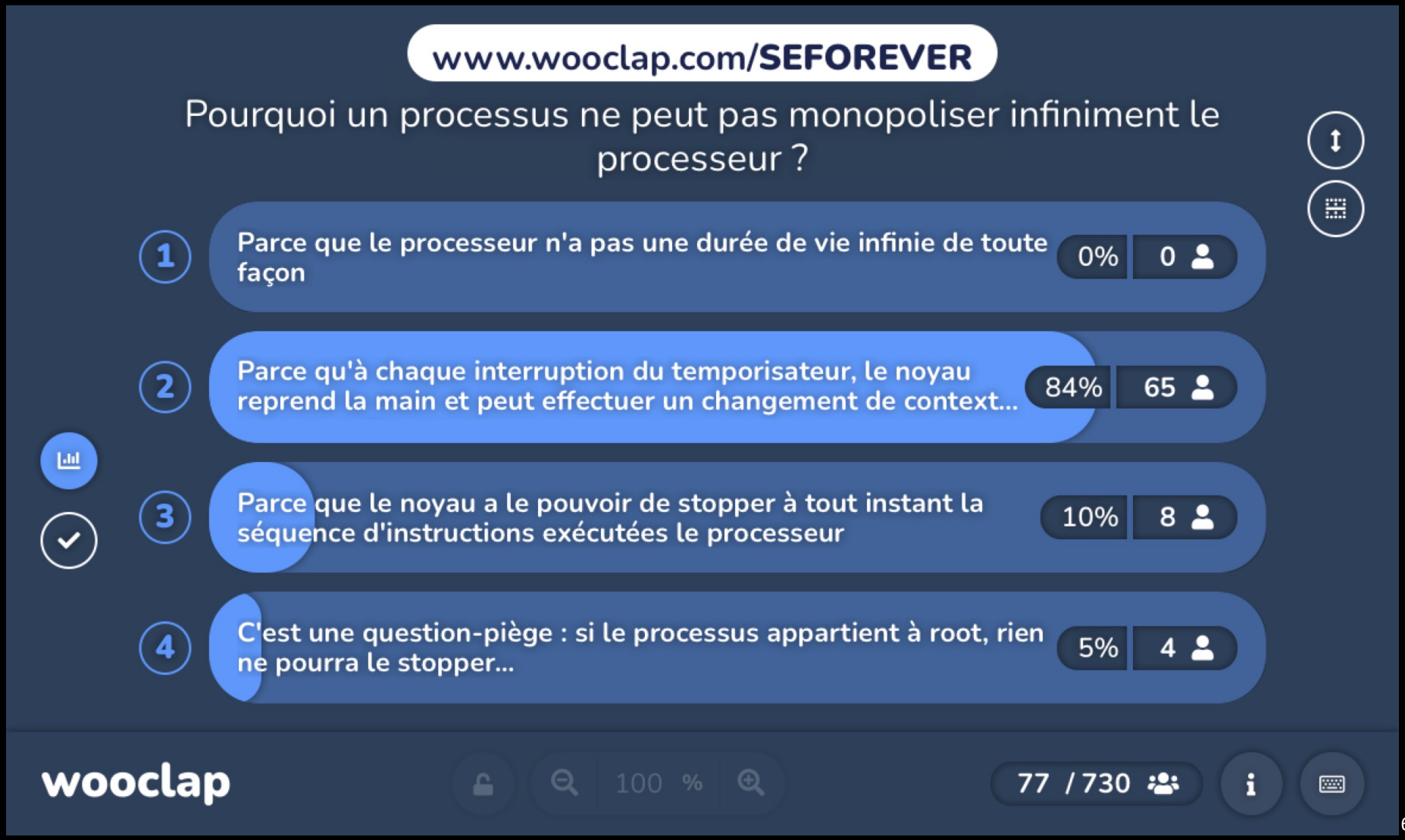
- As a result, NACHOS also simulates the hardware
  - (Single) CPU
  - Memory
  - Timer
  - I/O console
- The corresponding code should never be modified

Structure of directories

- `nachos/code/`
  - Only a convenient (?) place to trigger the whole compilation process
- `nachos/code/threads/`
  - ‘make’ will only build a simple auto-test kernel
- `nachos/code/userprog/`
  - ‘make’ will build a version of NACHOS able to run user programs
  - E.g.

```
./nachos -x ../test/halt
```

- `nachos/code/test/`
  - Contains the source files of user programs
  - See `halt.c` for instance


www.wooclap.com/SEFOREVER
Pourquoi un processus ne peut pas monopoliser infiniment le processeur ?
1
Parce que le processeur n'a pas une durée de vie infinie de toute façon
0%
0
2
Parce qu'à chaque interruption du temporisateur, le noyau reprend la main et peut effectuer un changement de context...
84%
65
3
Parce que le noyau a le pouvoir de stopper à tout instant la séquence d'instructions exécutées le processeur
10%
8
4
C'est une question-piège : si le processus appartient à root, rien ne pourra le stopper...
5%
4
wooclap
100 %
77 / 730

Additional resources available on http://gforgeron.gitlab.io/se/