Interrupts

Concurrency

Three Classic forms of Concurrency

Multiprocessing

Using two or more CPUs in order to execute processes simultaneously.

Time-Sharing

Sharing a given CPU resource amongst multiple processes via a protocol for sharing execution time. Can be ***preemptive*** or ***cooperative***.

Interrupts

Hardware interrupts the CPU under some circumstances, providing an opportunity to switch processes.

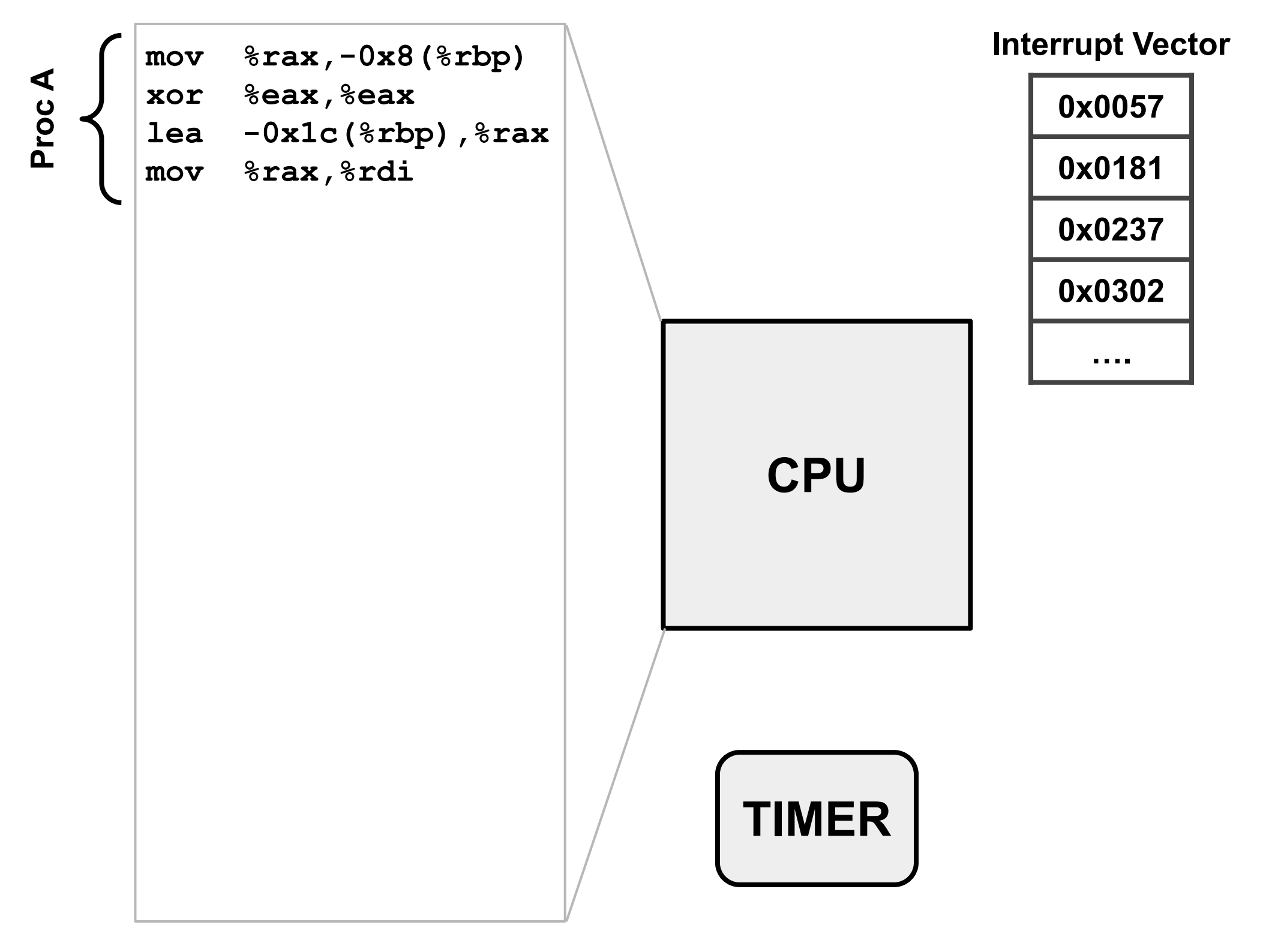
Interrupts

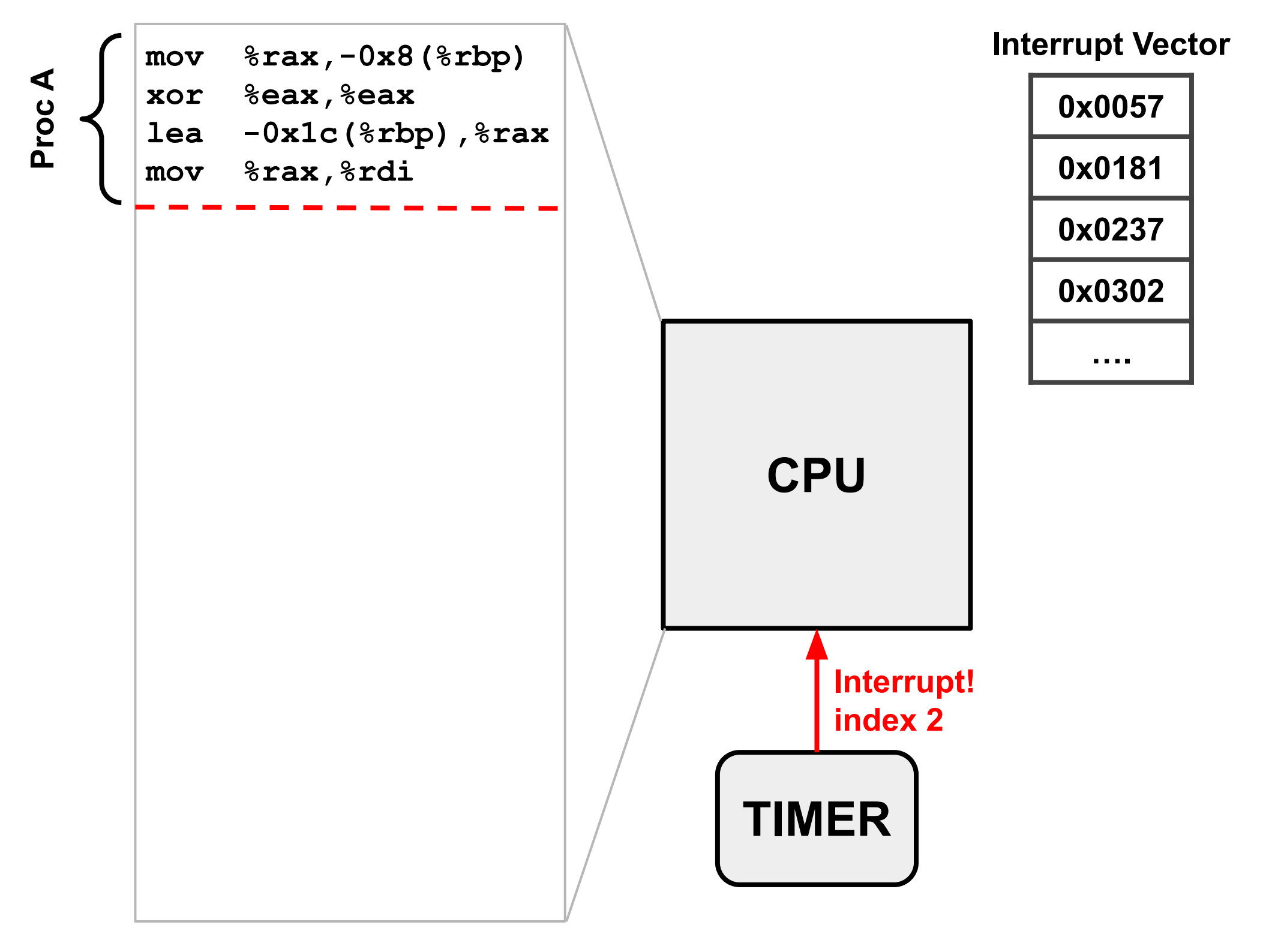
- An **interrupt** is a signal asking for CPU attention
  - CPU forces a jump to the **interrupt handler**
  - Like a function call
  - Can happen from hardware or software
- Original process doesn't run again until handler returns (maybe)

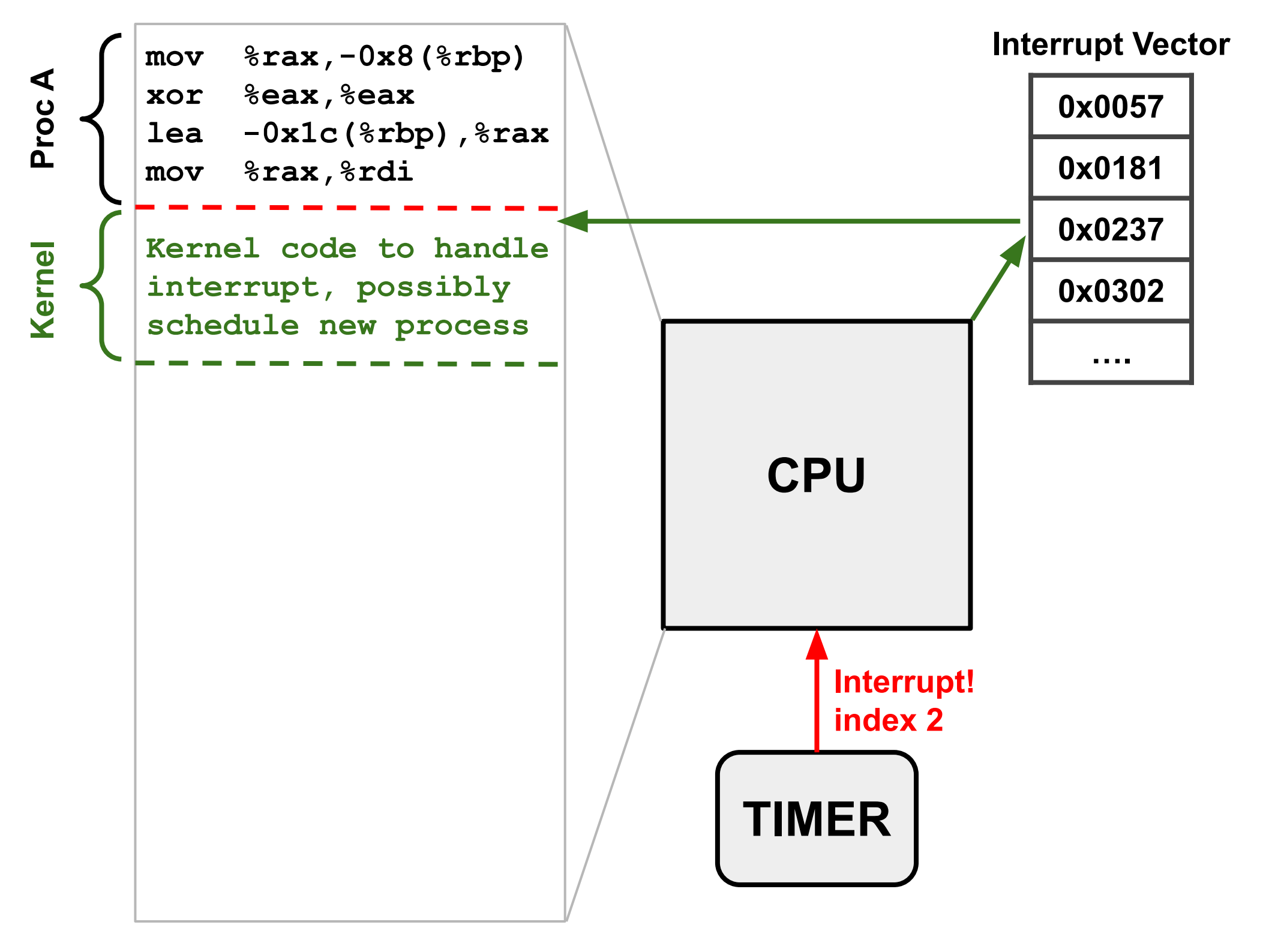
Interrupts

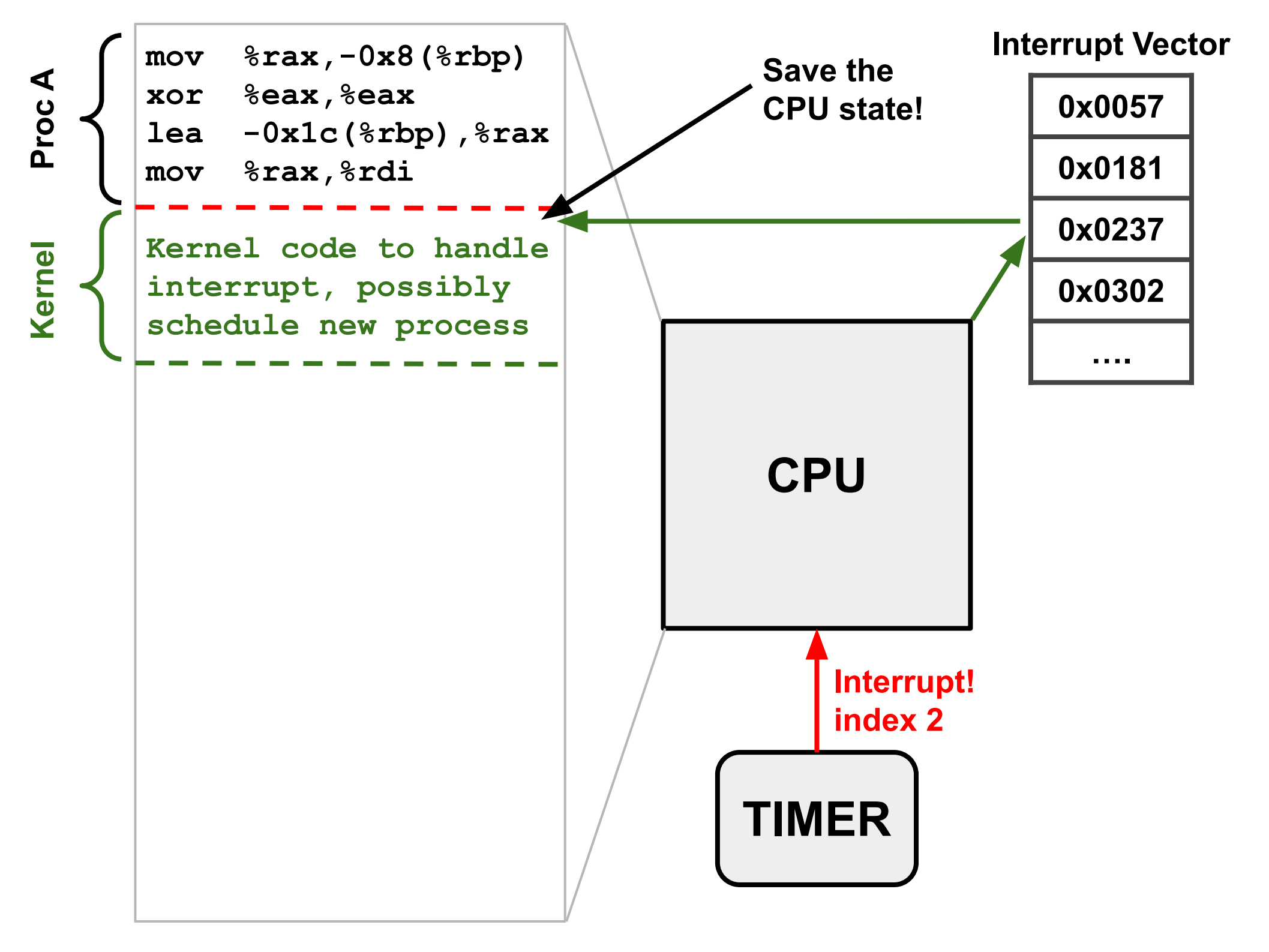
- If you don't **disable interrupts,** any program can be interrupted (user or kernel)
- An interrupt must be able to run on top of **any** other process
- Thus, interrupts provide a mechanism to build pseudo-concurrency on top of

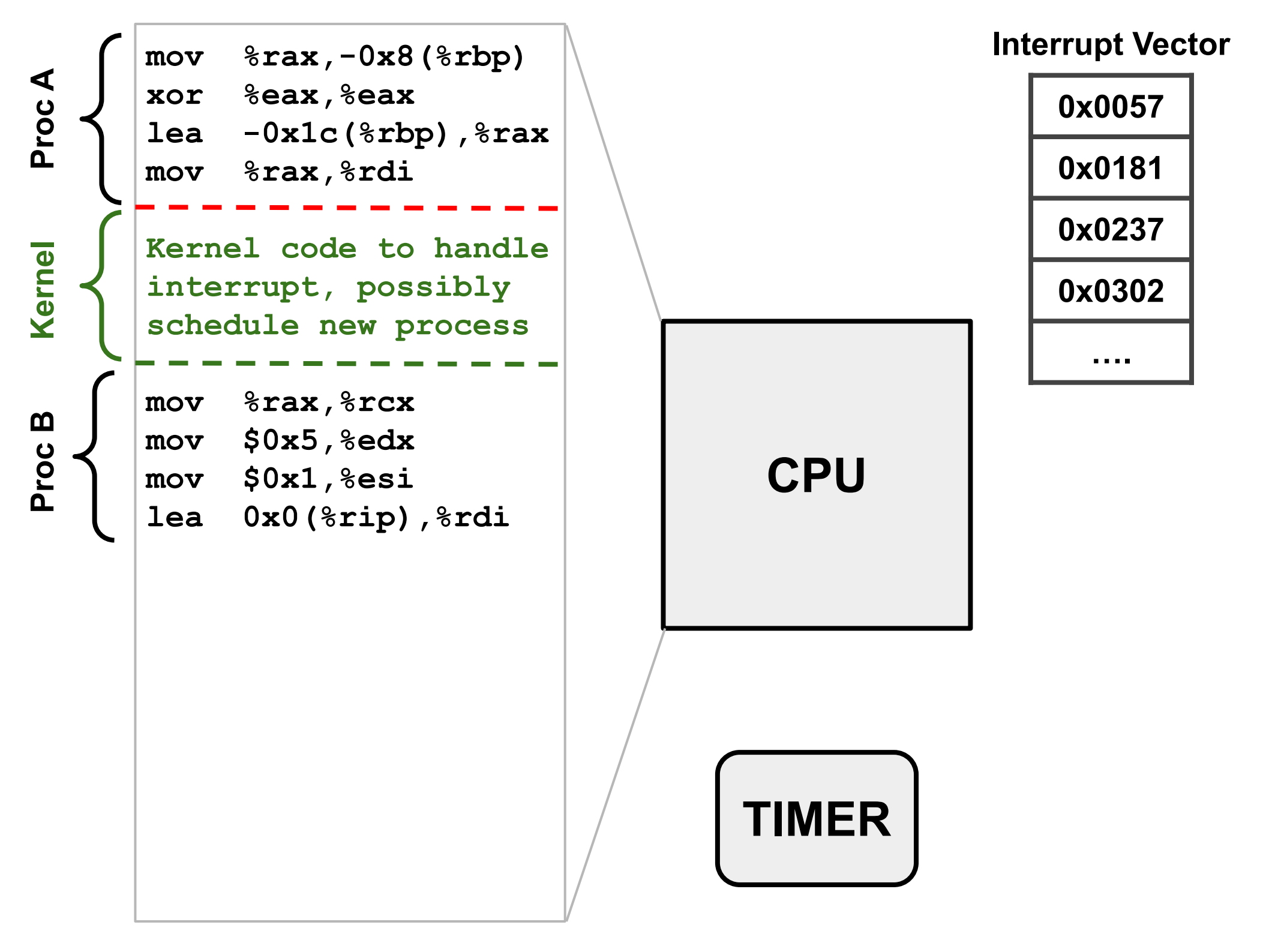
Timer Interrupts Visual Example

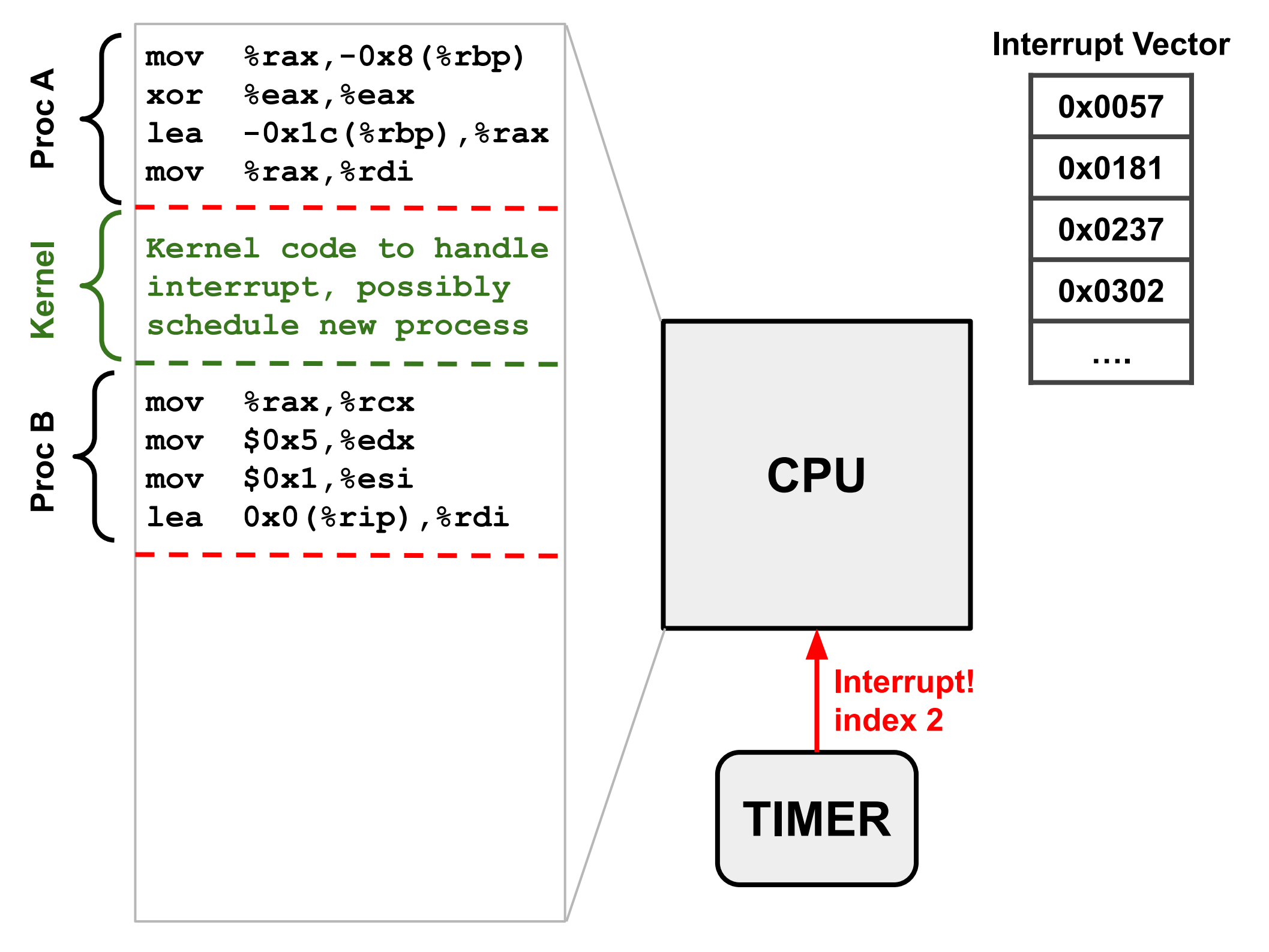

Proc A
mov %rax,-0x8(%rbp)
xor %eax,%eax
lea -0x1c(%rbp),%rax
mov %rax,%rdi
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
TIMER

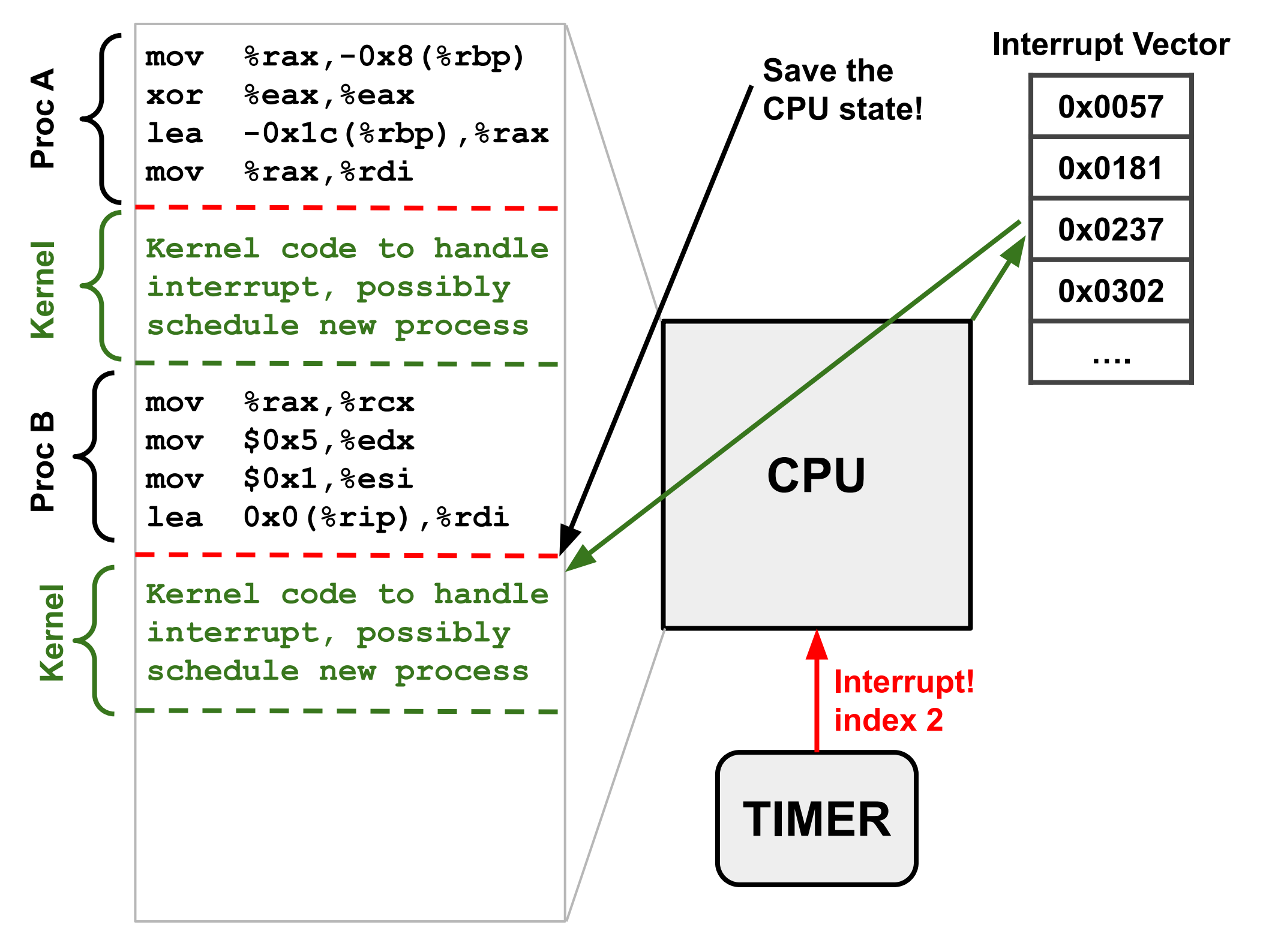

Proc A
mov %rax,-0x8(%rbp)
xor %eax,%eax
lea -0x1c(%rbp),%rax
mov %rax,%rdi
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
Interrupt!
index 2
TIMER

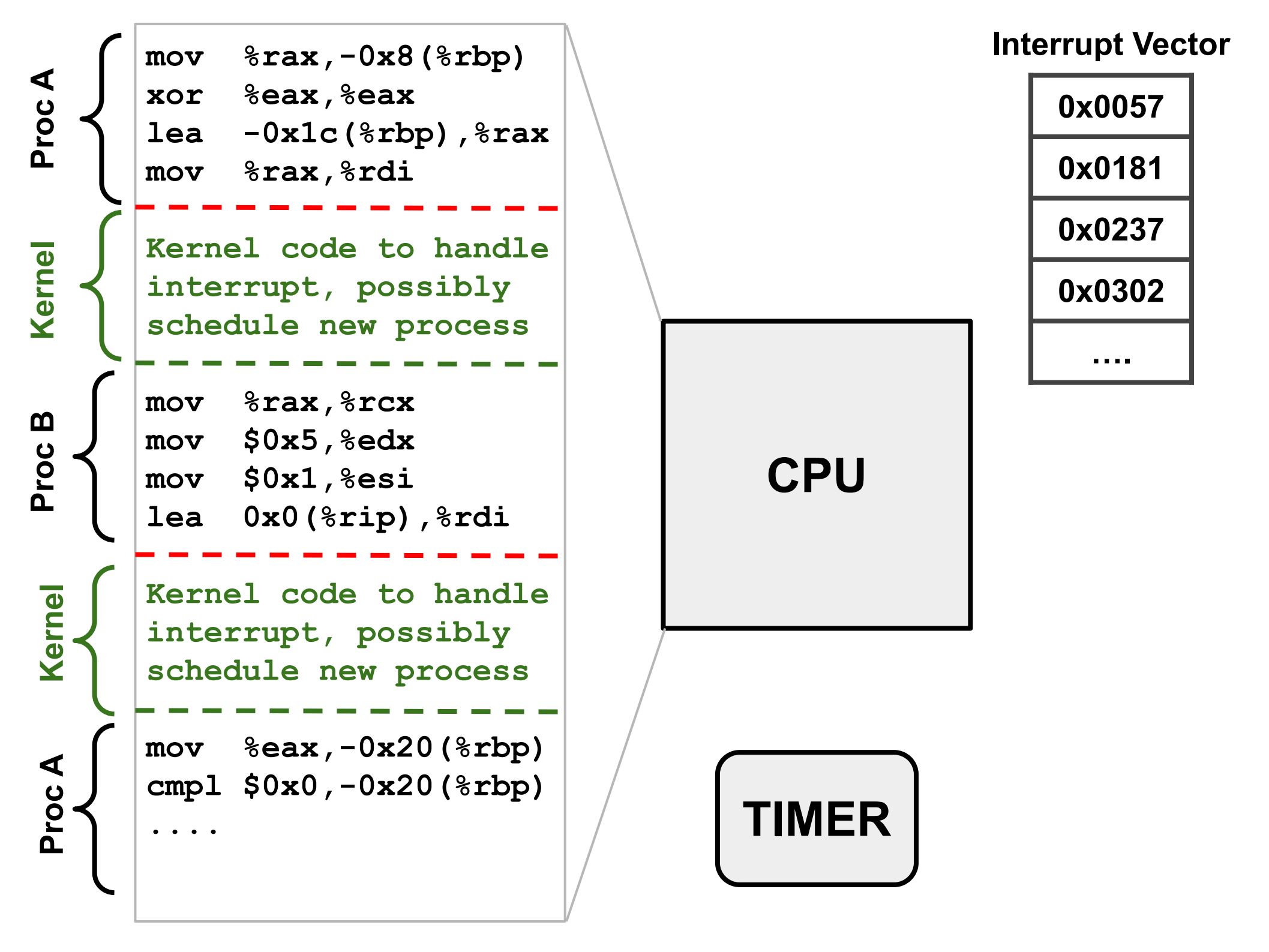

Proc A
mov %rax,-0x8(%rbp)
xor %eax,%eax
lea -0x1c(%rbp),%rax
mov %rax,%rdi
Kernel
Kernel code to handle interrupt, possibly schedule new process
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
Interrupt!
index 2
TIMER


Proc A
mov %rax,-0x8(%rbp)
xor %eax,%eax
lea -0x1c(%rbp),%rax
mov %rax,%rdi
Kernel
Kernel code to handle interrupt, possibly schedule new process
Save the CPU state!
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
Interrupt! index 2
TIMER


Proc A
mov %rax,-0x8(%rbp)
xor %eax,%eax
lea -0x1c(%rbp),%rax
mov %rax,%rdi
Kernel
Kernel code to handle interrupt, possibly schedule new process
Proc B
mov %rax,%rcx
mov $0x5,%edx
mov $0x1,%esi
lea 0x0(%rip),%rdi
CPU
TIMER
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....


Proc A
mov %rax,-0x8(%rbp)
xor %eax,%eax
lea -0x1c(%rbp),%rax
mov %rax,%rdi
Kernel
Kernel code to handle interrupt, possibly schedule new process
Proc B
mov %rax,%rcx
mov $0x5,%edx
mov $0x1,%esi
lea 0x0(%rip),%rdi
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
Interrupt!
index 2
TIMER


Proc A
mov %rax,-0x8(%rbp)
xor %eax,%eax
lea -0x1c(%rbp),%rax
mov %rax,%rdi
Kernel
Kernel code to handle interrupt, possibly schedule new process
Proc B
mov %rax,%rcx
mov $0x5,%edx
mov $0x1,%esi
lea 0x0(%rip),%rdi
Kernel
Kernel code to handle interrupt, possibly schedule new process
Save the CPU state!
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
Interrupt! index 2
TIMER

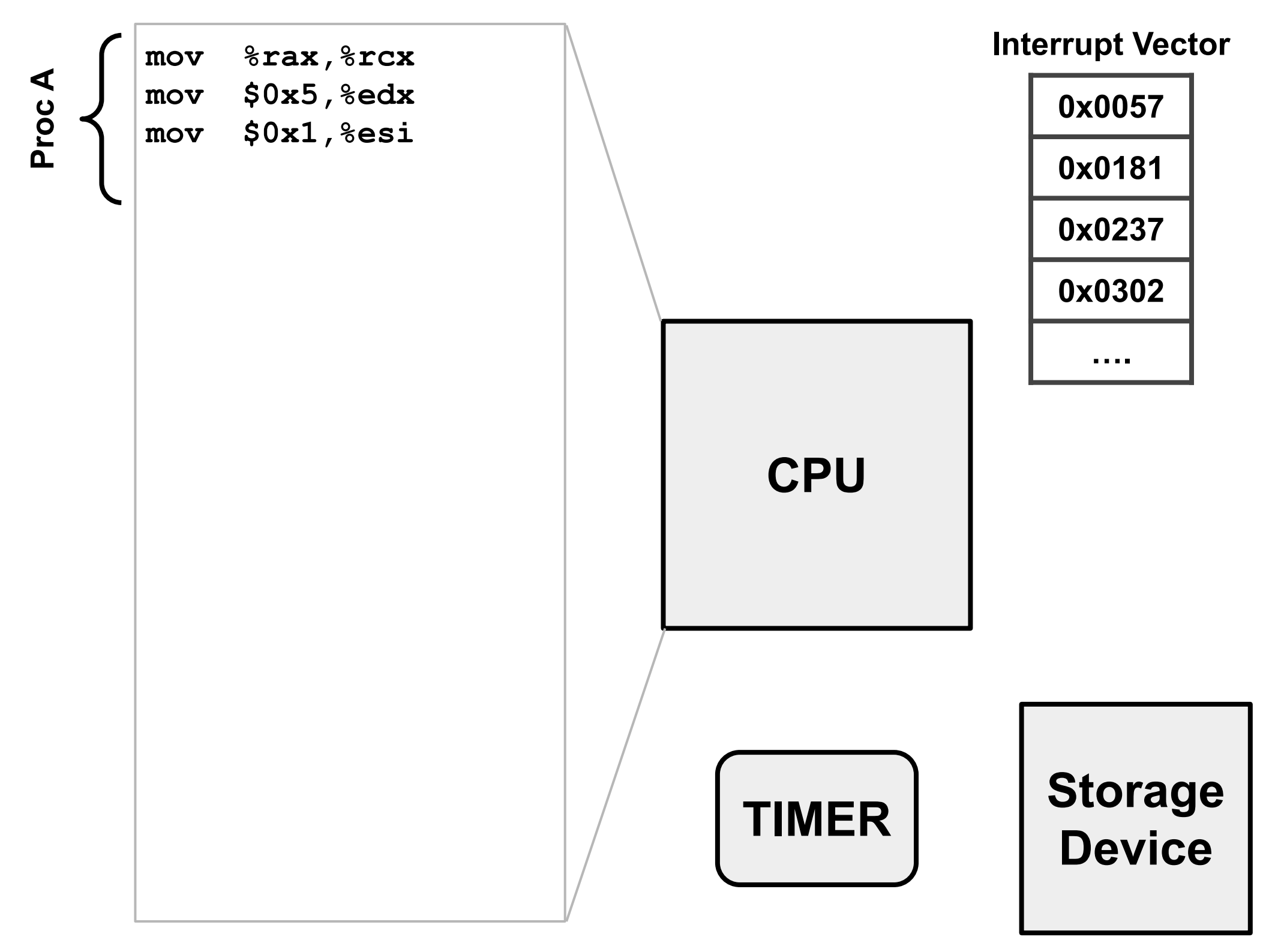

Proc A
mov %rax,-0x8(%rbp)
xor %eax,%eax
lea -0x1c(%rbp),%rax
mov %rax,%rdi
Kernel
Kernel code to handle interrupt, possibly schedule new process
Proc B
mov %rax,%rcx
mov $0x5,%edx
mov $0x1,%esi
lea 0x0(%rip),%rdi
Kernel
Kernel code to handle interrupt, possibly schedule new process
Proc A
mov %eax,-0x20(%rbp)
cmpl $0x0,-0x20(%rbp)
....
CPU
TIMER
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....

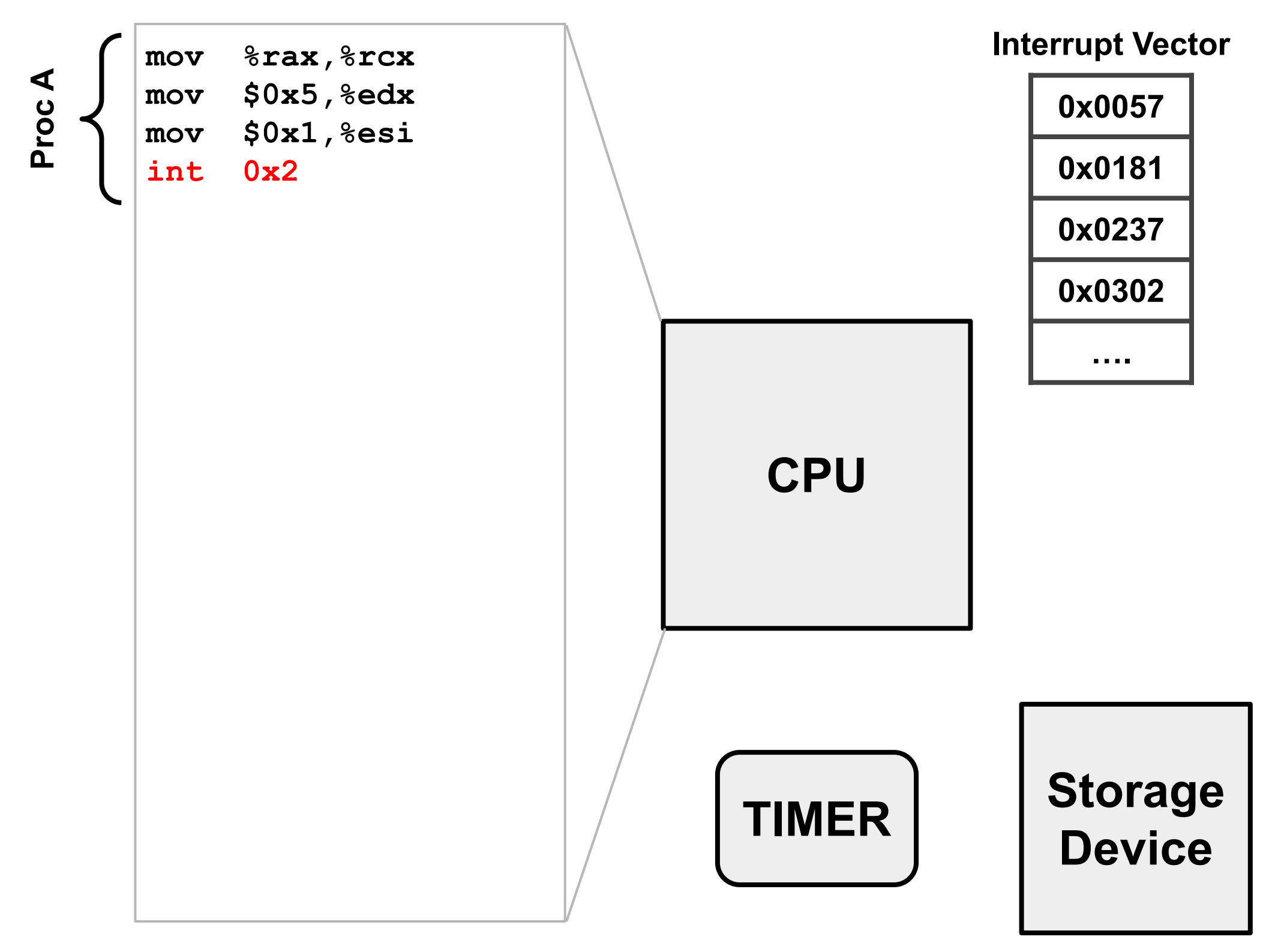
Activity: Interrupts

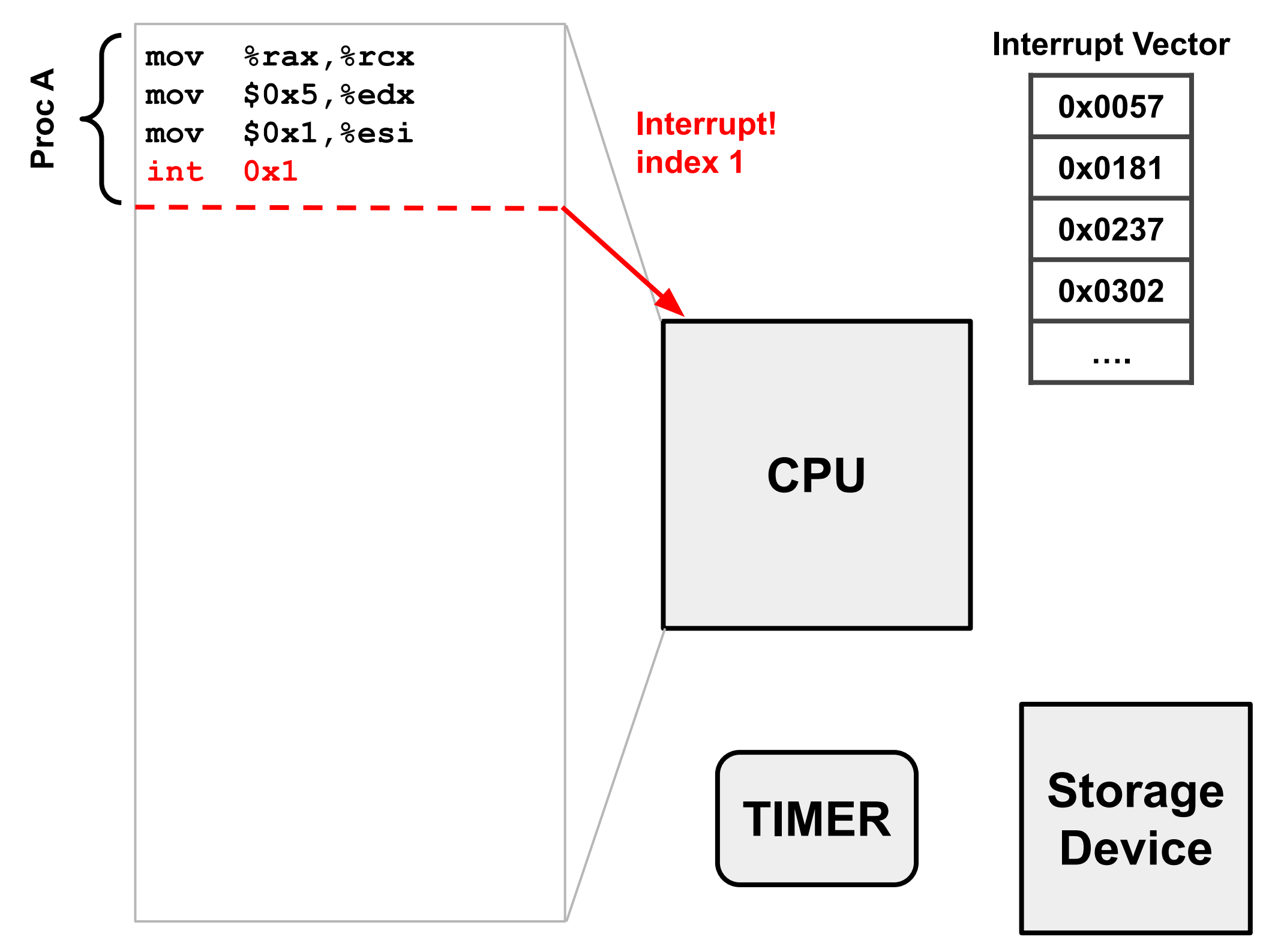
Consider this Scenario:

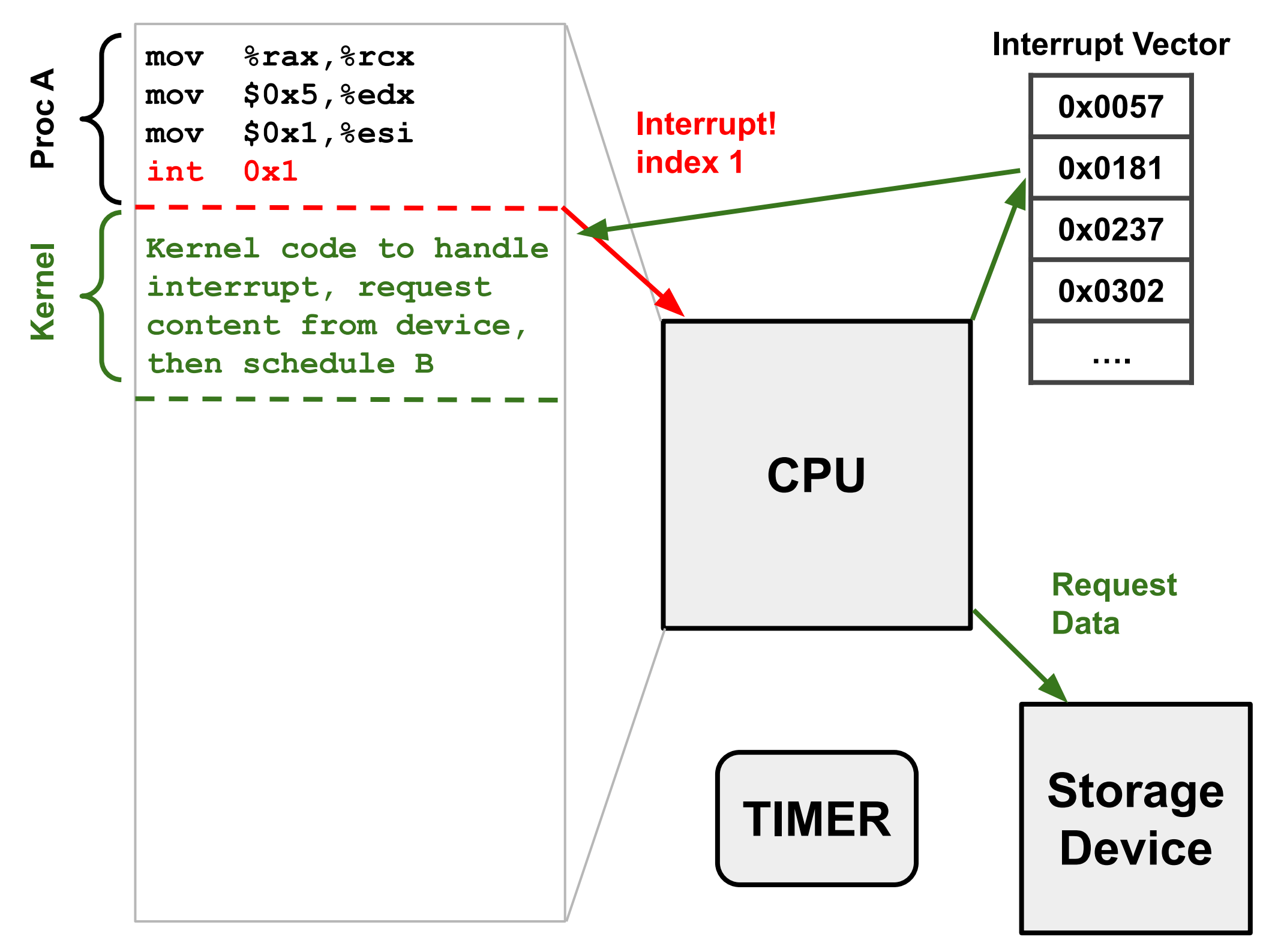
A CPU runs a 3.0Ghz. This is a simple CPU that executes exactly 1 instruction per clock cycle. The timer interrupt fires off 1000 times per second. How many instructions can be executed in each time slice?

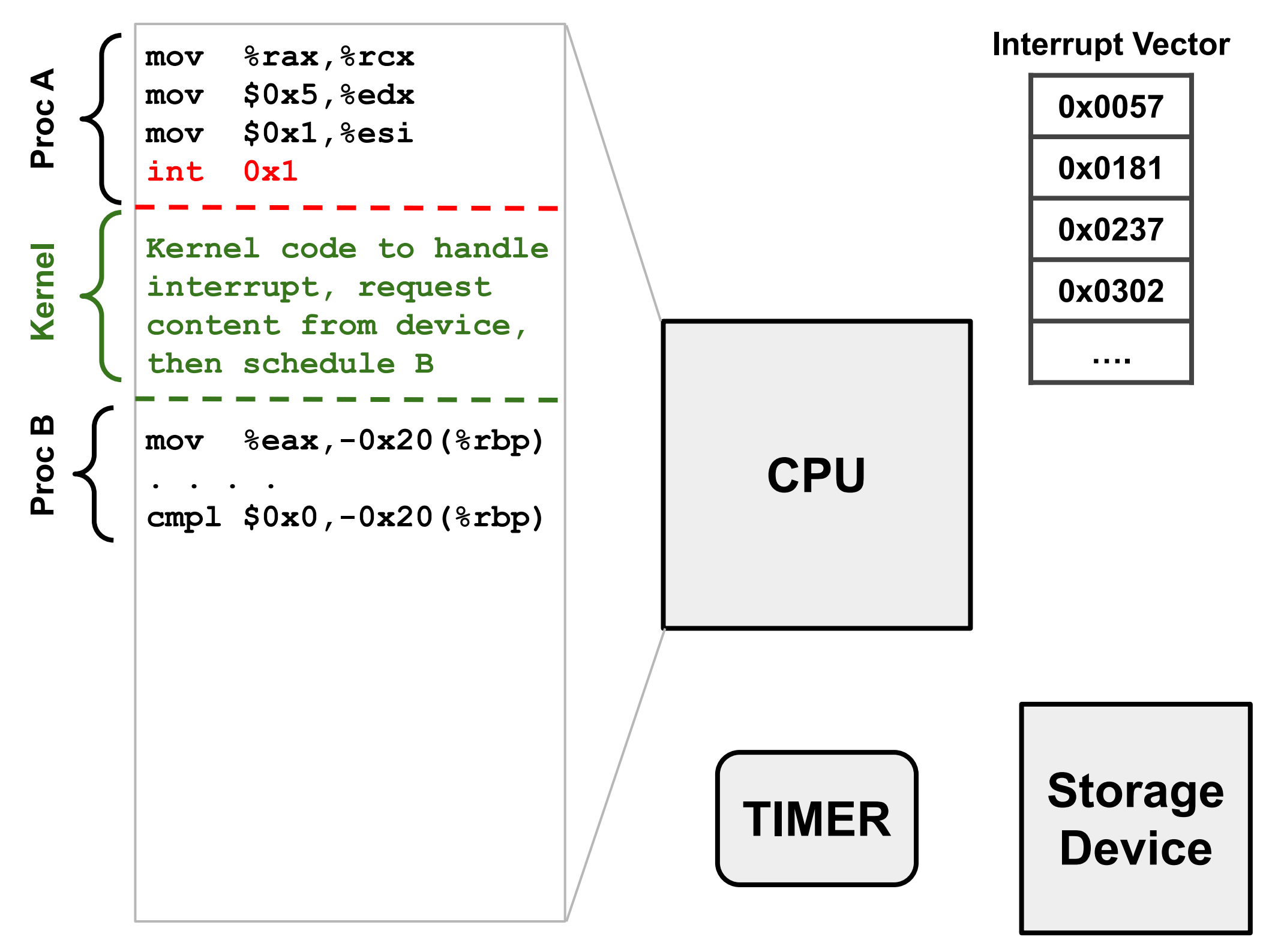
Software and Device Interrupts Visual Example

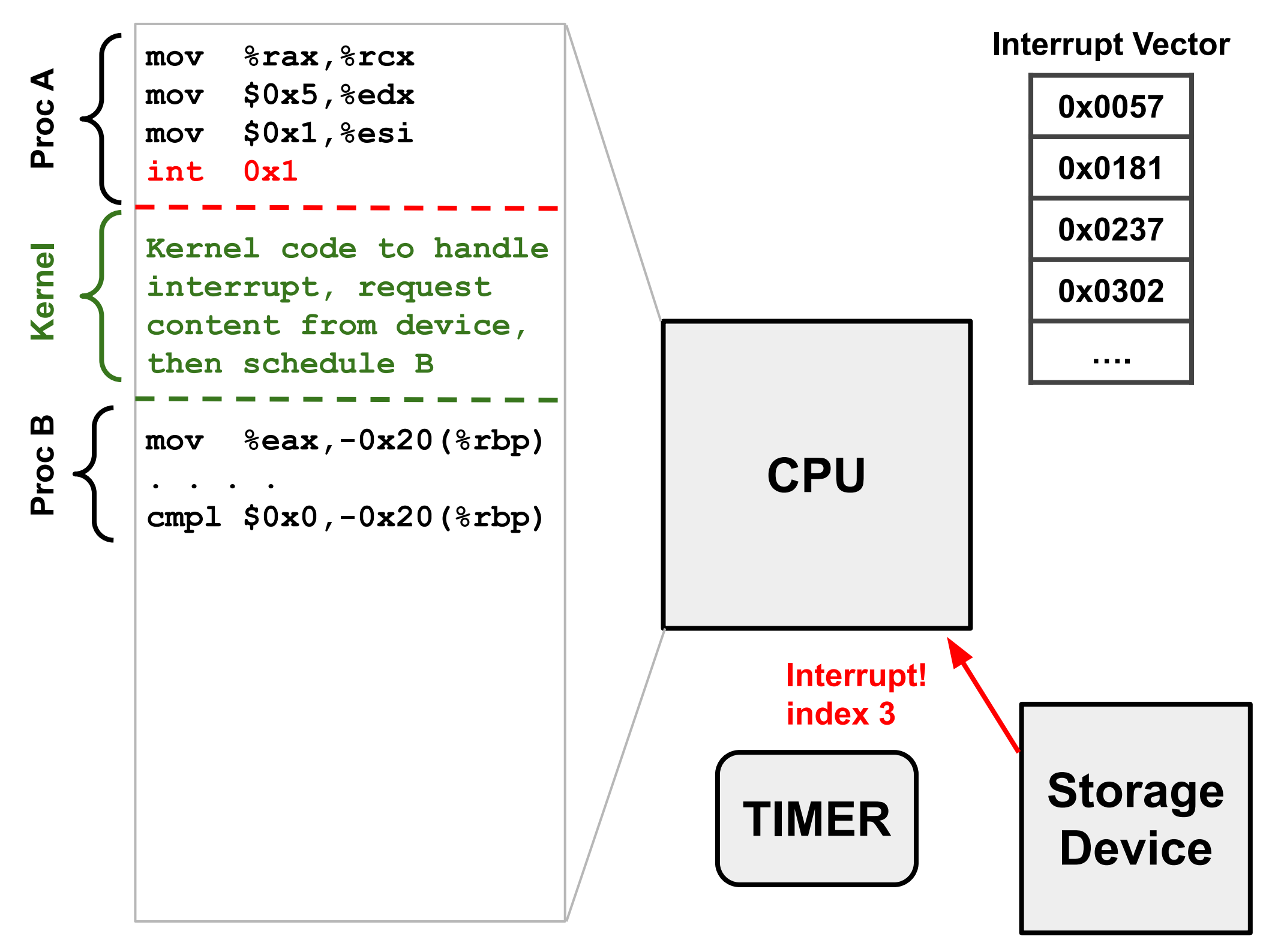

Proc A
mov %rax,%rcx
mov $0x5,%edx
mov $0x1,%esi
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
TIMER
Storage Device

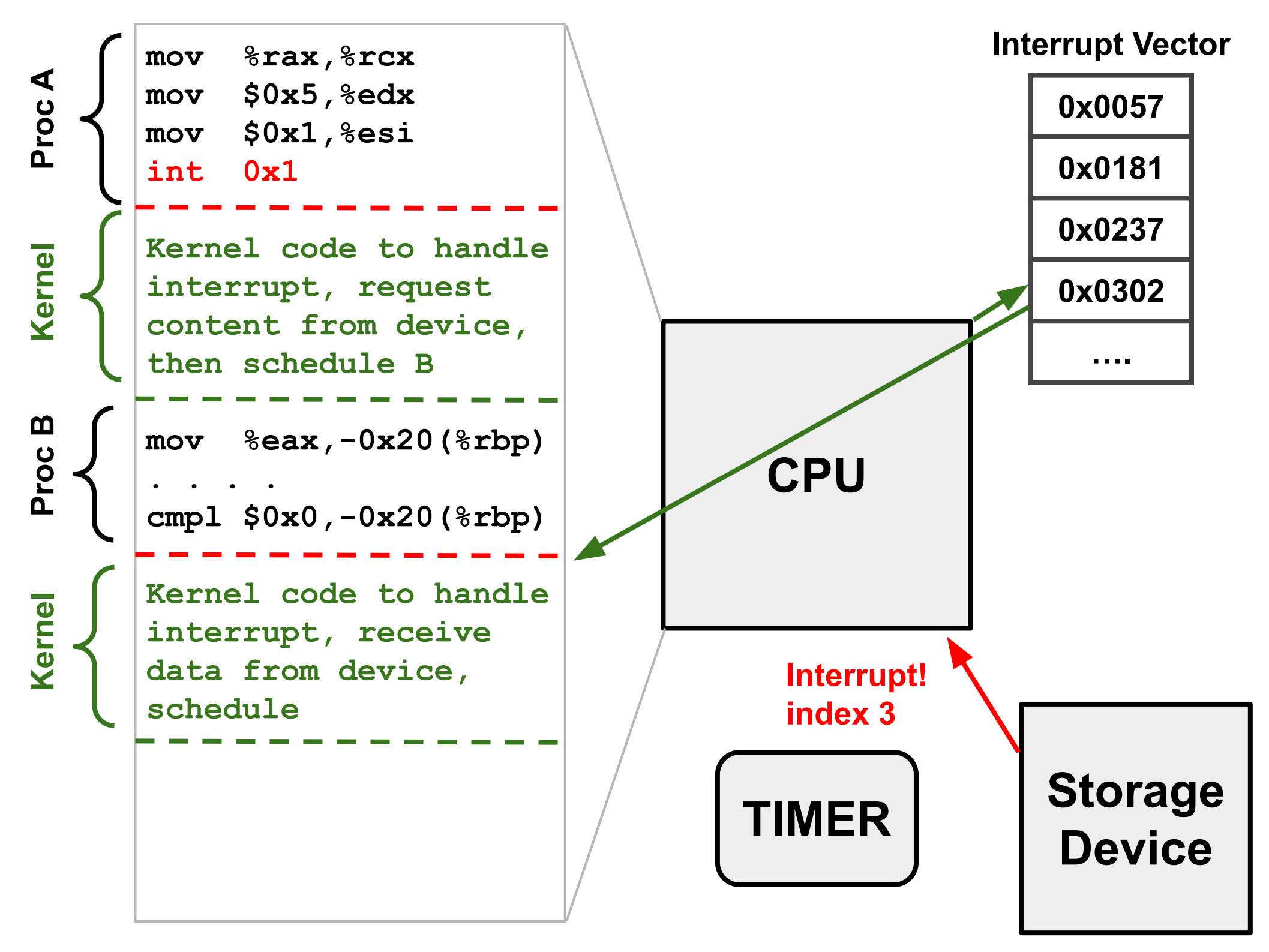

Proc A
mov %rax,%rcx
mov $0x5,%edx
mov $0x1,%esi
int 0x2
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
TIMER
Storage Device

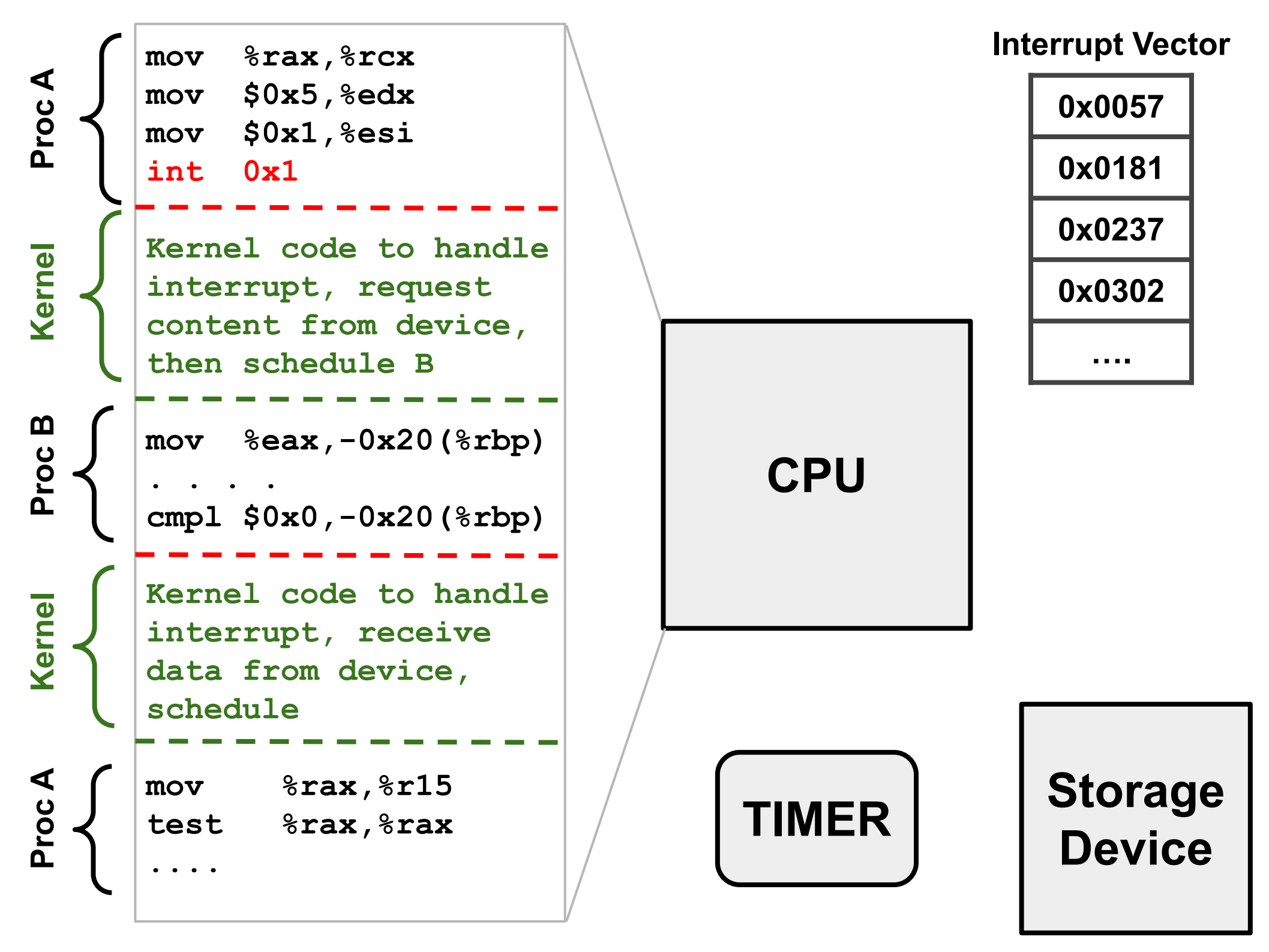

Proc A
mov %rax,%rcx
mov $0x5,%edx
mov $0x1,%esi
int 0x1
Interrupt!
index 1
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
TIMER
Storage Device


Proc A
mov %rax,%rcx
mov $0x5,%edx
mov $0x1,%esi
int 0x1
Kernel
Kernel code to handle interrupt, request content from device, then schedule B
Interrupt!
index 1
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
Request Data
TIMER
Storage Device


Proc A
mov %rax,%rcx
mov $0x5,%edx
mov $0x1,%esi
int 0x1
Kernel
Kernel code to handle interrupt, request content from device, then schedule B
Proc B
mov %eax,-0x20(%rbp)
. . . .
cmpl $0x0,-0x20(%rbp)
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
TIMER
Storage Device


Proc A
mov %rax,%rcx
mov $0x5,%edx
mov $0x1,%esi
int 0x1
Kernel
Kernel code to handle interrupt, request content from device, then schedule B
Proc B
mov %eax,-0x20(%rbp)
. . . .
cmpl $0x0,-0x20(%rbp)
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
Interrupt!
index 3
TIMER
Storage Device

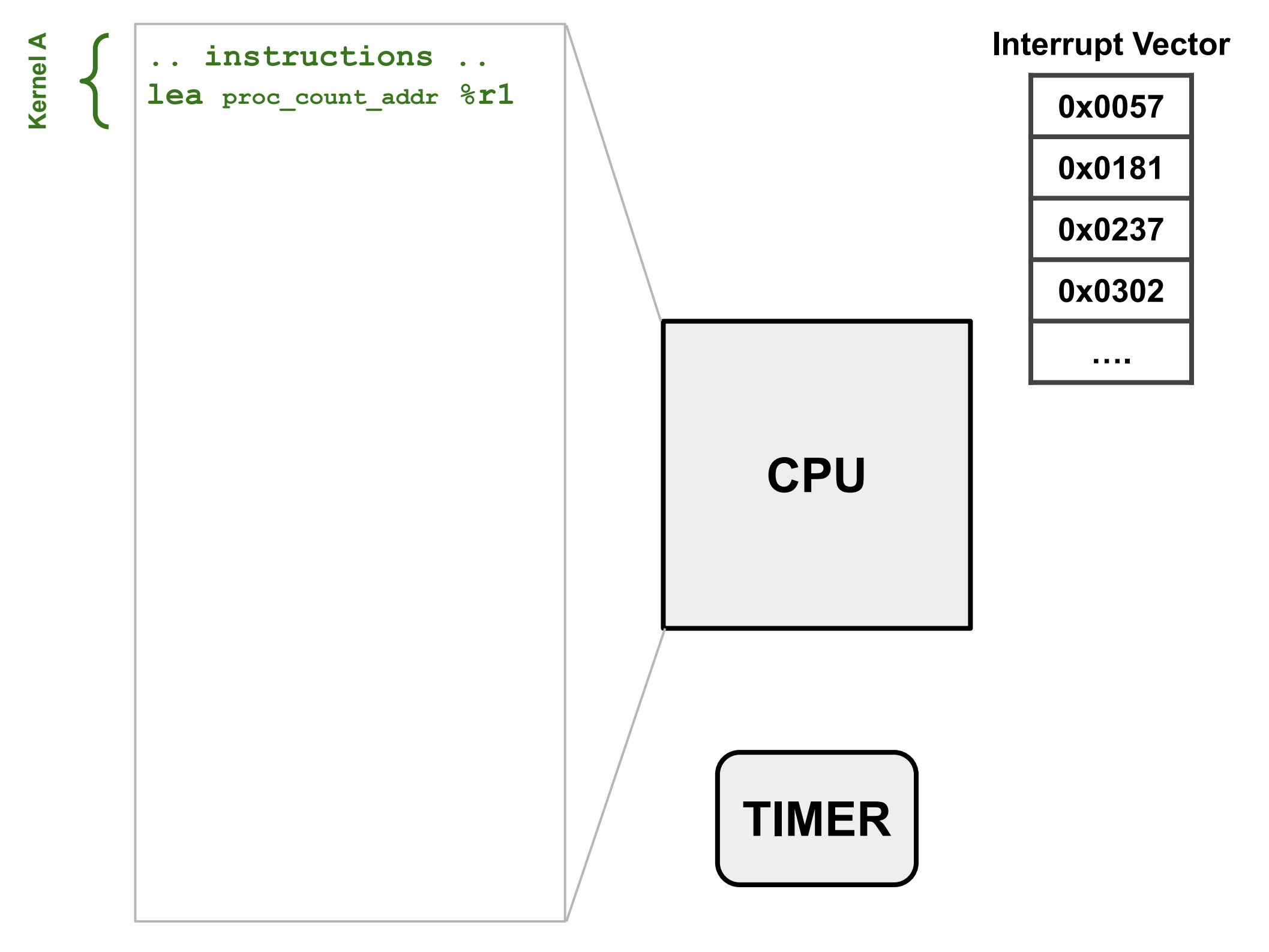

Proc A
mov %rax,%rcx
mov $0x5,%edx
mov $0x1,%esi
int 0x1
Kernel
Kernel code to handle interrupt, request content from device, then schedule B
Proc B
mov %eax,-0x20(%rbp)
. . . .
cmpl $0x0,-0x20(%rbp)
Kernel
Kernel code to handle interrupt, receive data from device, schedule
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
Interrupt!
index 3
TIMER
Storage Device

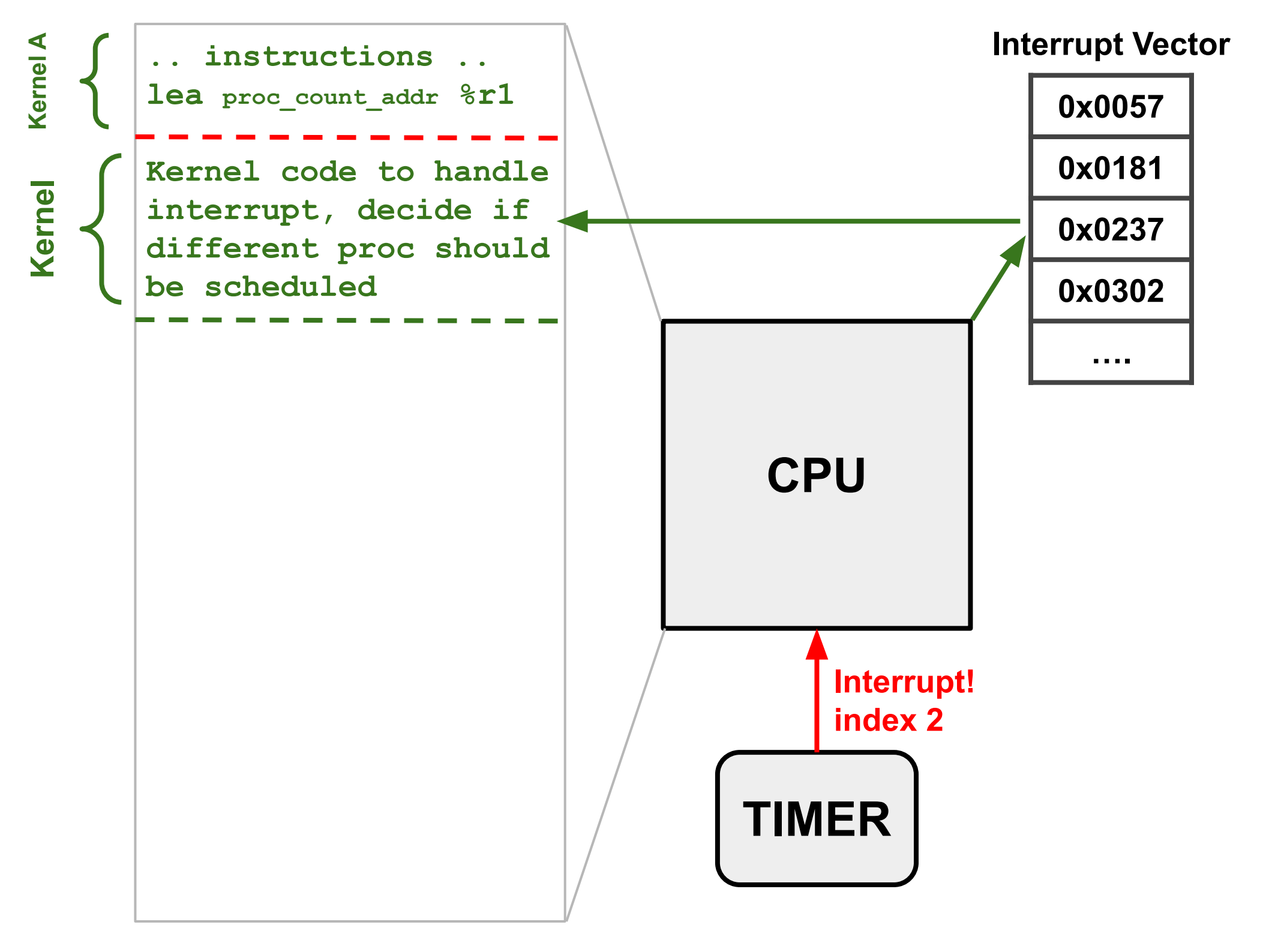

Proc A
mov %rax,%rcx
mov $0x5,%edx
mov $0x1,%esi
int 0x1
Kernel
Kernel code to handle interrupt, request content from device, then schedule B
Proc B
mov %eax,-0x20(%rbp)
. . . .
cmpl $0x0,-0x20(%rbp)
Kernel
Kernel code to handle interrupt, receive data from device, schedule
Proc A
mov %rax,%r15
test %rax,%rax
....
CPU
TIMER
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
Storage Device

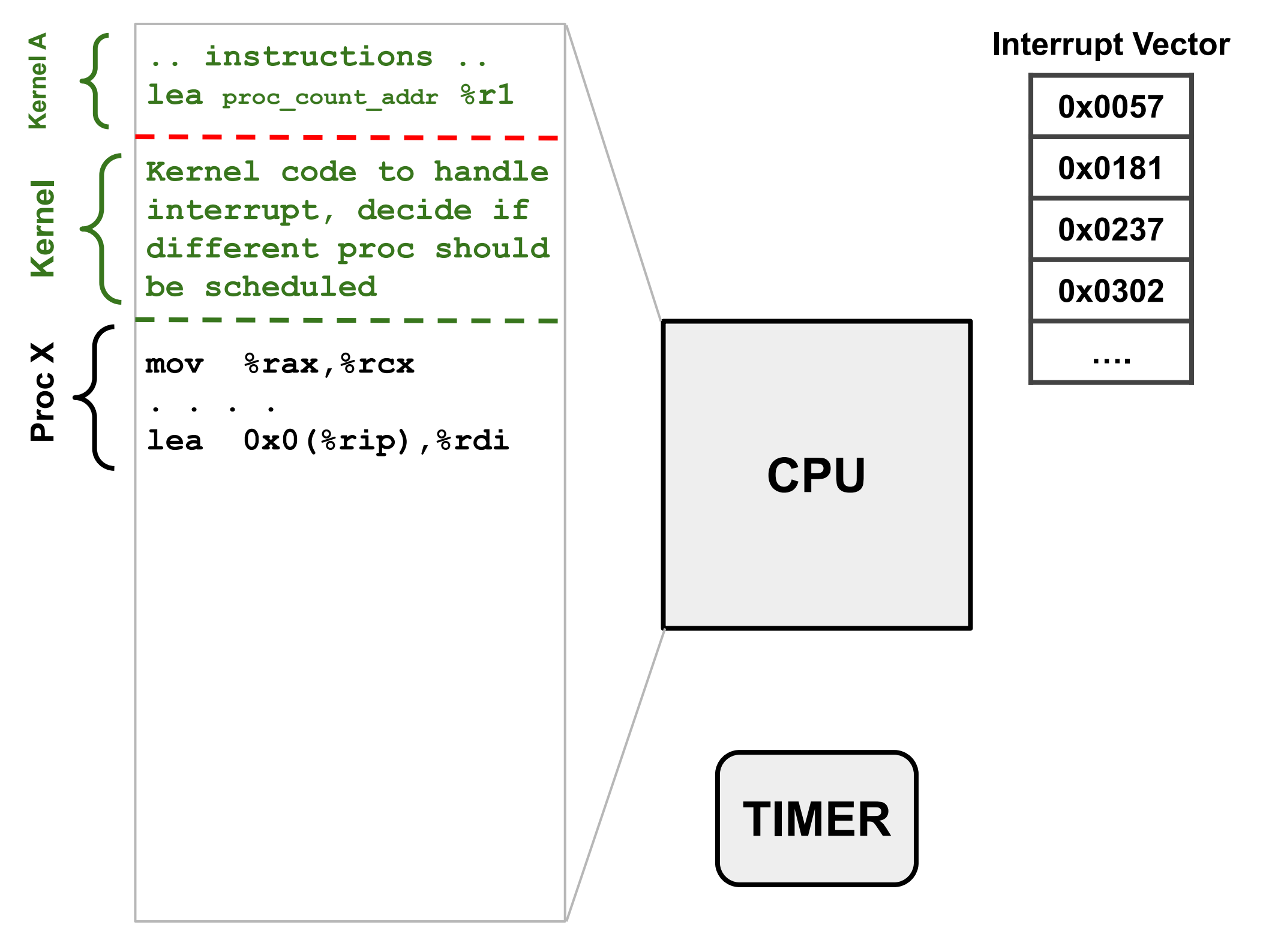
Activity: Interrupts

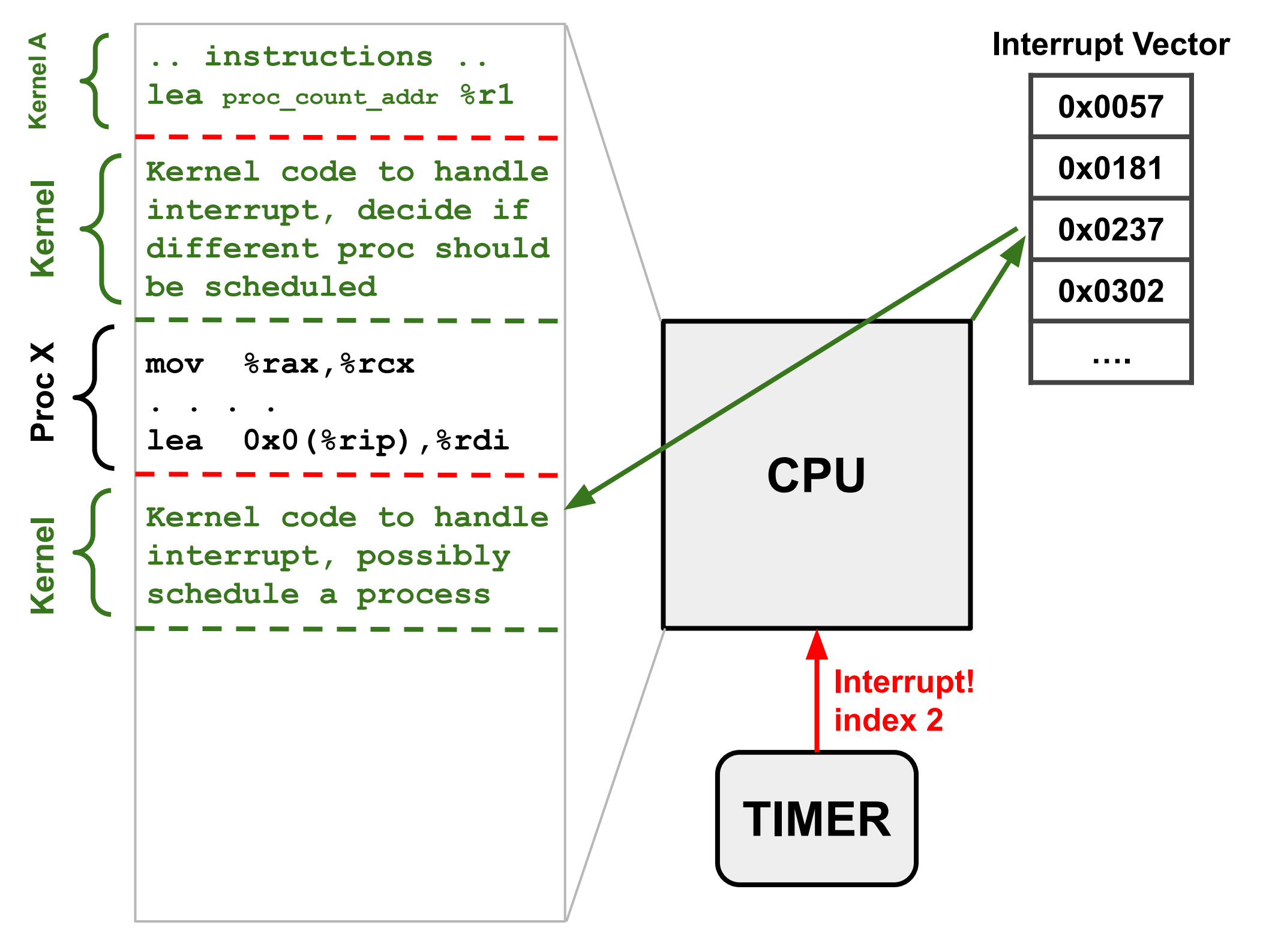
Consider this scenario: Process A was running, then a timer interrupt occurs and the interrupt handling code starts executing

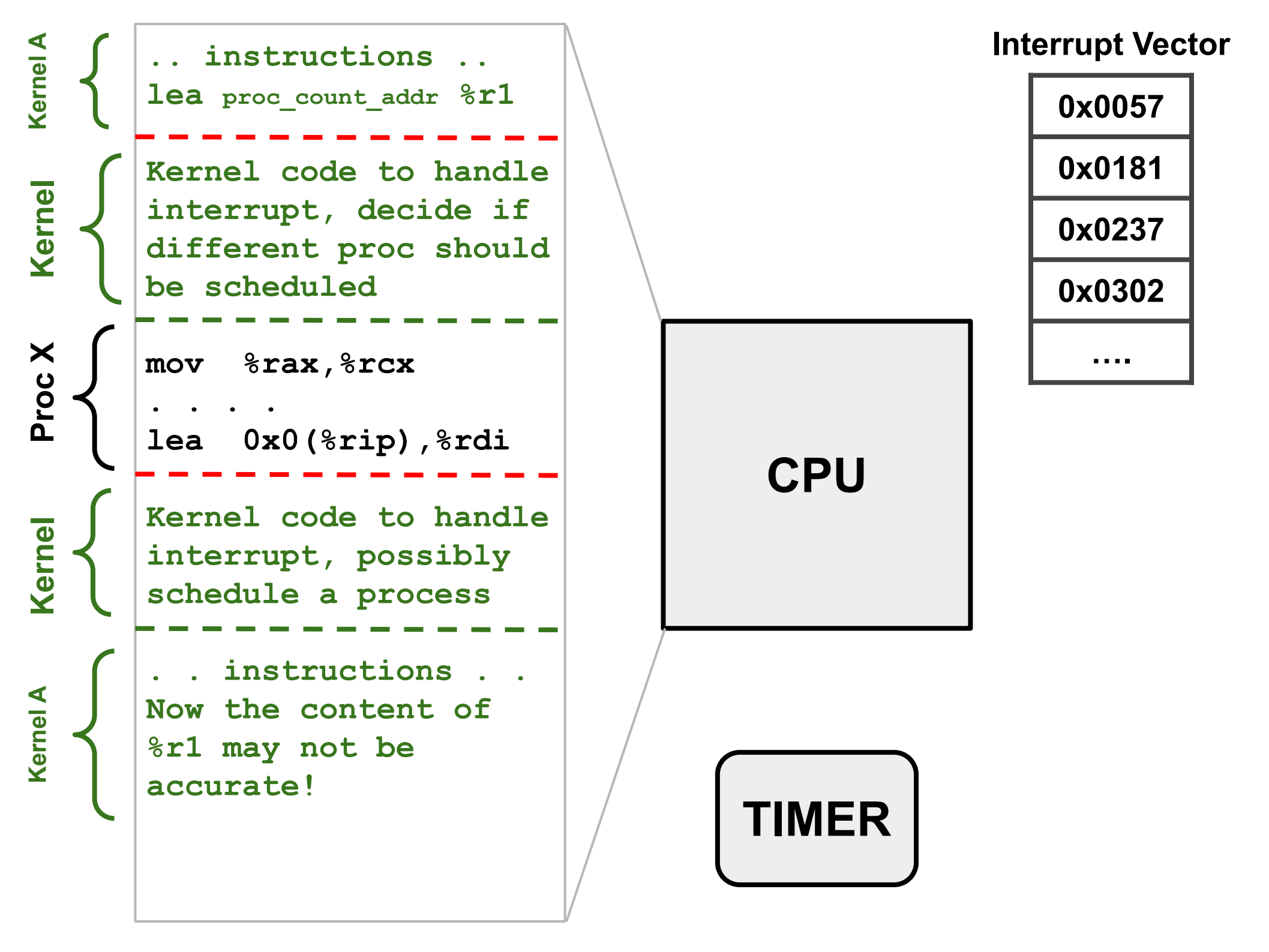
- What if the interrupt handler *modifies* a variable that process A cares about?
- What if the interrupt handler *uses* a variable that process A was changing?

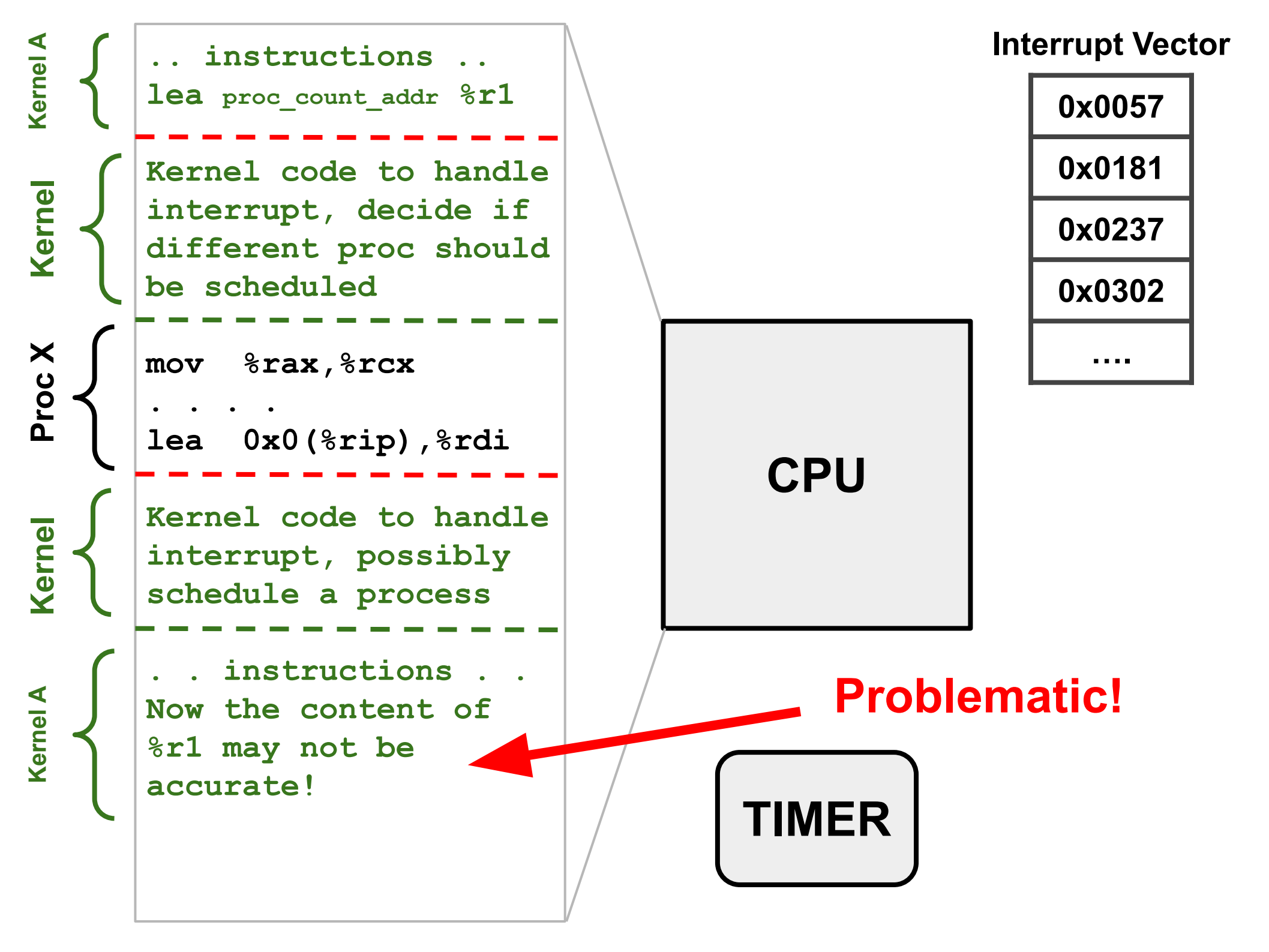

Kernel A
.. instructions ..
lea proc_count_addr %r1
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
TIMER


Kernel A
.. instructions ..
lea proc_count_addr %r1
Kernel
Kernel code to handle interrupt, decide if different proc should be scheduled
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
Interrupt!
index 2
TIMER


Kernel A
.. instructions ..
lea proc_count_addr %r1
Kernel
Kernel code to handle interrupt, decide if different proc should be scheduled
Proc X
mov %rax,%rcx
. . . .
lea 0x0(%rip),%rdi
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
TIMER


Kernel A
.. instructions ..
lea proc_count_addr %r1
Kernel
Kernel code to handle interrupt, decide if different proc should be scheduled
Proc X
mov %rax,%rcx
. . . .
lea 0x0(%rip),%rdi
Kernel
Kernel code to handle interrupt, possibly schedule a process
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
Interrupt!
index 2
TIMER


Kernel A
.. instructions ..
lea proc_count_addr %r1
Kernel
Kernel code to handle interrupt, decide if different proc should be scheduled
Proc X
mov %rax,%rcx
. . . .
lea 0x0(%rip),%rdi
Kernel
Kernel code to handle interrupt, possibly schedule a process
Kernel A
. . instructions . .
Now the content of %r1 may not be accurate!
CPU
TIMER
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....


Kernel A
.. instructions ..
lea proc_count_addr %r1
Kernel
Kernel code to handle interrupt, decide if different proc should be scheduled
Proc X
mov %rax,%rcx
. . . .
lea 0x0(%rip),%rdi
Kernel
Kernel code to handle interrupt, possibly schedule a process
Kernel A
. . instructions . .
Now the content of %r1 may not be accurate!
Interrupt Vector
0x0057
0x0181
0x0237
0x0302
....
CPU
Problematic!
TIMER

Races

- A **race condition** is when the outcome of a calculation depends on “accidental” (that is, unpredictable) details of how quickly it runs
- Remember: it is **impossible** to reliably predict your speed
  - Might be interrupted
  - Might be context-switched
  - Cache, paging, etc.

Races

- Races are **really bad!**

- Often will seem to work, but fail randomly
- Very difficult to replicate
- Very difficult to test your fix

- Conclusion: **prevent them before they happen!!!**

Interrupts

- Kernel code (not user!) can **disable interrupts** at any time
  - No interrupts will fire
  - No interrupt handlers will run
  - External interrupts still happen, CPU remembers them
  - Interrupts fire **immediately** when user re-enables interrupts
  - Some interrupts cannot be disabled

Interrupts

- In Phase 1 you will need to ensure interrupts get disabled whenever a transition to the kernel happens.
- Set the ***Processor Status Register*** (PSR) to the appropriate value.

Activity: Interrupts

- Disabling interrupts works well to solve these problems on a ***single-core*** CPU system
- What about multi-core or multi-cpu systems?

Preventing Concurrency

- In a **single-CPU** OS is simpler! If you disable interrupts:
  - Time-slicing never happens
  - Interrupts can't run
- In a multi-CPU OS, concurrency is **real**
  - Use locks to protect data
  - When self-deadlock is a worry, disable interrupts **before** you gain a lock

Concurrency

Consider this code

```
while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Critical Sections

A **critical section** is a portion of the program where interrupting it might cause a race

```
while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Non-Critical

Critical Section

Locks

More on this in a future lecture

A **lock** is a mechanism that guarantees only one process at a time is allowed to execute a segment of code. Must be **gained** and **released**.

```
while True:                 # not in CS
    gain(my_lock)
    load  x → $r1           # in CS
    inc   $r1               # in CS
    store $r1 → x           # in CS
    release(my_lock)
```

Locks

- A **spinlock** is a lock where `gain()` is implemented as a tight `while` loop
  - In some implementations, CPU can be stuck forever
  - In others, the process eventually gives up and goes to sleep

Locks

- A lock can only have one owner

- To become the owner, you “gain” (or “lock”) it
  - Will block if it is already owned
- To release ownership, you “release” (or “unlock”) it

Locks

- **WARNING**
- Locks are only useful if you use them in all the right places. Locks don't truly protect data; they just block processes from reaching code.

- If you forget to use `gain()/release()` on one CS, it will be a *danger* to all the other CSes (and vice-versa).

Interrupts

- **Self-deadlock** is the condition when a process owns a given lock, but is also blocked, trying to gain the lock a second time
- Because the lock will never be released, the lock will never be gained
- Thus, we're stuck forever
- Interrupts can trigger self-deadlock
  - (Other things can cause it, too...)

Interrupts

- What are the tradeoffs of disabling interrupts?

  Good: self-deadlock impossible

  Bad: preemptive context switches never happen

Preventing Concurrency

- Instead of gaining & releasing any locks, we will simply enable and disable interrupts!

- ```
  while True:                    # not in CS
      old_psr = disable_ints()
      load  x → $r1              # in CS
      inc   $r1                  # in CS
      store $r1 → x              # in CS
      restore_ints(old_psr)
  ```
  ```

Concurrency

Three Classic forms of Concurrency

Multiprocessing

We won't be doing this in USLOSS

Time-Sharing

Will need to be considered

Interrupts

Exist in USLOSS!