Operating Systems: Process Management

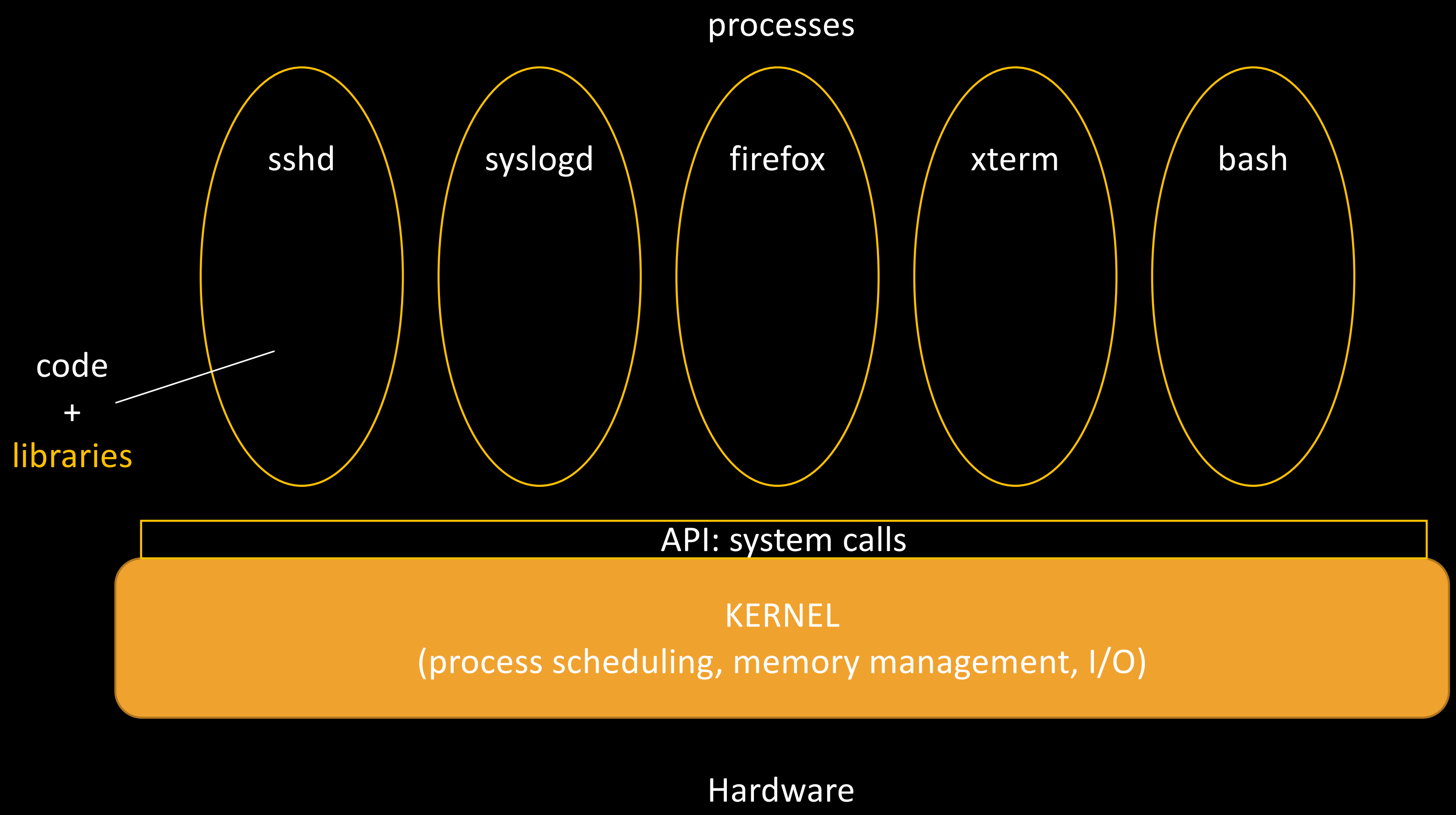
Raymond Namyst

Dept. of Computer Science

University of Bordeaux, France

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

Structure of an OS



Processes

- Processes are lively instances of programs
  - Program = binary code stored on disk
  - Multiple processes can run the same program independently

- Process = Address Space + Execution Context
  - Address space
    - Set of visible memory addresses
      - Code, Data, Heap, Stack, Shared Libraries, etc.
  - Execution Context
    - Stack + content of processor registers

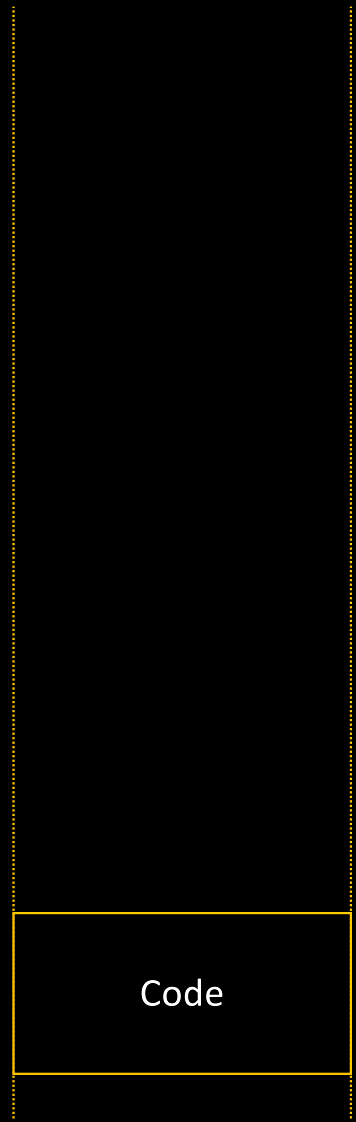
Address Space

- Typically composed of distinct memory regions
  - A region being a contiguous range of valid addresses

Address Space

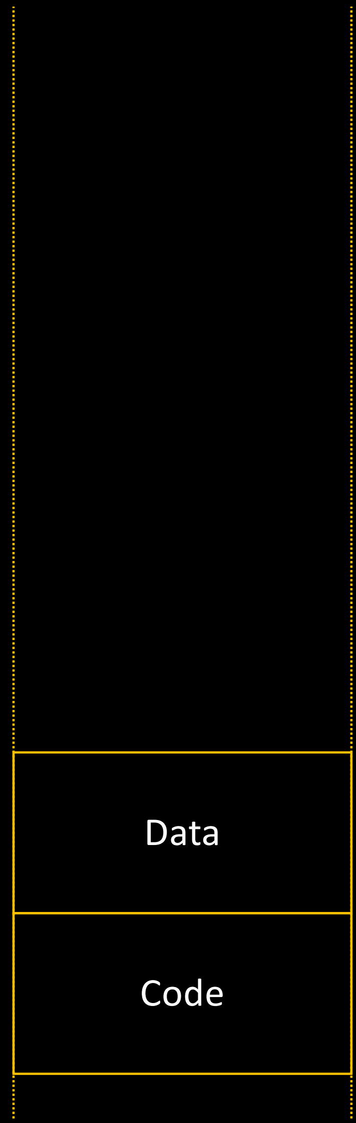
- Typically composed of the following regions
  - Code
    - (aka text segment)
    - Contains executable instructions
    - Usually, a read-only region

    - It also hosts constants
      - E.g. constant strings
        - “hello world!”



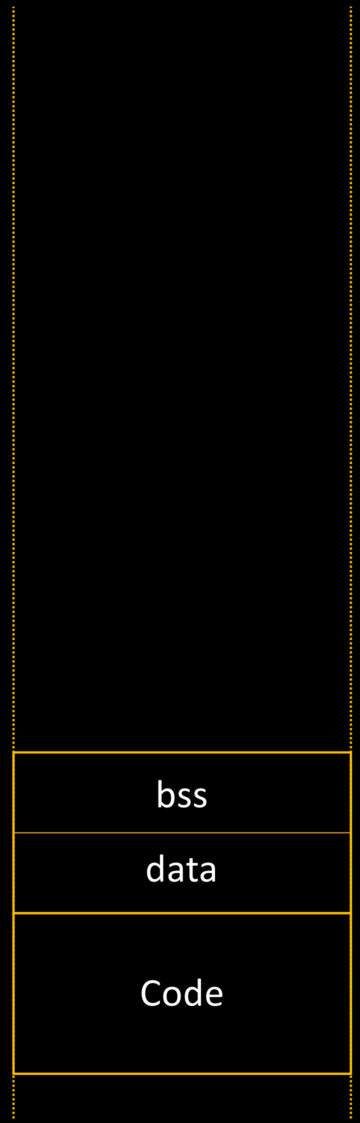
Address Space

- Typically composed of the following regions
  - Code
  - Data
    - Allocation of static variables
      - `int i;`



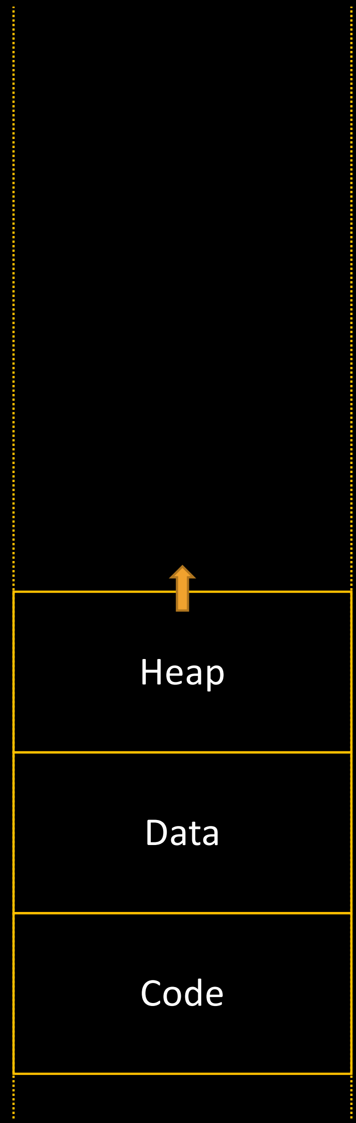
Address Space

- Typically composed of the following regions
  - Code
  - Data
    - Allocation of static variables
    - Actually two segments
      - Initialized data (data segment)
        - `float pi = 3.1415;`
        - Stored in object file
      - Uninitialized data (bss segment)
        - `int i;`
        - Only segment size is stored in object file



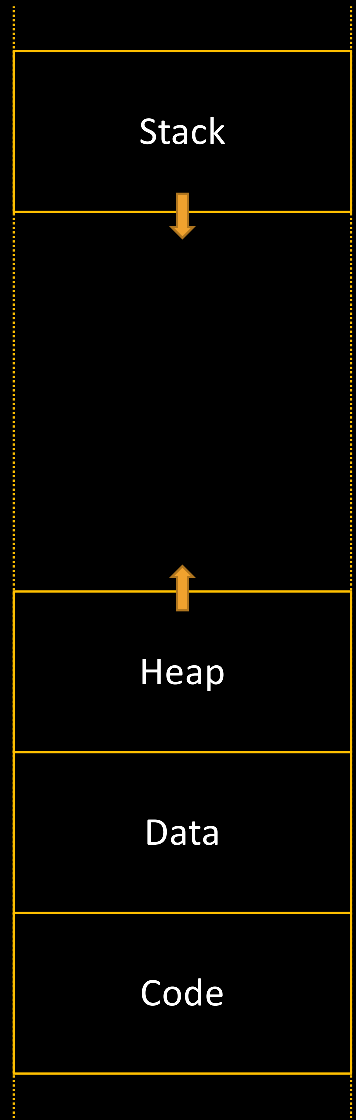
Address Space

- Typically composed of the following regions
  - Code
  - Data
  - Heap
    - Dynamic allocations
      - `malloc/free`
    - Managed by libc
      - Dynamic expansion
      - OS cannot (always) detect accesses outside malloc'ed buffers…



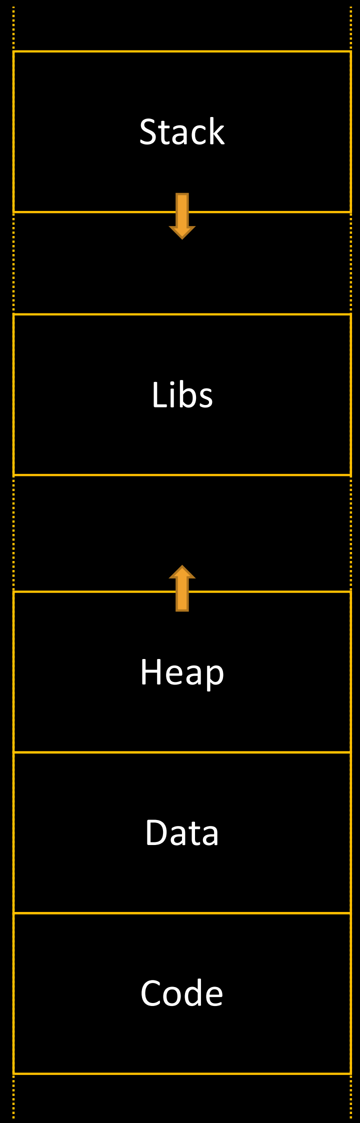
Address Space

- Typically composed of the following regions
  - Code
  - Data
  - Heap
  - Stack
    - Allocation of function parameters and local variables
    - Automatic growth
    - 8 MiB default limit under Linux



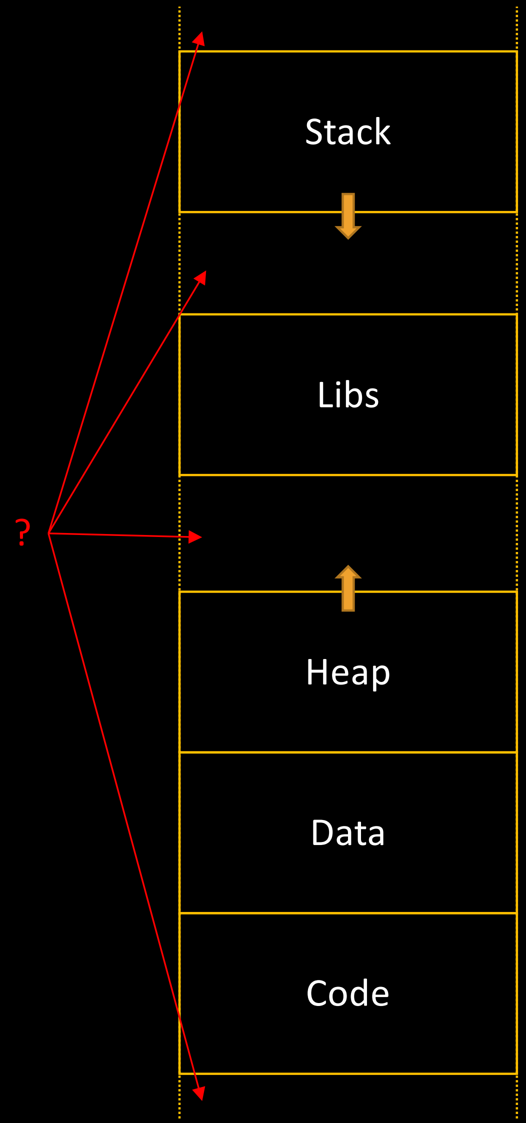
Address Space

- Typically composed of the following regions
  - Code
  - Data
  - Heap
  - Stack
  - Shared Libraries
    - libc, libm, libGL, etc.
    - Mapped on demand



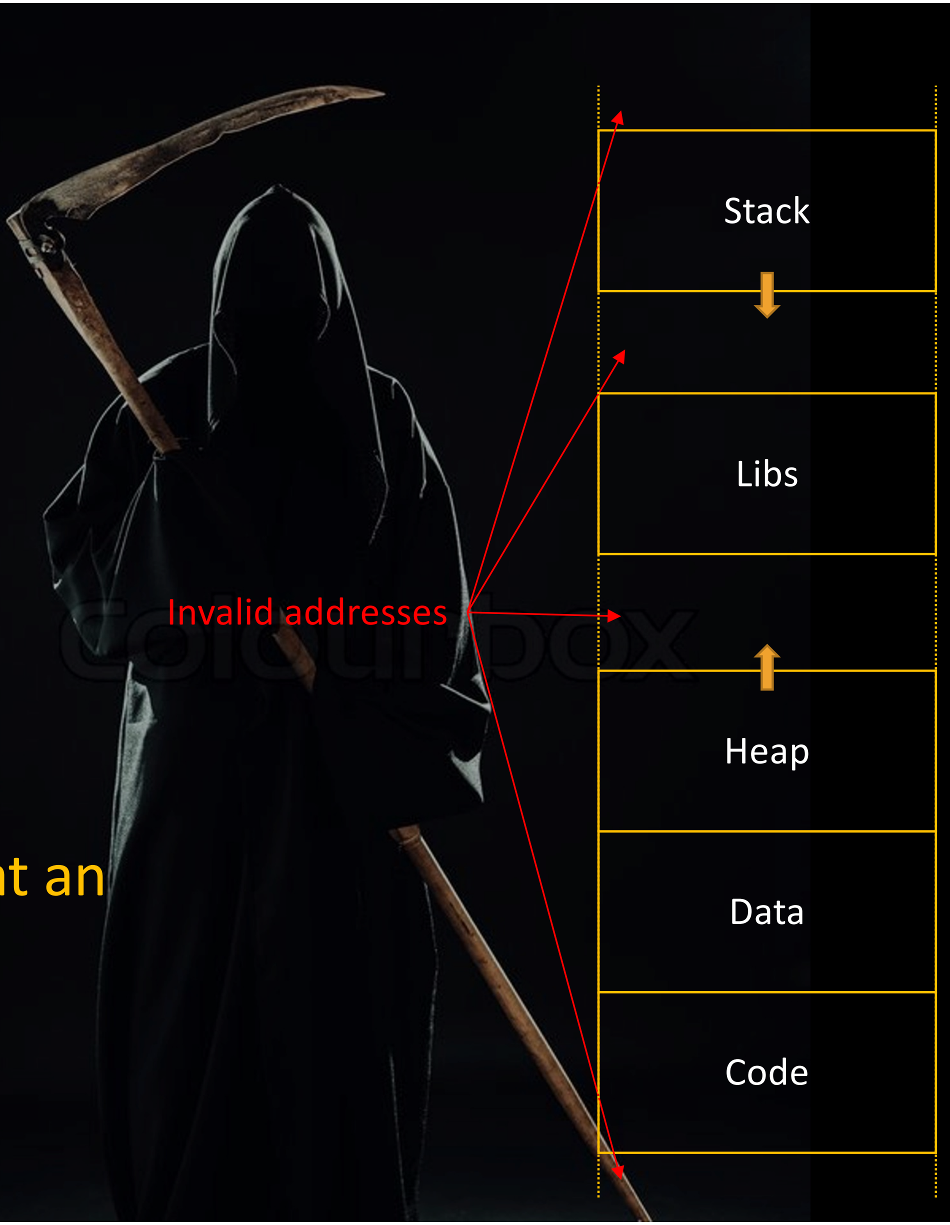
Address Space

- Typically composed of the following regions
  - Code
  - Data
  - Heap
  - Stack
  - Shared Libraries
    - libc, libm, libGL, etc.
    - Mapped on demand



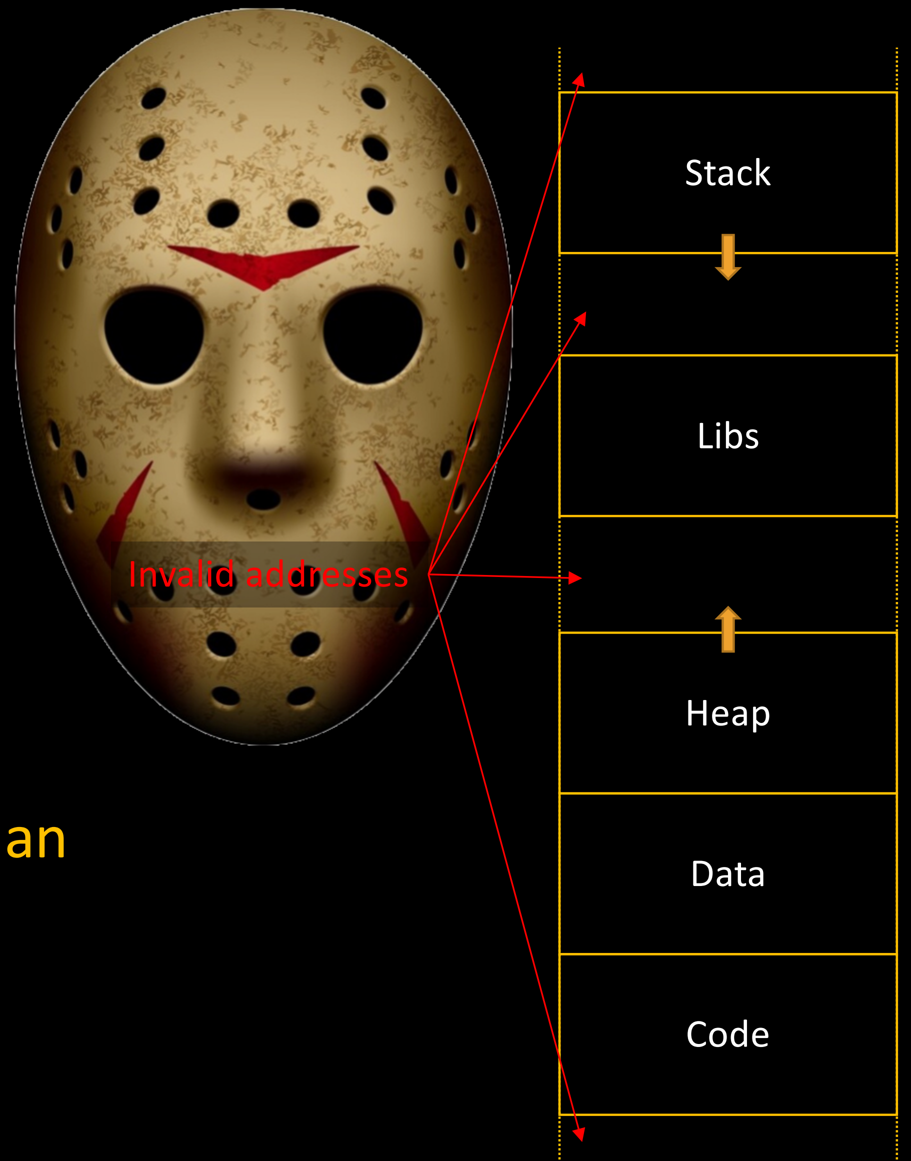
Address Space

- Typically composed of the following regions
  - Code
  - Data
  - Heap
  - Stack
  - Shared Libraries
- Attempt to access memory at an invalid address leads to a Segmentation Fault



Address Space

- Typically composed of the following regions
  - Code
  - Data
  - Heap
  - Stack
  - Shared Libraries
- Attempt to access memory at an invalid address leads to a Segmentation Fault

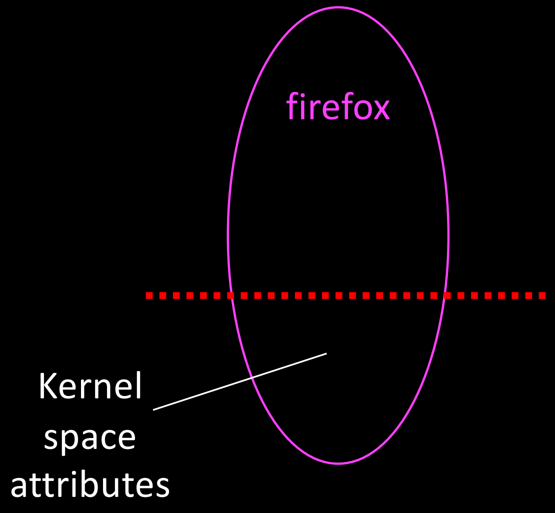


Inspecting Memory Regions under Linux

```
[jolicoeur] cat /proc/self/maps
    55ad0226e000-55ad02276000 r-xp 00000000 08:01 1573289                /bin/cat
    55ad02475000-55ad02476000 r--p 00007000 08:01 1573289                /bin/cat
    55ad02476000-55ad02477000 rw-p 00008000 08:01 1573289                /bin/cat
    55ad02c0d000-55ad02c2e000 rw-p 00000000 00:00 0                      [heap]
    7f9a1646b000-7f9a1669e000 r--p 00000000 08:01 7079259                /usr/lib/locale/locale-archive
    7f9a166a3000-7f9a16838000 r-xp 00000000 08:01 8131225                /lib/x86_64-linux-gnu/libc-2.24.so
    7f9a16838000-7f9a16a38000 ---p 00195000 08:01 8131225                /lib/x86_64-linux-gnu/libc-2.24.so
    7f9a16a38000-7f9a16a3c000 r--p 00195000 08:01 8131225                /lib/x86_64-linux-gnu/libc-2.24.so
    7f9a16a3c000-7f9a16a3e000 rw-p 00199000 08:01 8131225                /lib/x86_64-linux-gnu/libc-2.24.so
    7f9a16a43000-7f9a16a66000 r-xp 00000000 08:01 8128192                /lib/x86_64-linux-gnu/ld-2.24.so
    7f9a16c66000-7f9a16c67000 r--p 00023000 08:01 8128192                /lib/x86_64-linux-gnu/ld-2.24.so
    7f9a16c67000-7f9a16c68000 rw-p 00024000 08:01 8128192                /lib/x86_64-linux-gnu/ld-2.24.so
    7ffeaea77000-7ffeaea98000 rw-p 00000000 00:00 0                      [stack]
```

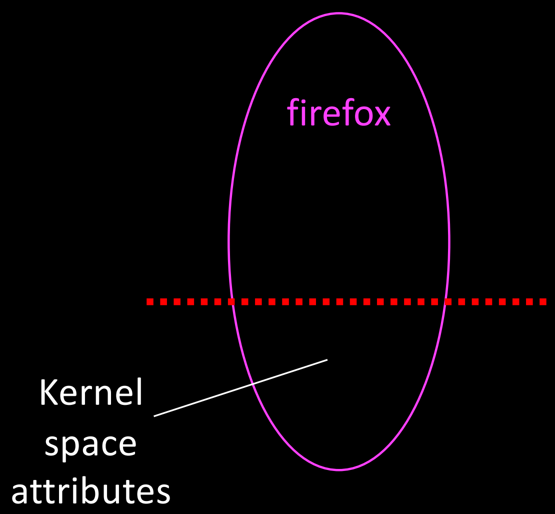
Process Attributes

- In addition to Address Space description, the kernel stores the following information about each process:
  - Process ID (pid)
  - Priority
  - User ID (real/effective)
  - File descriptor table
  - Signal handling table
  - Space for registers backup
  - Etc.

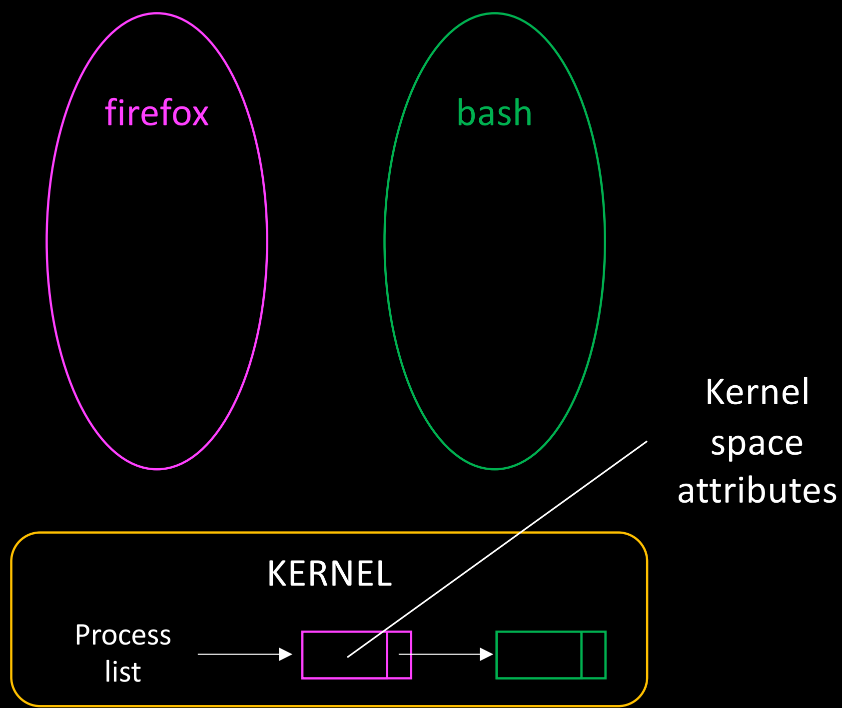


Process Attributes

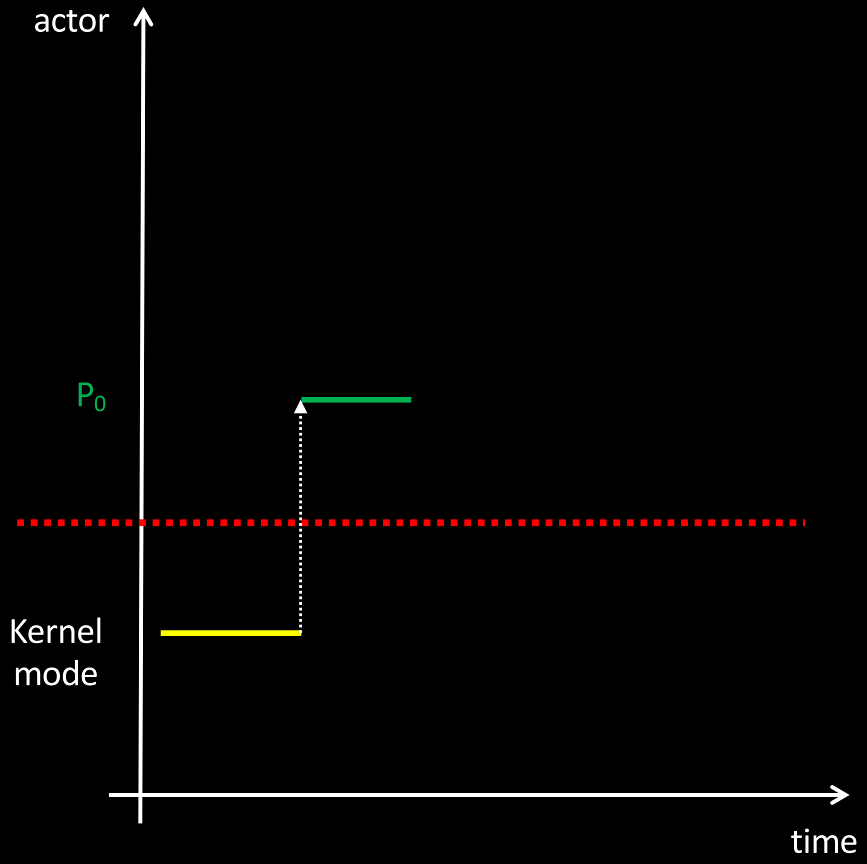
Processes can be represented this way:



But reality is (obviously) more like:

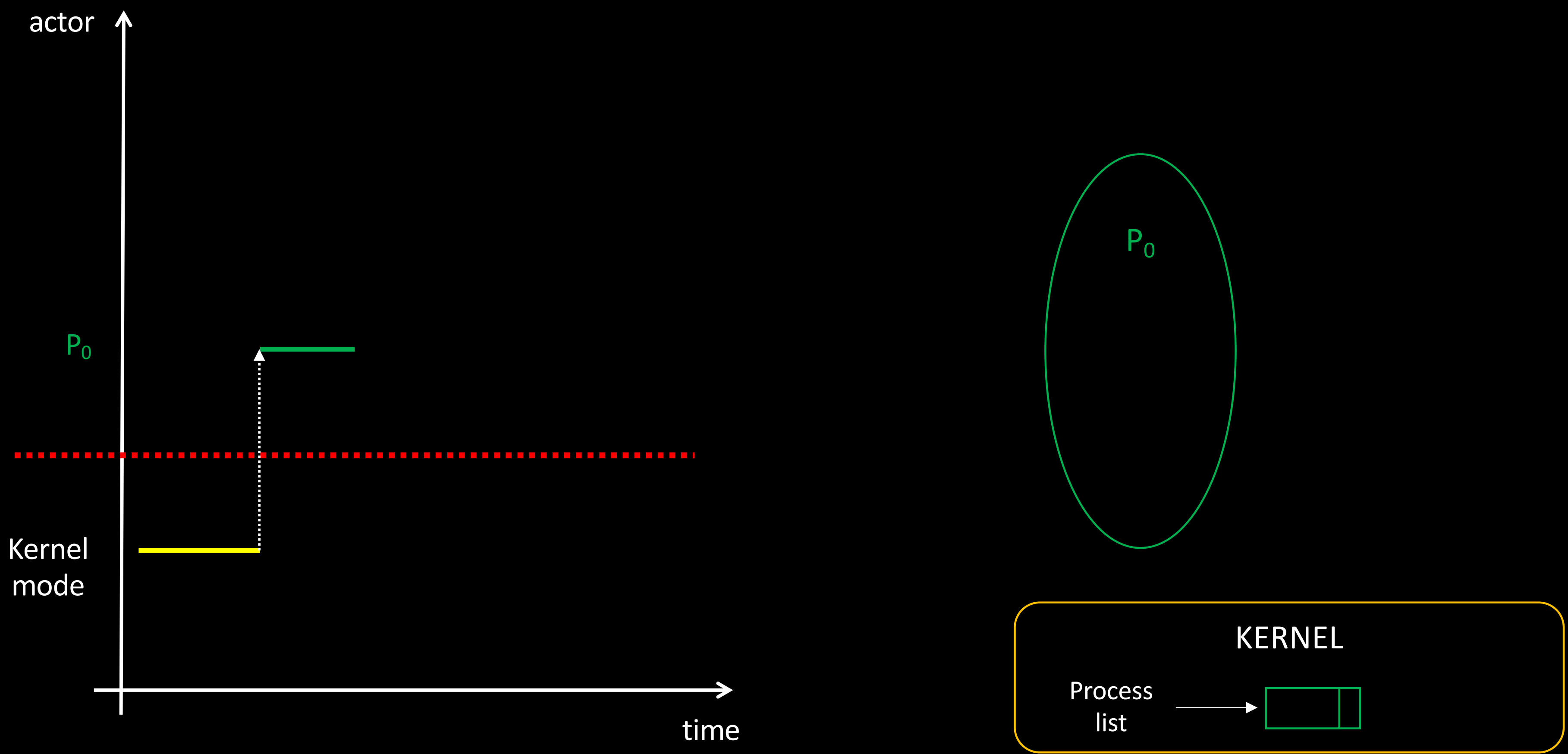


Process Creation

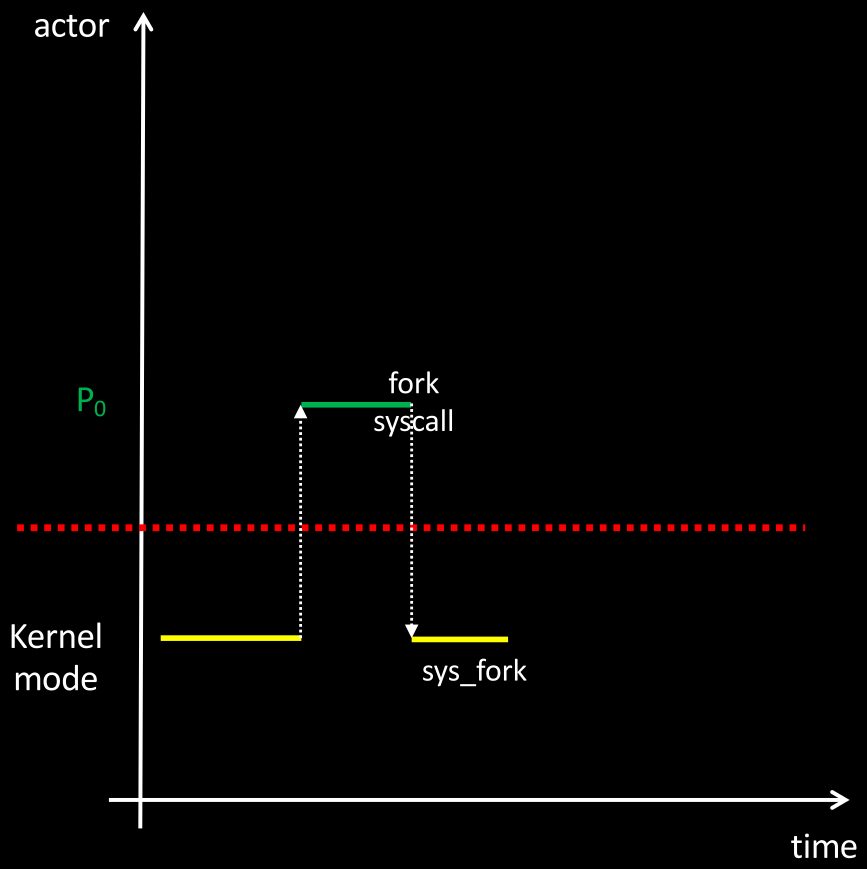


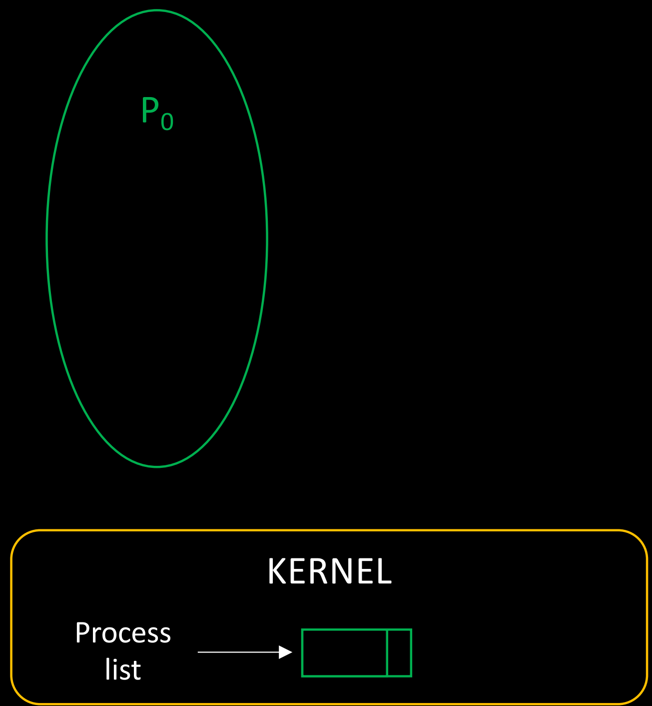
- The Kernel originally spawns one process (P_0)
  - This process will in turn create several processes (background DAEMONs)
    - Using a system call (what else?)

Process Creation

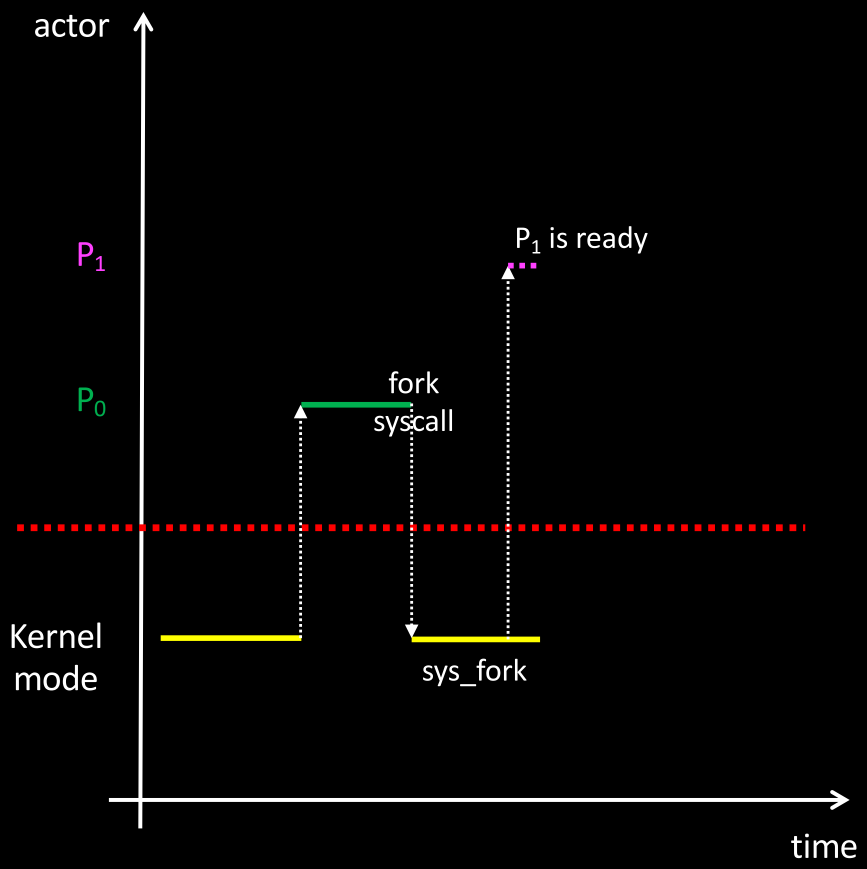


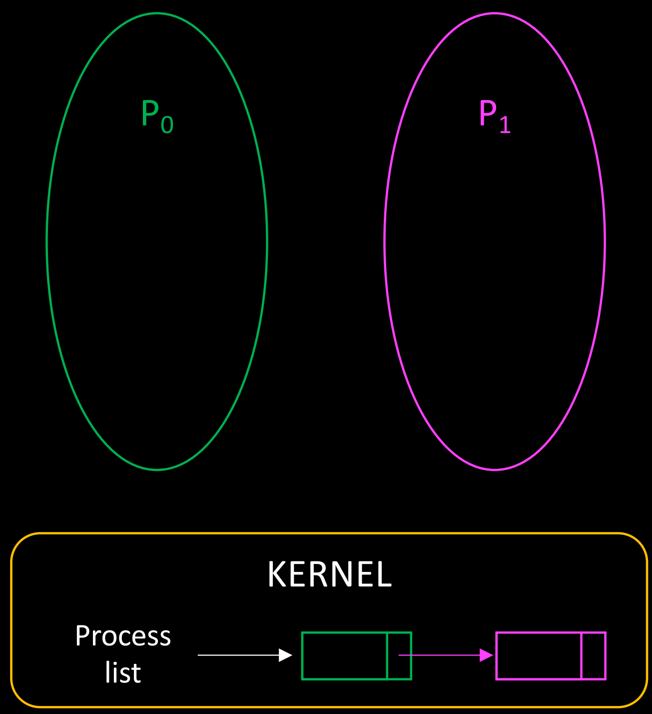
Process Creation



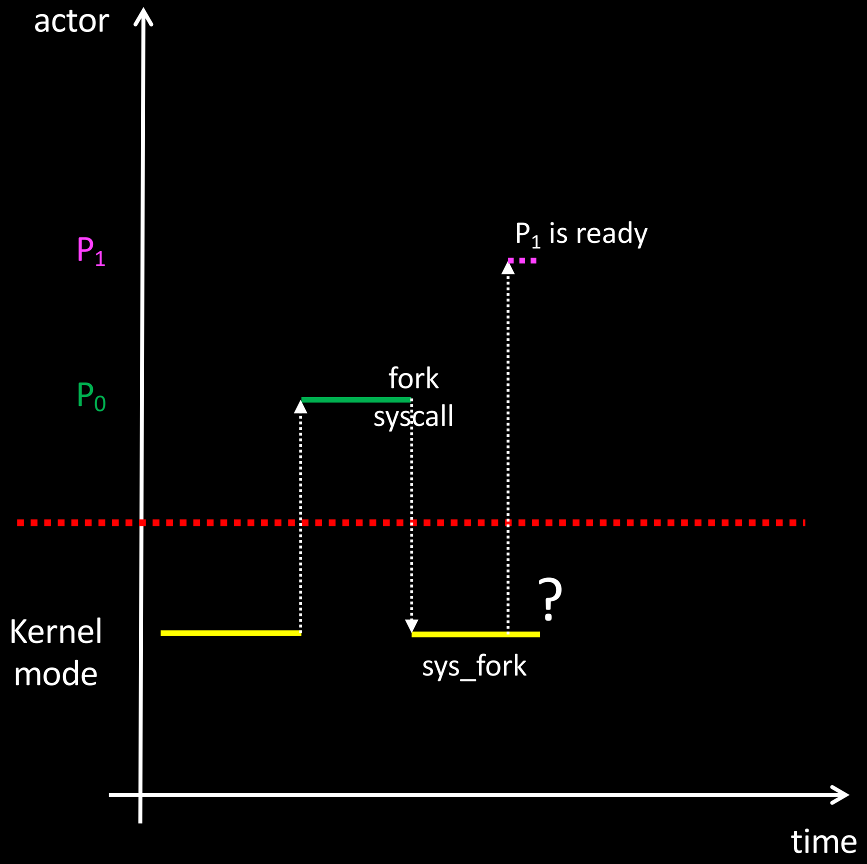


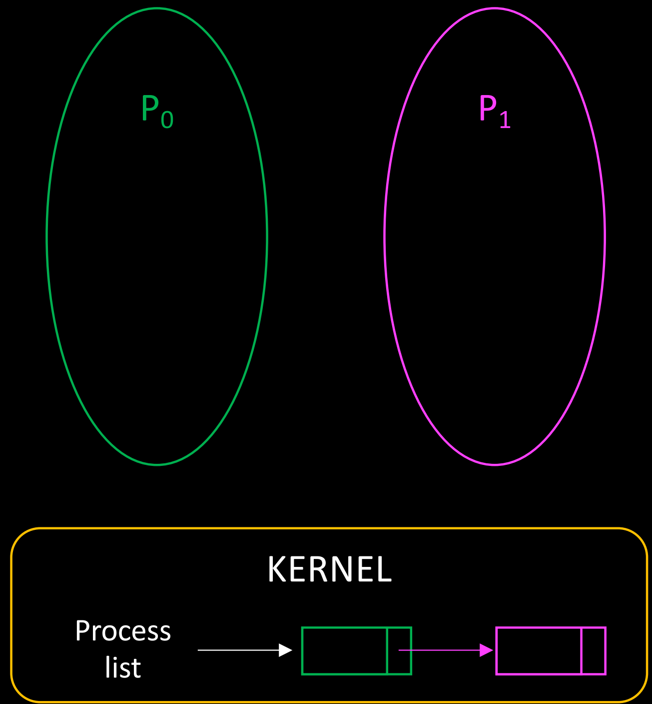
Process Creation



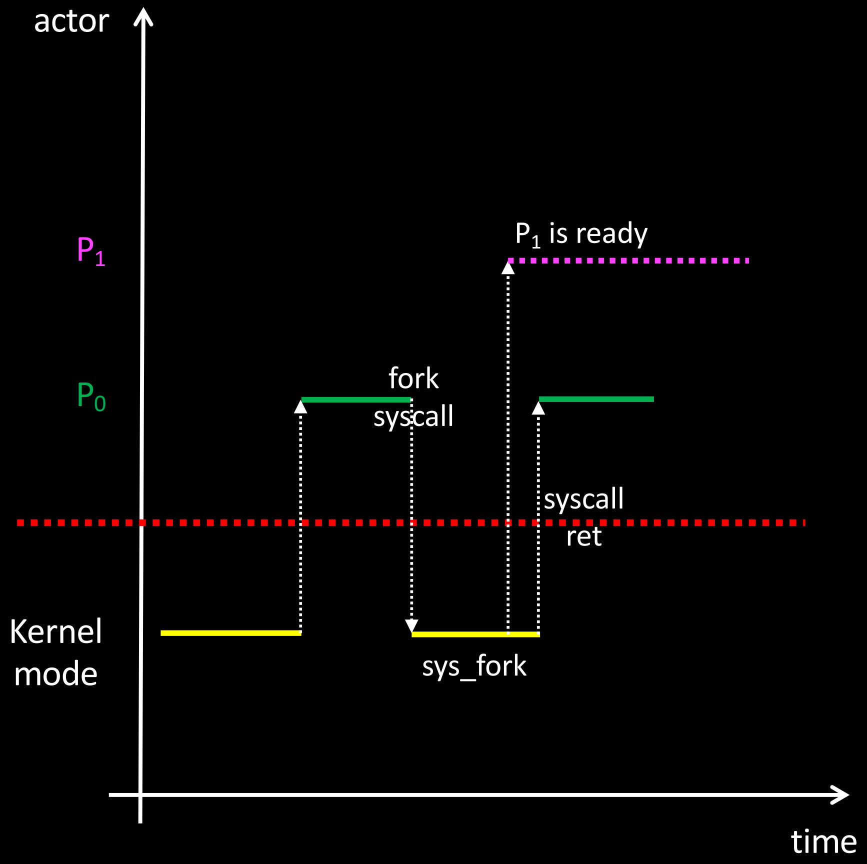


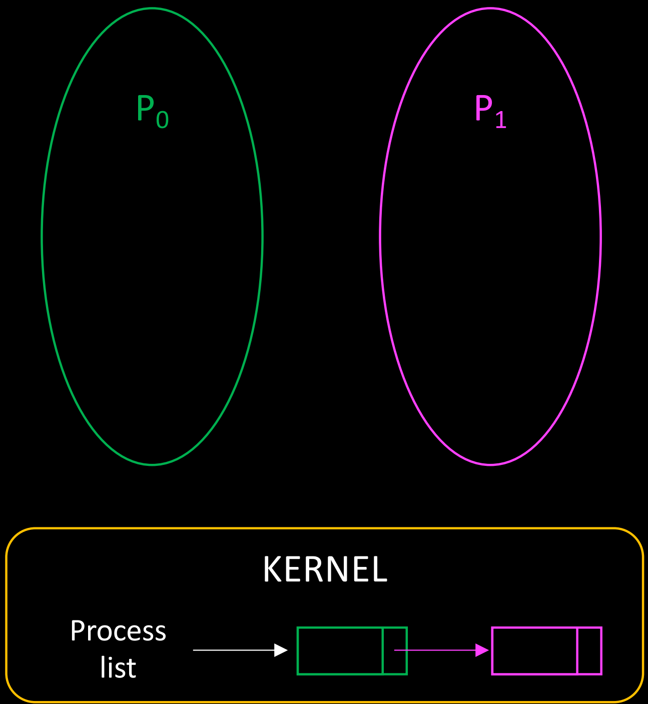
Process Creation



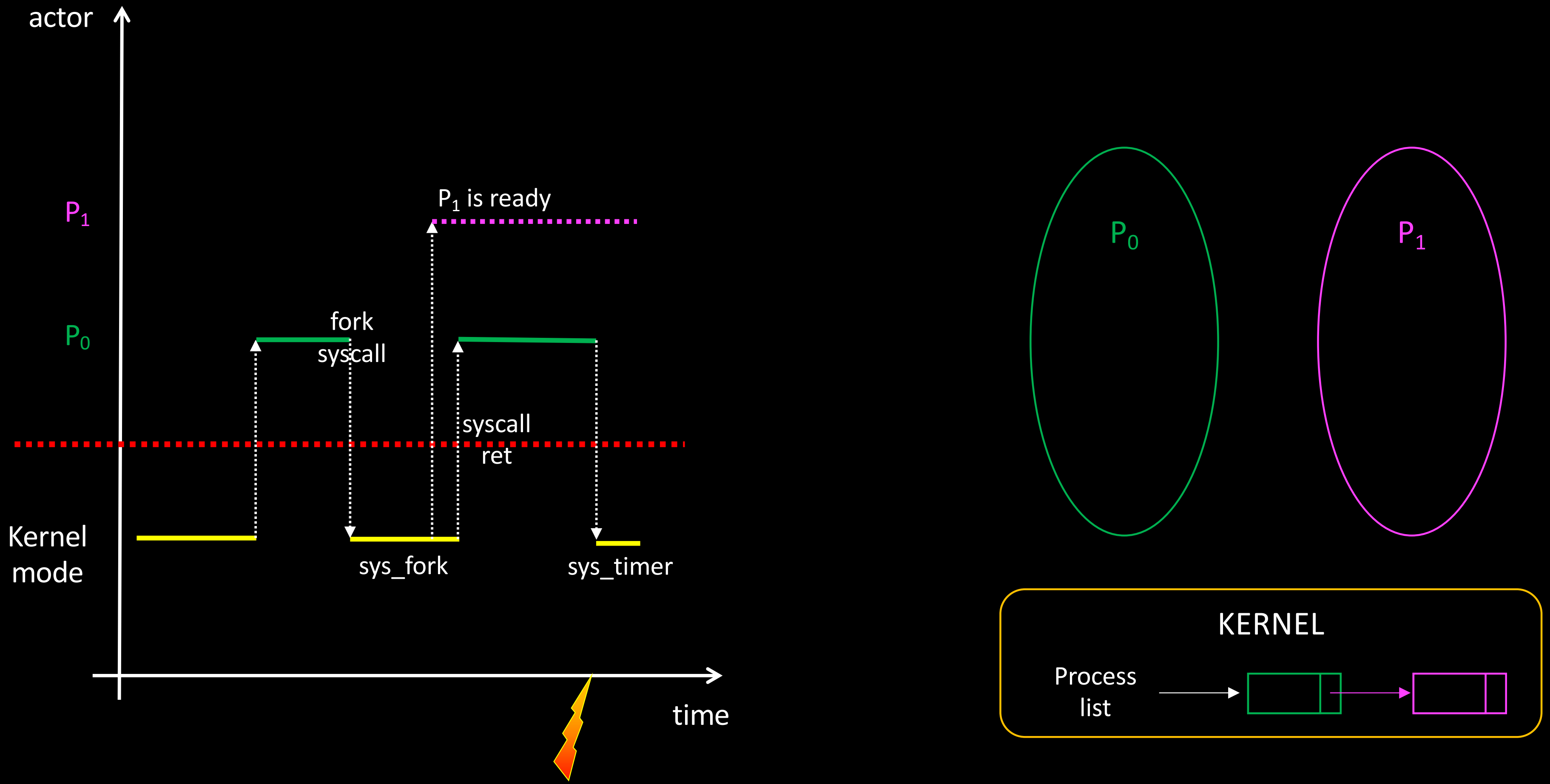


Process Creation

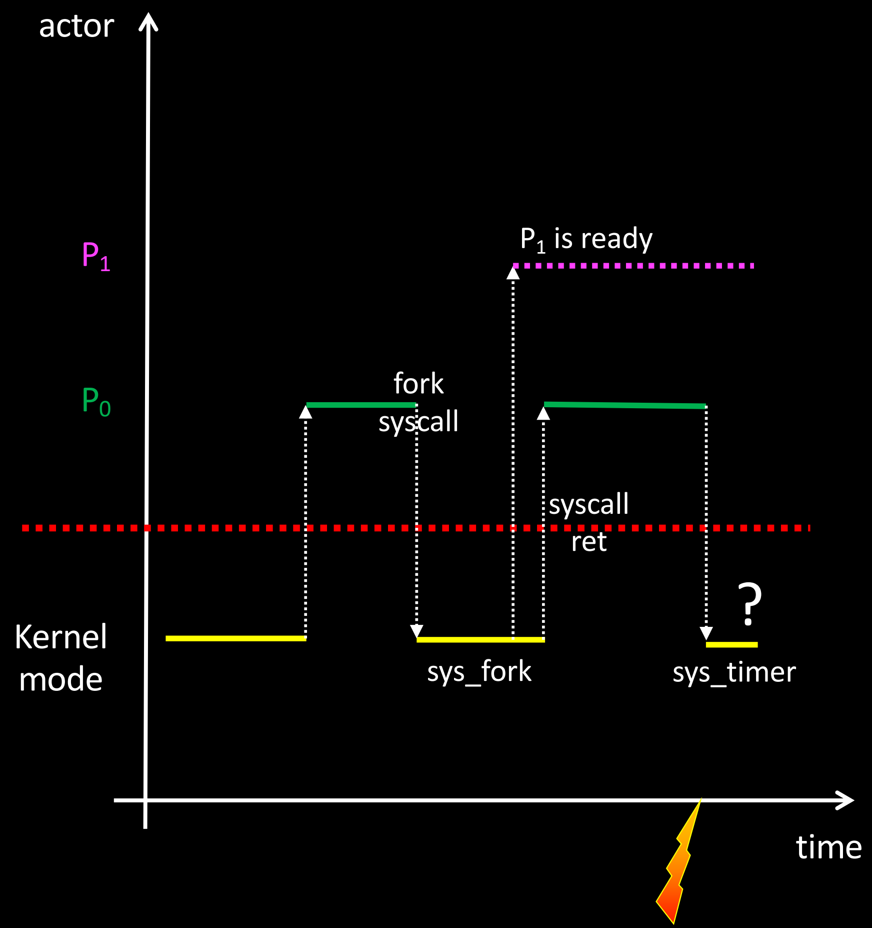




Process Creation



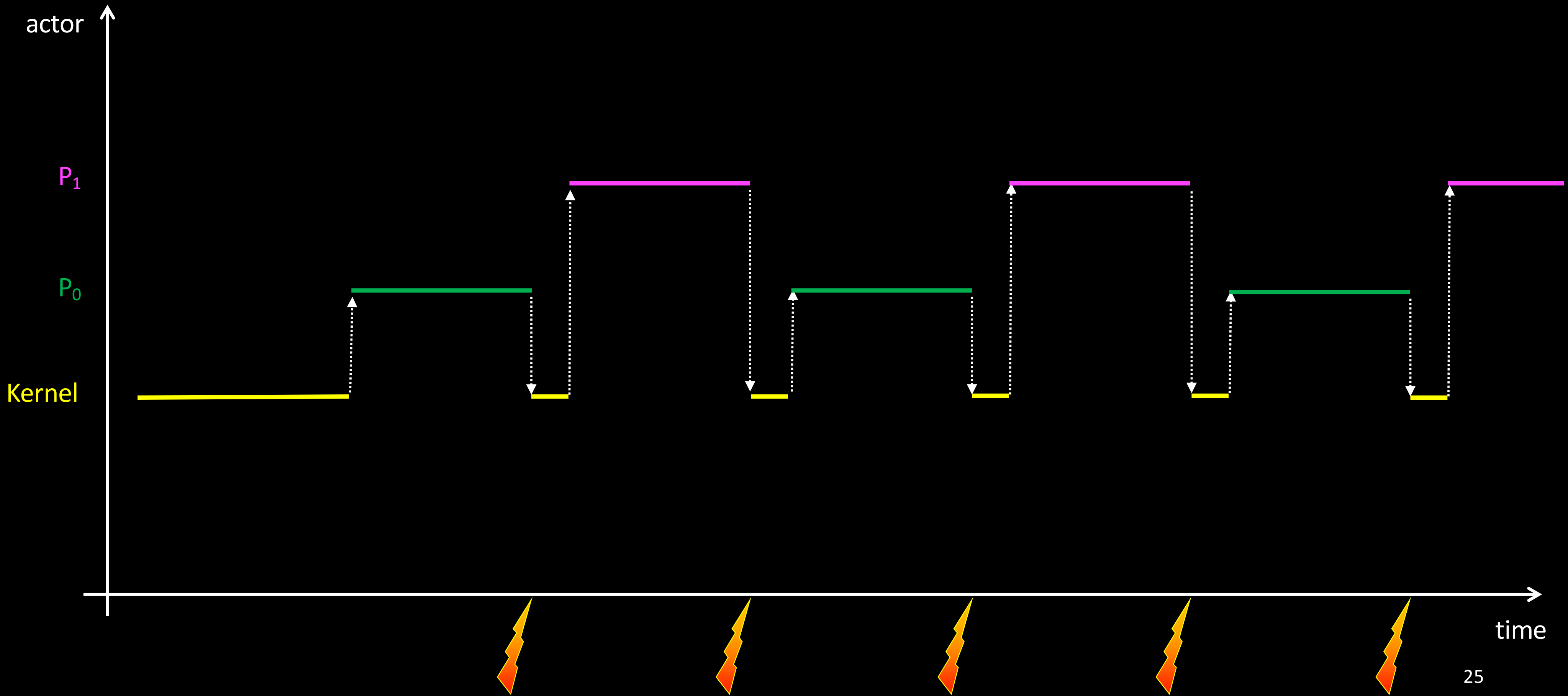
Process Scheduling



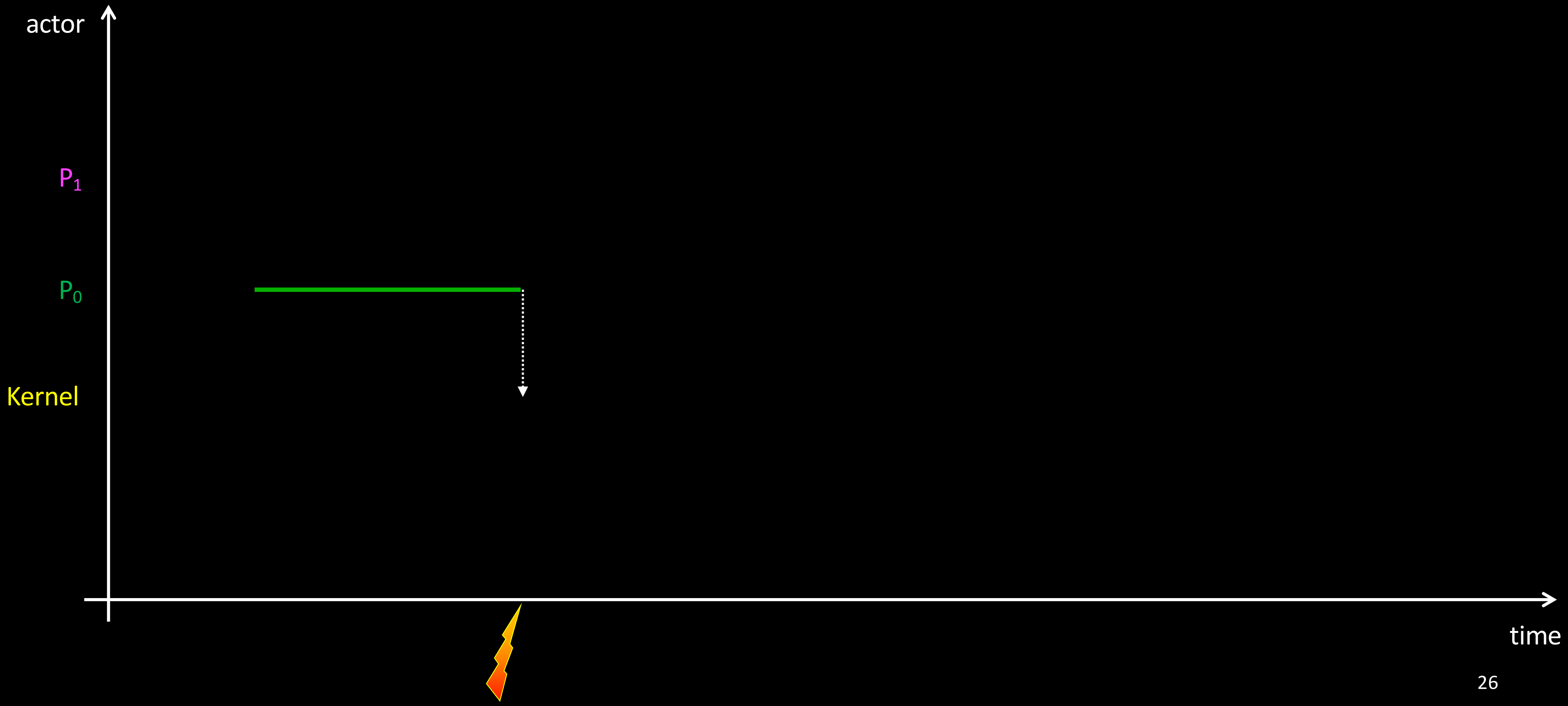
- At some point, the kernel must decide "which process should run now?"
  = Process Scheduling

- NB
  - A CPU executes one program at a time
  - There can be at most #CPU processes running simultaneously

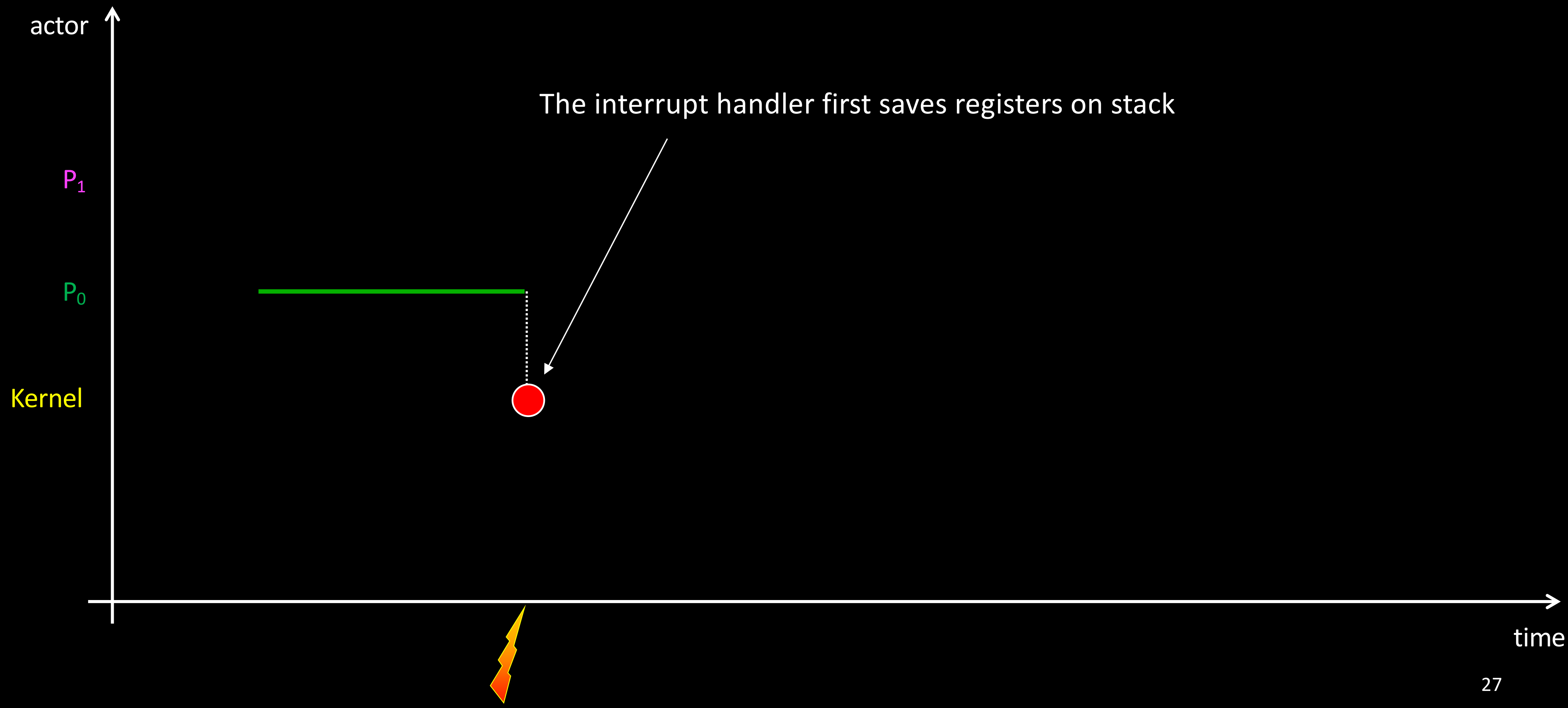
Process Scheduling



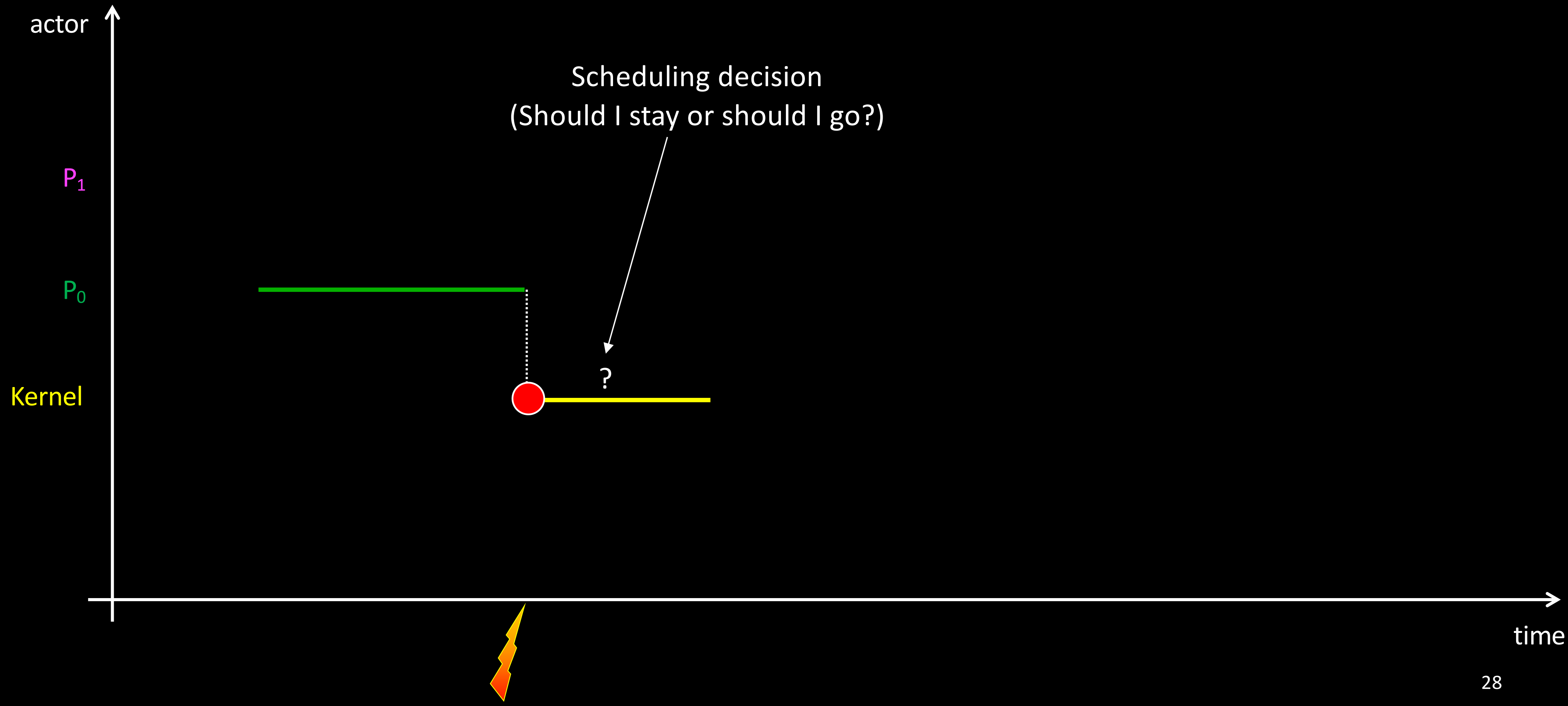
Process Scheduling (close up)



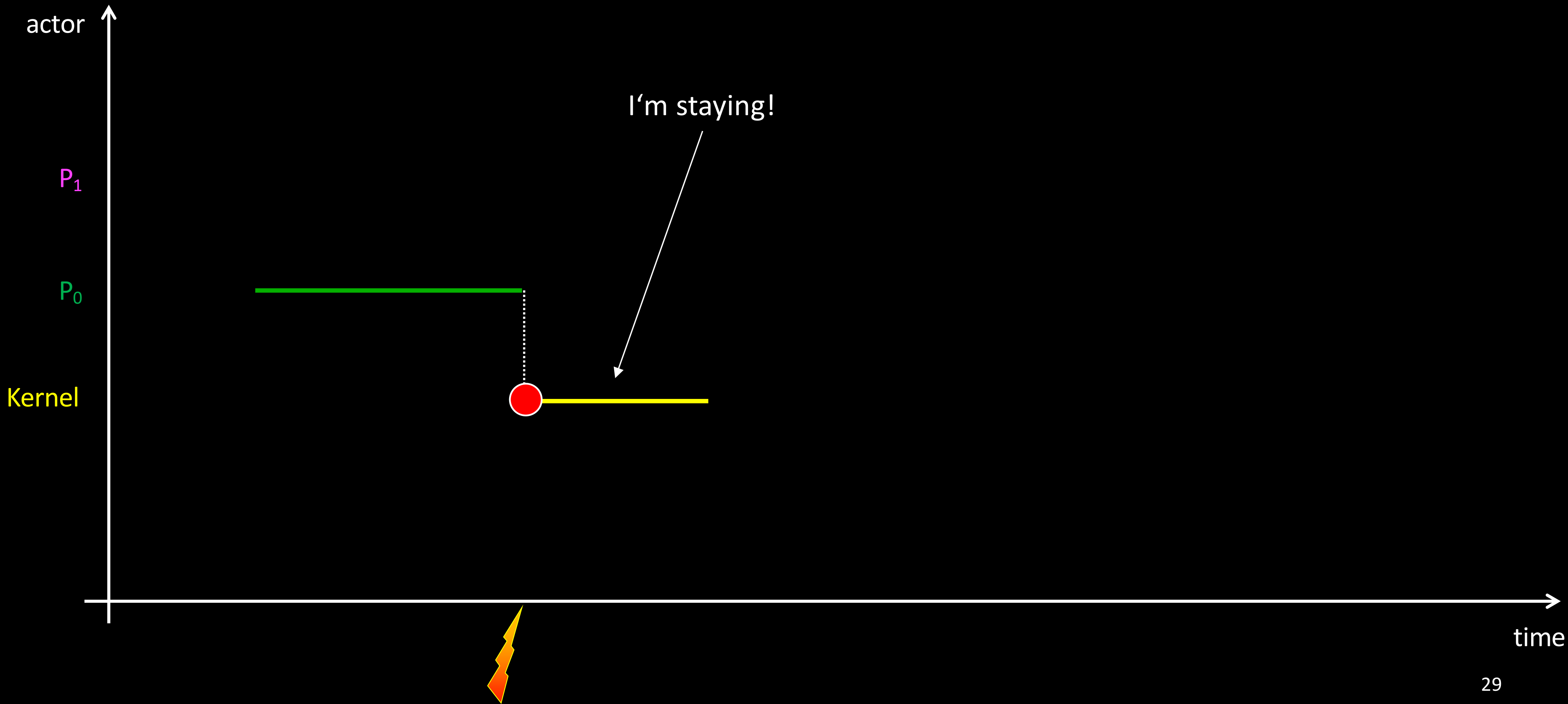
Process Scheduling (close-up)



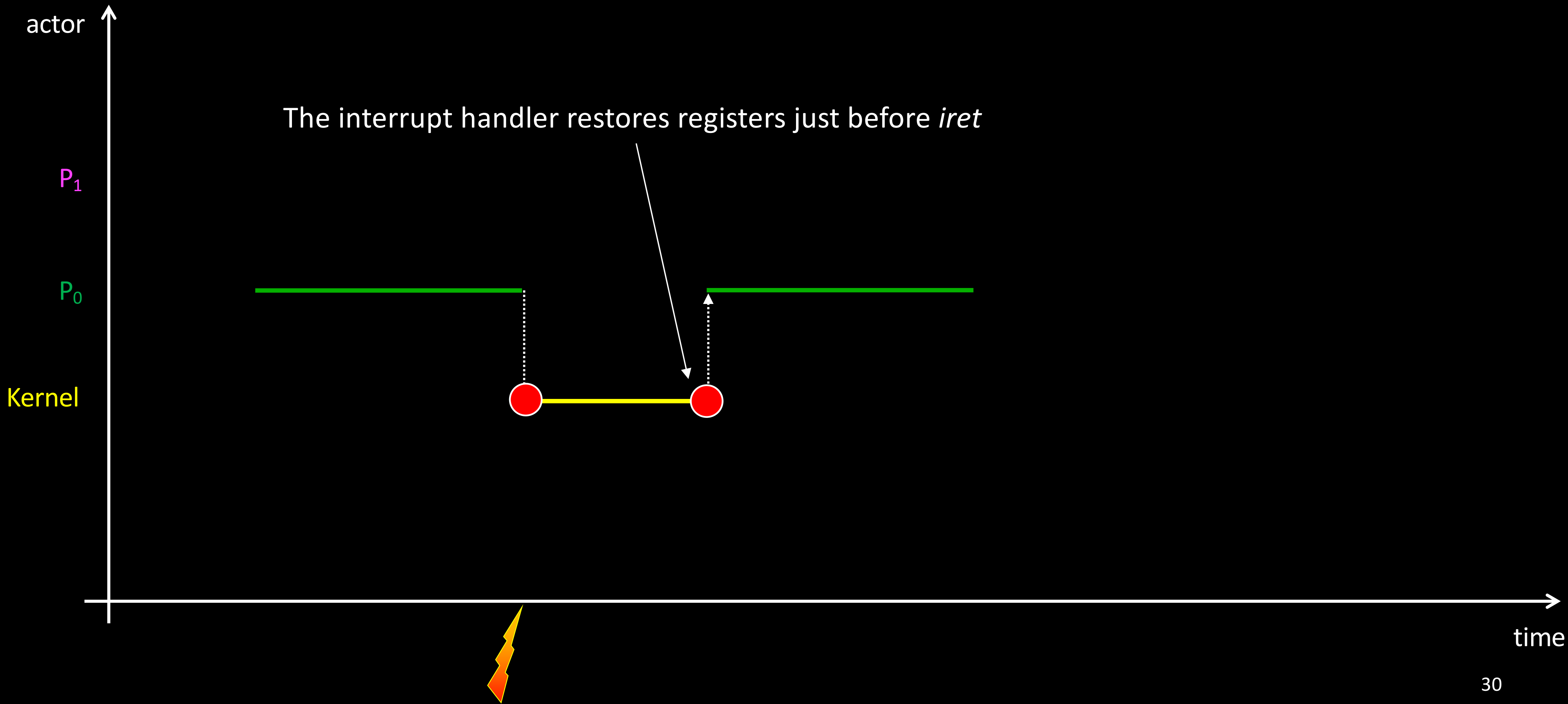
Process Scheduling (close-up)



Process Scheduling (close-up)

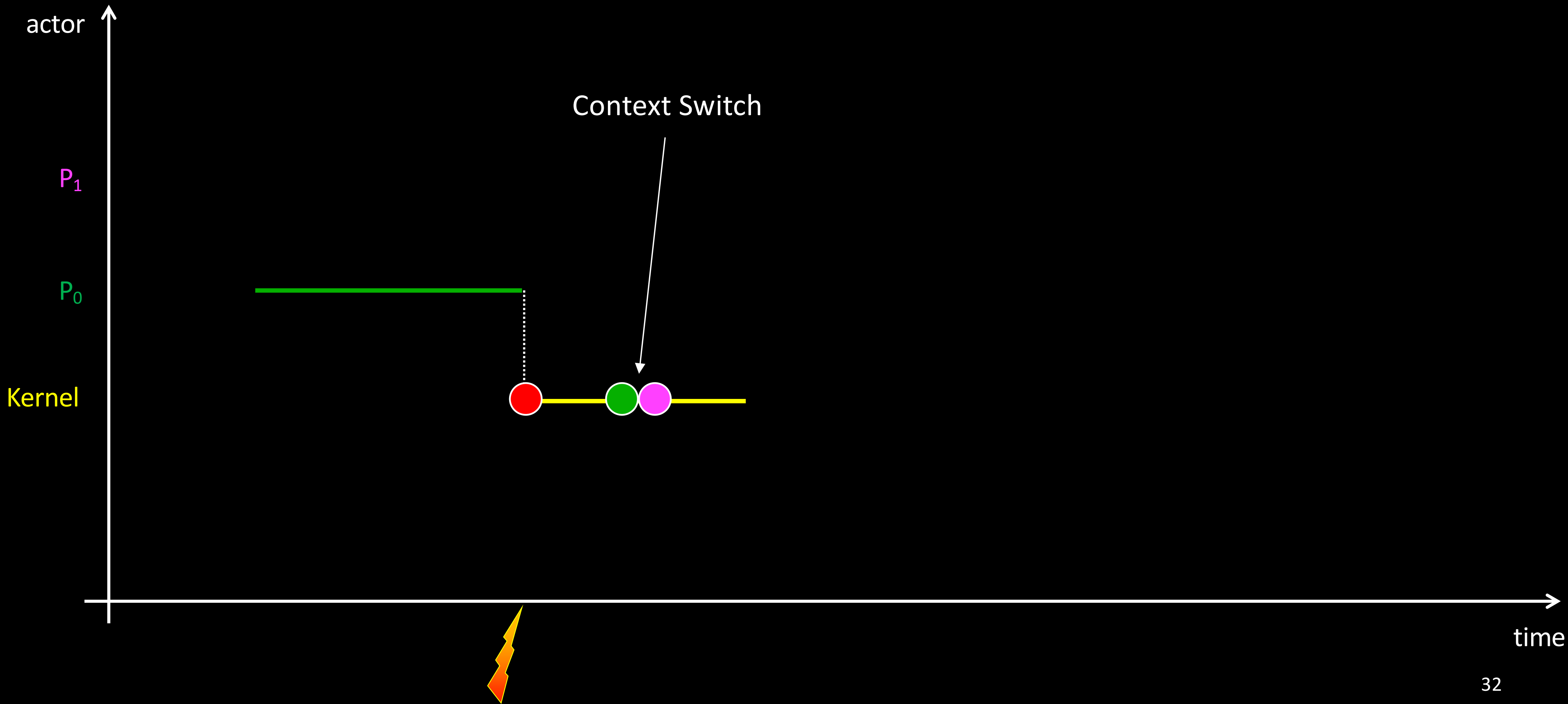


Process Scheduling (close-up)

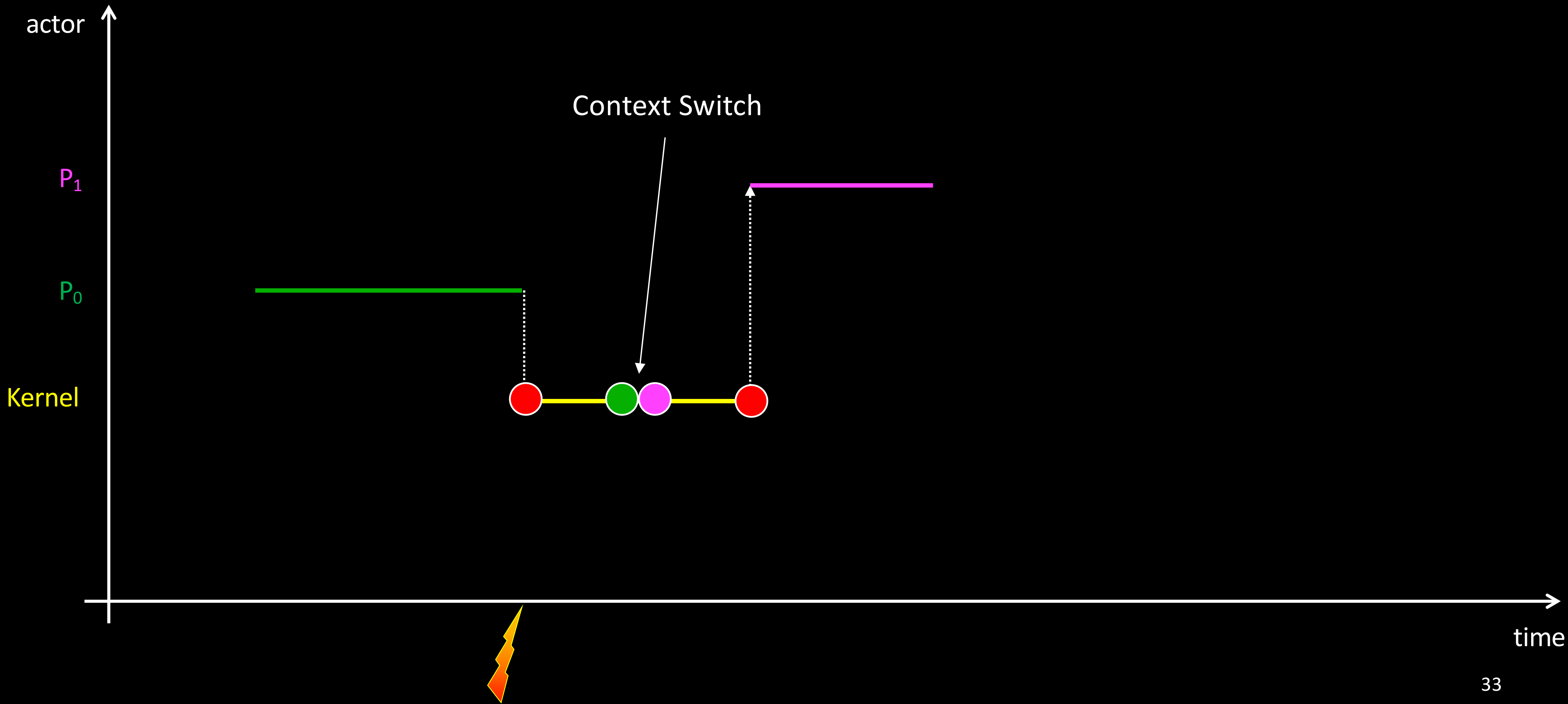


Process Scheduling (close-up)

Process Scheduling (close-up)



Process Scheduling (close-up)



Context Switching

- switch_to (P_{prev}, P_{next})
  - Save P_{prev} registers
  - Restore P_{next} registers

  - P_{prev} becomes P_{next}
  - P_{next} resumes execution and returns from "one" `switch_to` call

  - P_{prev} will resume execution when some process will switch back to it

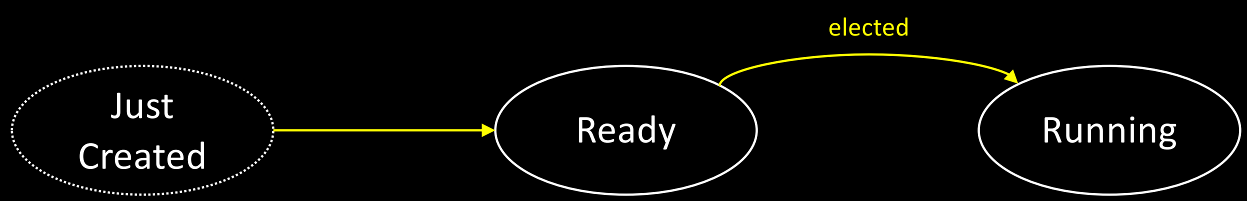
```
kernel_f()
{
  …
  switch_to (prev, next);
  …
}

kernel_g()
{
  …
  switch_to (prev, next);
  …
}
```

Process States

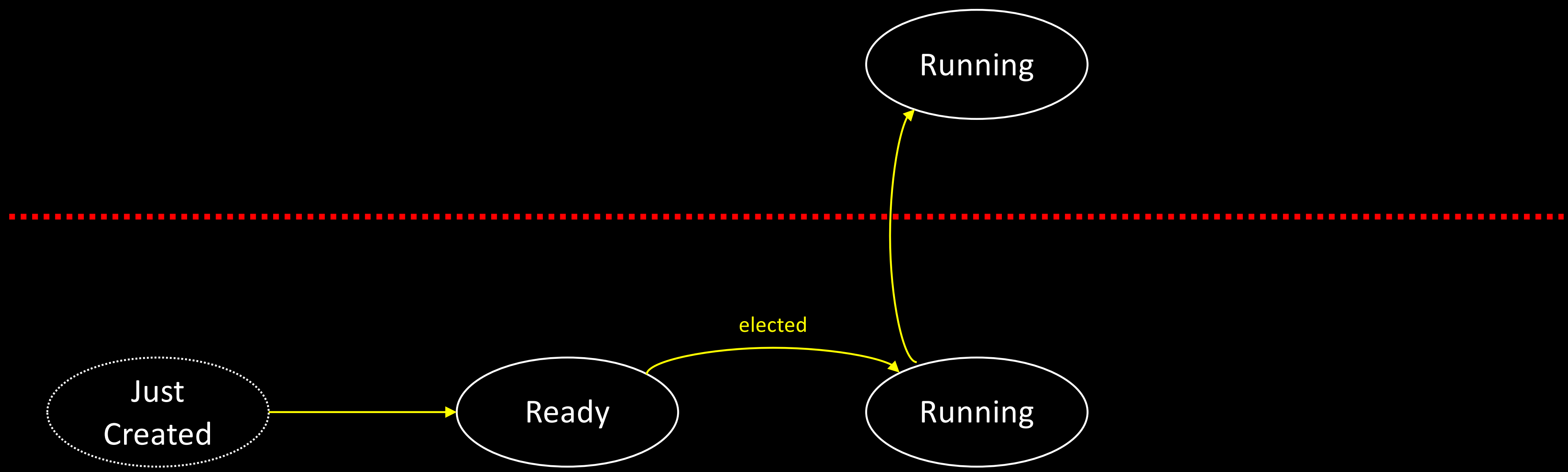
Just
Created

Process States

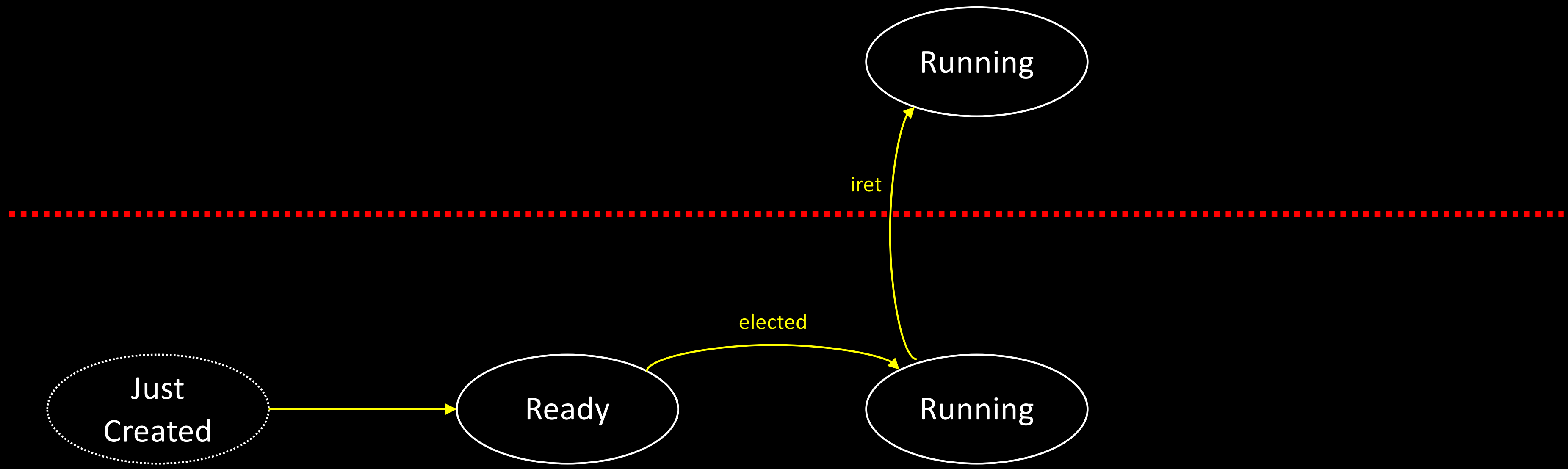
Process States



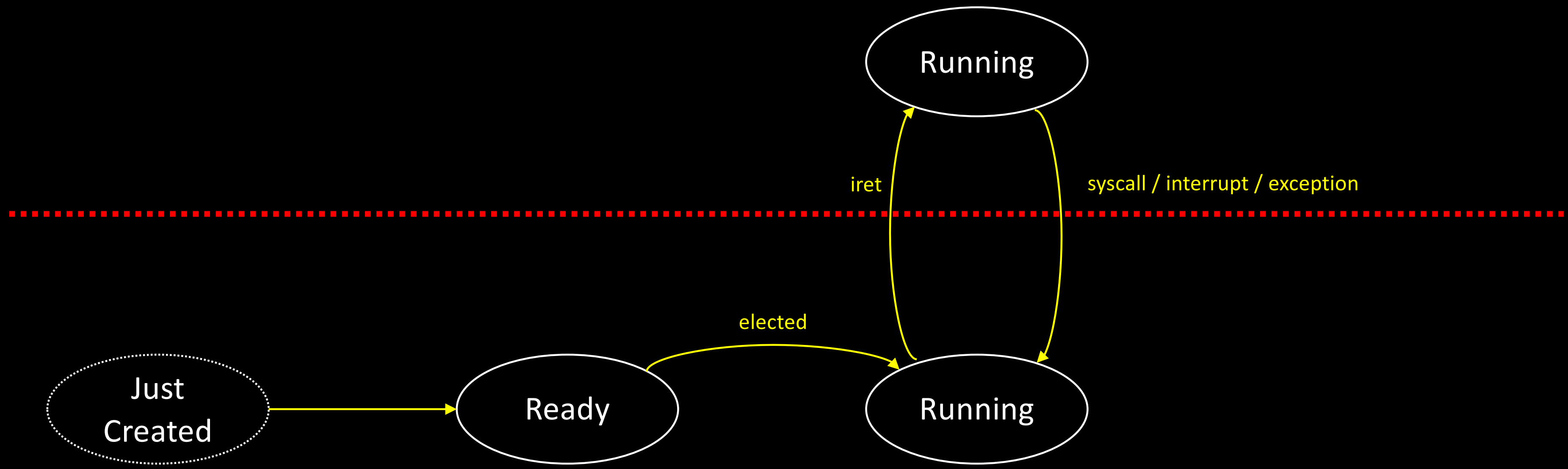
Process States



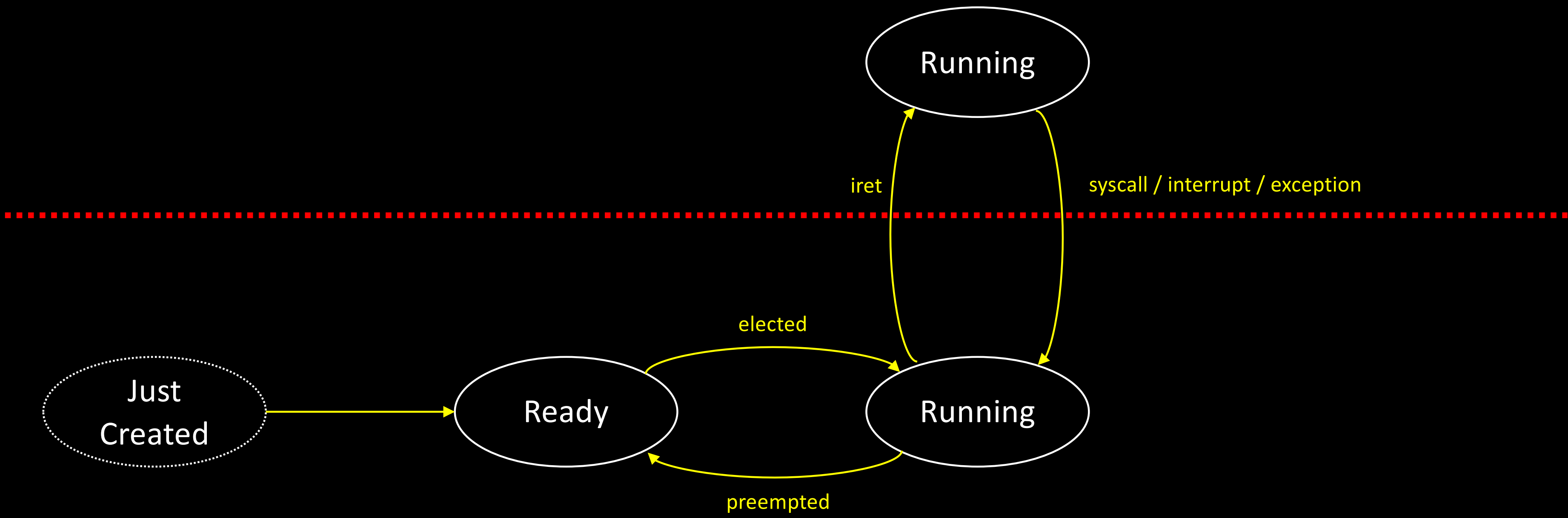
Process States



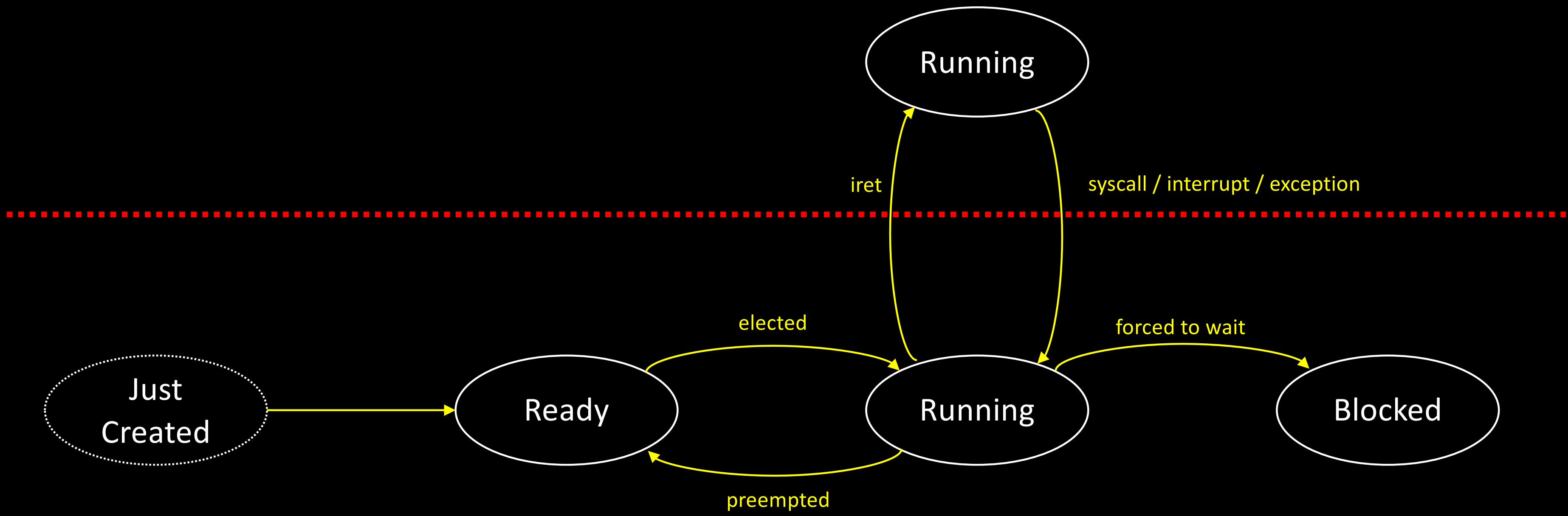
Process States



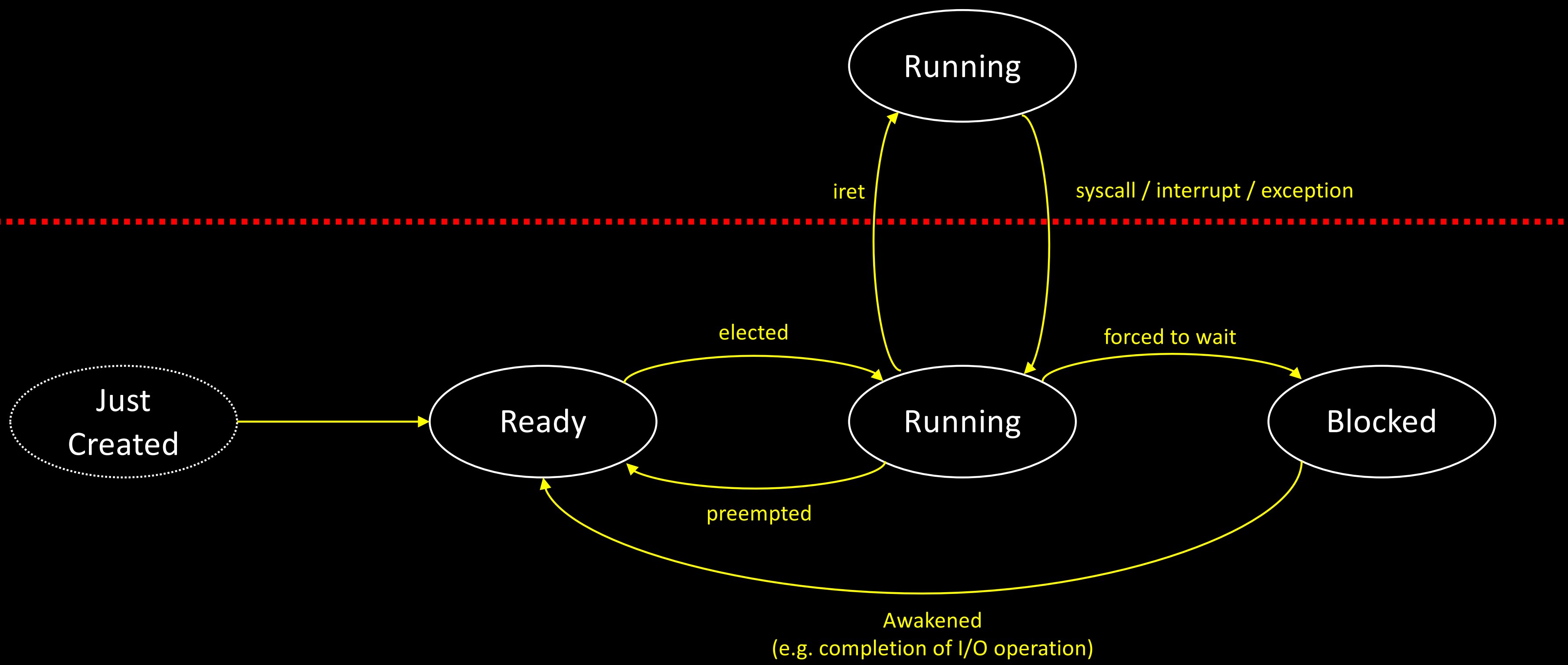
Process States



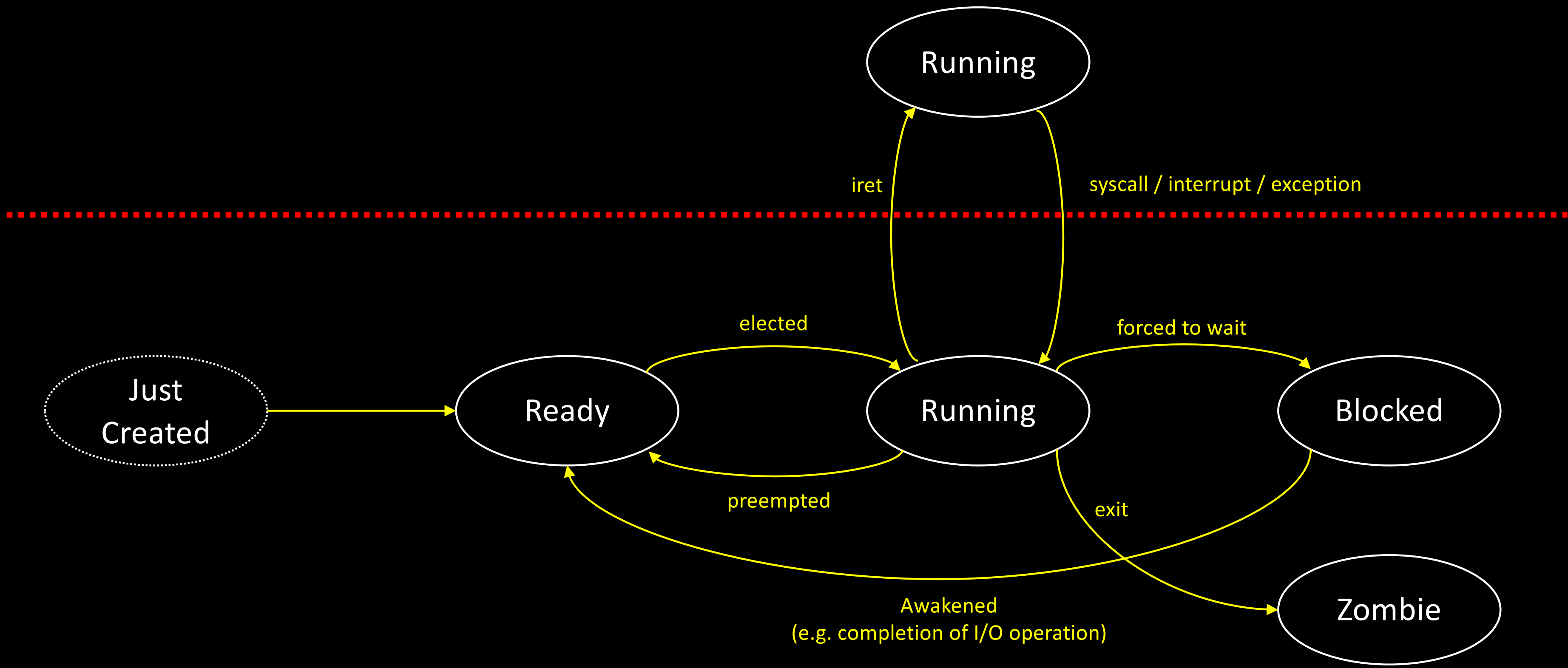
Process States



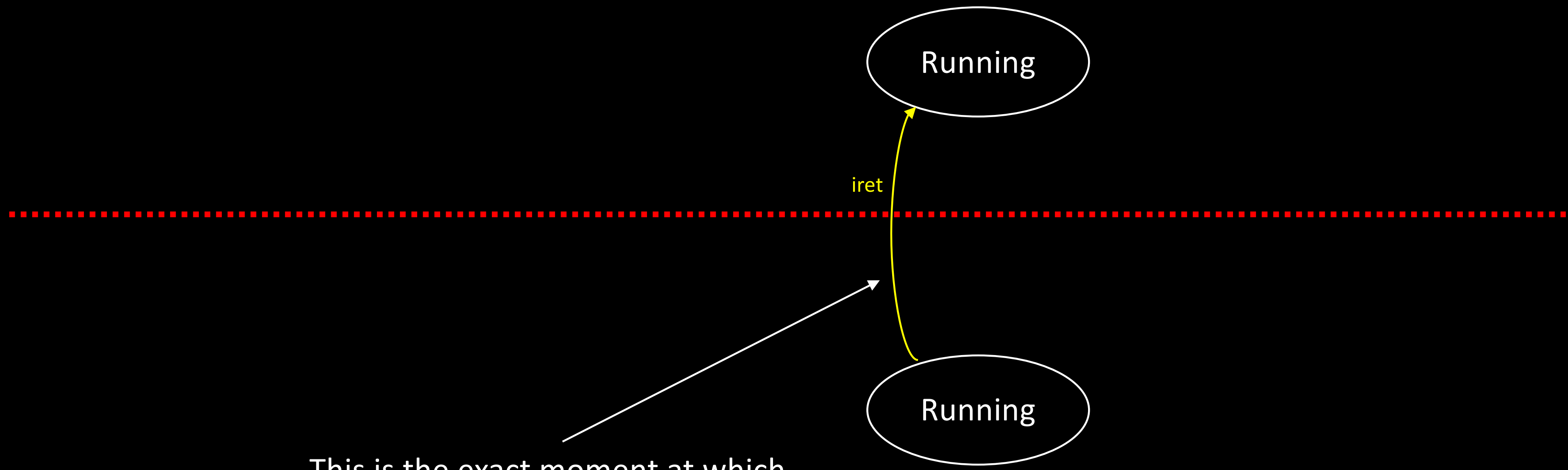
Process States



Process States

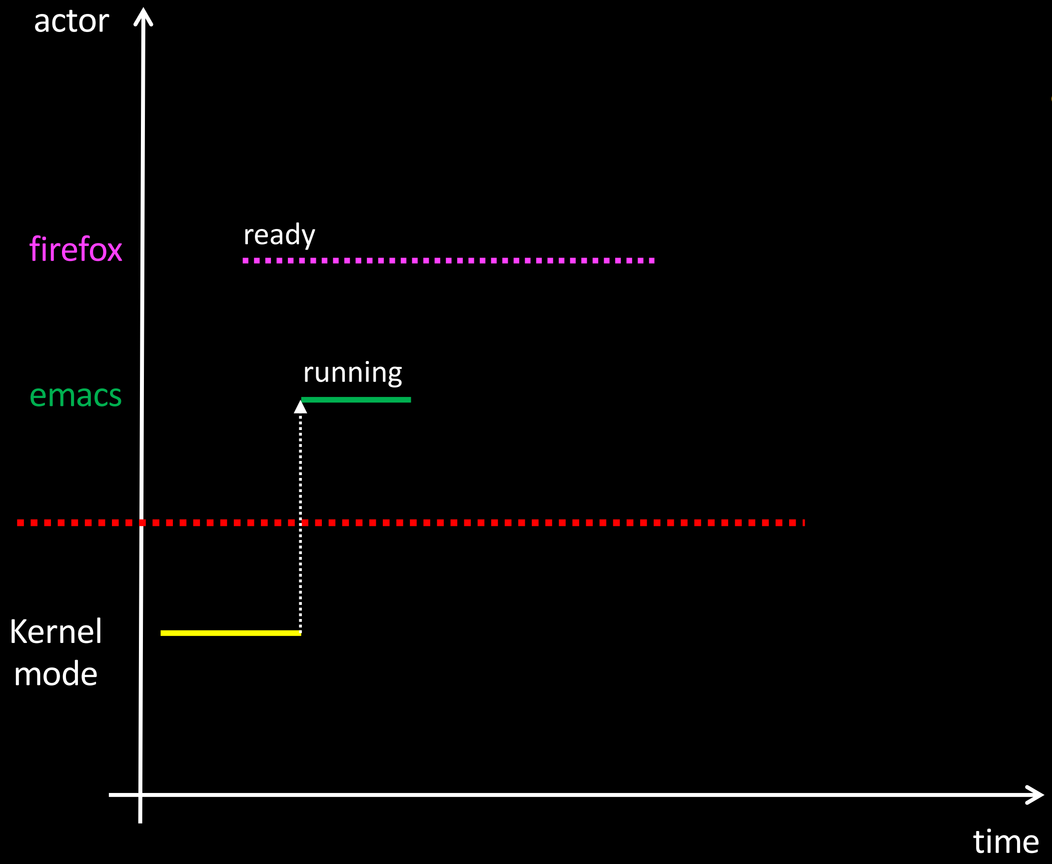


Oh, by the way…



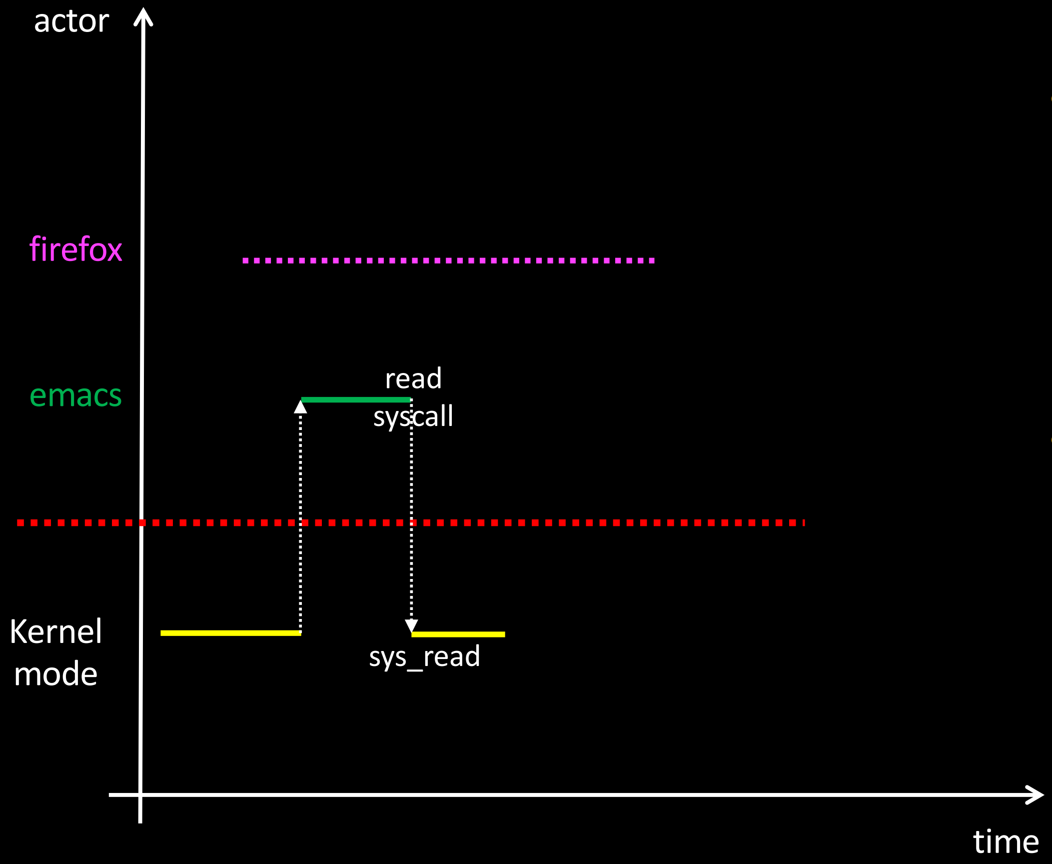
This is the exact moment at which
pending signals are delivered

Handling of Blocking Calls



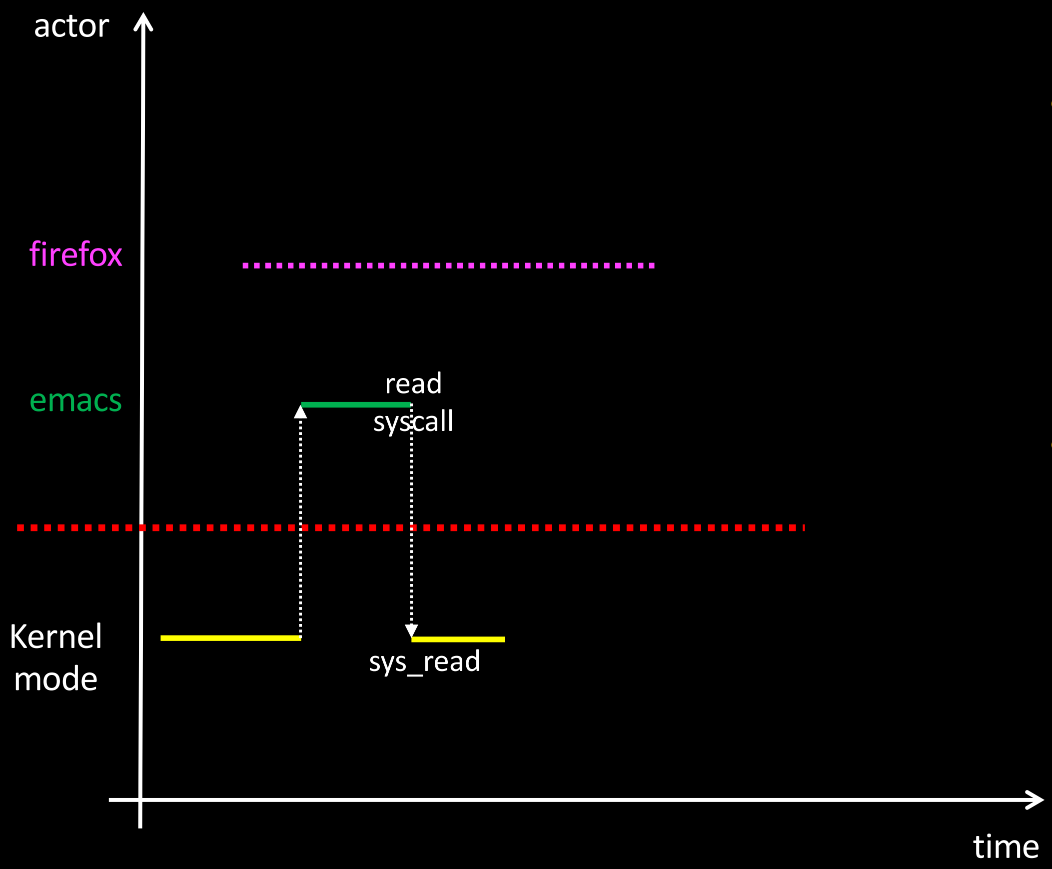
- Let's say emacs is running
  - Emacs spends its life
    - Waiting for keyboard input
    - Refreshing display

Handling of Blocking Calls



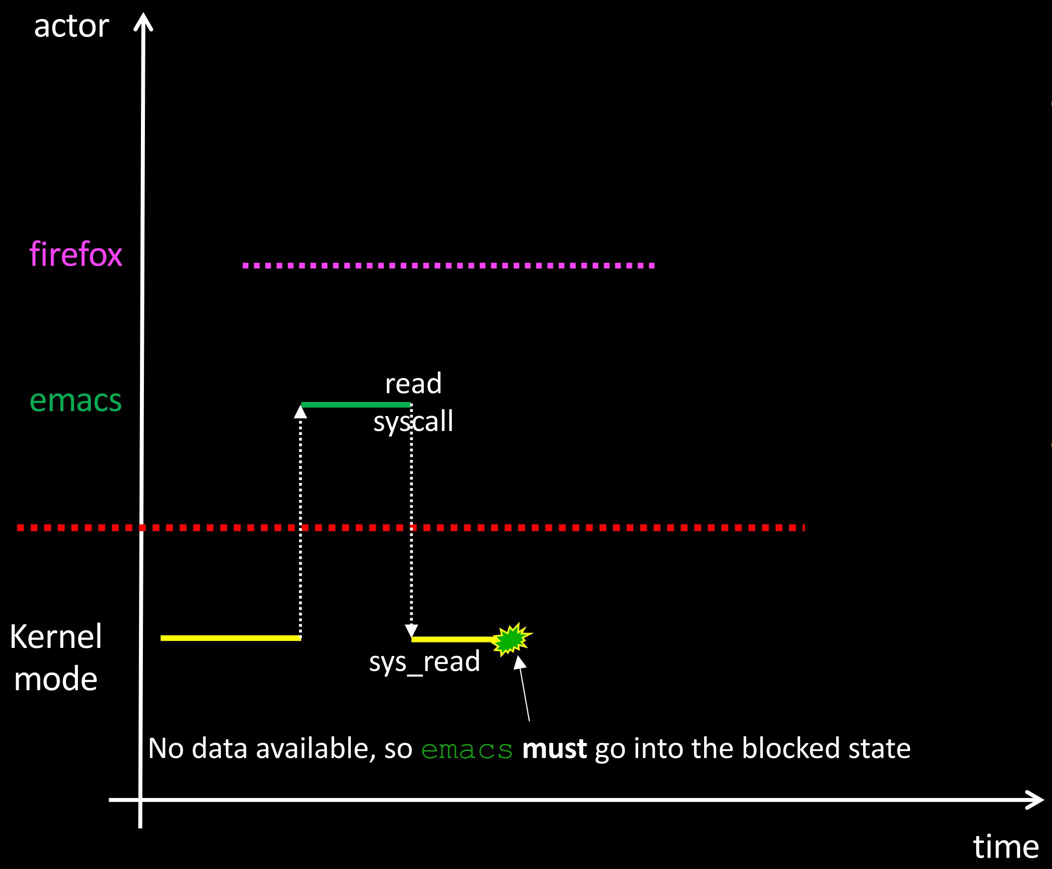
- Let’s say emacs is running
  - Emacs spends its life
    - Waiting for keyboard input
    - Refreshing display
- Waiting for keyboard input
  - `read` system call

Handling of Blocking Calls



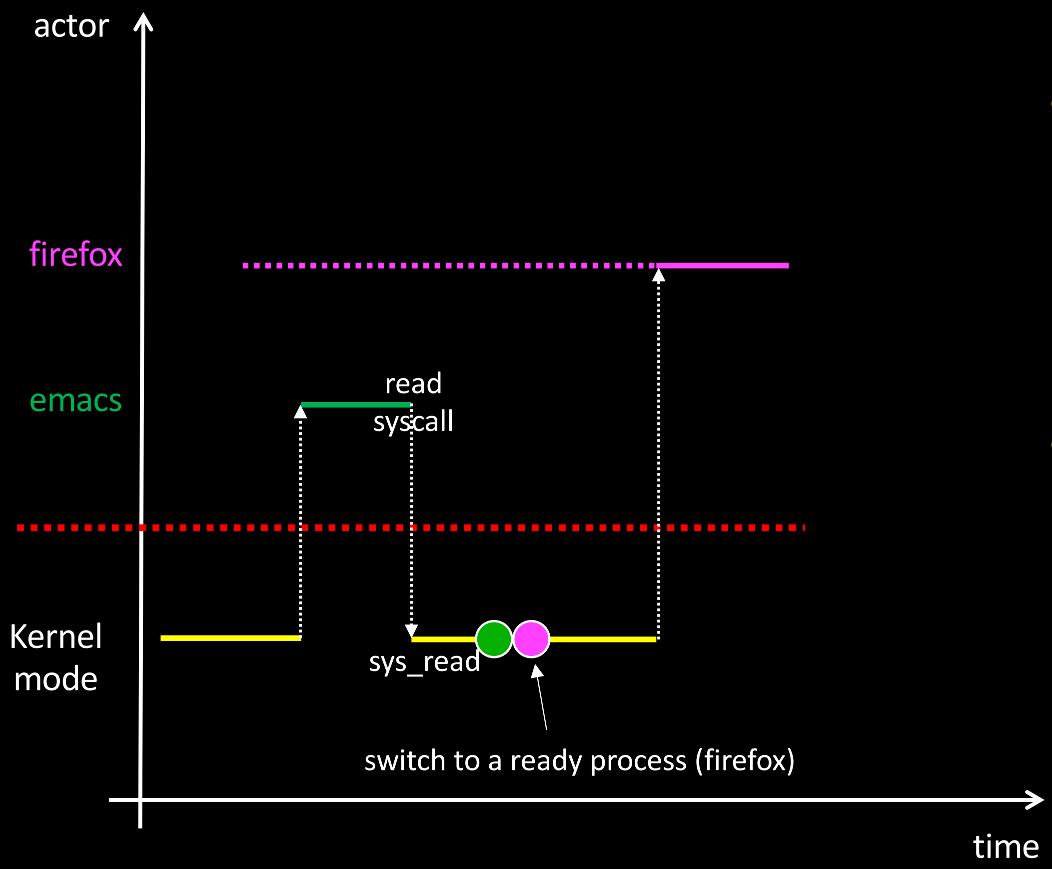
- Let's say emacs is running
  - Emacs spends its life
    - Waiting for keyboard input
    - Refreshing display
- Waiting for keyboard input
  - `read` system call
  - Most of the time, keyboard buffer is empty

Handling of Blocking Calls



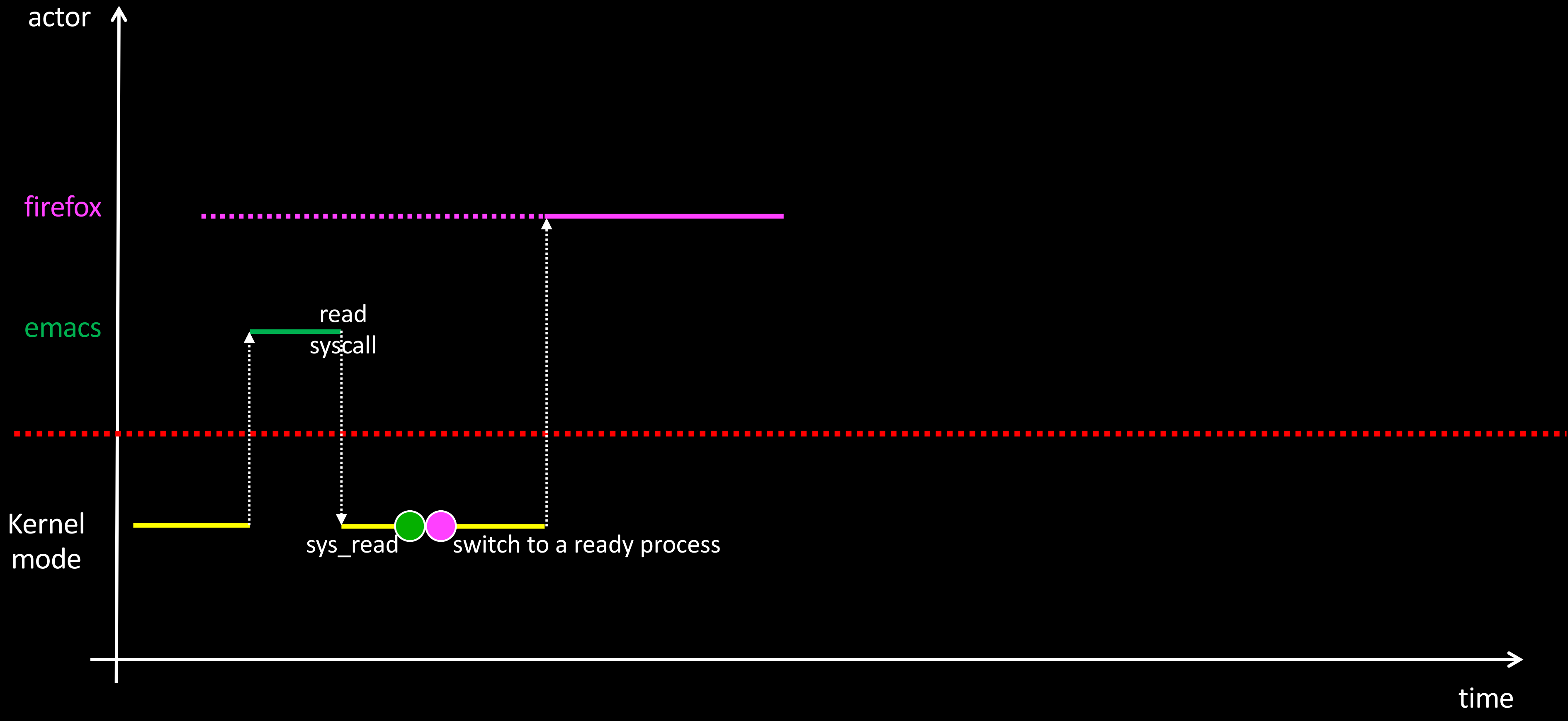
- Let's say emacs is running
  - Emacs spends its life
    - Waiting for keyboard input
    - Refreshing display
- Waiting for keyboard input
  - `read` system call
  - Most of the time, keyboard buffer is empty

Handling of Blocking Calls

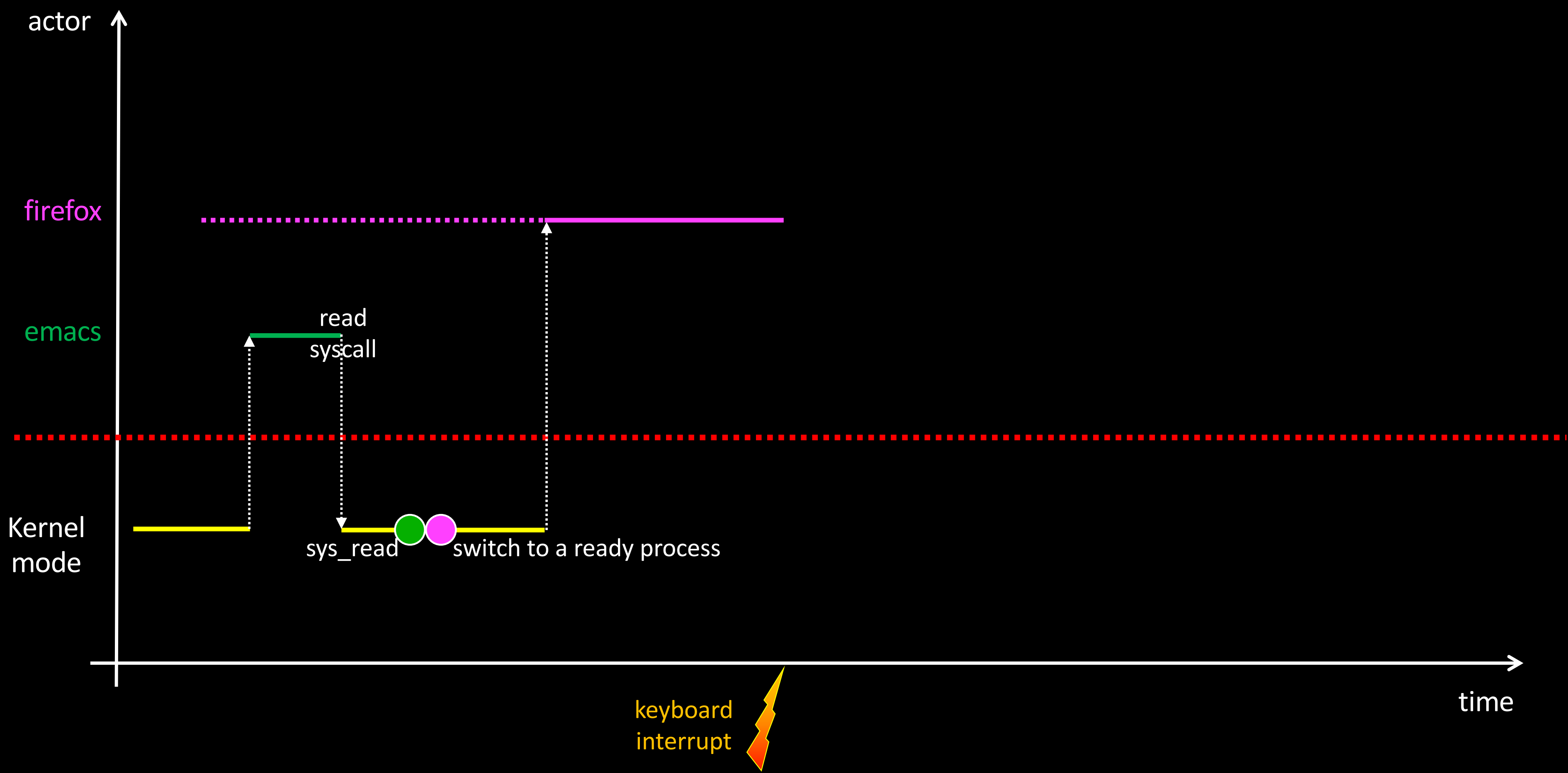


- Let's say emacs is running
  - Emacs spends its life
    - Waiting for keyboard input
    - Refreshing display
- Waiting for keyboard input
  - `read` system call
  - Most of the time, keyboard buffer is empty

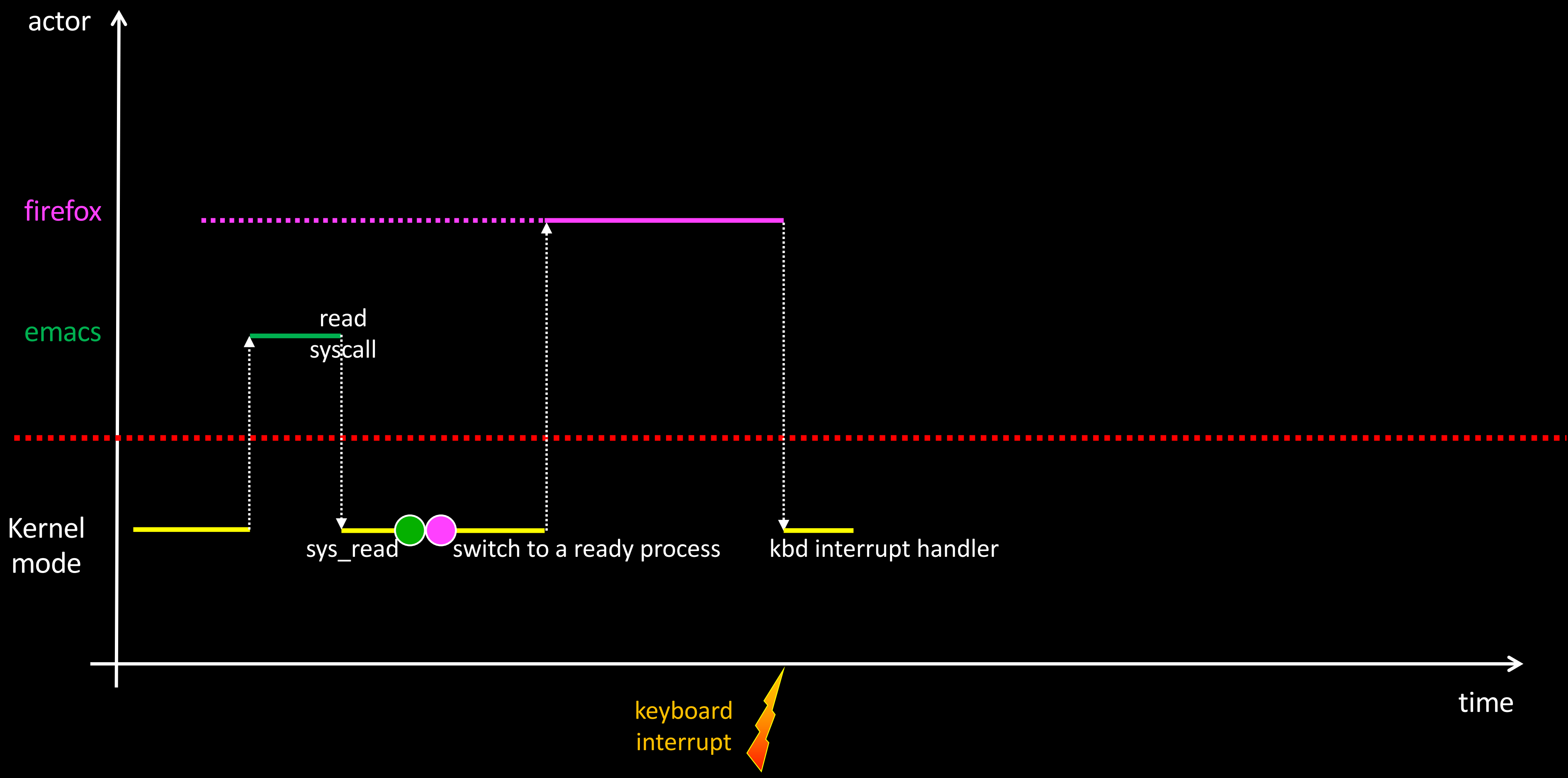
Handling of Blocking Calls



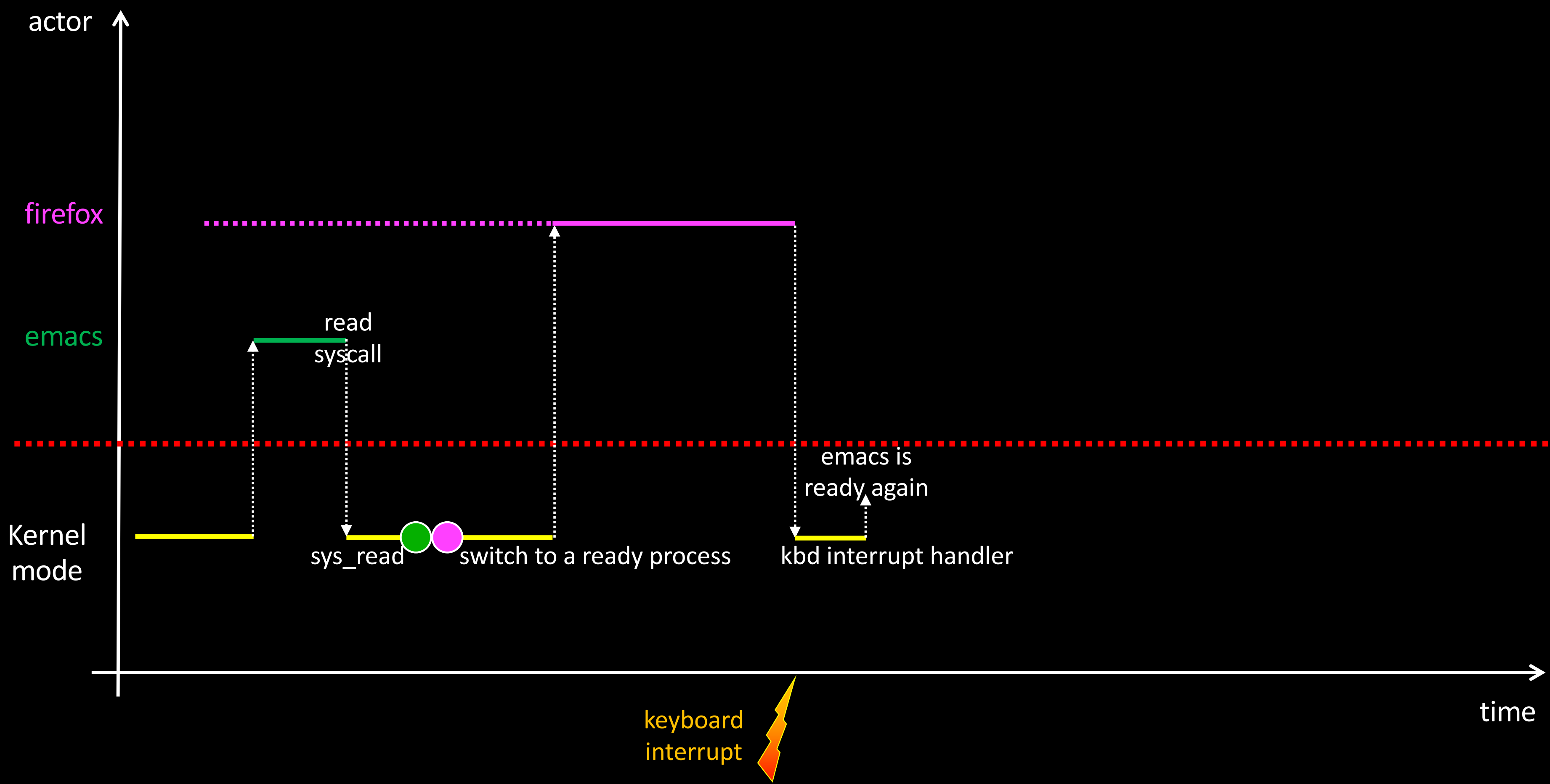
Handling of Blocking Calls



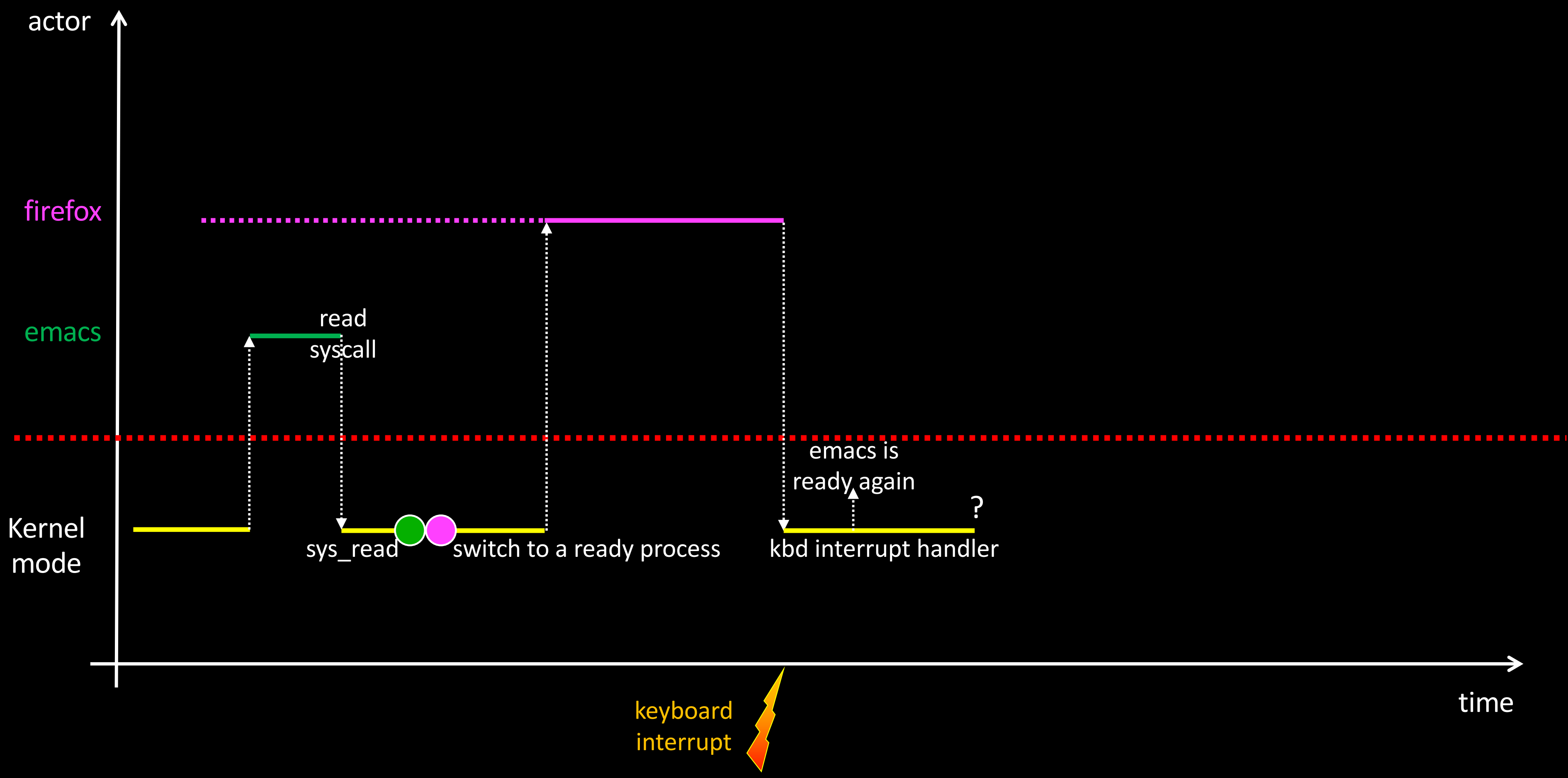
Handling of Blocking Calls



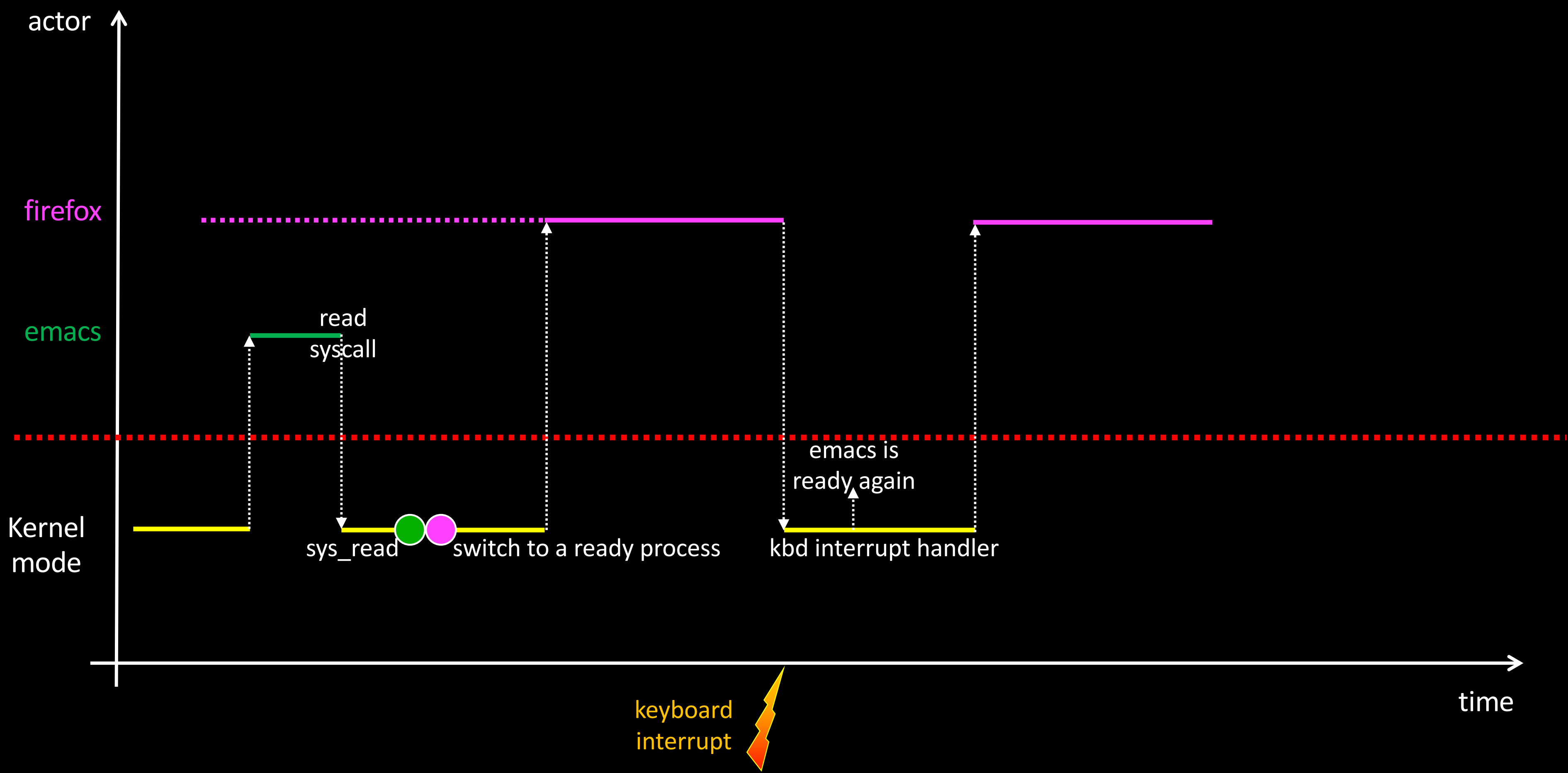
Handling of Blocking Calls



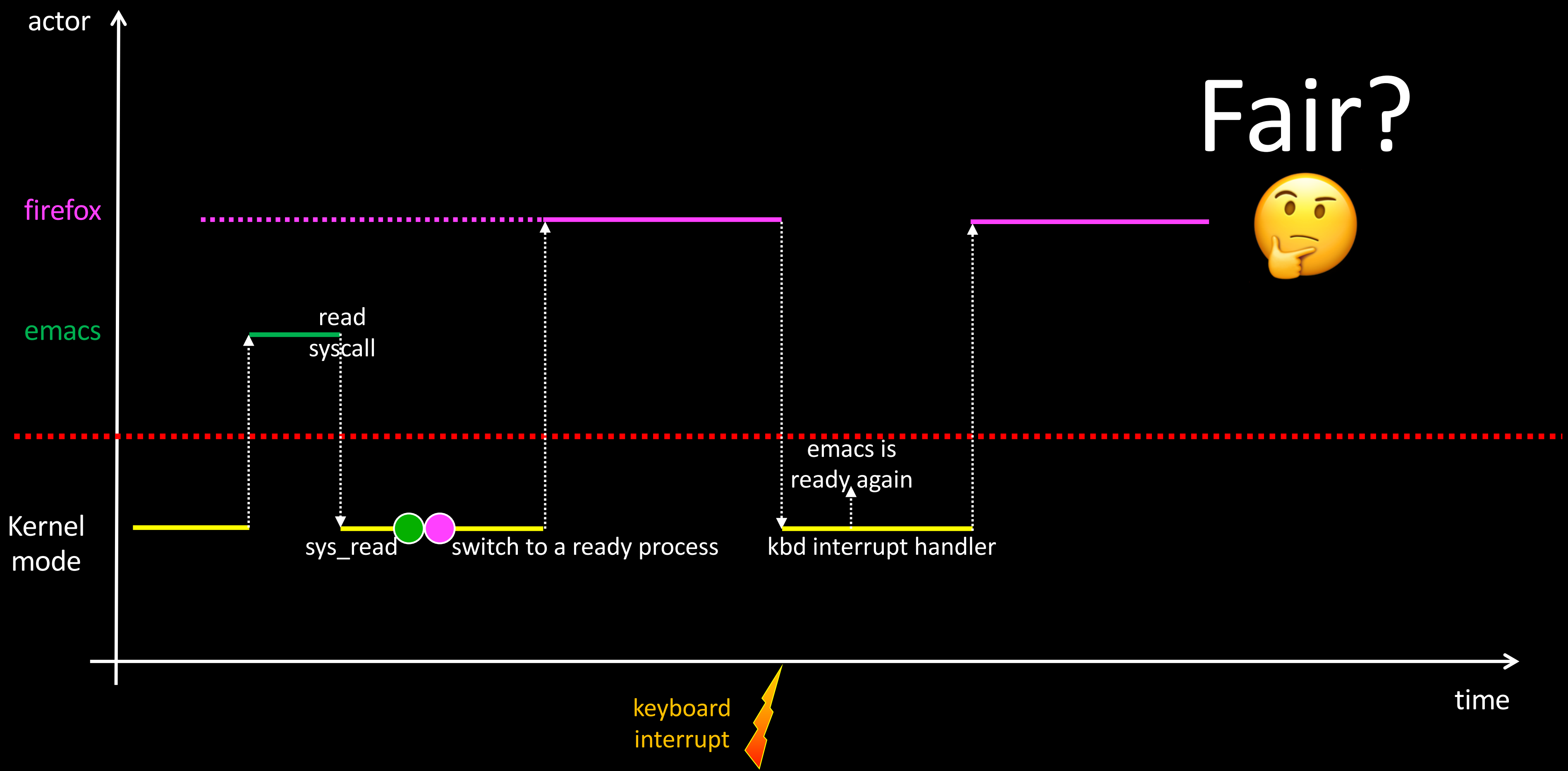
Handling of Blocking Calls



Handling of Blocking Calls



Handling of Blocking Calls



Scheduling

- General goal of a process scheduler
  - Optimize CPU usage and maximize user happiness
    - Each process has a fair access to the CPU
    - CPU is always running at 100%
    - Responsiveness of interactive processes is optimal
    - Completion time of long-running processes is minimal
    - Etc.
  - Satisfying these rules altogether is impossible
    - There is no such thing as a *Universal ~~Soldier~~ Scheduler*
    - Scheduling heavily depends on OS type
      - Interactive
      - Real-time
      - Batch server

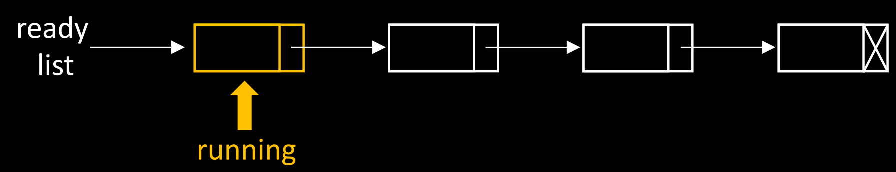
Scheduling in an interactive world

- Most critical property
  - Responsiveness of interactive processes is optimal
- Interactive processes
  - Processes reacting to I/O events
- Scheduling strategy
  - Scheduling algorithm
    - Election of next running process among the pool of ready ones
  - Places where the scheduling code is executed

The FIFO Scheduler

- Running process = head of ready list
    - Removed only when blocking or terminating
    - No periodic preemption

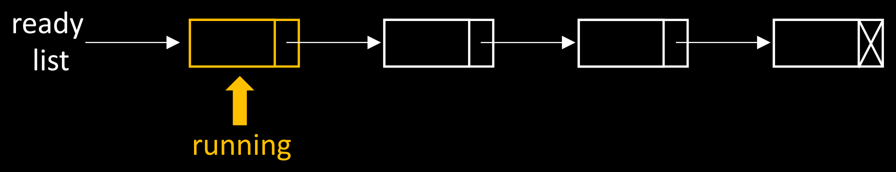
- Pros
    - ?
    - ?
- Cons
    - ?



The FIFO Scheduler

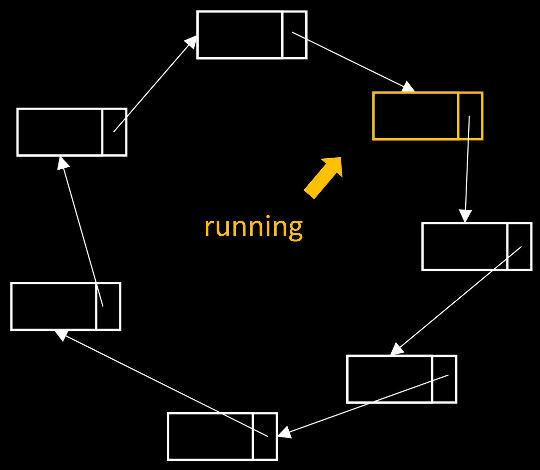
- Running process = head of ready list
  - Removed only when blocking or terminating
  - No periodic preemption

- Pros
  - Very small overhead
  - O(1) election algorithm
- Cons
  - Starvation



The Round-Robin Scheduler

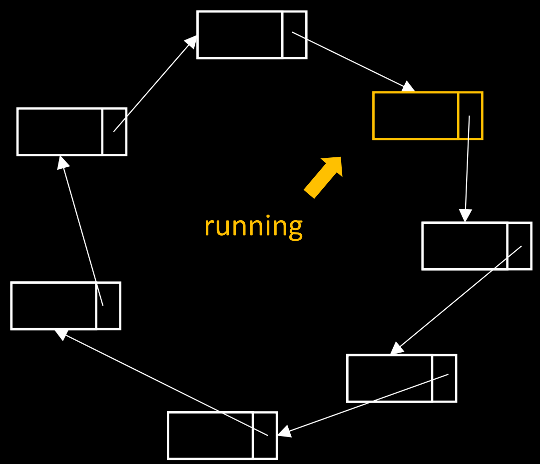
- FIFO + preemption
  - At each timer interrupt, the running process yields CPU to its successor



- Pros
  - ?
  - ?
- Cons
  - ?

The Round-Robin Scheduler

- FIFO + preemption
  - At each timer interrupt, the running process yields CPU to its successor
- Pros
  - No starvation
  - O(1) scheduler
- Cons
  - No priority

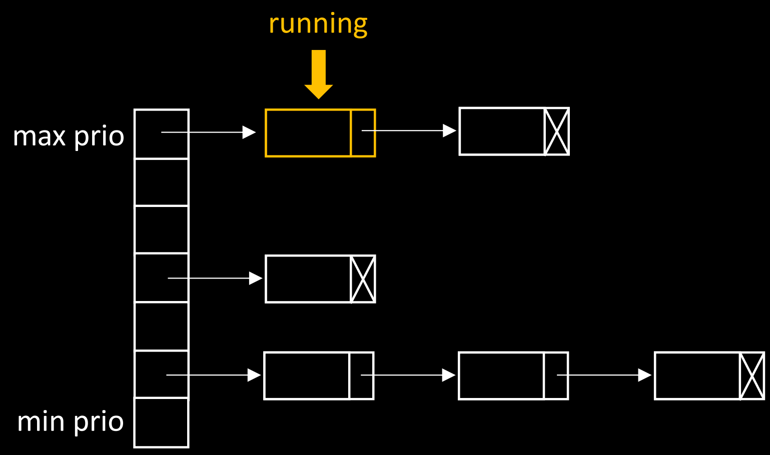


The (strict) Priority Scheduler

- Used in Real-time systems
- One FIFO list per priority level

- Running process = head of highest non-empty priority list

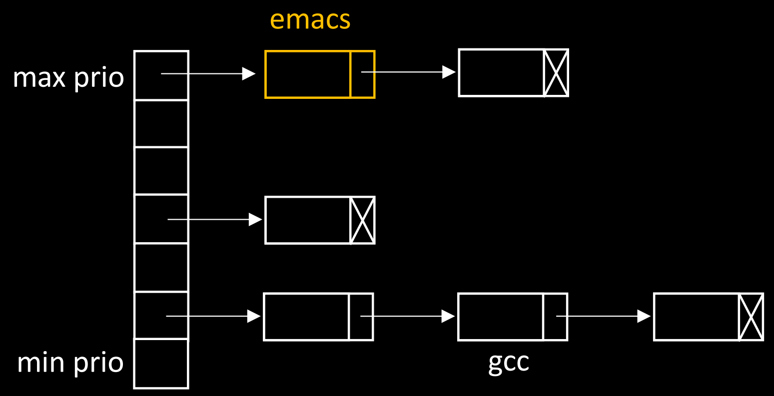
- Pros
  - O(#priorities) scheduler
- Cons
  - ?



The (strict) Priority Scheduler

- Used in Real-time systems
- One FIFO list per priority level

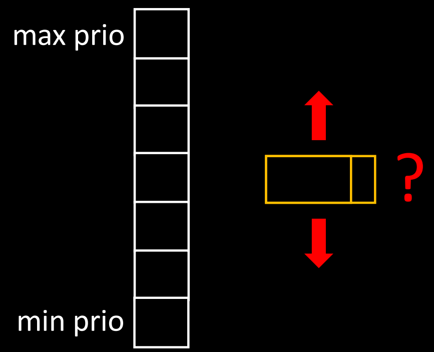
- Running process = head of highest non-empty priority list

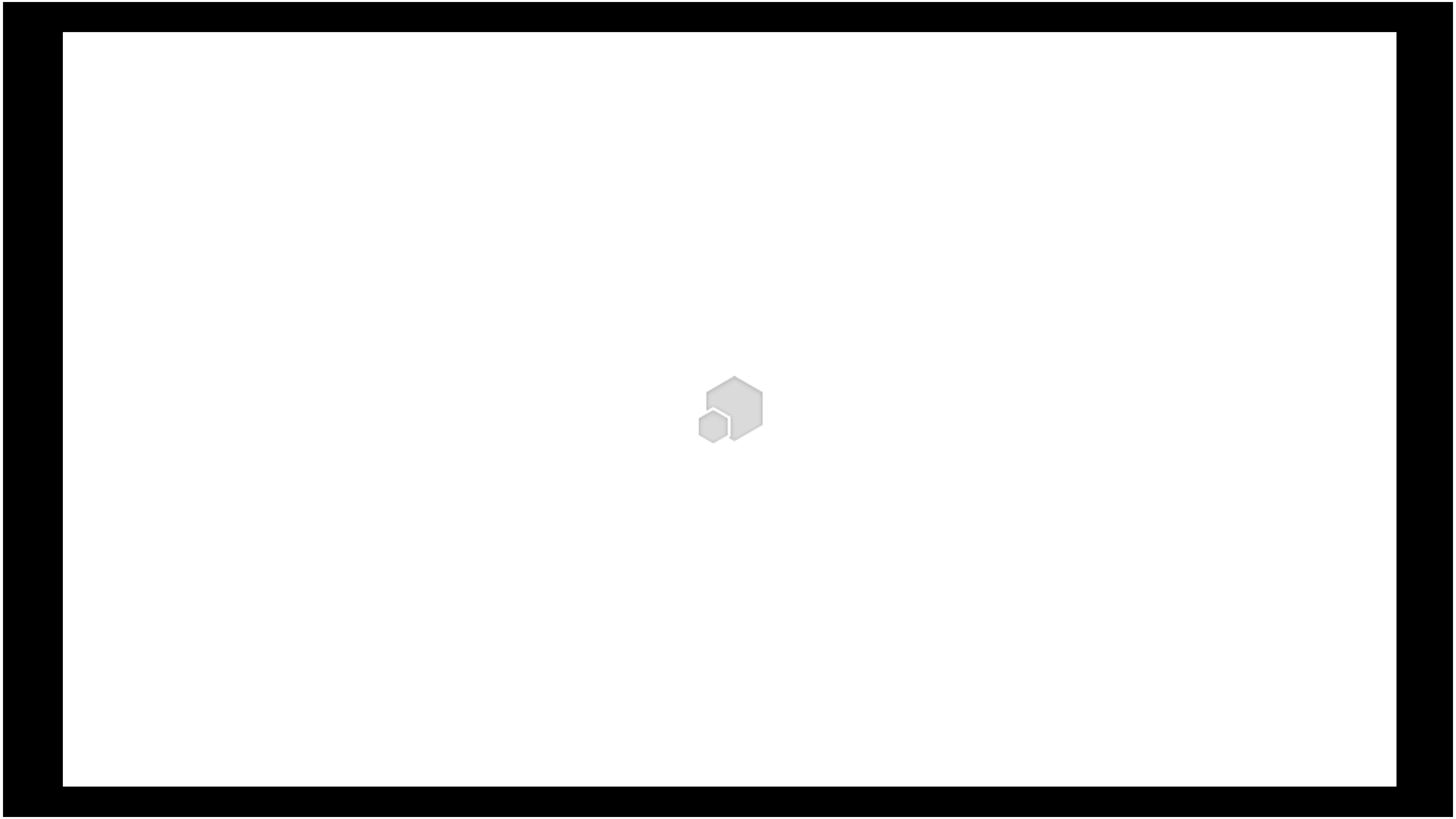


- Pros
  - O(#priorities) scheduler
- Cons
  - How to assign priorities to processes?

Assigning dynamic priorities to processes

- We'd like to assign higher priorities to "cool" processes
    - Which need to react quickly to events?
    - Which perform a lot of I/O?
    - Which won't use a full quantum of time (10ms) next time?

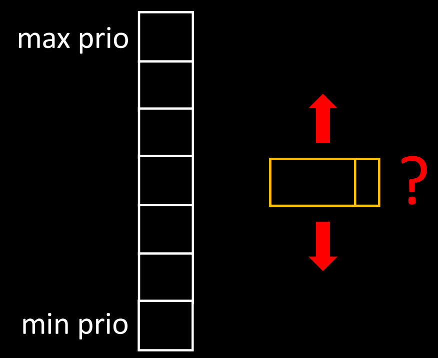




Assigning dynamic priorities to processes

- We'd like to assign higher priorities to "cool" processes
  - Which need to react quickly to events?
  - Which perform a lot of I/O?
  - Which won't use a full quantum of time (10ms) next time?

- How do we know?
  - People can change…
    - *"If I can change, and you can change, everybody can change!"*
      [Rocky Balboa, 1985]



Predicting the Future ?

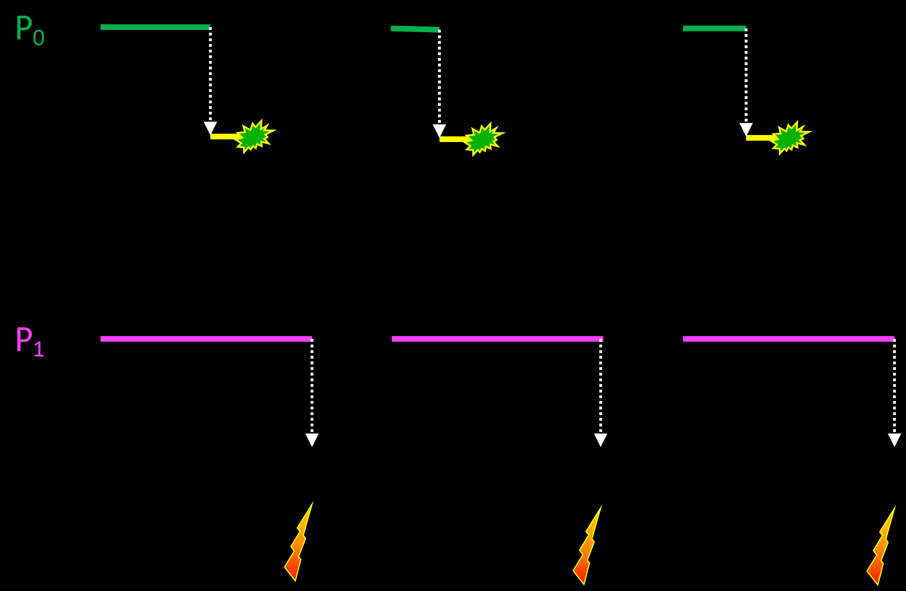
Predicting the Future

- By looking at the past!
  - If a process kept behaving well so far…
    …it will probably do so next time we schedule it!

Predicting the Future

- By looking at the past!
  - If a process kept behaving well so far…
    …it will probably do so next time we schedule it!

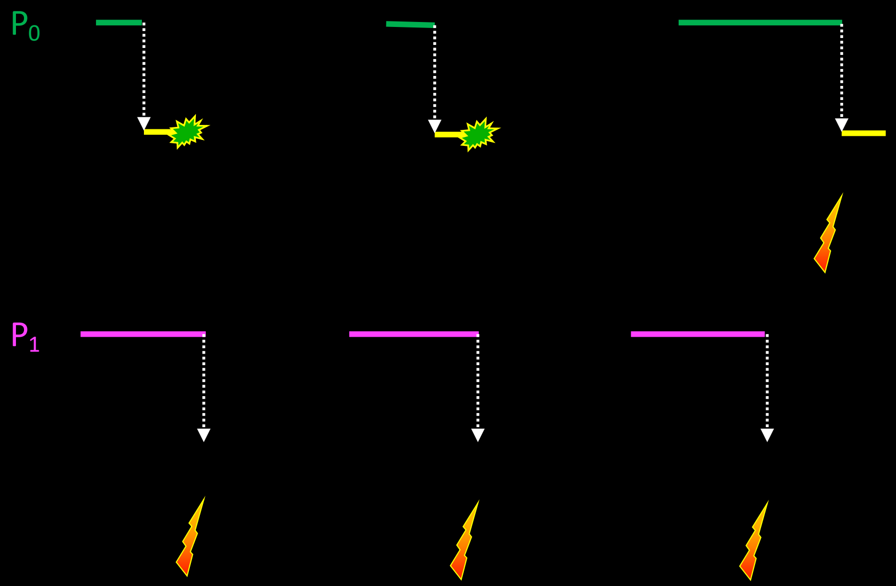
- P_0 looks more friendly than P_1



Predicting the Future

- By looking at the past!
  - If a process kept behaving well so far…
    …it will probably do so next time we schedule it!

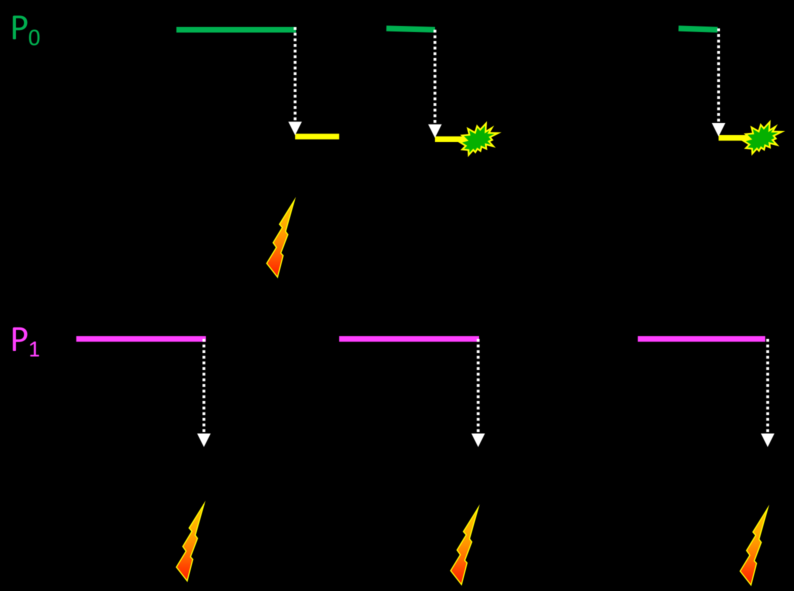
- P_0 looks more friendly than P_1
  - Really?



Predicting the Future

- By looking at the past!
  - If a process kept behaving well so far…
    …it will probably do so next time we schedule it!

- P_0 looks more friendly than P_1
  - Really?
  - Can we forgive P_0?



Estimating duration of the next quantum

- T_n: CPU utilization observed at step n
- E_n: estimation of the CPU utilization time at step n

Estimating duration of the next quantum

- T_n: CPU utilization observed at step n
- E_n: estimation of the CPU utilization time at step n
  - $E_n = \alpha(T_{n-1}) + (1 - \alpha)E_{n-1}$

Estimating duration of the next quantum

- T_n: CPU utilization observed at step n
- E_n: estimation of the CPU utilization time at step n
  - $E_n = \alpha(T_{n-1}) + (1-\alpha)E_{n-1}$

- $\alpha = 0$
  - Fixed, a priori estimation
- $\alpha = 1$
  - We only look at the last period
- $\alpha = \frac{1}{2}$
  - $E_1 = T_0$
  - $E_2 = \frac{1}{2}T_1 + \frac{1}{2}T_0$
  - $E_3 = \frac{1}{2}T_2 + \frac{1}{4}T_1 + \frac{1}{4}T_0$

From Estimation to Priority

- OK, we can predict how long each process will run next time it is scheduled
  - Which process do we choose?
- Try to maximize average happiness!
  - Think about queues at the supermarket!

From Estimation to Priority

- To maximize average happiness
  - We should minimize average waiting time
    - Schedule shortest jobs first!

From Estimation to Priority

- To maximize average happiness
  - We should minimize average waiting time
    - Schedule shortest jobs first!
- Priority should be inversely proportional to E_n
  - Interactive Operating Systems schedulers try, more or less, to follow this strategy

Strategy used in Linux 2.4.x kernels

- Credits are assigned to processes, based on their fixed priority
  - Sort of “*pocket money*”
- To run on the CPU, a process must spend money
  - No more money = no CPU
- At some point, no more ready processes have money left
  - The kernel restarts a new epoch and redistributes credits

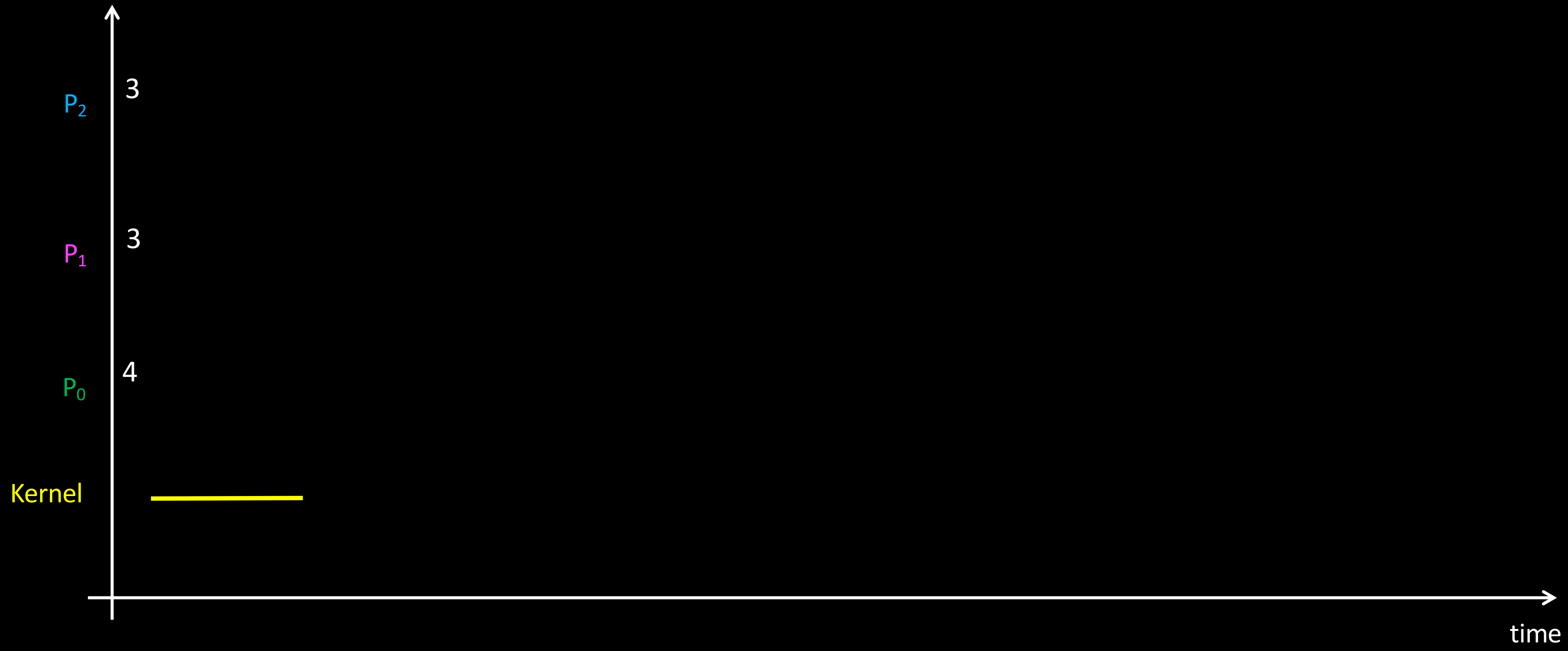
Strategy used in Linux 2.4.x kernels

- Credits are assigned to processes, based on their fixed priority
  - Let us take a concrete, simple example with 3 processes
  - Initially:
    - P_0 has 4 credits
    - P_1 has 3 credits
    - P_2 has 3 credits

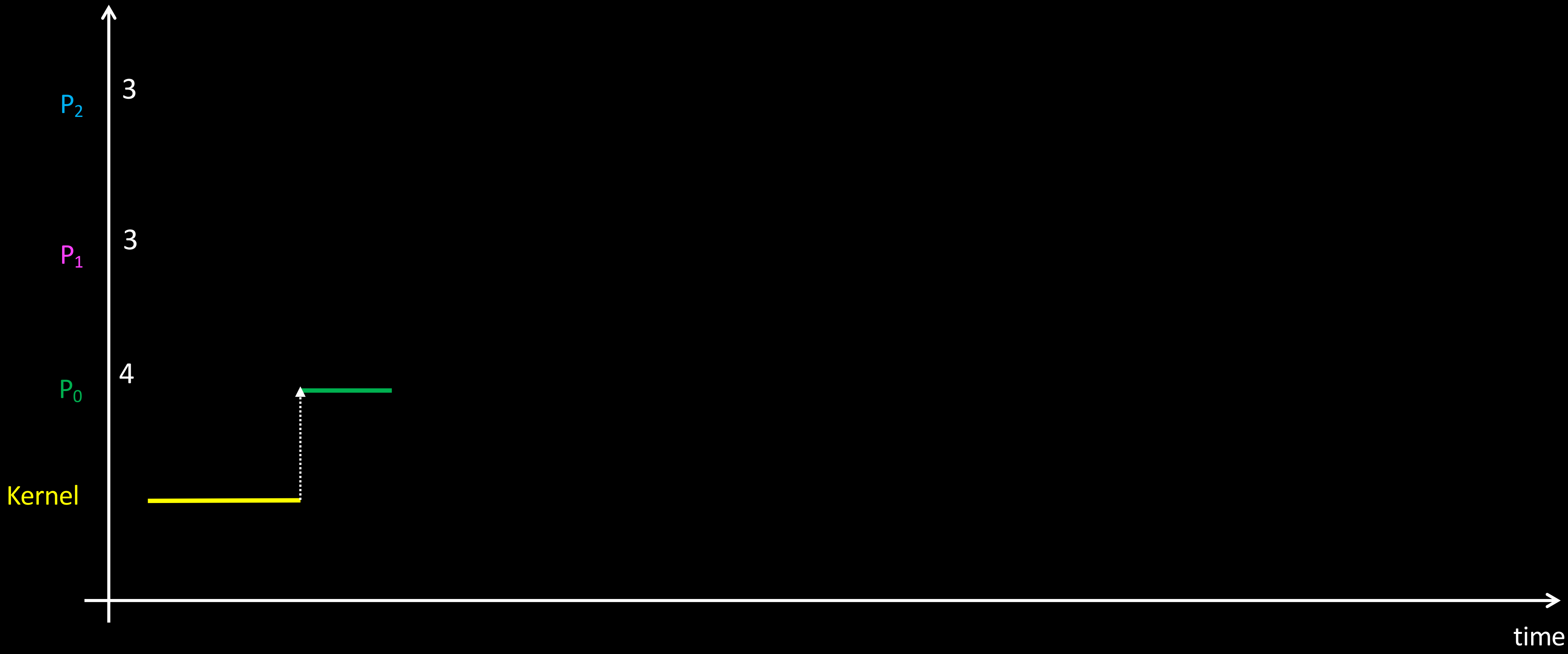
Strategy used in Linux 2.4.x kernels

- Credits are assigned to processes, based on their fixed priority
  - Let us take a concrete, simple example with 3 processes
  - Initially:
    - P_0 has 4 credits
    - P_1 has 3 credits
    - P_2 has 3 credits
  - Rich people are usually privileged, aren't they?
    - So P_0 will be the next running process

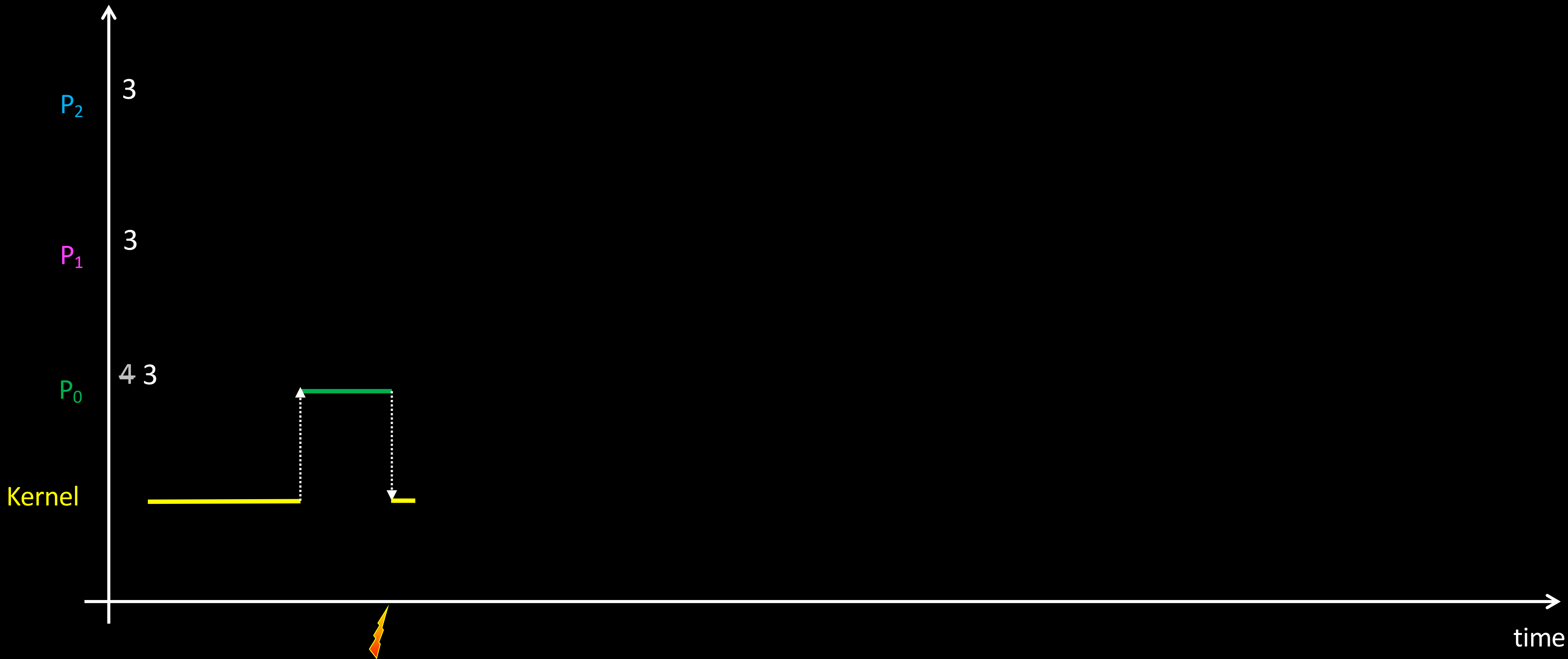
Strategy used in Linux 2.4.x kernels



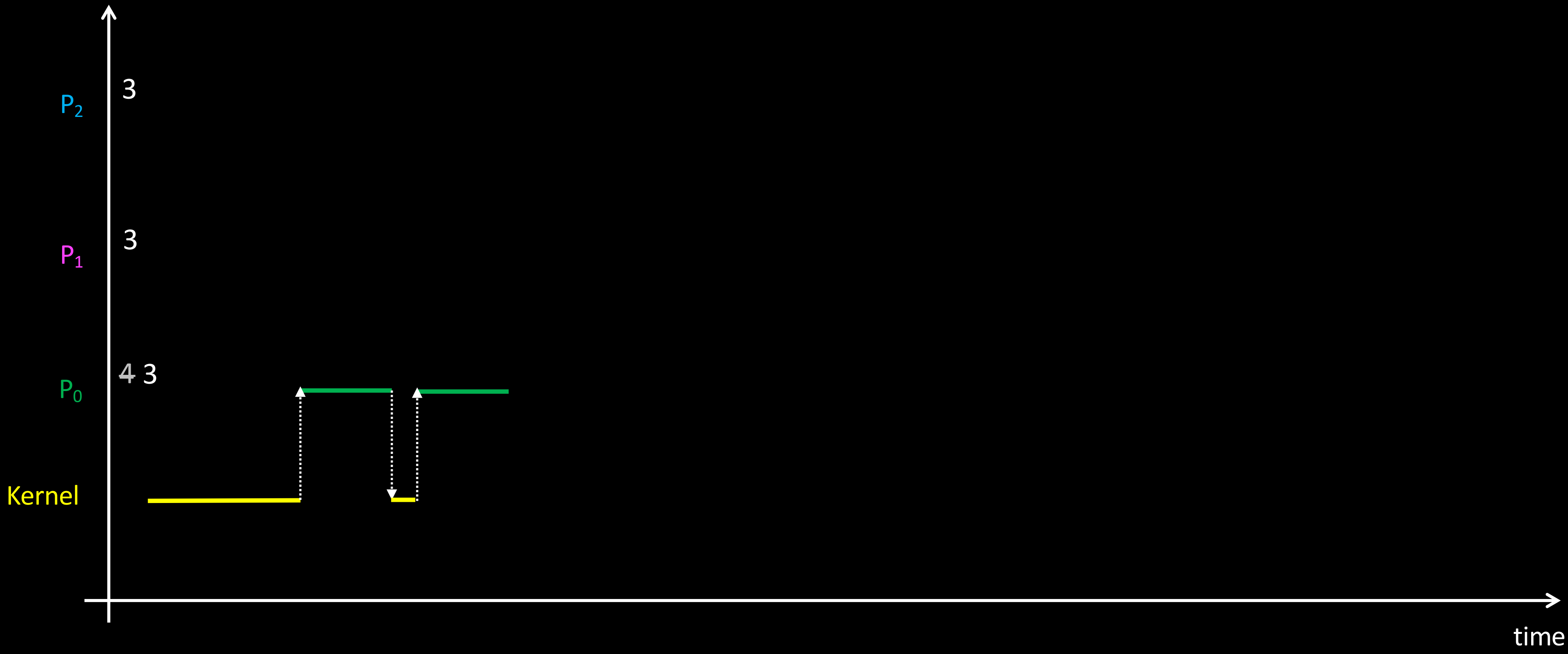
Strategy used in Linux 2.4.x kernels



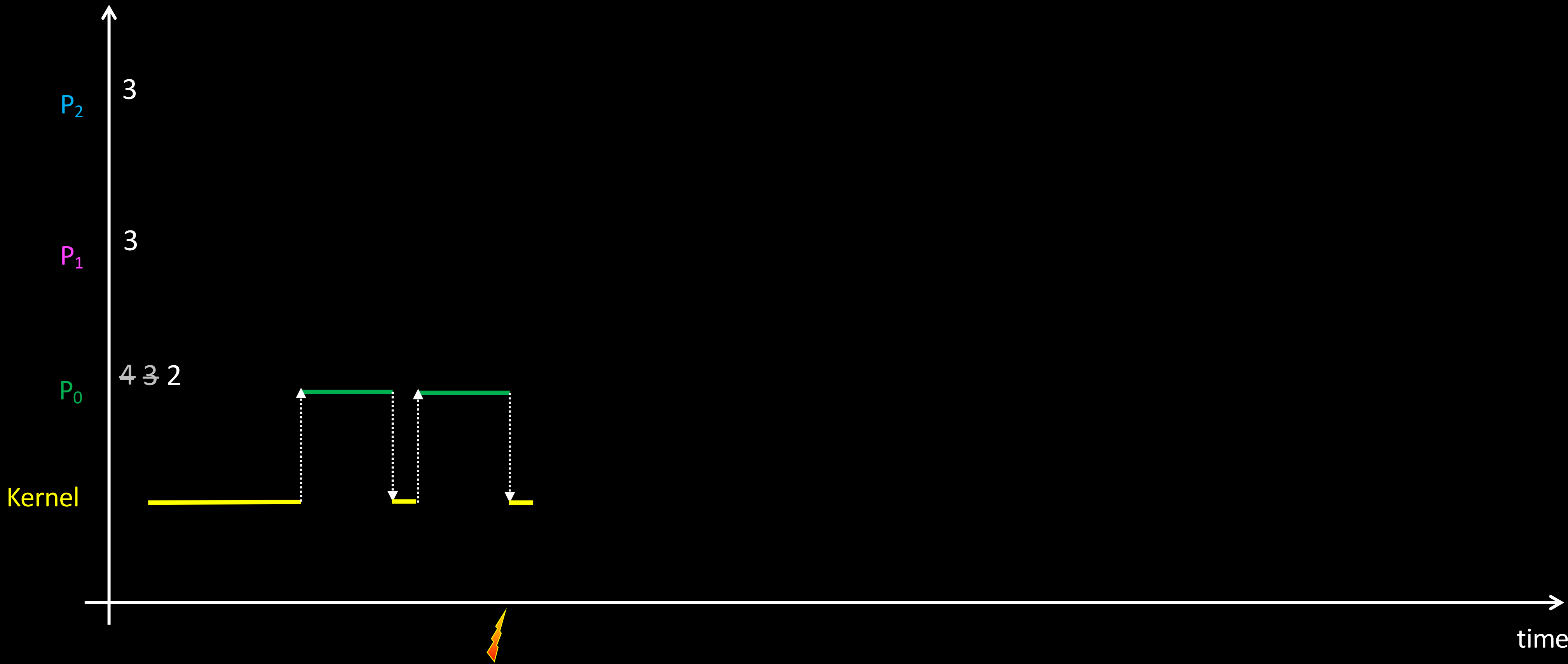
Strategy used in Linux 2.4.x kernels



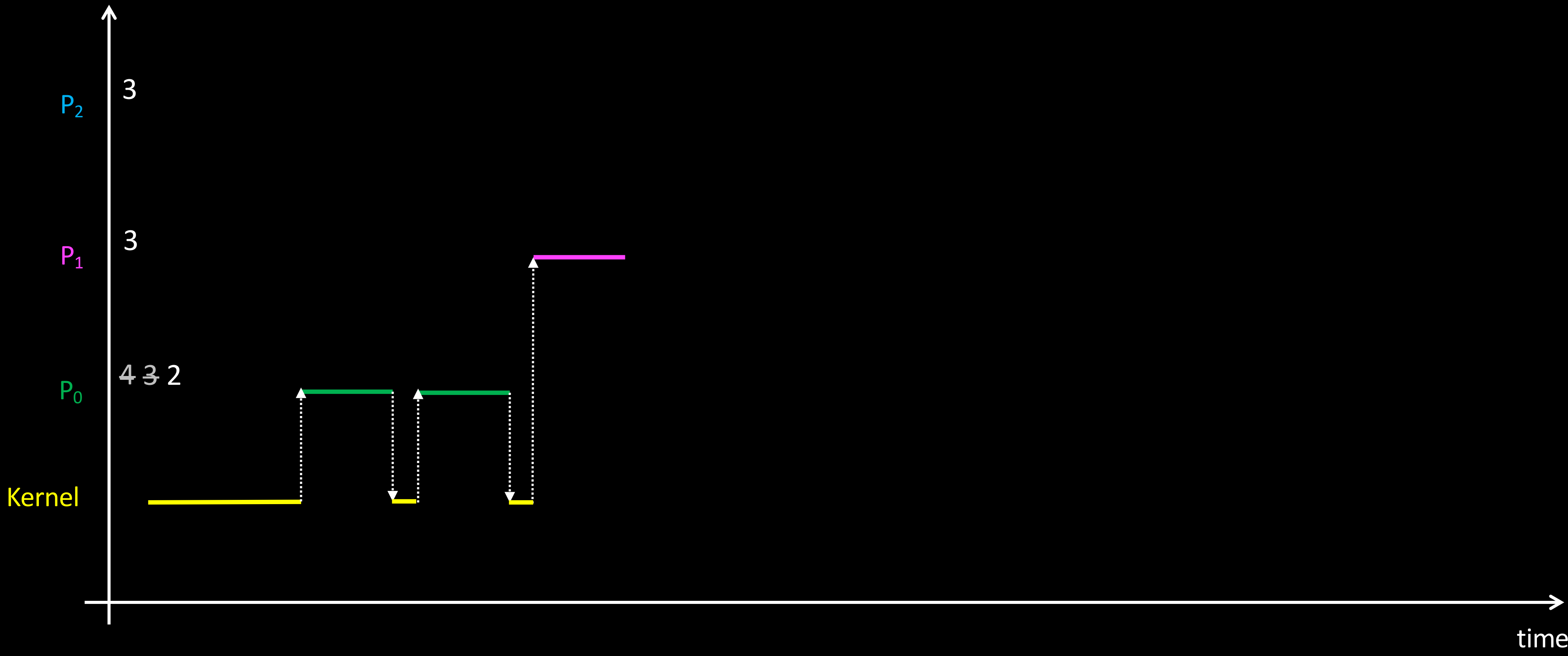
Strategy used in Linux 2.4.x kernels



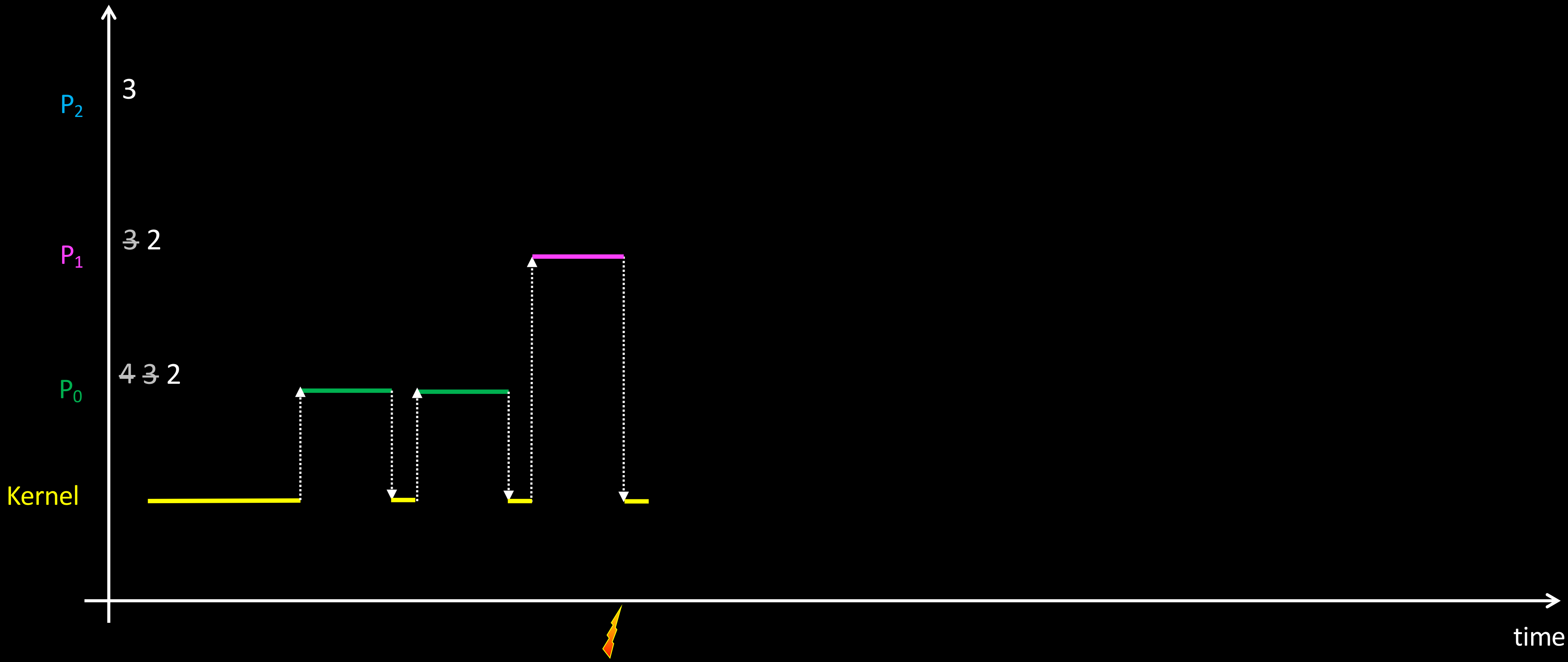
Strategy used in Linux 2.4.x kernels



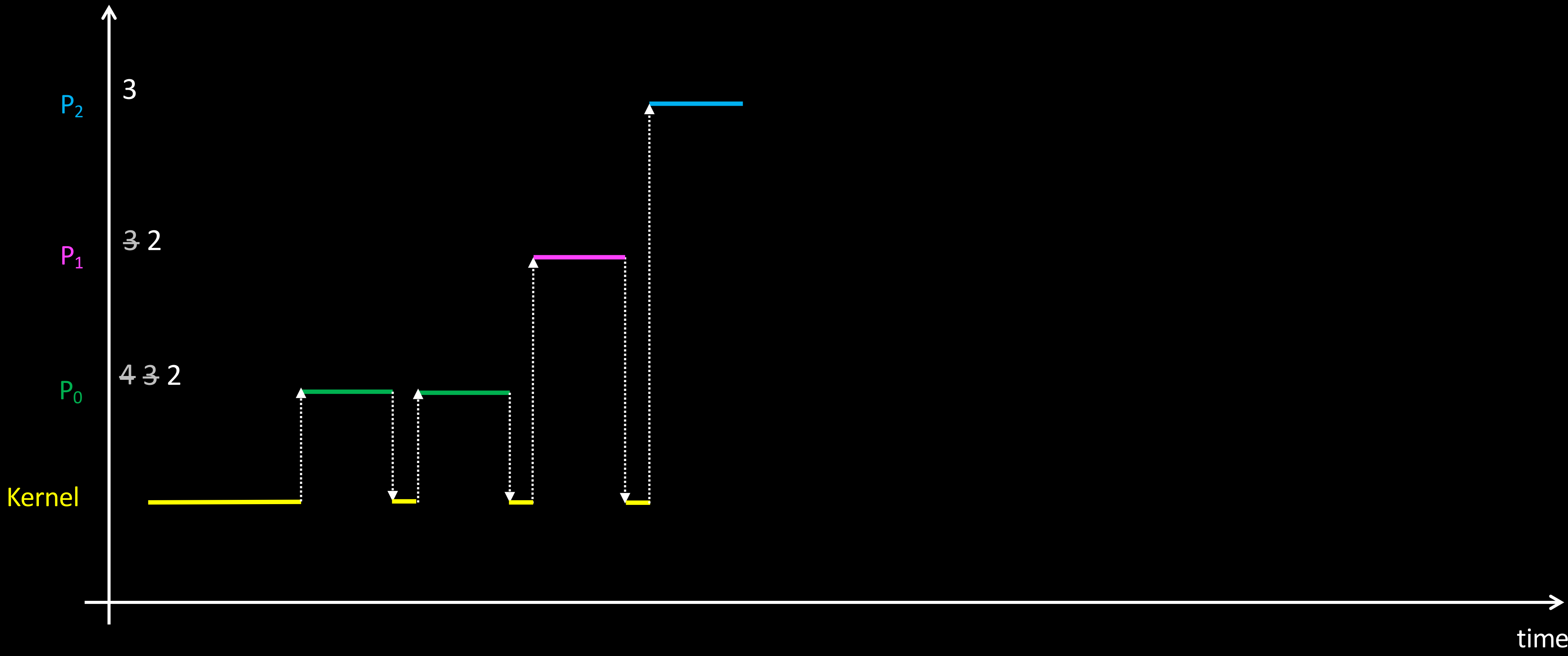
Strategy used in Linux 2.4.x kernels



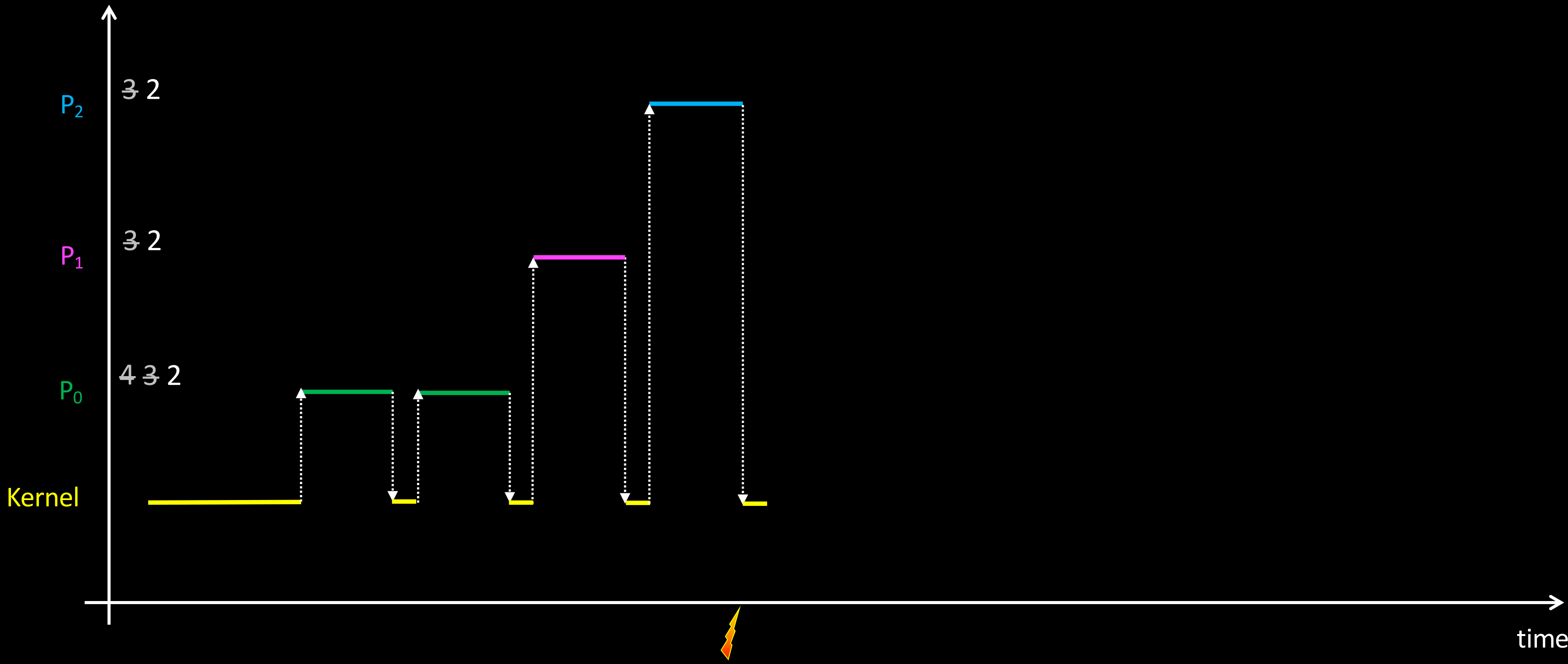
Strategy used in Linux 2.4.x kernels



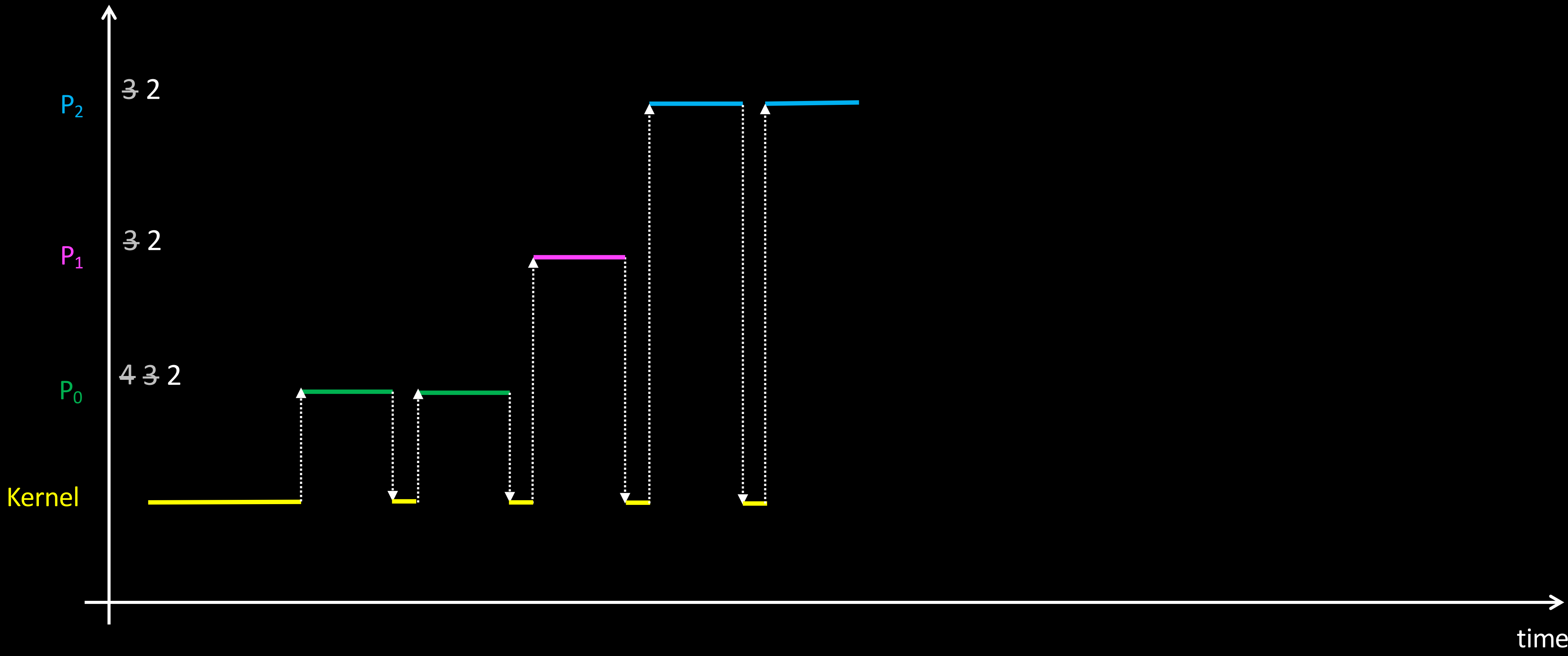
Strategy used in Linux 2.4.x kernels



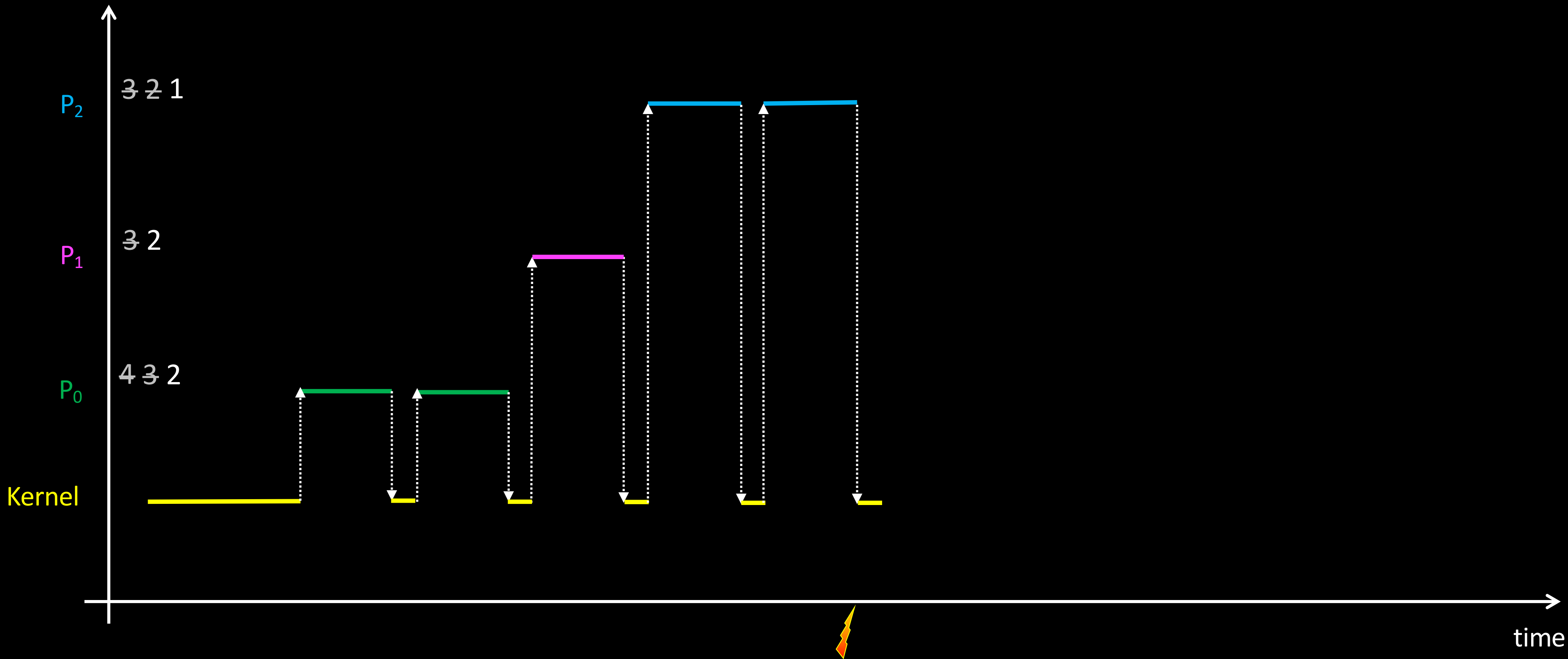
Strategy used in Linux 2.4.x kernels



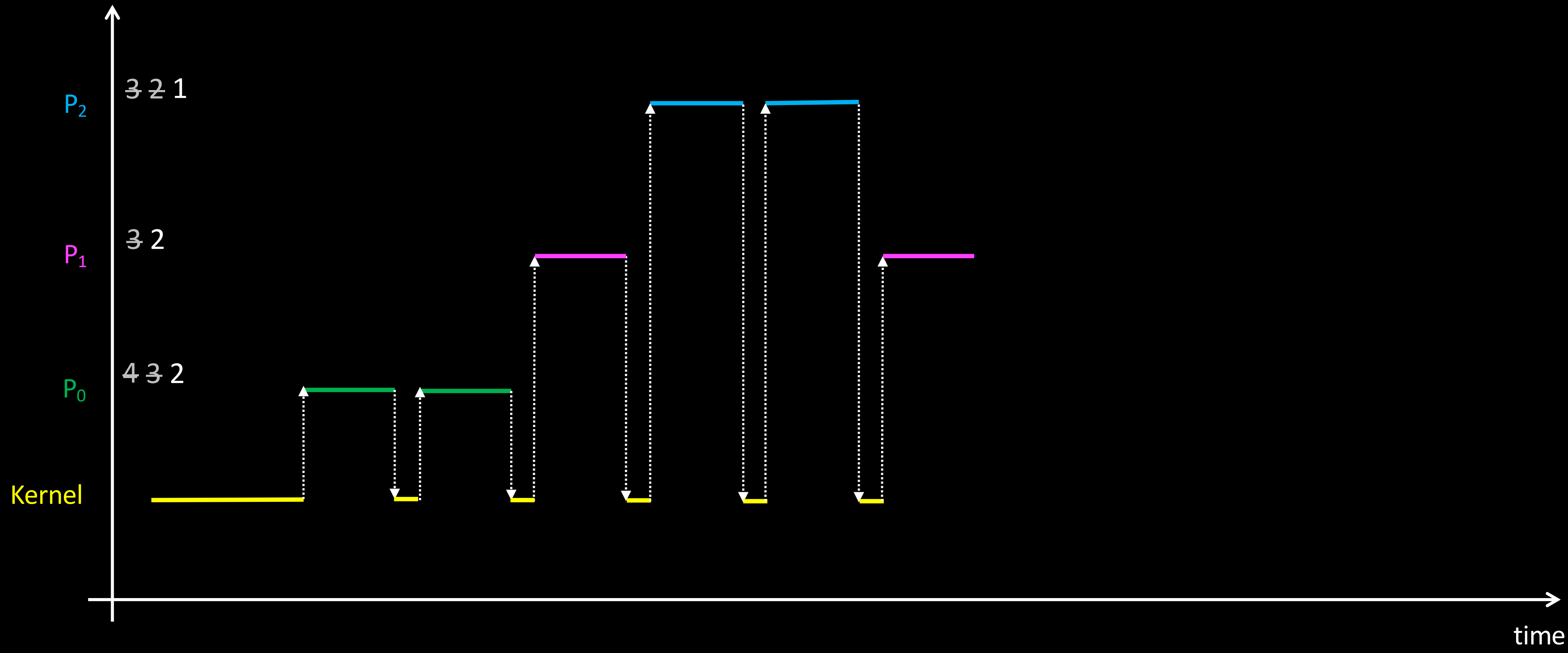
Strategy used in Linux 2.4.x kernels



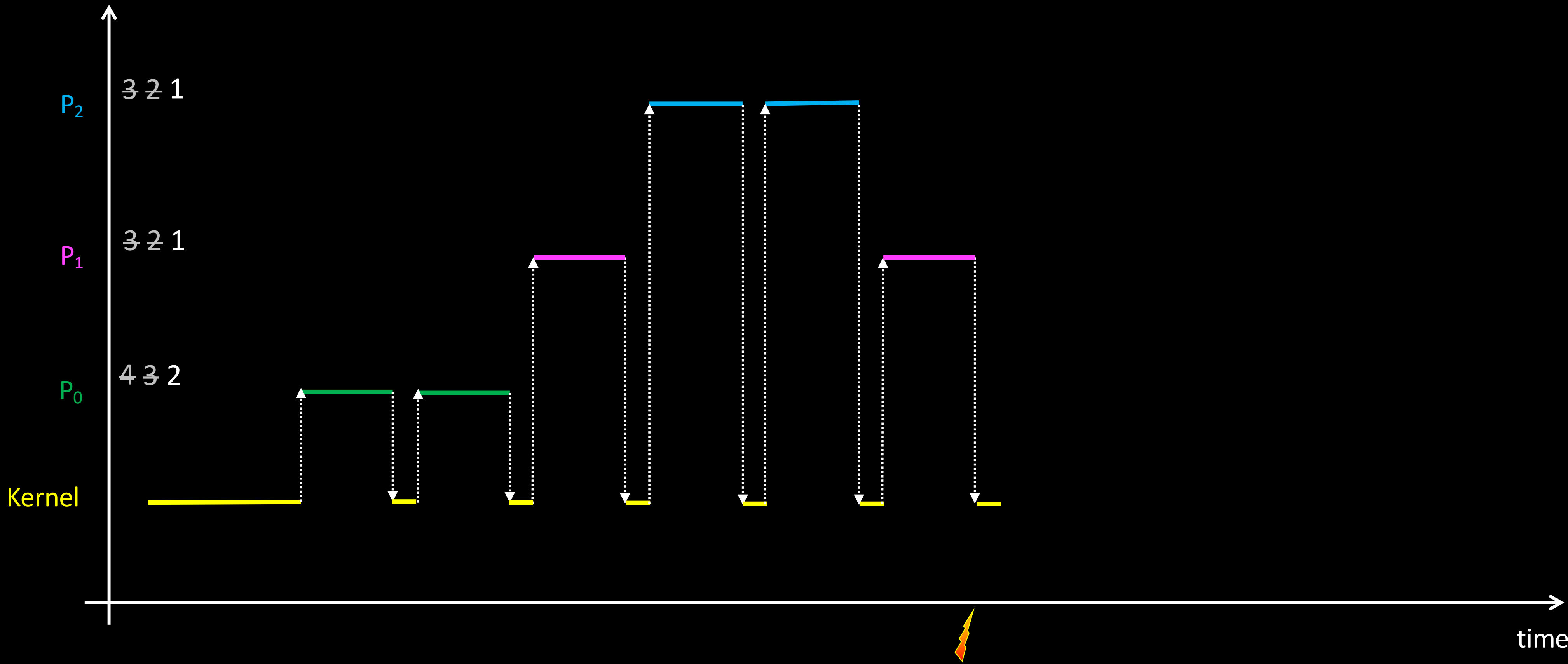
Strategy used in Linux 2.4.x kernels



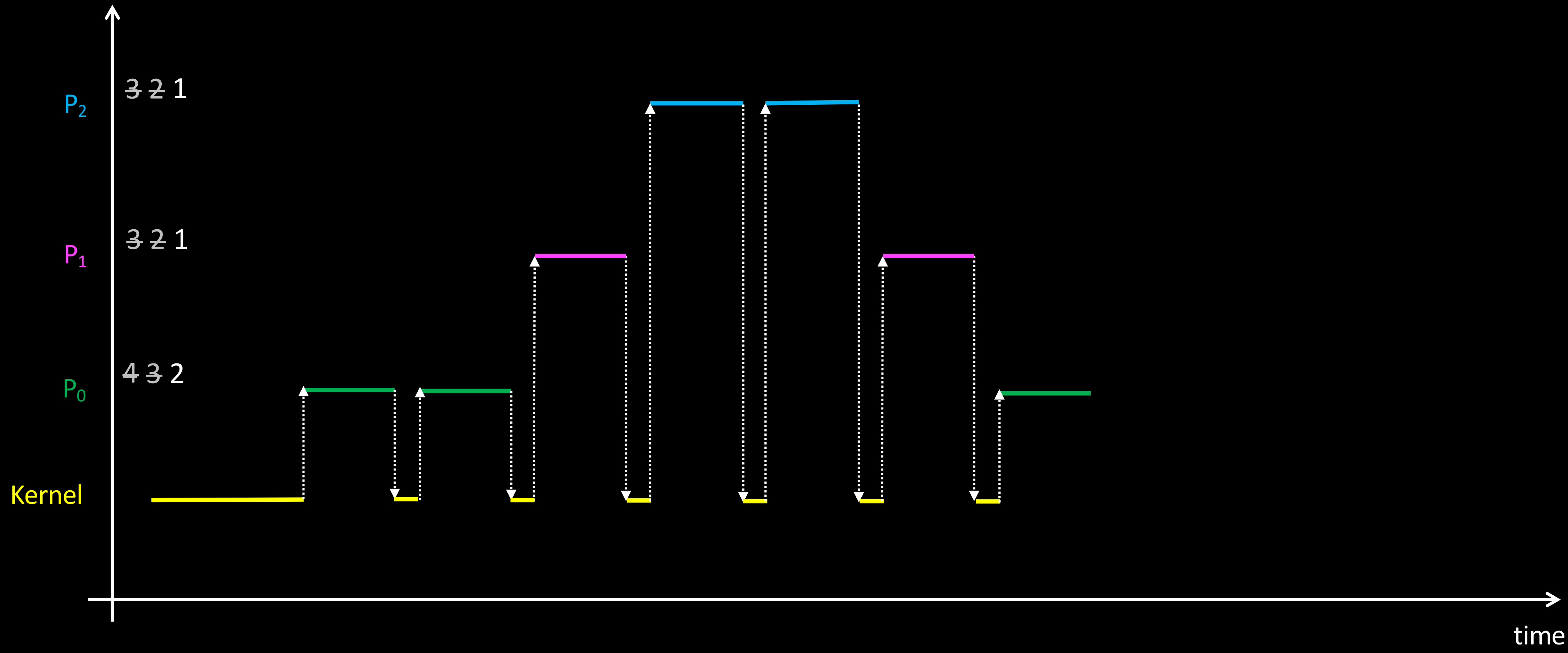
Strategy used in Linux 2.4.x kernels



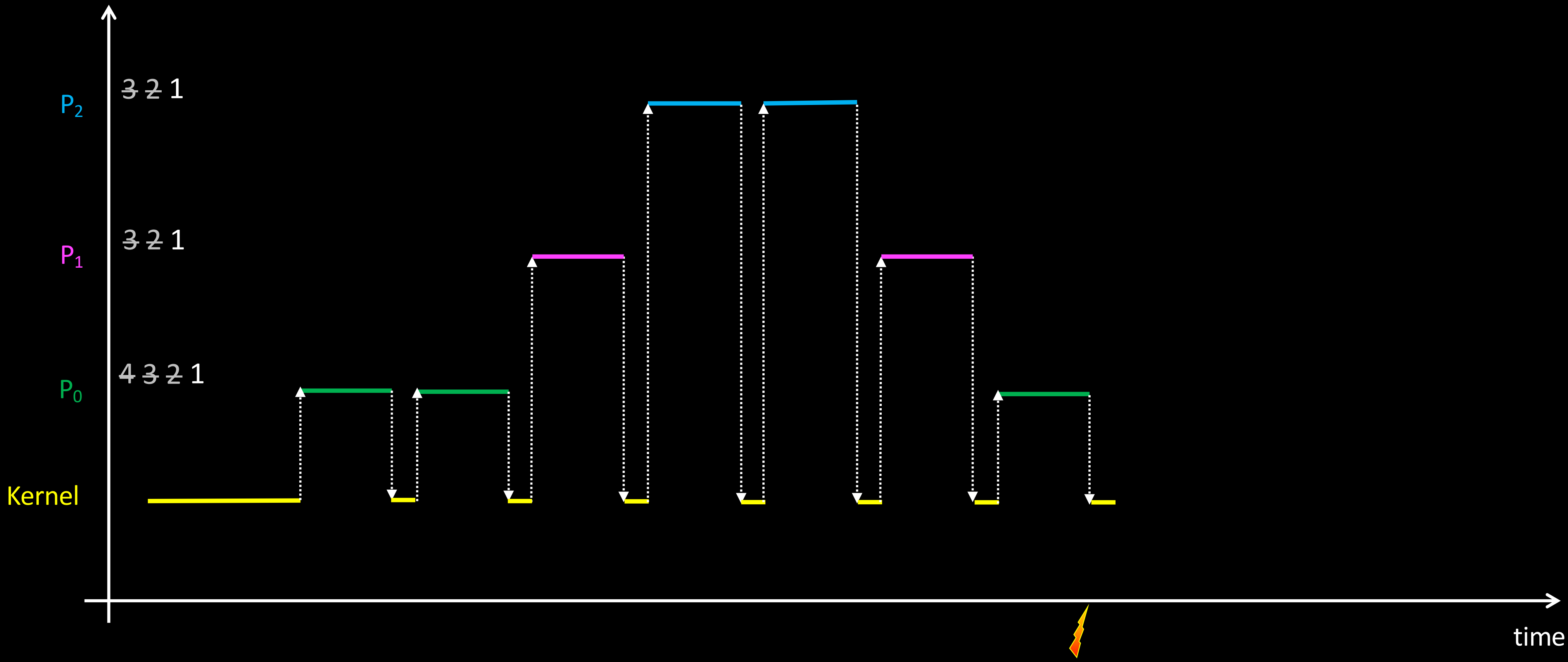
Strategy used in Linux 2.4.x kernels



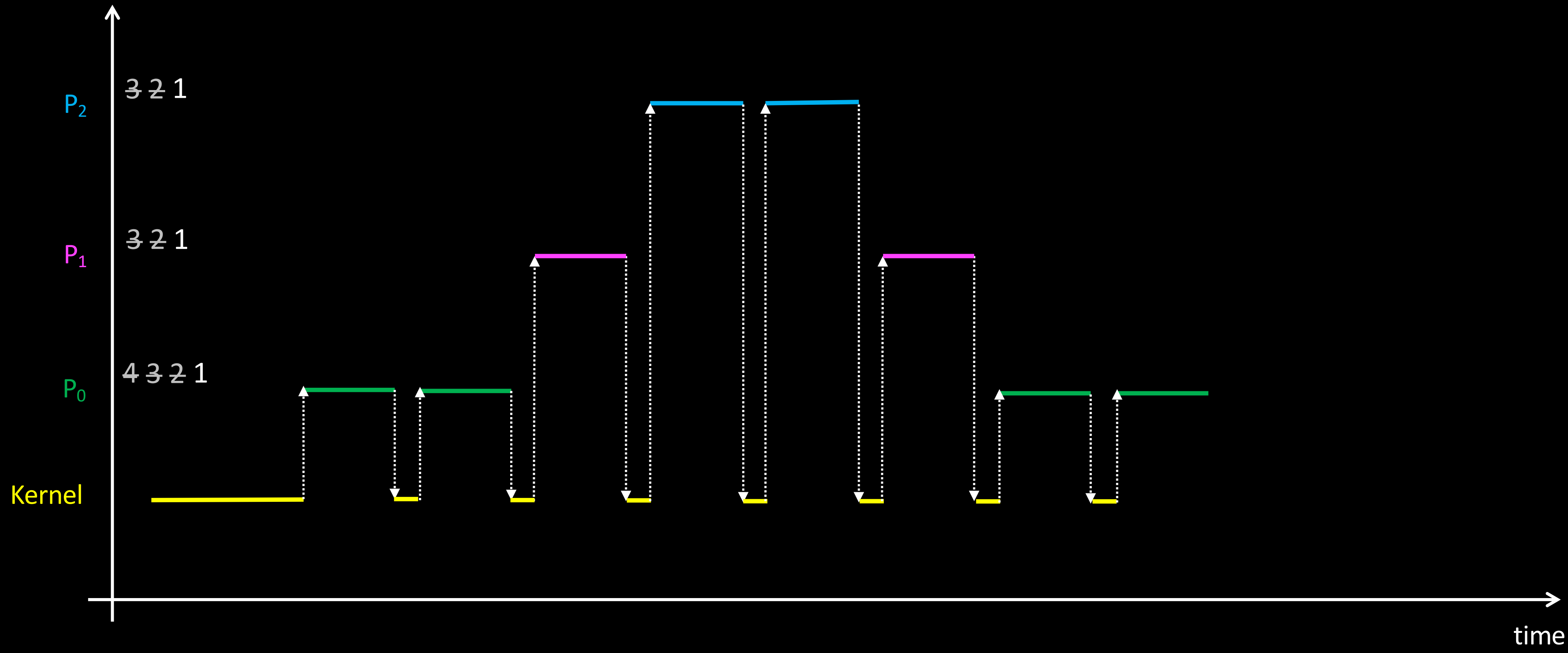
Strategy used in Linux 2.4.x kernels



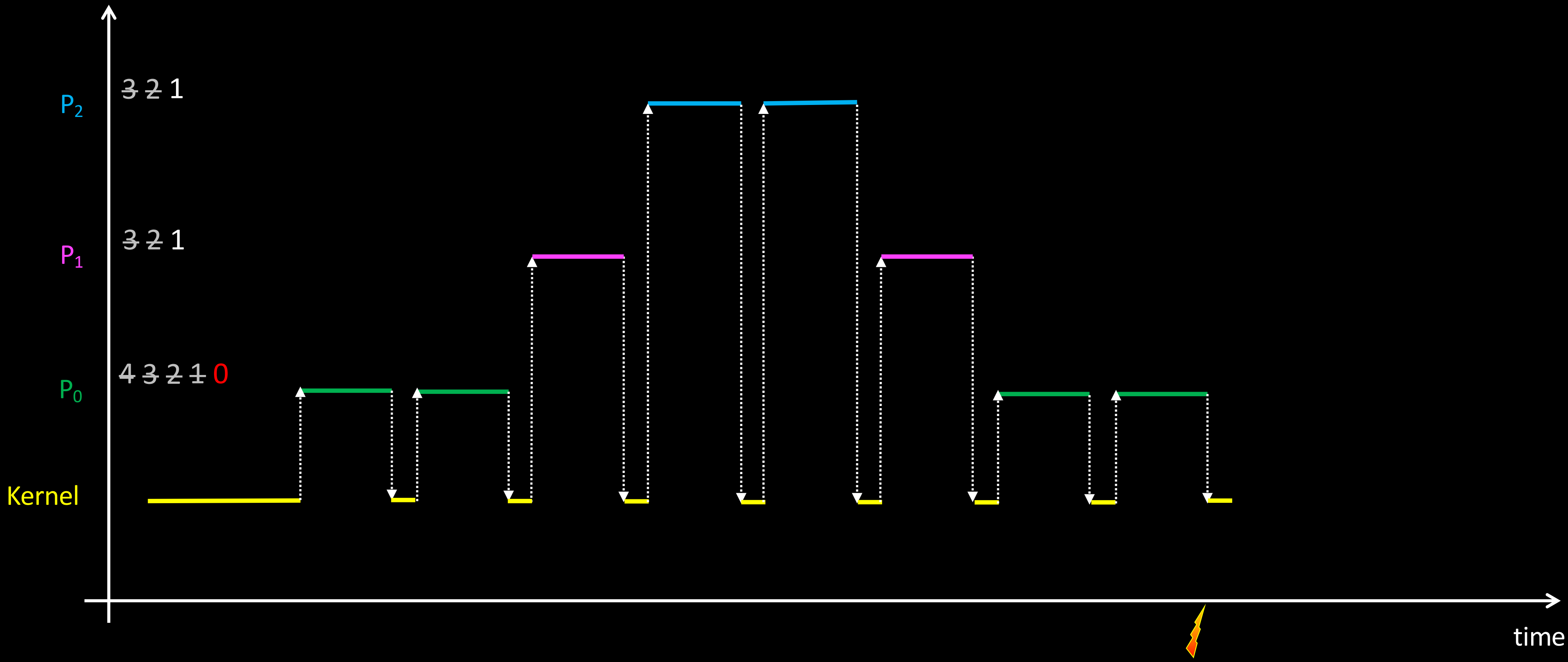
Strategy used in Linux 2.4.x kernels



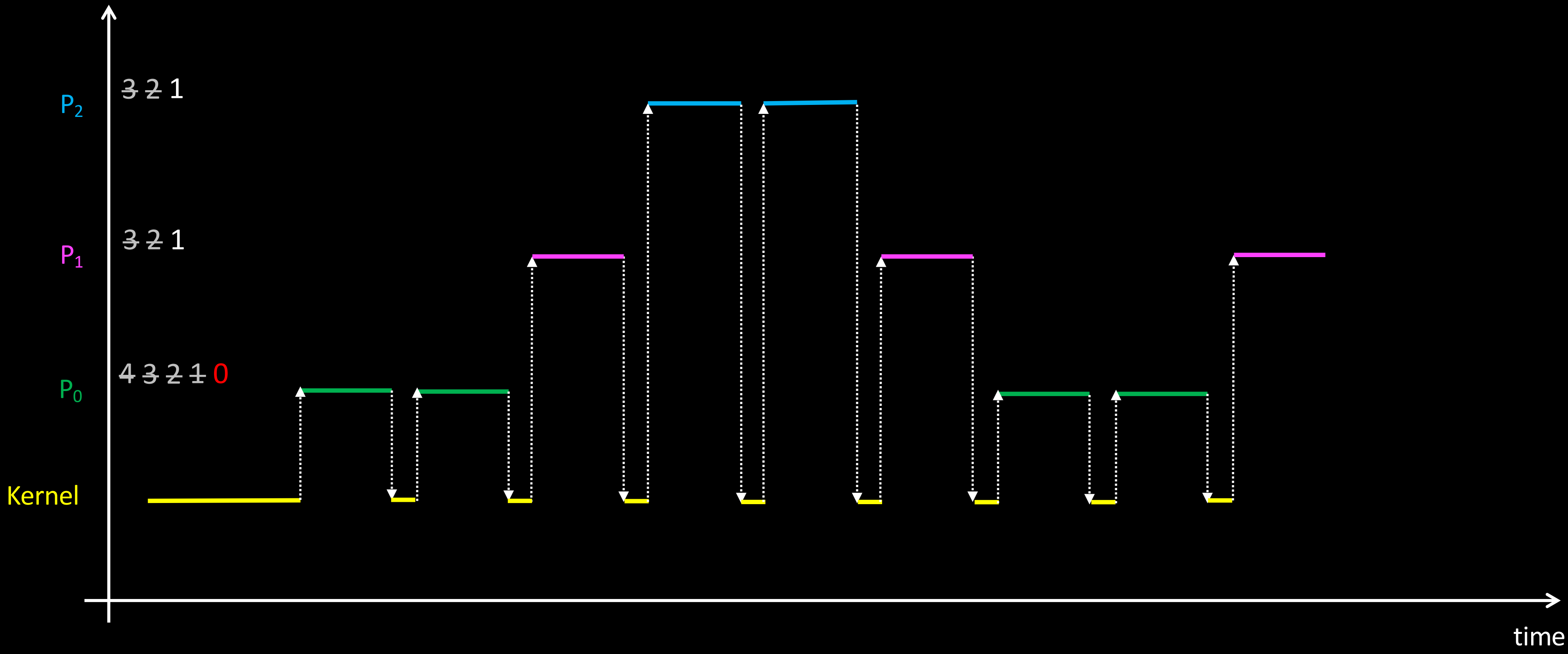
Strategy used in Linux 2.4.x kernels



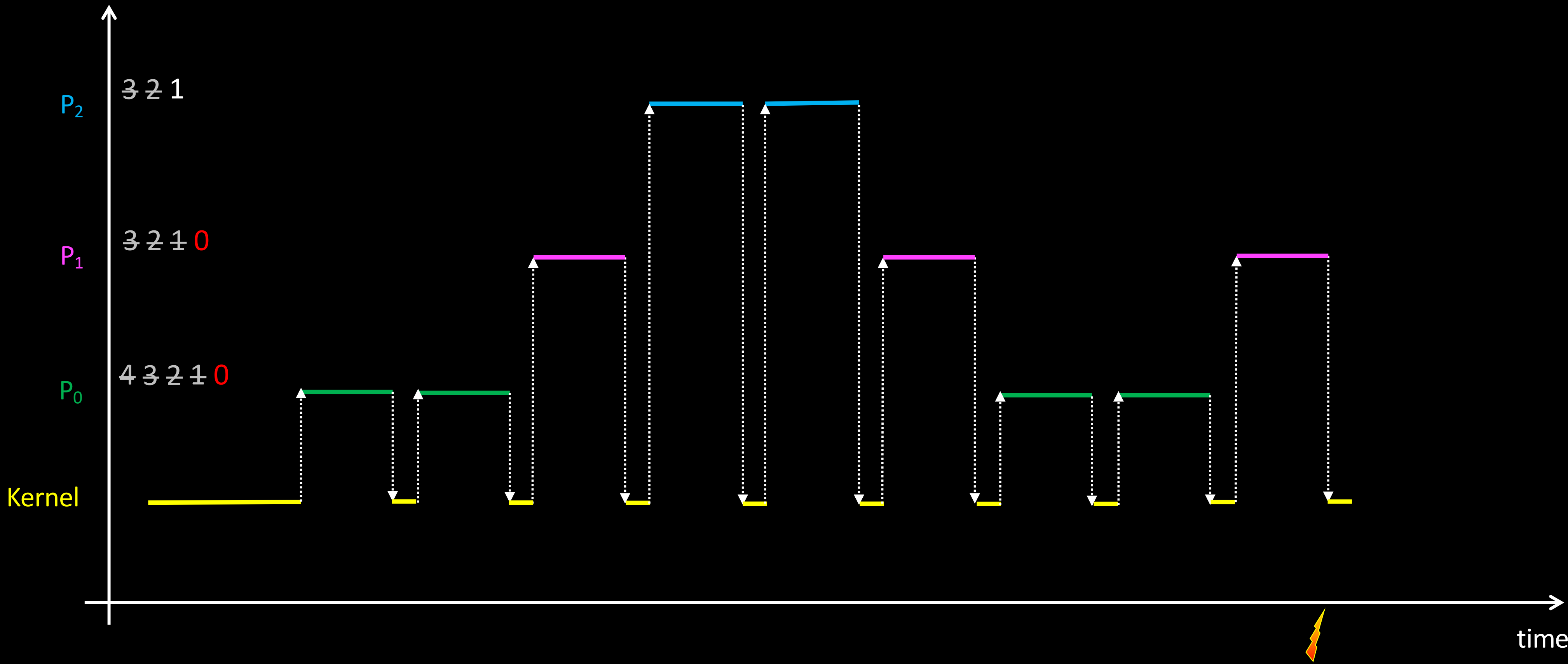
Strategy used in Linux 2.4.x kernels



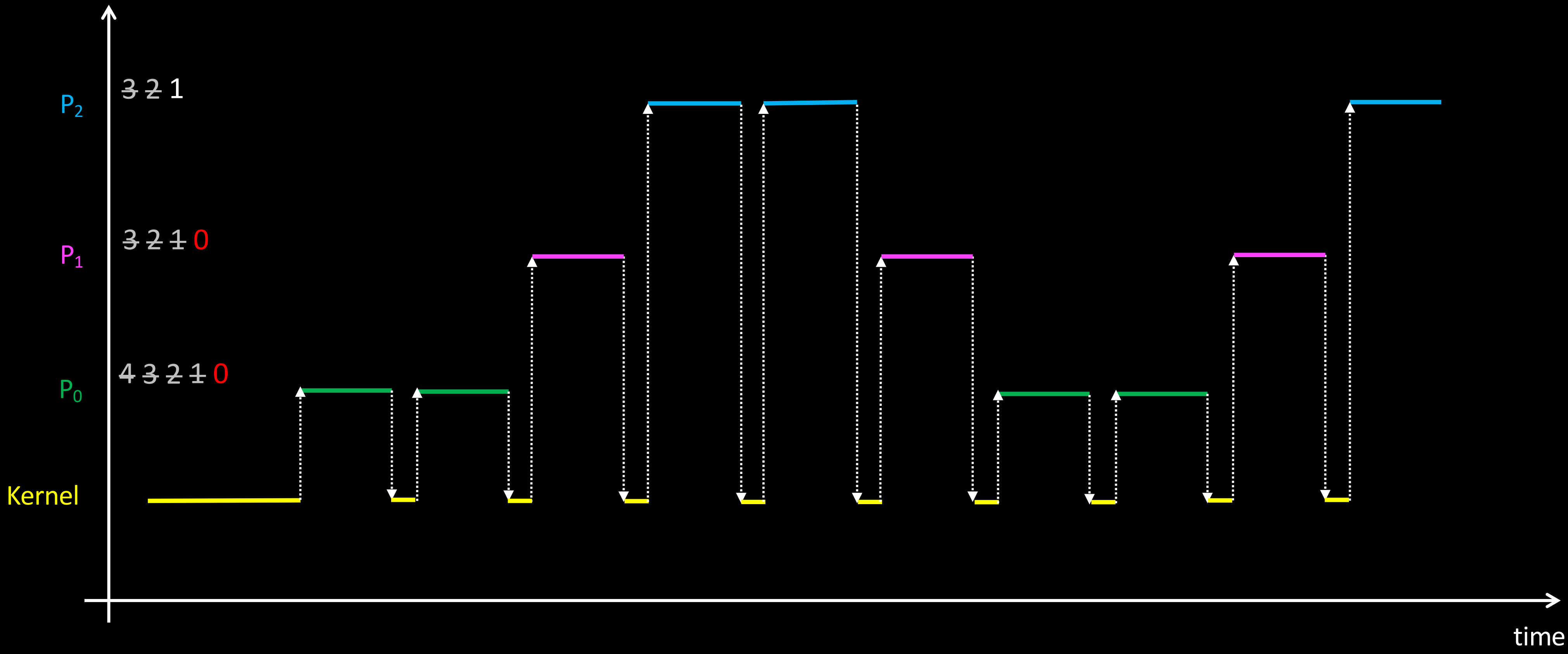
Strategy used in Linux 2.4.x kernels



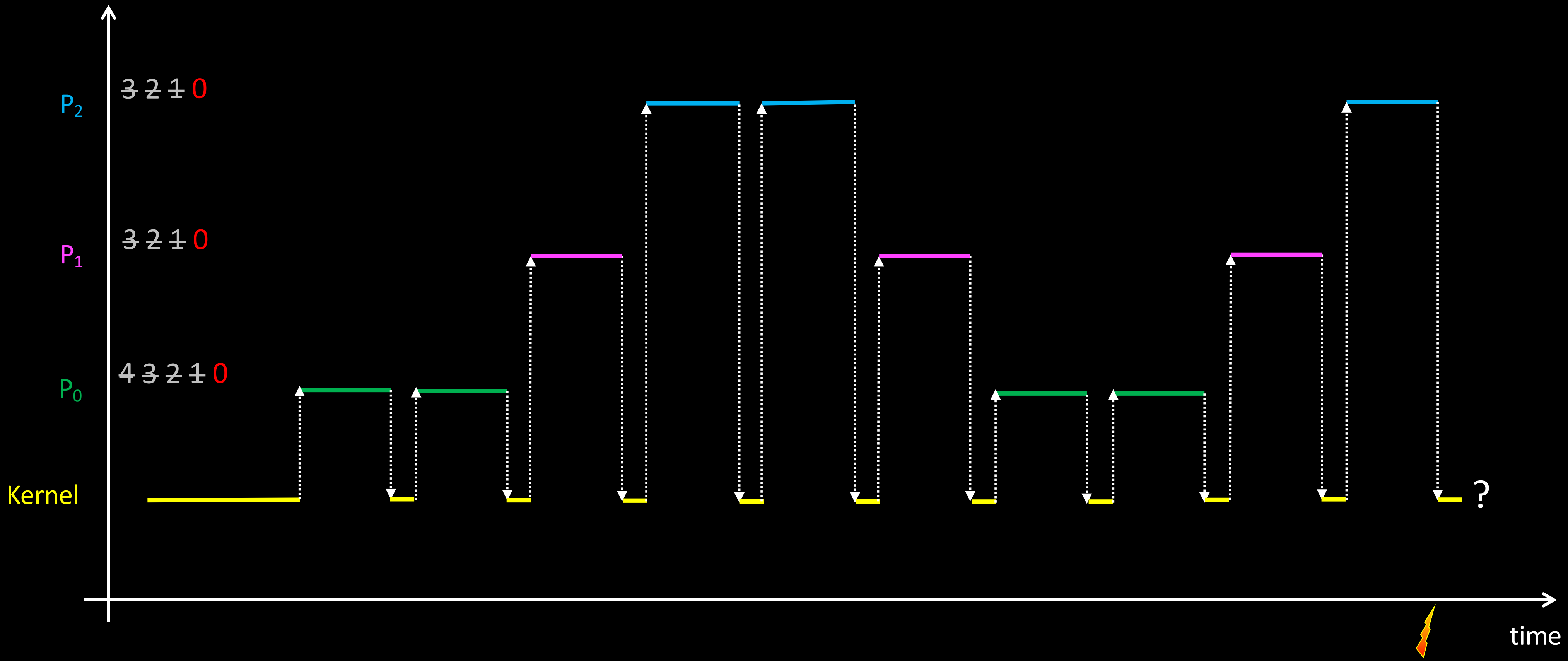
Strategy used in Linux 2.4.x kernels



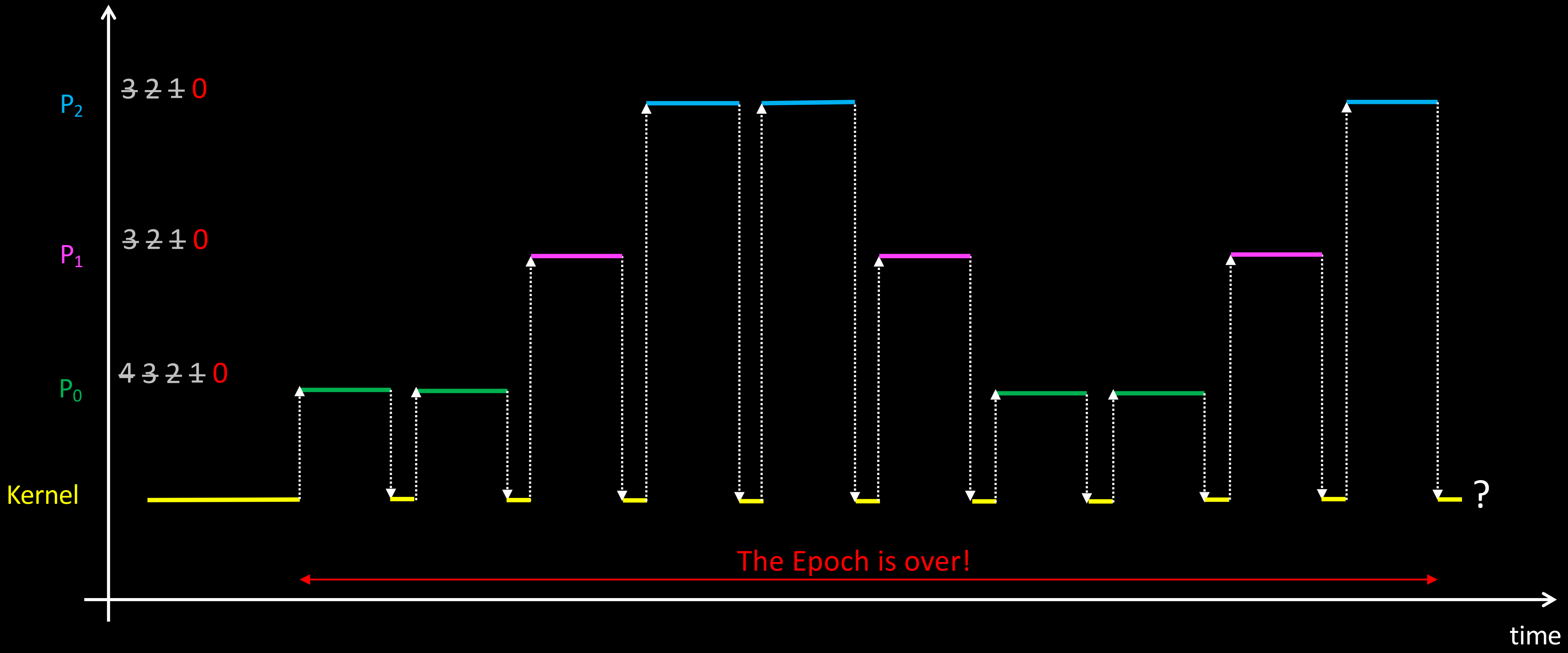
Strategy used in Linux 2.4.x kernels



Strategy used in Linux 2.4.x kernels



Strategy used in Linux 2.4.x kernels



Strategy used in Linux 2.4.x kernels

- When all “ready processes” are short of credits, Linux starts a new Epoch
  - Money is credited back to all processes
  - The same way you give money to your kids every month…
    - Duration of an Epoch is unknown, though

Strategy used in Linux 2.4.x kernels

- When all “ready processes” are short of credits, Linux starts a new Epoch
  - Money is credited back to all processes
  - The same way you give money to your kids every month…
    - Duration of an Epoch is unknown, though
- Uh, wait… Really?
  - What if a process did not spend all its credits?
    - In other words: one of your kids is secretly saving money…

Strategy used in Linux 2.4.x kernels

- Uh, wait… Really?
  - What if a process did not spend all its credits?
    - In other words: one of your kids is secretly saving money…

Strategy used in Linux 2.4.x kernels

- To avoid infinite accumulation of credits
  - One solution is to introduce a tax!

- At the beginning of a new Epoch, each process receives
  - to_credits(priority) + remaining_credits/2

Strategy used in Linux 2.4.x kernels

- To avoid infinite accumulation of credits
  - One solution is to introduce a tax!

- At the beginning of a new Epoch, each process receives
  - to_credits(priority) + remaining_credits/2

- In the worst case, a process can accumulate
  - C
  - C + C/2
  - C + C/2 + C/4
  - C + C/2 + C/4 + C/8
  - C + C/2 + C/4 + C+8 + …

- Bounded by 2C

Strategy used in Linux 2.4.x kernels

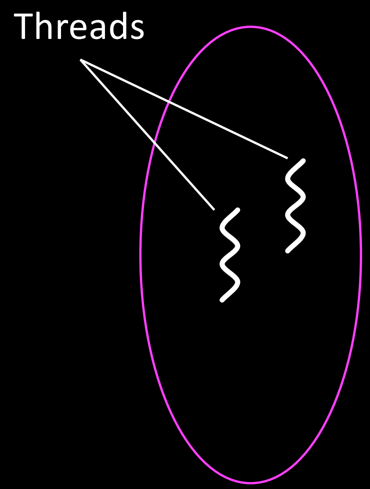
- We’re now ready to explore how this is implemented!

Scheduling on multicore machines

- Each core runs the scheduler asynchronously
  - Timer interrupts not necessarily synchronized
- The ready list can be
  - Shared by all cores
    - How to prevent multiple cores from choosing the same process simultaneously?
  - Distributed among cores
    - How to balance ready threads fairly? How often?
- Local scheduling decisions can require “reschedule” operations on other cores

Processes and Threads

- Threads = Execution context
- Process = Thread + Address Space
- Several threads can share the same address space



Process featuring 2 threads

Processes and Threads

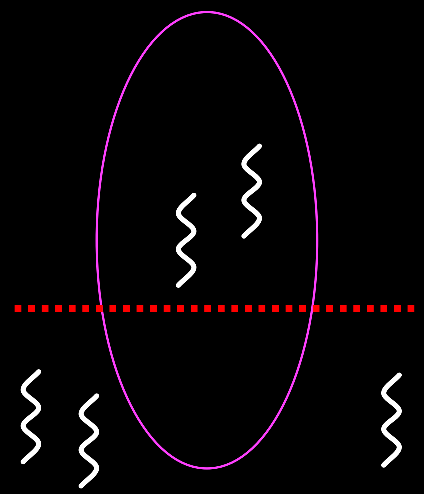
```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int NBTHREADS = 1;

void *func (void *arg)
{
  printf ("Hello from %s\n", arg);

  return NULL;
}
```

```c
int main (int argc, char *argv[])
{
  pthread_t pid;

  pthread_create (&pid, NULL, func, “thread”);

  printf ("Hello from main\n");

  pthread_join (pid, NULL);

  return 0;
}
```

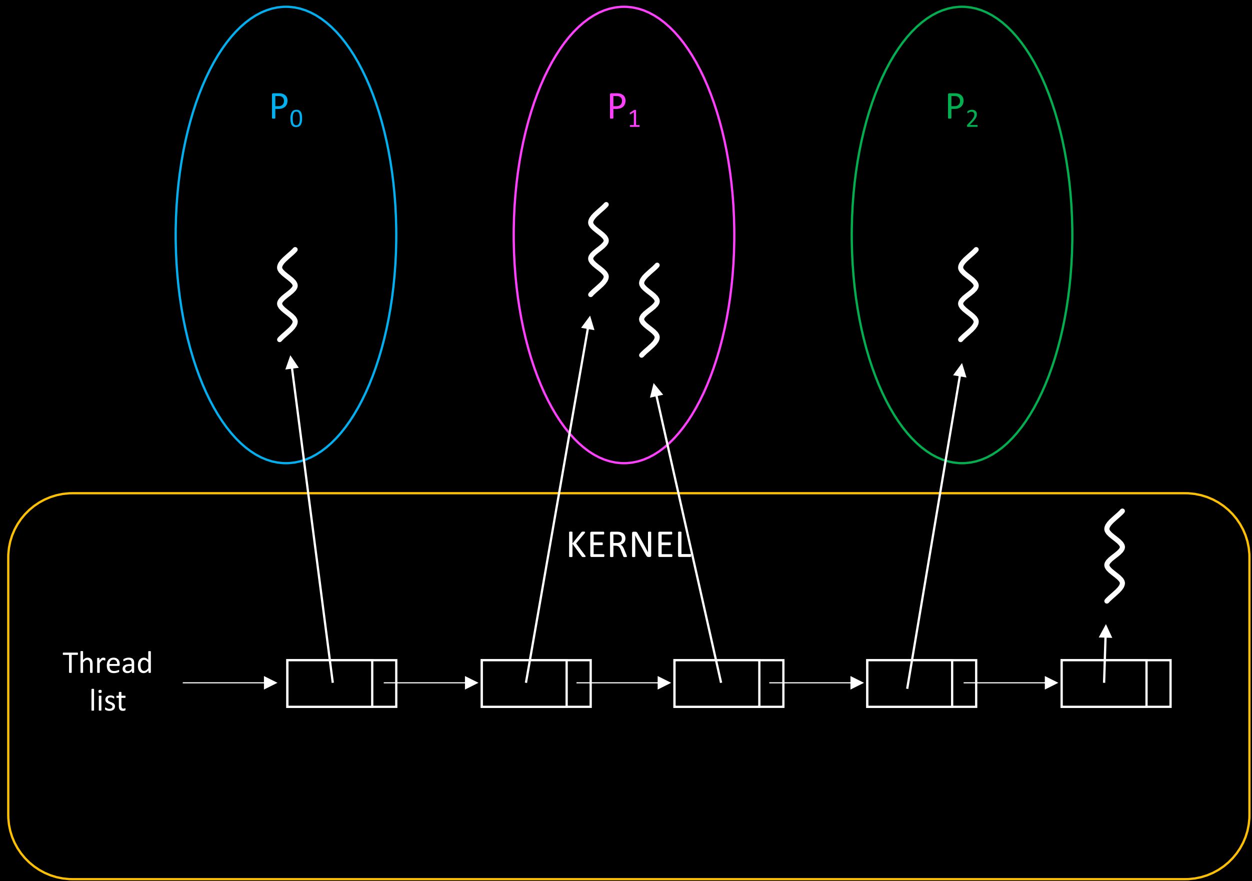
Processes and Threads

- Some (daemons) threads only run inside the kernel

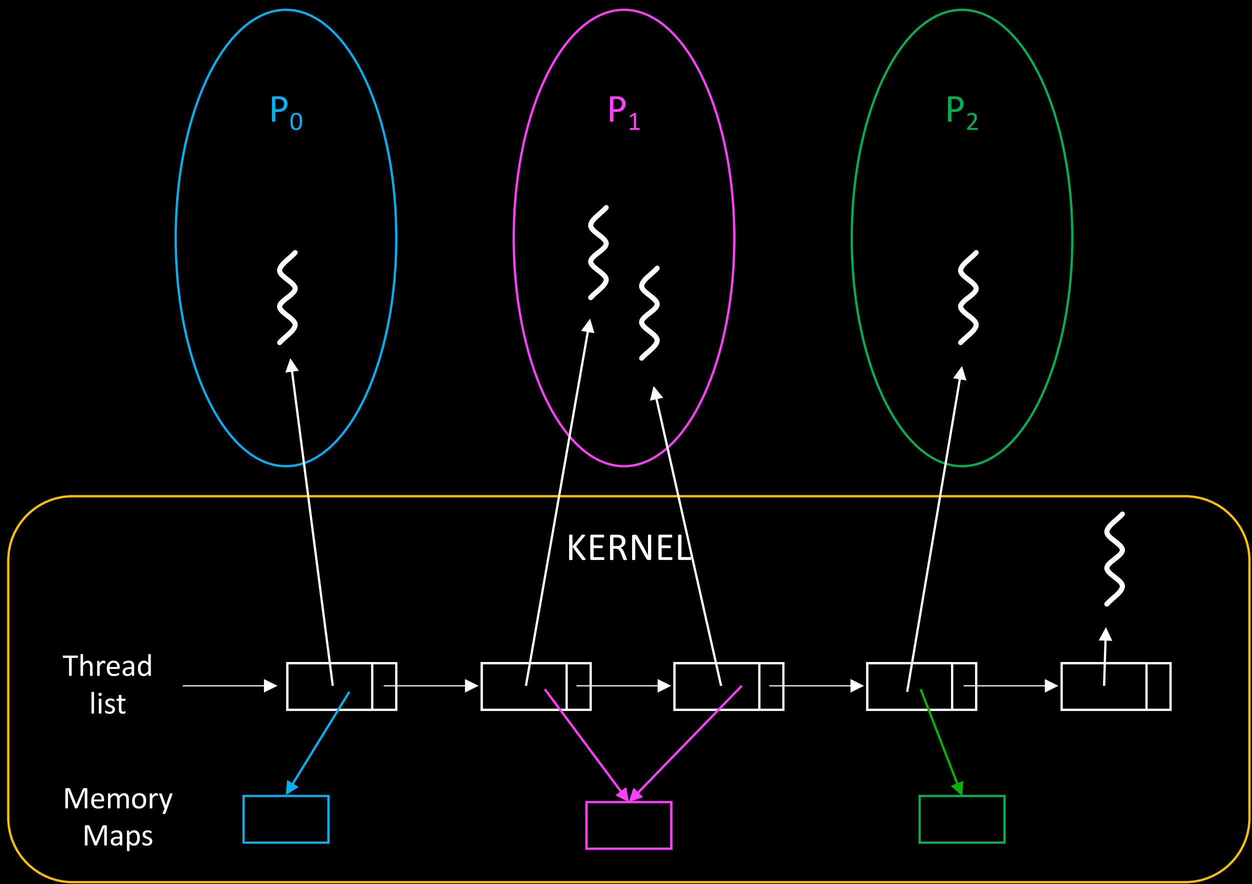
- Modern kernels manage only threads



Processes and Threads: the Big Picture



Processes and Threads: the Big Picture



Race conditions

- Threads can access the same data simultaneously
  - May lead to undefined behavior, data corruption, …
  - Think about
    - Linked lists, graphs, hash tables
    - Structures where several fields must be updated consistently
    - Or just integers…
- When executing kernel code, processes share data as well
  - So the kernel must enforce synchronization

Race conditions

```
volatile int n = 0;
```

```
for (int i = 0; i < 100; i++)
  n++;
```

```
for (int i = 0; i < 100; i++)
  n++;
```

pthread_join

```
printf ("n = %d\n", n);
```

$n = 200$?

Race conditions

```
volatile int n = 0;
```

```
for (int i = 0; i < 100; i++)
  n++;
```

```
for (int i = 0; i < 100; i++)
  n++;
```

pthread_join

```
printf ("n = %d\n", n);
```

$n \in [100,200]$?

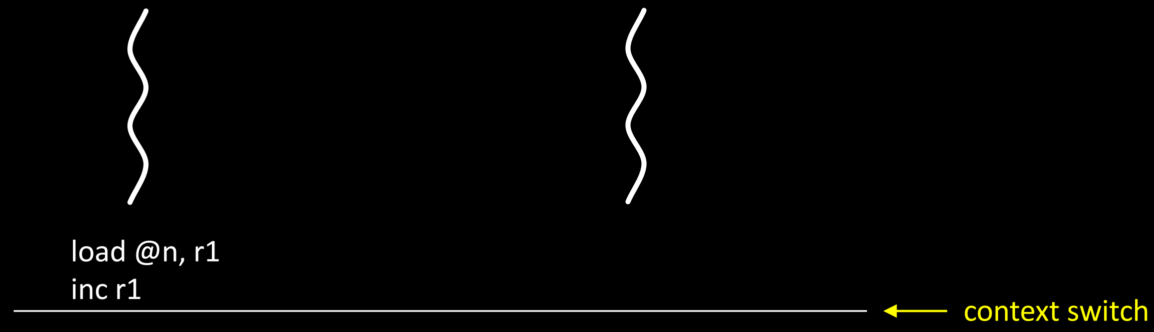
Possible scenario

	load @n, r1	; load from memory
n++ ⇔	inc r1	; increment register
	store r1, @n	; store in memory

n : 0

Possible scenario

n++ ⇔	load @n, r1	; load from memory
	inc r1	; increment register
	store r1, @n	; store in memory



n : 0

Possible scenario

n++ ⇔

```
load @n, r1      ; load from memory
inc r1           ; increment register
store r1, @n     ; store in memory
```

n : ~~0~~ 99

```
load @n, r1
inc r1
```

```
load @n, r1
inc r1
store r1, @n
...
```

99x

Possible scenario

n++ ⇔

```
load @n, r1      ; load from memory
inc r1           ; increment register
store r1, @n     ; store in memory
```

load @n, r1
inc r1

---

load @n, r1
inc r1
store r1, @n
...

} 99x

---

store r1, @n

---

n : ~~0 99~~ 1

Possible scenario

n++ ⇔

```
load @n, r1      ; load from memory
inc r1           ; increment register
store r1, @n     ; store in memory
```

n : ~~0 99~~ 1

load @n, r1 inc r1		
	load @n, r1 inc r1 store r1, @n …	99x
store r1, @n		
	load @n, r1 inc r1	

Possible scenario

```
        load @n, r1      ; load from memory
n++ ⇔  inc r1           ; increment register
        store r1, @n     ; store in memory
```

n : ~~0~~ ~~99~~ ~~1~~ 100

	load @n, r1 inc r1		
		load @n, r1 inc r1 store r1, @n …	99x
	store r1, @n		
		load @n, r1 inc r1	
99x	load @n, r1 inc r1 store r1, @n …		

Possible scenario

n++ ⇔	load @n, r1	; load from memory
	inc r1	; increment register
	store r1, @n	; store in memory

n : ~~0 99 1 100~~ 2

	load @n, r1 inc r1	
		load @n, r1 inc r1 store r1, @n … (99x)
	store r1, @n	
		load @n, r1 inc r1
99x	load @n, r1 inc r1 store r1, @n …	
		store r1, @n

Possible scenario

n++ ⇔
load @n, r1 ; load from memory
inc r1 ; increment register
store r1, @n ; store in memory

n : ~~0 99 1 100~~ 2

$n \in [2{,}200]$!

ad

load @n, r1
inc r1
store r1, @n
…
} 99x

store r1, n

load @n, r1
inc r1

99x {
load @n, r1
inc r1
store r1, @n
…

store r1, @n

Race conditions

- Even the simple ++ operator is not an *atomic* operation
  - So we must prevent multiple threads to execute this operation concurrently!

- To do so, we need synchronization tools
  - This is the topic of the fascinating next chapter! ☺

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