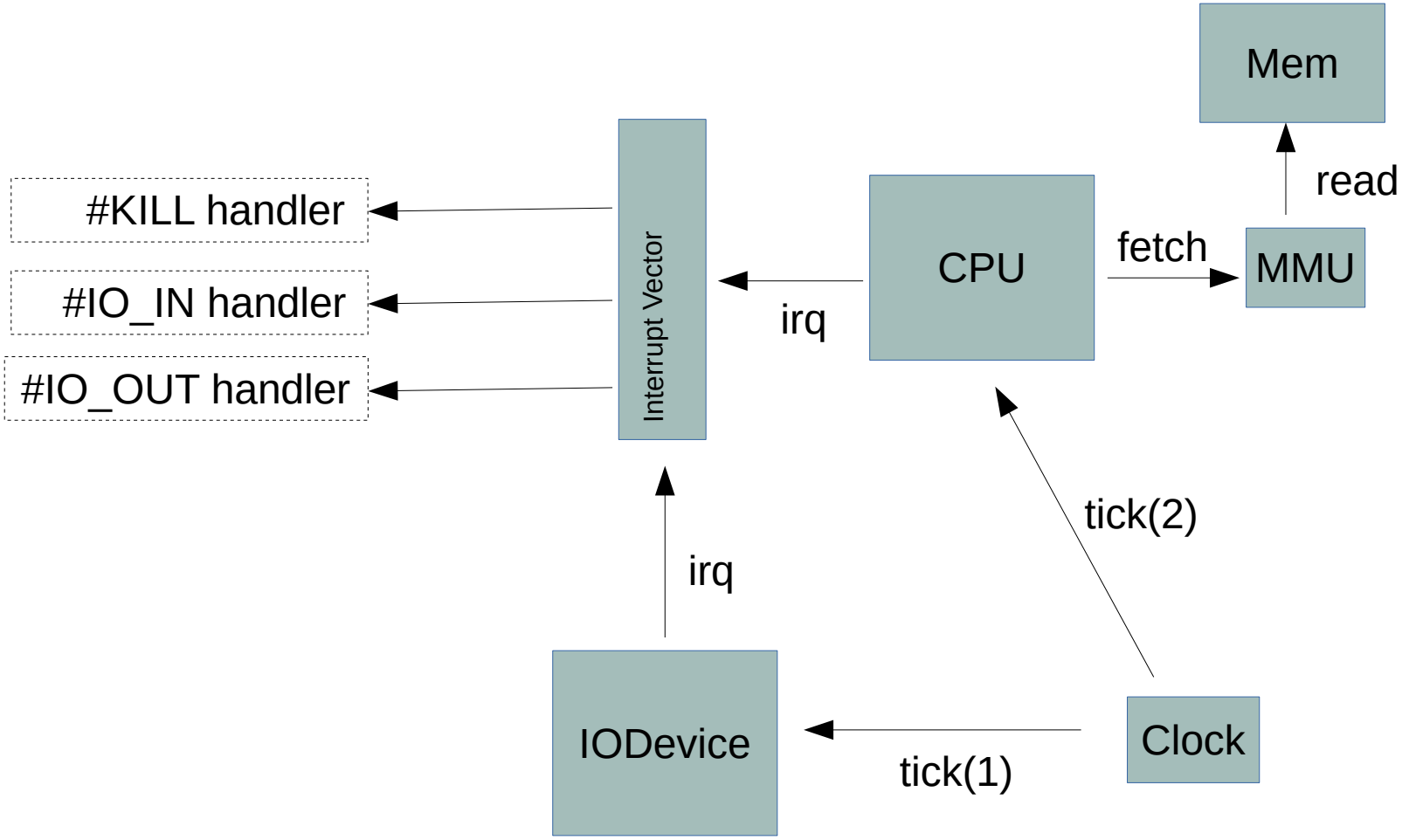
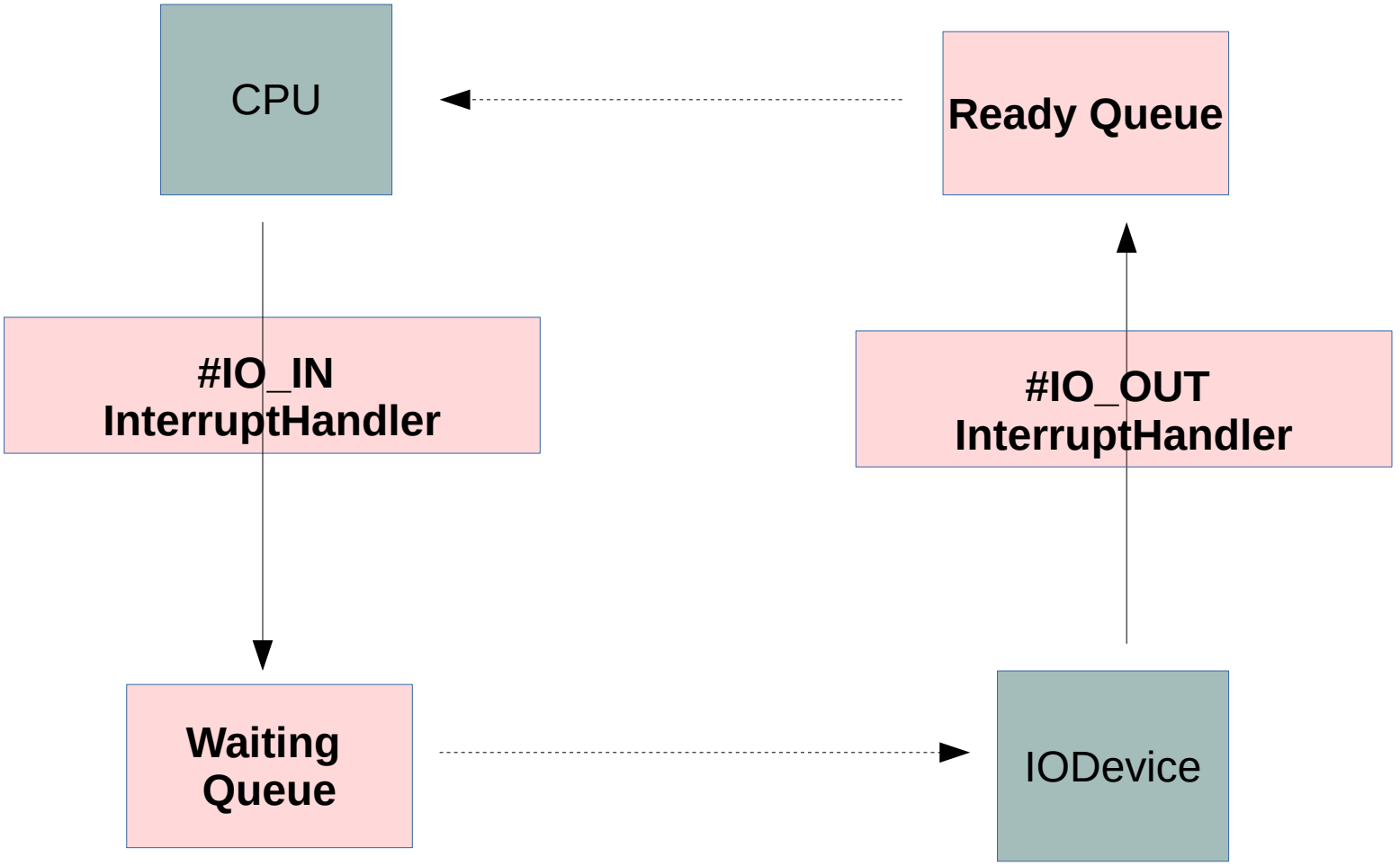
Simulador de S.O.

Práctica 3

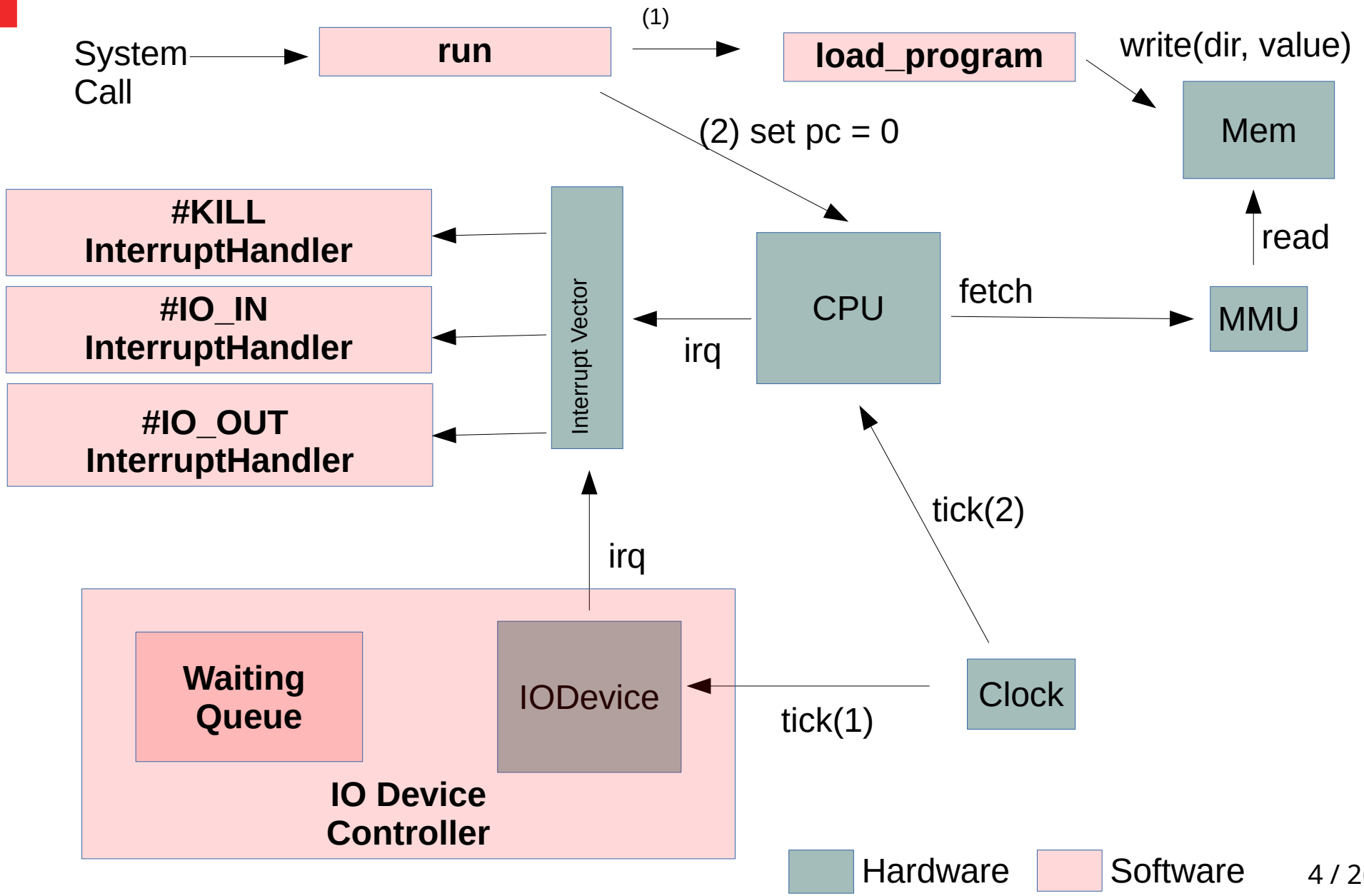
Hardware


Mem
read
#KILL handler
#IO_IN handler
#IO_OUT handler
Interrupt Vector
CPU
fetch
MMU
irq
tick(2)
irq
IODevice
Clock
tick(1)
Hardware
Software

Input/Output



Versión Inicial



CPU: IDLE / BUSY

- Manejo de CPU **IDLE** y **BUSY**
  - **IDLE**: _pc = -1
  - **BUSY**: _pc > -1
  - El cpu se inicia IDLE
  - Si el CPU esta IDLE,
    - Tick() no hace nada

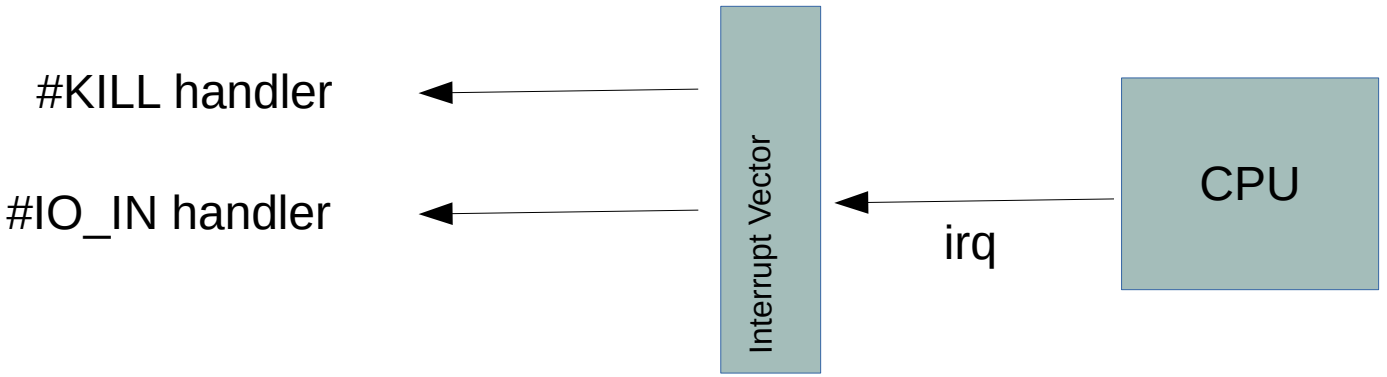
```
Cpu.__init__():
        self._pc = -1
```

```
Cpu.tick(self):
    if (self._pc > -1):
        self._fetch()
        self._decode()
        self._execute()
    else:
        log.logger.info("cpu - NOOP")
```

CPU: Interrupciones

- El CPU debe reconocer el “tipo” de instrucción a ejecutar y lanza **interrupt requests** (señal de hardware)
  - **CPU → logger.info("Exec: …”)**
    - “Ejecuta” la instucción.
  - **EXIT → lanza irq (#KILL)**
    - Avisa que el proceso en CPU terminó
  - **IO → lanza irq (#IO_IN)**
    - Avisa que el proceso en CPU necesita “ir” a I/O

CPU: Interrupciones

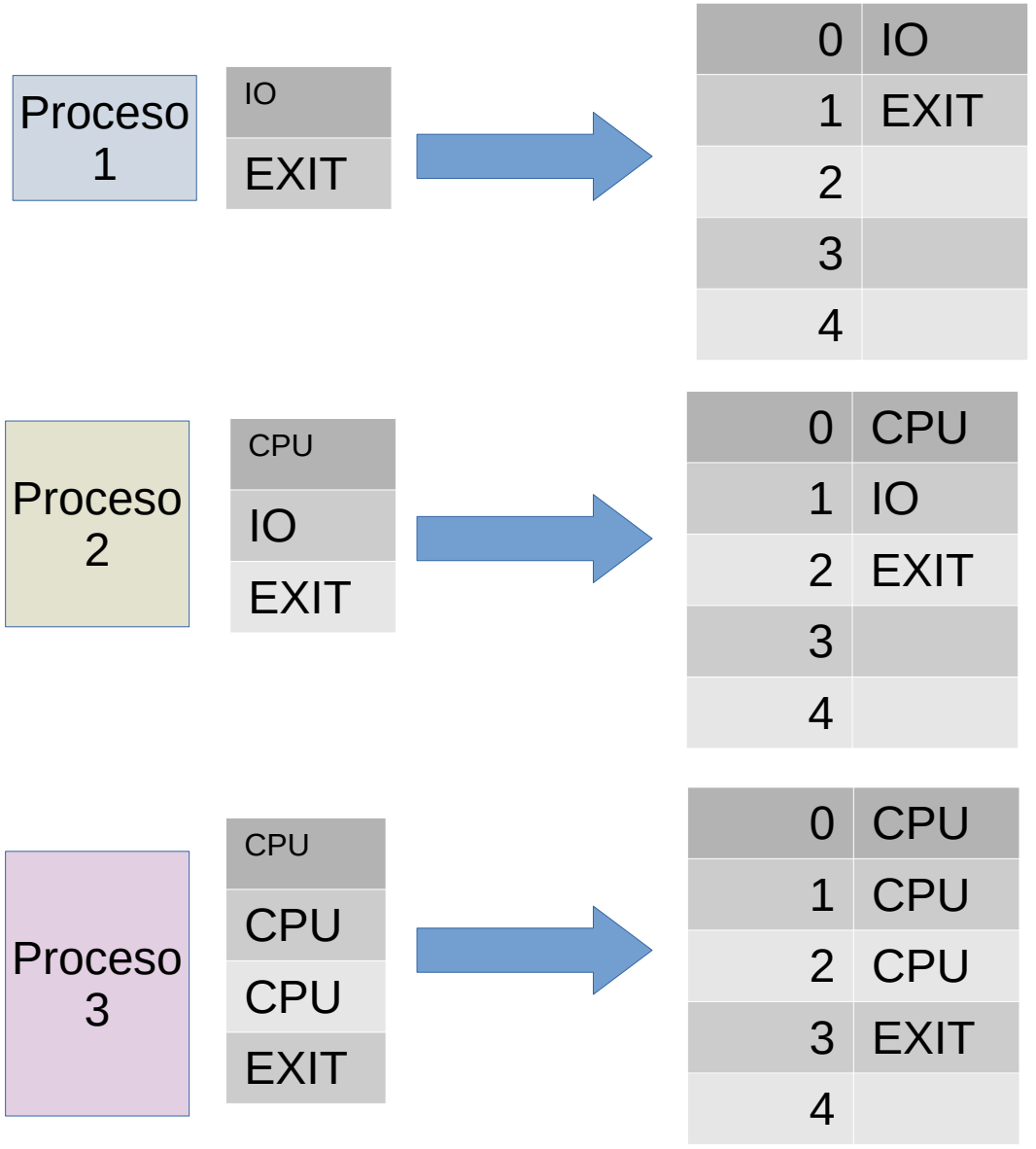


CPU: Interrupciones

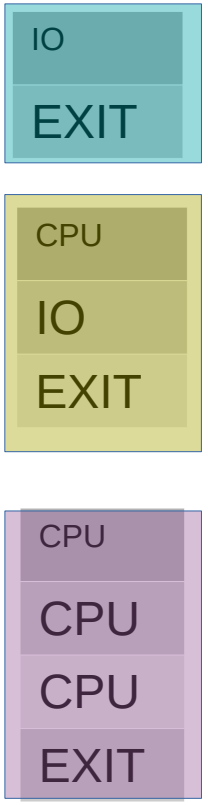
- El CPU debe reconocer el “tipo” de instrucción a ejecutar y lanza **interrupt requests** (señal de hardware)

```
def _execute(self):

        if ASM.isEXIT(self._ir):
            killIRQ = IRQ(KILL_INTERRUPTION_TYPE)
            self._interruptVector.handle(killIRQ)

        elif ASM.isIO(self._ir):
            ioInIRQ = IRQ(IO_IN_INTERRUPTION_TYPE, self._ir)
            self._interruptVector.handle(ioInIRQ)

        else:
            log.logger.info("cpu - Exec: {instr}, PC={pc}".format(instr=self._ir, pc=self._pc))
```

Memoria: Batch



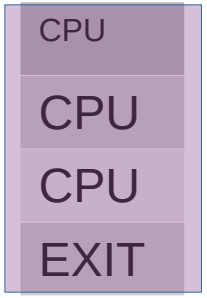
Memoria: Multiprogramación



0	IO	Proceso 1
1	EXIT	
2	CPU	Proceso 2
3	IO	
4	EXIT	
5	CPU	Proceso 3
6	CPU	
7	CPU	
8	EXIT	
9		
10		
11		
12		
13		

Dirección Lógica y Física

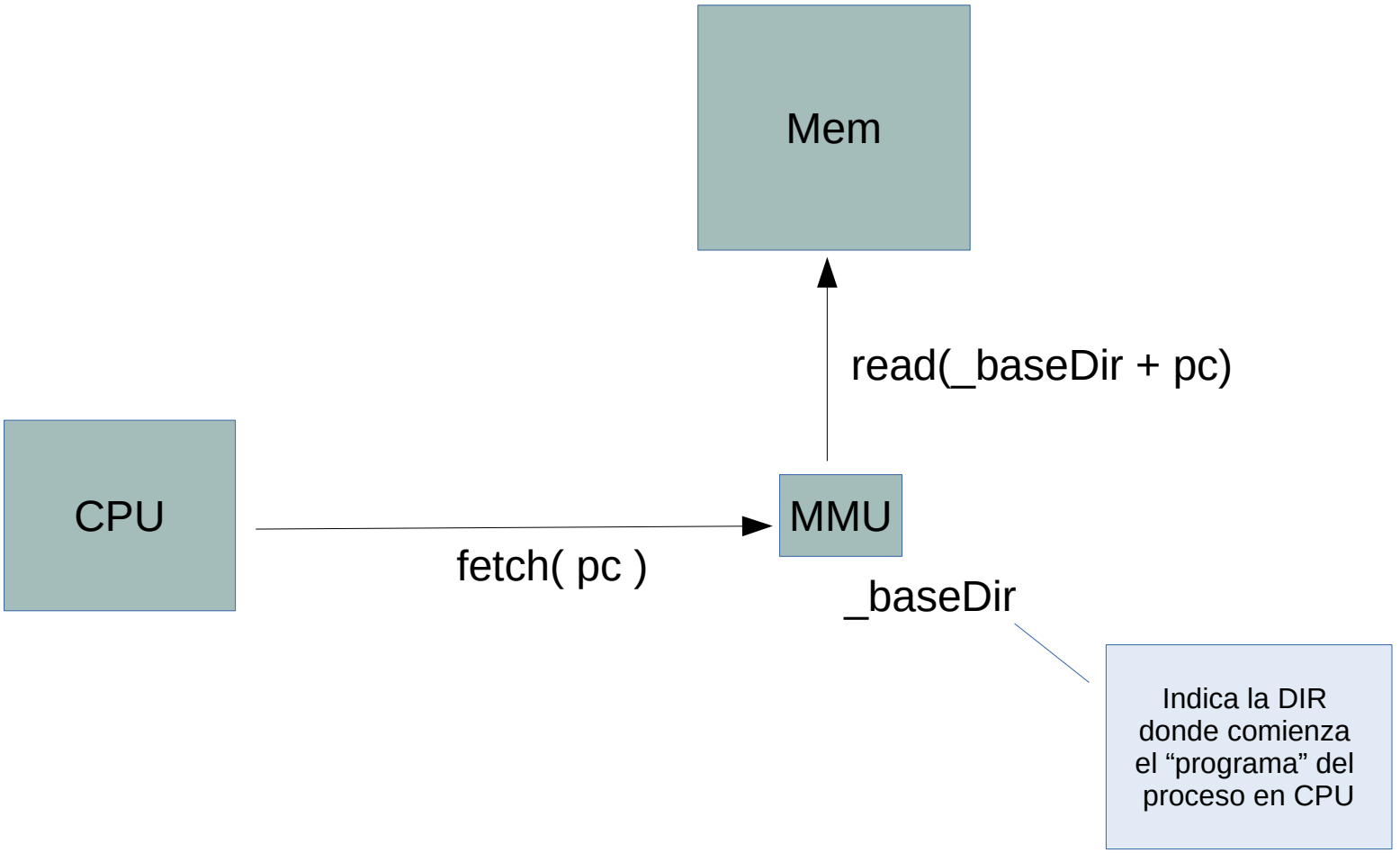
- Cuando un proceso se esta ejecutando, la CPU conoce la dirección lógica de la instrucción (pc) a ejecutar.



Dir Lógica (pc)	Instr	Dir Mem Física
0	CPU	5
1	CPU	6
2	CPU	7
3	EXIT	8

- Ej: que pasa si se quiere ejecutar el proceso 3 ??
  - 1: Necesito tener “el programa” cargado en memoria
  - 2: Inicializo el pc de la CPU
    - CPU._pc = 0
  - 3: **Como hago el fetch de la instruccion ??**

Memory Managment Unit



MMU

- Maneja la transformación de direcciones **lógicas** a direcciones **físicas** de la instrucción a “fetchear”
- La BaseDir del MMU **debe** ser la del proceso que está ejecutando la CPU.

```
def fetch(self,  logicalAddress):

    if (logicalAddress > self._limit):
        raise Exception("Invalid Address "

    physicalAddress = logicalAddress + self._baseDir
    return self._memory.read(physicalAddress)
```

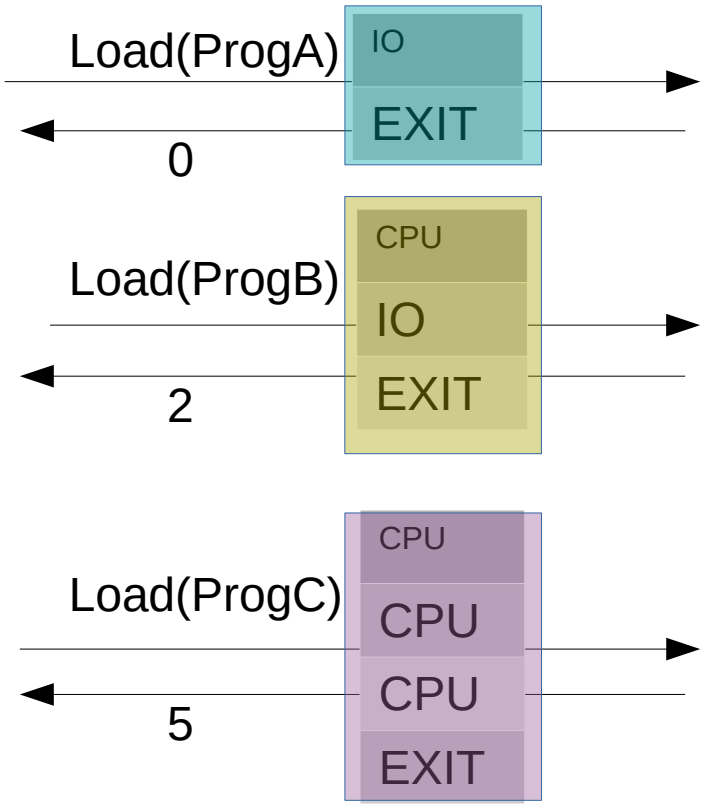
- El MMU contiene la baseDir del proceso “running”.

Loader

Para simplificar el proceso de ejecucion de un programas, podriamos tener un componente específico que se encargue de cargar en memoria los programas

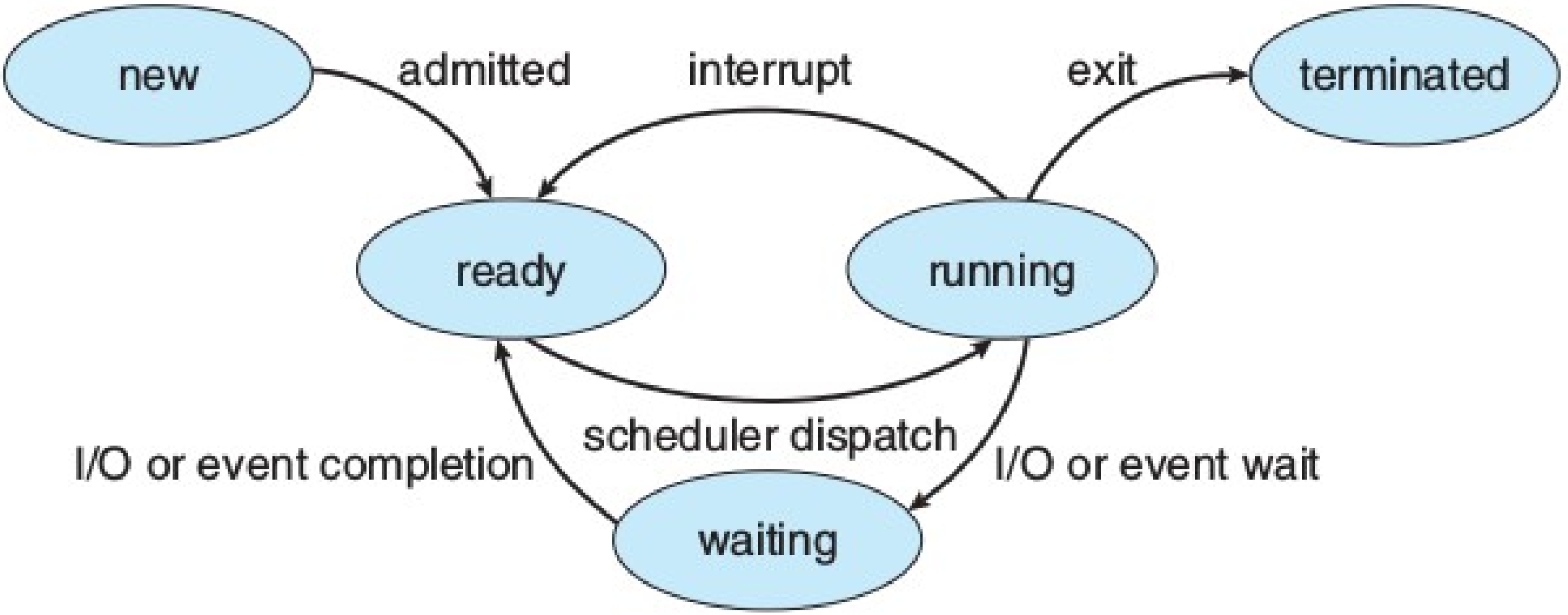
- El loader puede guardar el “próximo” lugar de inicio a guardar el programa
- Loader.load(program) : carga el programa en la memoria y retorna el “BaseDir” donde esta cargado el mismo

Loader



0	IO	Proceso 1
1	EXIT	
2	CPU	Proceso 2
3	IO	
4	EXIT	
5	CPU	Proceso 3
6	CPU	
7	CPU	
8	EXIT	
9		
10		
11		
12		
13		

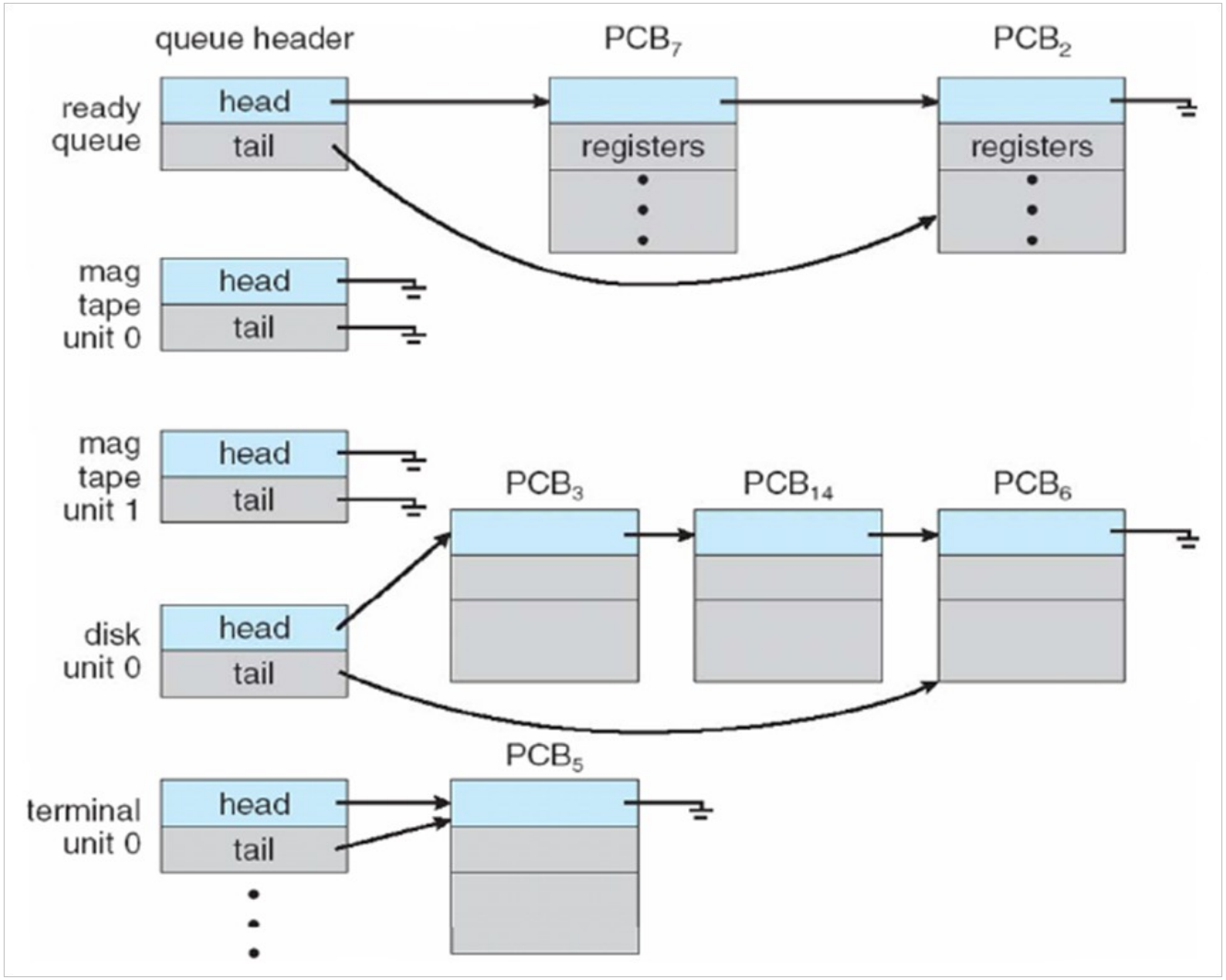
Procesos: Estados



PCB: Process Control Block

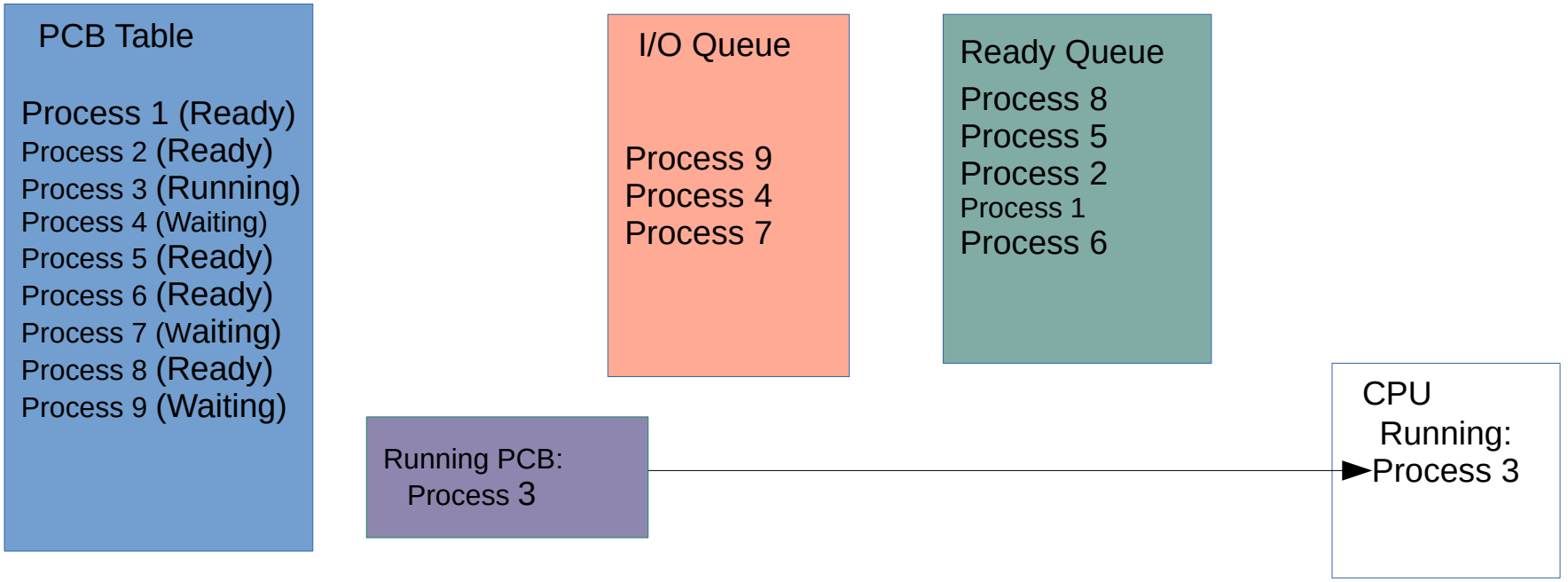
- Mantiene el estado del proceso.
- Como mínimo necesitamos:
  - pid: entero (positivo)
  - baseDir: entero (positivo)
  - pc: entero (positivo)
  - state: [new| ready | running |waiting | terminated] **(enum o constante)**
  - path: ej "test.exe"

Ready Queue


queue header
PCB7
PCB2
ready queue
head
tail
registers
registers
mag tape unit 0
head
tail
mag tape unit 1
head
tail
PCB3
PCB14
PCB6
disk unit 0
head
tail
PCB5
terminal unit 0
head
tail

Queues y PCB Table

- Por el momento nos interesa saber que procesos están listos para correr, esperando por I/O y cuál esta corriendo en el CPU



PCB Table

- Es una tabla que maneja el Kernel donde están todos los PCBs del Sistema.

  Operaciones de ejemplo:

  - **get**(pid) → retorna el PCB con ese pid
  - **add**(pcb) → agrega el PCB a la tabla
  - **remove**(pid) → elimina el PCB con ese PID de la tabla

- También genera PIDs únicos:
  - **getNewPID**() → retorna un PID único
    - Cuando se crea un PCB, se le asigna un PID único
    - Los PIDs no se reutilizan

Running PCB

- El Kernel va a mantener el pcb que esta en el CPU.
  - **runningPCB**: (setter y getter) para “dejar a mano” el PCB que esta en la CPU

Context Switch

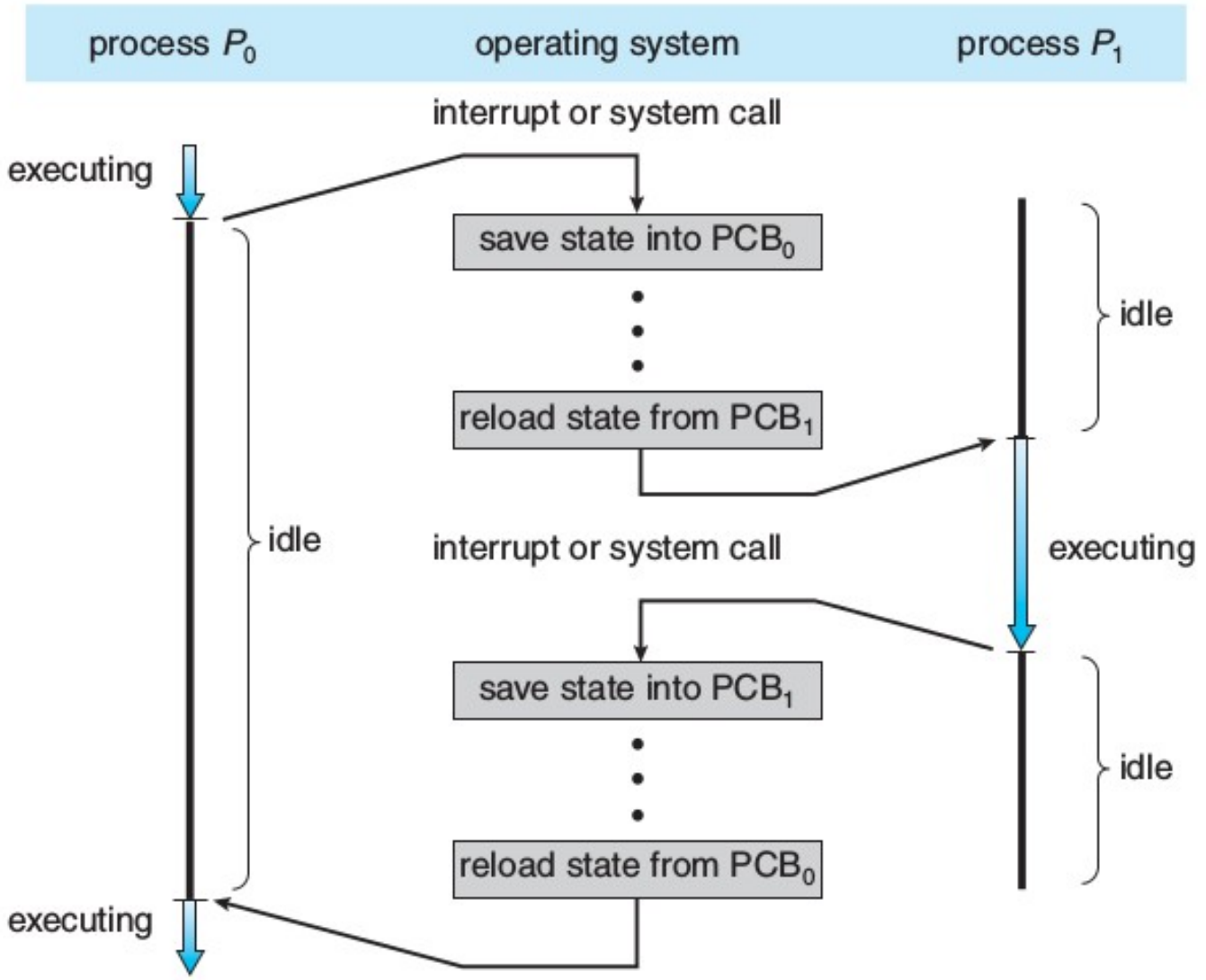
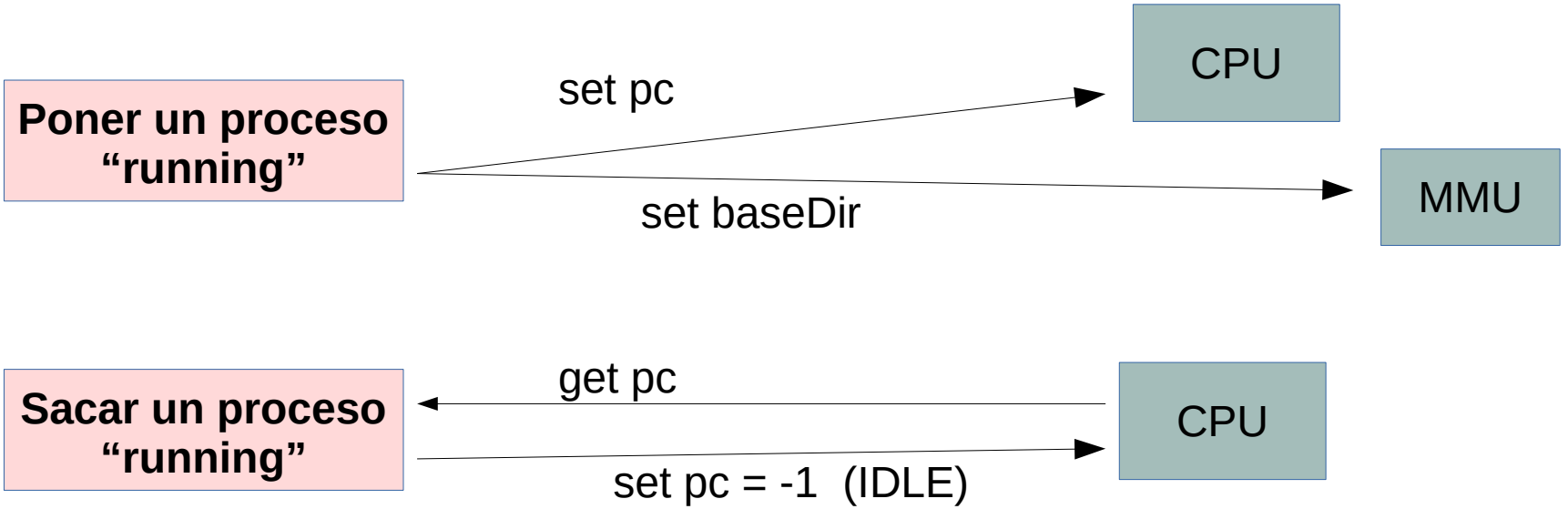


Figure 3.4 Diagram showing CPU switch from process to process.

Práctica 3: Dispatcher



Dispatcher

- Sirve como “Driver” del CPU + MMU:
- “Carga” un proceso en el CPU
- “Salva” el estado del proceso en CPU

- Operaciones:
  - **load(pcb)** →

    CPU.??? = pcb.???

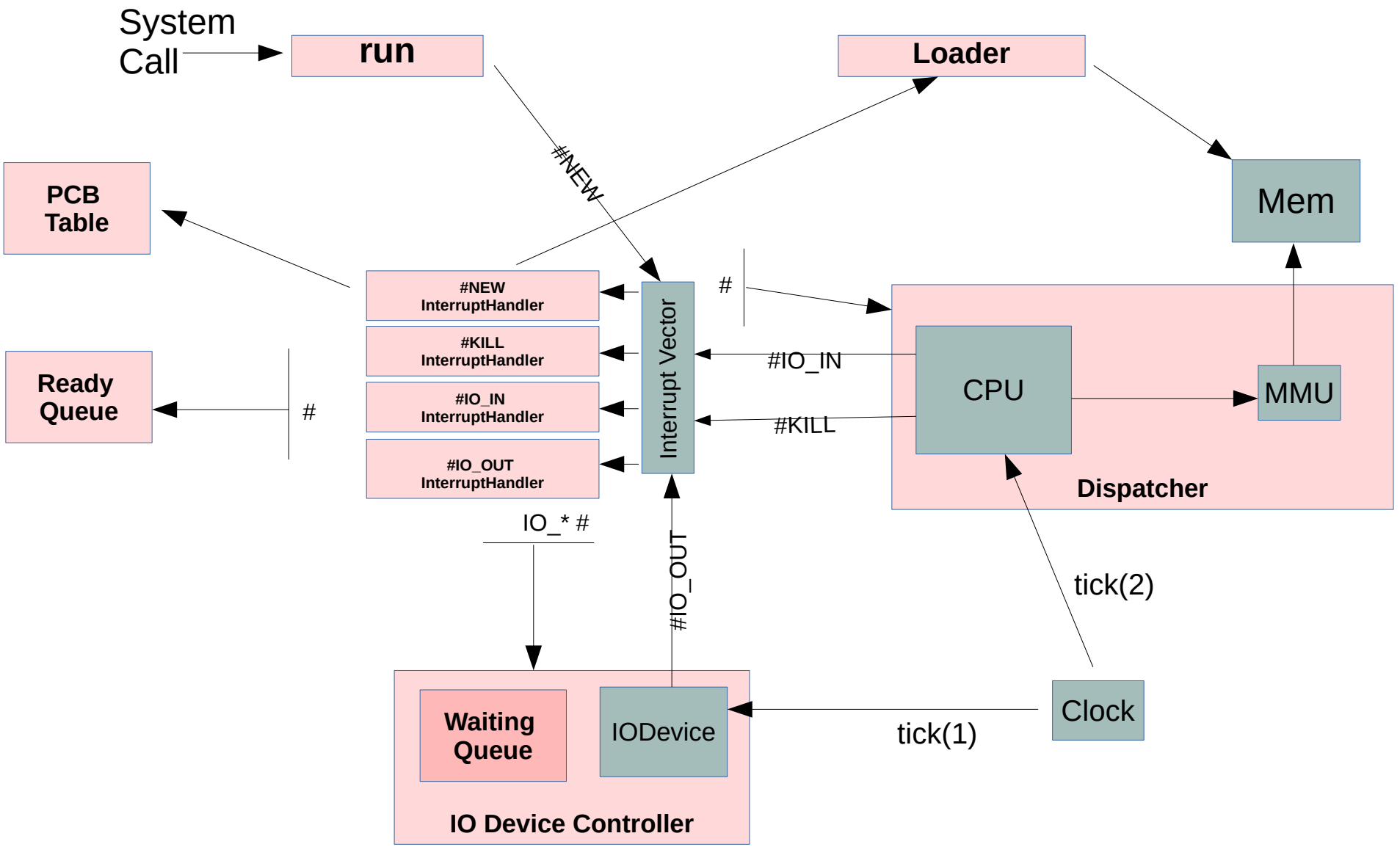
    MMU.??? = pcb.???

  - **save(pcb)** → salva el estado de la CPU en el PCB y deja el CPU IDLE

    pcb.??? = CPU.???

    CPU.pc = -1 # CPU queda IDLE hasta el proximo load()

Practica 3: S.O. Esperado



Handler de #New

- Una vez que tengamos funcionando los otros InterruptorsHandlers debemos transformar el run() en una IRQ de #New. Esto lo hacemos para para emular la creación de procesos en “Kernel-Mode”

```
def run(self, program):

    newIRQ = IRQ(NEW_INTERRUPTION_TYPE, program)
    HARDWARE.interruptVector.handle(newIRQ)
```

- Tienen que crear una clase (NewInterruptionHandler) para manejar este tipo de interrupciones
- Y luego hay que configurarlo en el kernel.__init__()

```
newHandler = NewInterruptionHandler(self)
HARDWARE.interruptVector.register(NEW_INTERRUPTION_TYPE, newHandler)
```