Simulador de S.O.

Refactor 1

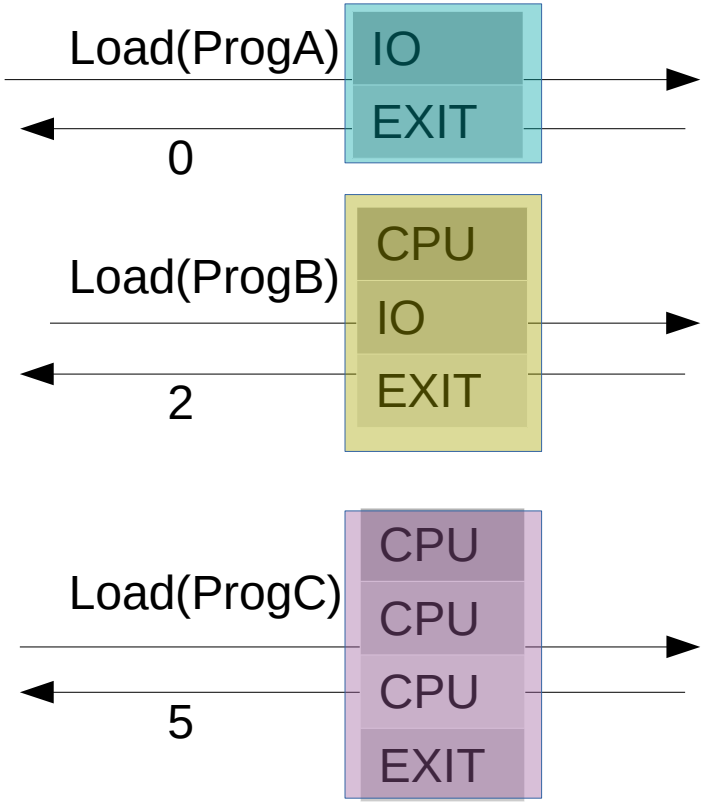
Evolución del simulador

- Cómo podemos optimizar el uso del CPU?

Memoria (por ahora Híbrida)

- clase **Memory** debe emular:
  - “Chip” de memoria [Hard]
    - Tamaño de Memoria: Infinito (**Imposible**)
  - Manejo de Memoria Lógica [Soft]
  - Loader (carga los programas) [Soft]
  - Soporte para cargar varios programas a la vez (multiprogramación) - [Soft]
  - **… Por ahora la dejamos así por simplicidad**

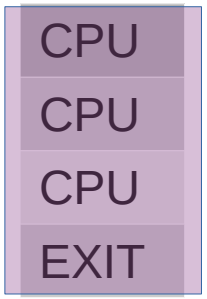
Memoria: Multiprogramación



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

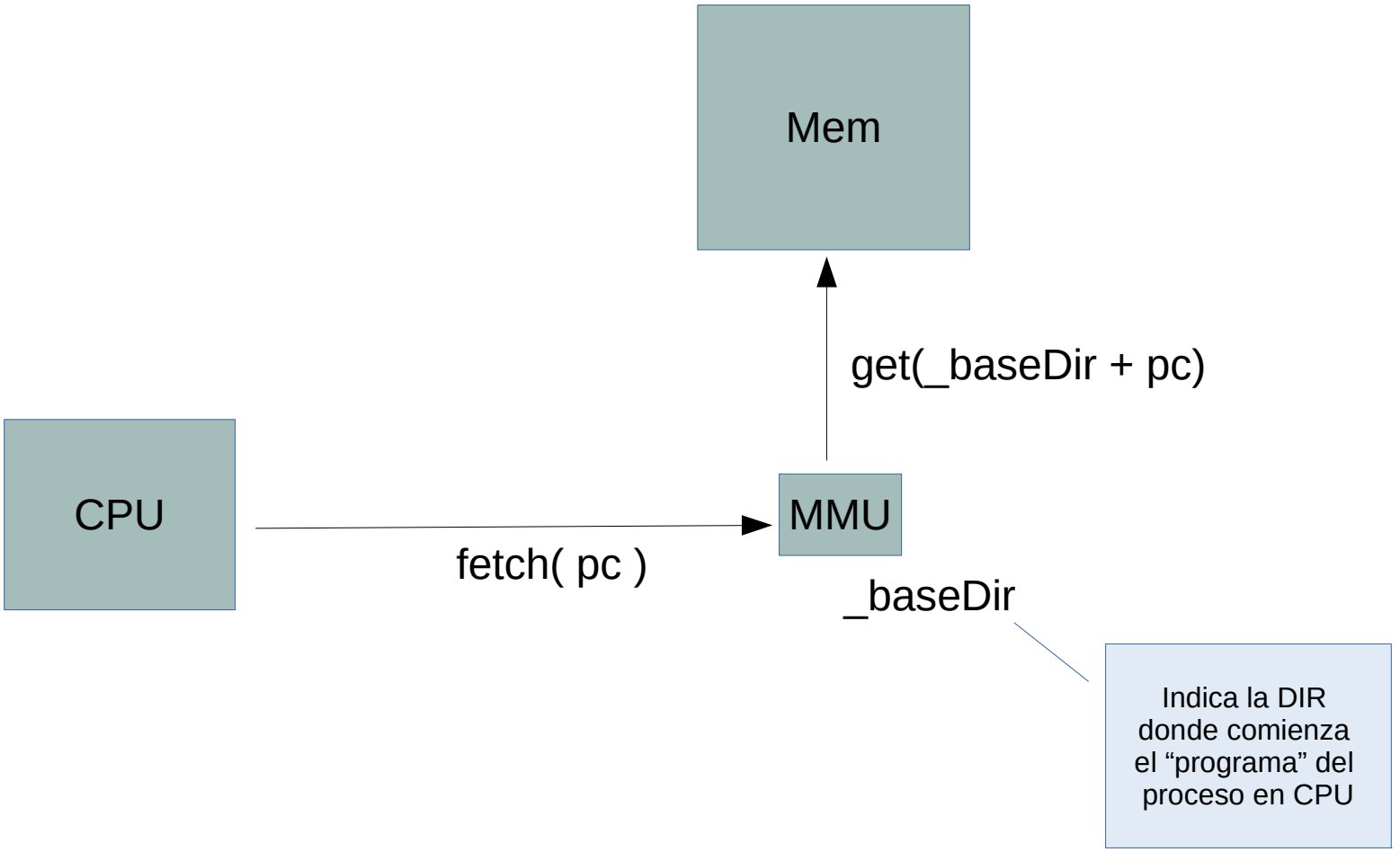
- Cuando un proceso se esta ejecutando, la CPU conoce la dirección lógica de la instrucción (pc) a ejecutar.



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.

```
MMU.fetch(self, pc):
        dirFisica  = self._baseDir + pc
        return self._memory.get(dirFisica)
```

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

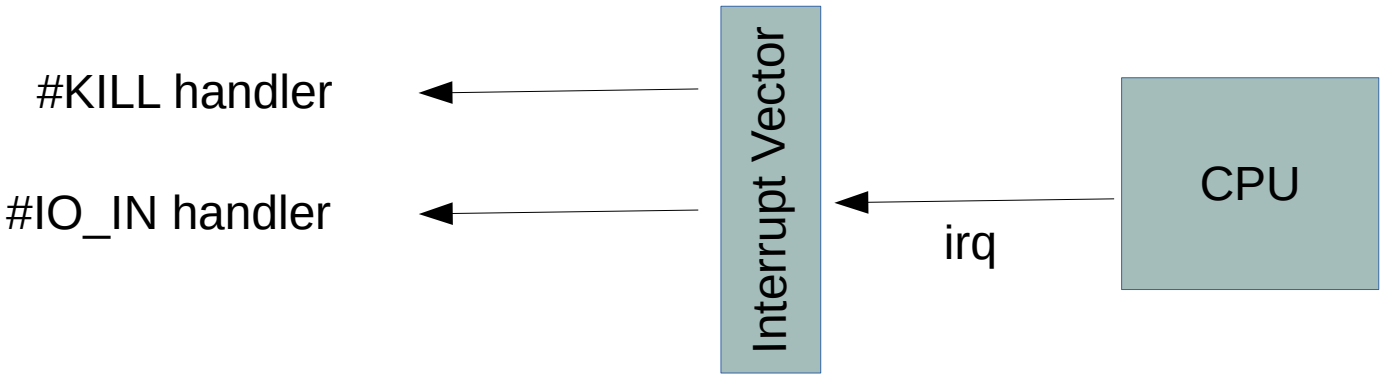
Evolución del simulador

- Cómo hacemos para determinar que el proceso quiere hacer I/O o terminó (EXIT) ?

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
    - **_interruptVector.kill()**
  - **IO → lanza irq (#IO_IN)**
    - Avisa que el proceso en CPU necesita “ir” a I/O
    - **_interruptVector.ioIn()**

Interrupt Vector



Clock

- Loop “infinito” avisando a los componentes de hardware que se inicia un ciclo (tick)

```
class Clock():

def __init__(self, cpu):
  self._cpu = cpu

def start(self):
   while true:
       self._cpu.tick()
       sleep(1)
```

- El clock “conoce” a todos los componentes de hardware que deben recibir el “tick”... por ahora solo CPU.
- Ahora podemos sacar el sleep(1) del cpu._execute()

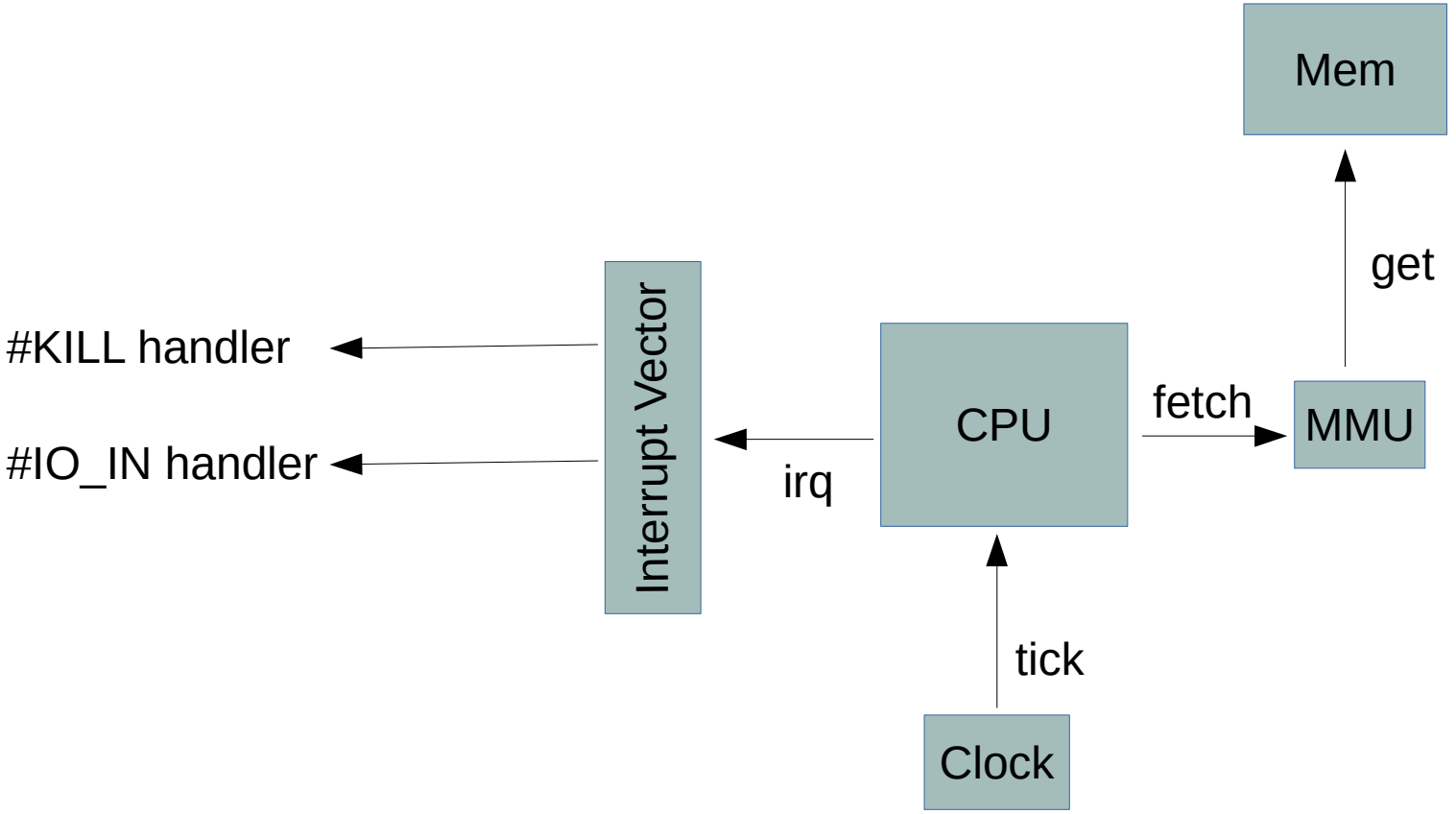
CPU: IDLE / BUSY

- Manejo de CPU **IDLE** y **BUSY**
  - **IDLE**: _pc = -1
  - **BUSY**: _pc > -1
  - Inicializar el _pc en -1
  - Si el CPU esta IDLE,
    - Tick() no hace nada

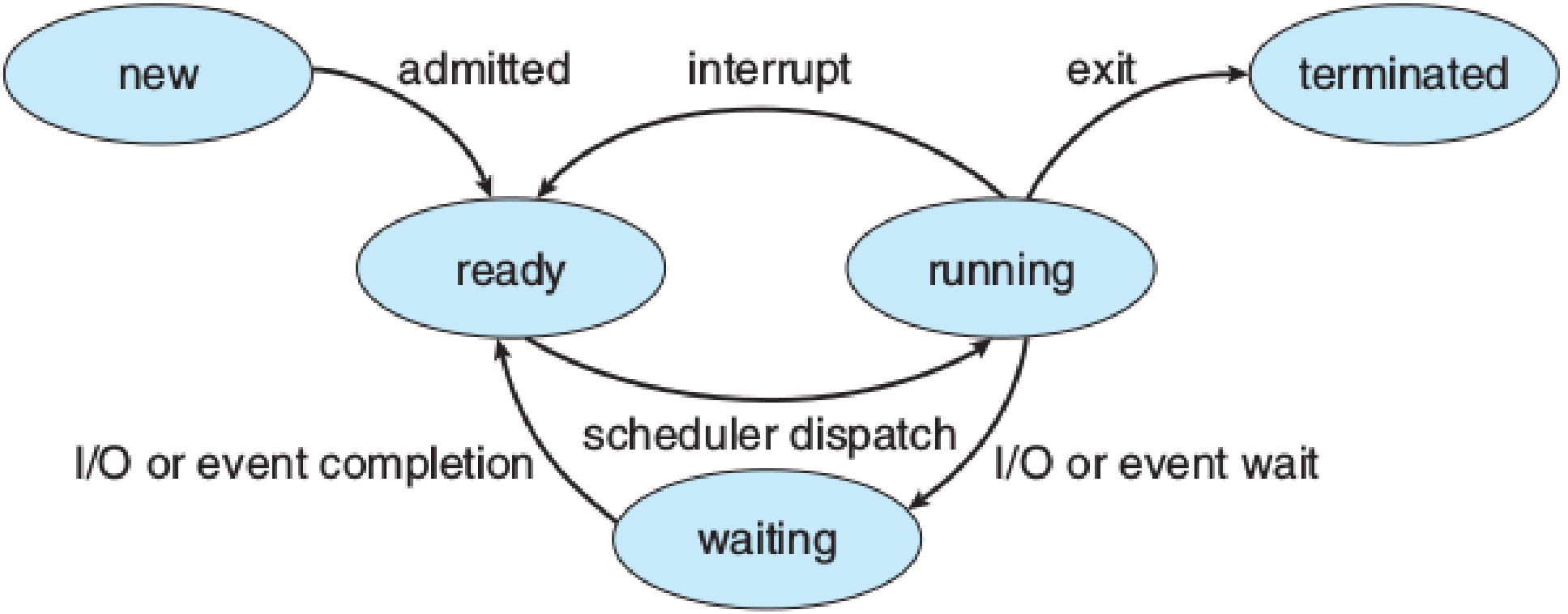
```
Cpu.__init__():
        self._pc = -1
```

```
Cpu.tick(self):
        if self._pc >= 0:
                self._fetch()
                self._decode()
                self._execute()
```

El Hardware queda así



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"

PCB Table

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

  Operaciones:

  - **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

Context Switch

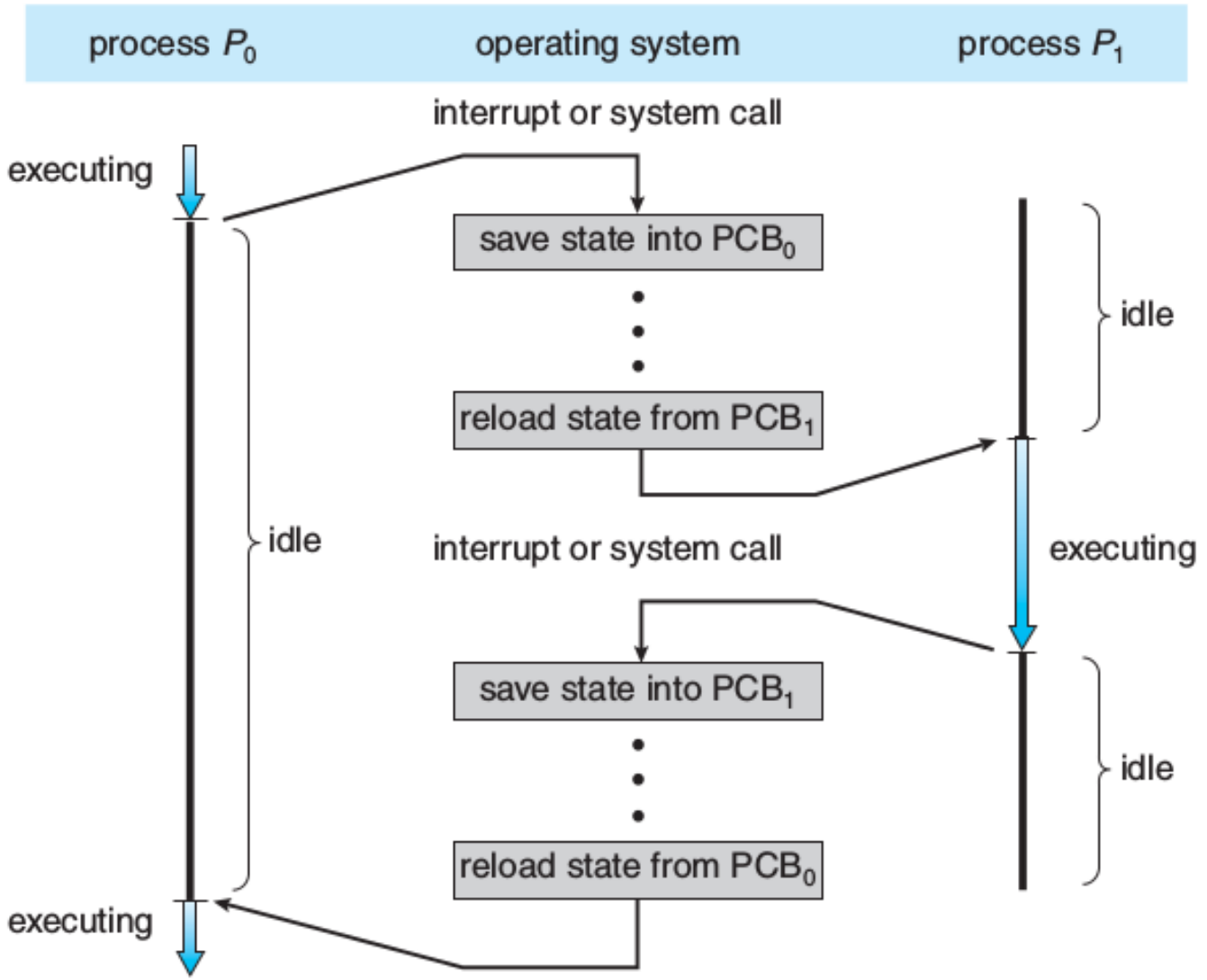


Figure 3.4 Diagram showing CPU switch from process to process.

Dispatcher

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

- Operaciones:
  - **load(pcb)** →

    self._cpu.??? = pcb.???
    self._mmu.??? = pcb.???

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

    pcb.??? = self._cpu.???

Scheduler

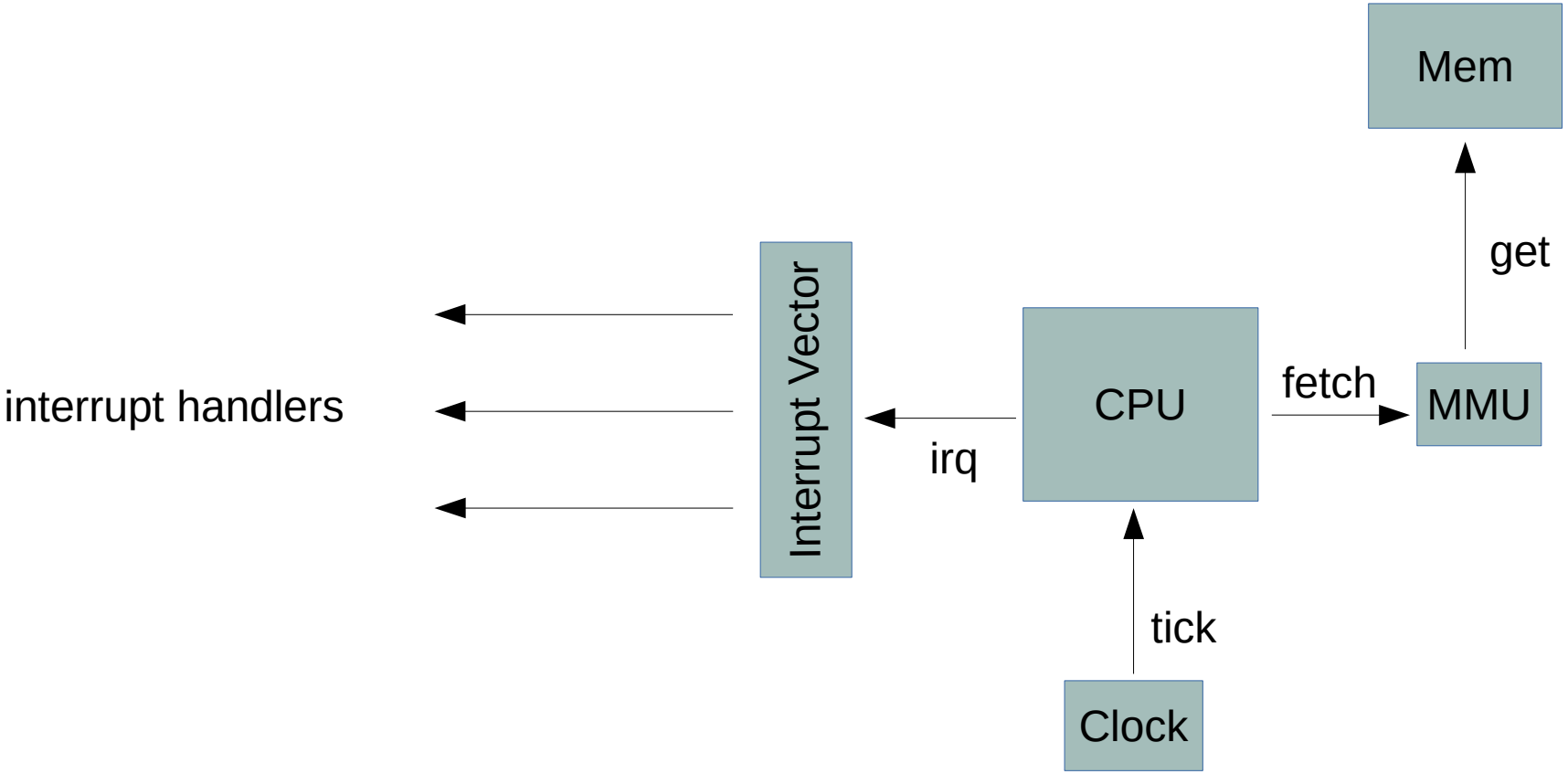
- Administra la **“Ready Queue”**
  - Cola de procesos listos para ejecutar (“ready”).

- Operaciones:
  - **add(PCB)** → agrega el PCB a la Ready Queue
  - **getNext()** → desencola el proximo PCB a ser ejecutado

Interrupt Handlers (híbrido)

- Por ahora, la clase **InterruptVector** emula el Vector de Interrupciones (Hardware) y el manejo de las mismas **Interrupt Handlers** (Software)
  - Interrupt Handlers:
    - **#KILL**: **Saca** el proceso del CPU y lo **elimina** del sistema operativo
    - **#NEW**: **Agrega** un nuevo proceso al sistema y lo deja **listo** para ejecutar
    - **#IO_IN**: **Saca** el proceso del CPU y lo deja **esperando** al dispositivo de **I/O** (no simulamos el I/O... por ahora)

Hardware


Mem
get
interrupt handlers
Interrupt Vector
irq
CPU
fetch
MMU
tick
Clock

Kernel (software)

- Mem Lógica
- Loader

#NEW InterruptHandler

#KILL InterruptHandler

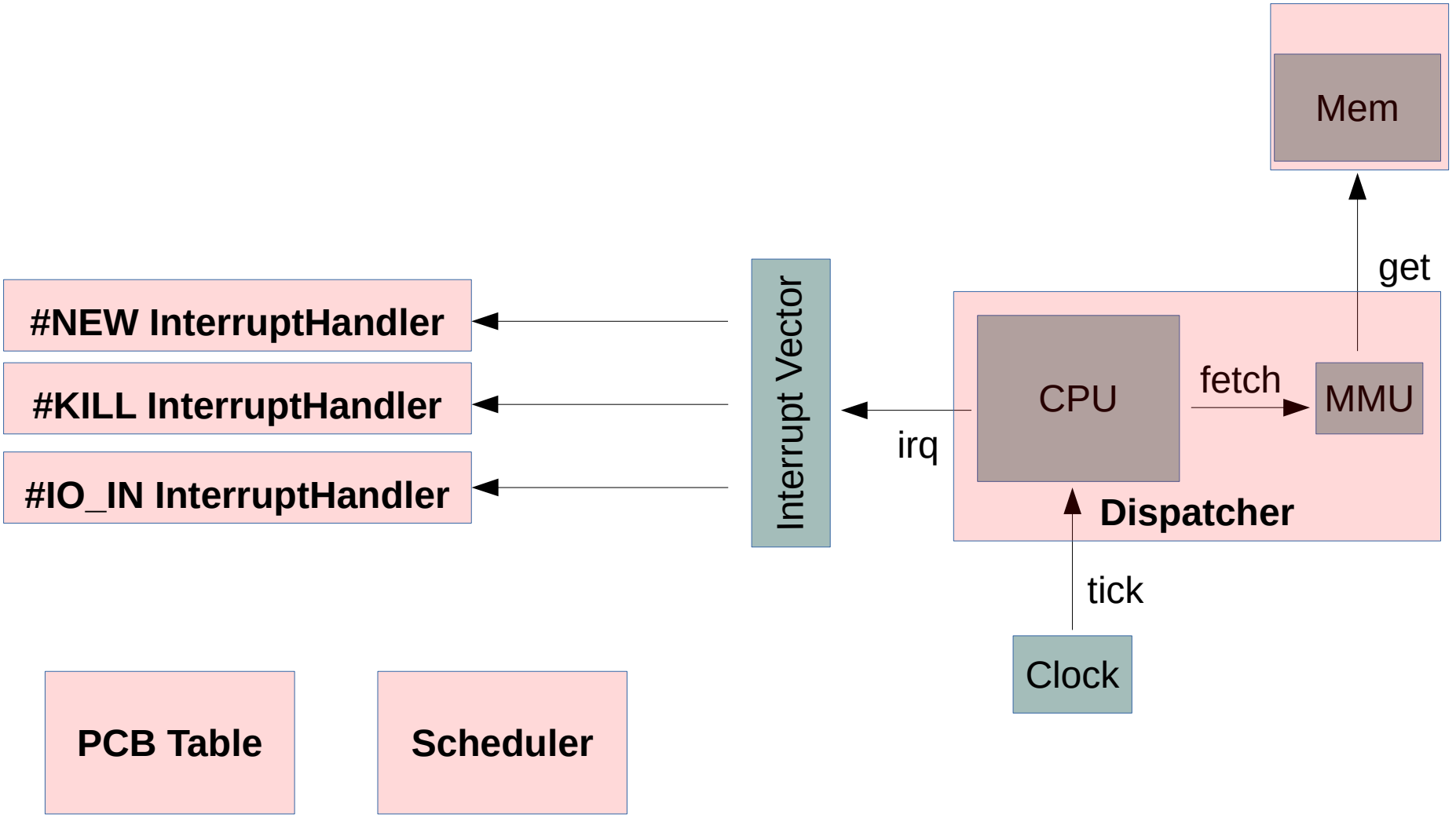
#IO_IN InterruptHandler

Dispatcher

PCB Table

Scheduler

Simulador


Mem
get
#NEW InterruptHandler
#KILL InterruptHandler
#IO_IN InterruptHandler
Interrupt Vector
irq
CPU
fetch
MMU
Dispatcher
tick
Clock
PCB Table
Scheduler

Main

```
if __name__ == '__main__':

   ## Configure Logger
   ... (queda igual)
   logger.info('Starting emulator')

   so = SO()


   ## Cargo 3 programas
   pA = Program("progA.exe", [CPU(5), IO(1), CPU(5)])
   so.exec(pA)
   pB = Program("progB.exe", [IO(1), CPU(3), IO(1), CPU(2)])
   so.exec(pB)
   pC = Program("progC.exe", [CPU(6)])
   so.exec(pC)

   ## inicio la ejecución
    so.start()
```

Clase S.O.

```
class SO():

    def __init__(self):
        ## Hardware
        self._hardware = Hardware()

        ## Software
        self._kernel = Kernel(self._hardware)


    def start(self):
        self._hardware.clock.start()



    def exec(self, prog):
        self._hardware.interruptVector.new(prog)
```

Clase Hardware

```
class Hardware():

    def __init__(self):
        ## Hardware
        self._memory = Memory()
        self._mmu = MMU(self._memory)
        self._cpu = Cpu(self._mmu)
        self._clock =  Clock(self._cpu)
        self._interruptVector = InterruptVector()
```

Clase Kernel

```
class Kernel():

    def __init__(self, hardware):
        ## Hardware
        self._hardware = hardware
        self._dispatcher = Dispatcher(self._hardware.cpu, self._hardware.mmu)
        self._scheduler = Scheduler()
        self._pcbTable = PCBTable()

        ## por ahora interruptVector es un híbrido (Hard/Soft)
        ## Interrupt Handlers config
        self._interruptVector.kernel = self
        self._interruptVector.hardware = self._hardware
```