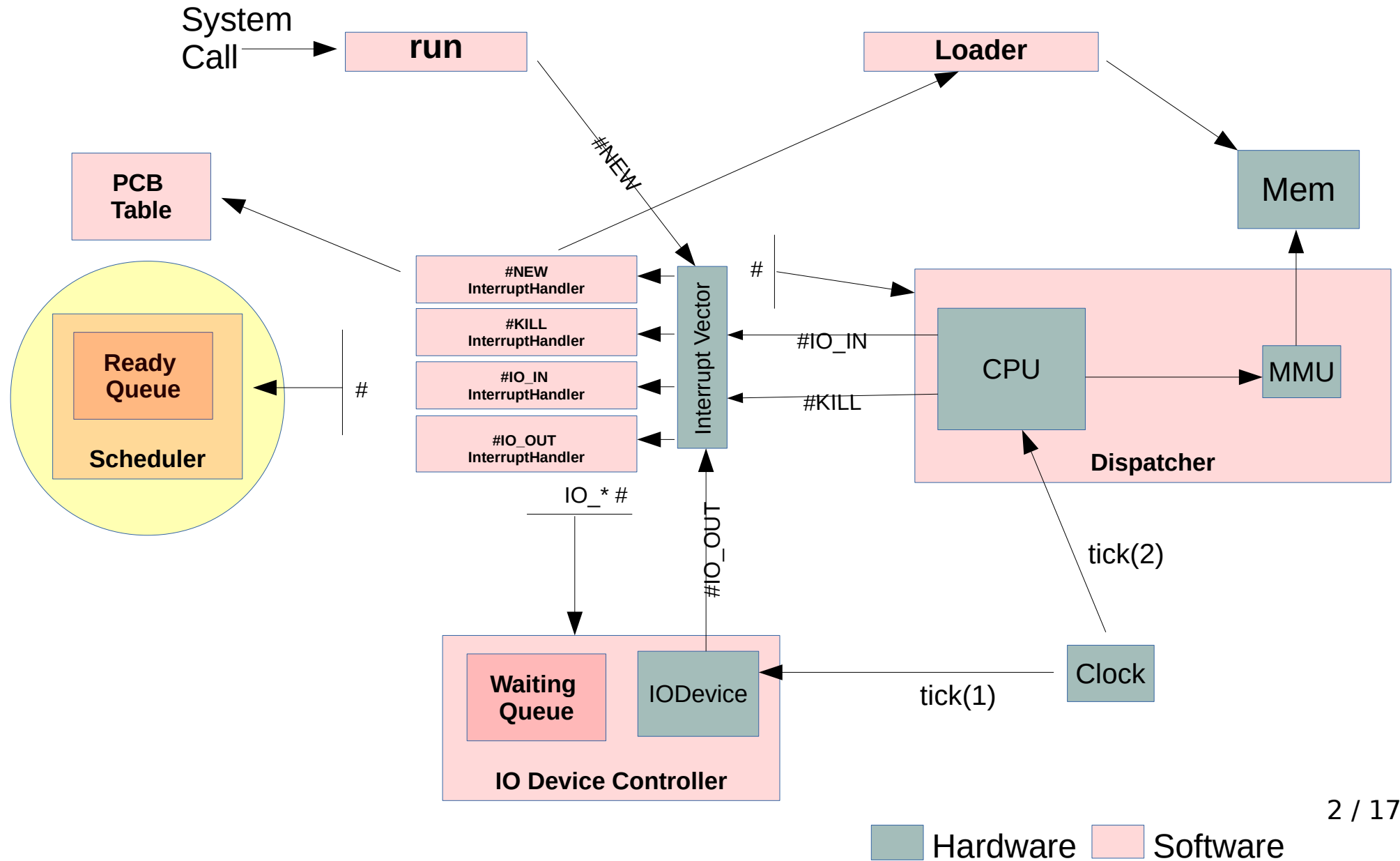




Simulador de S.O.

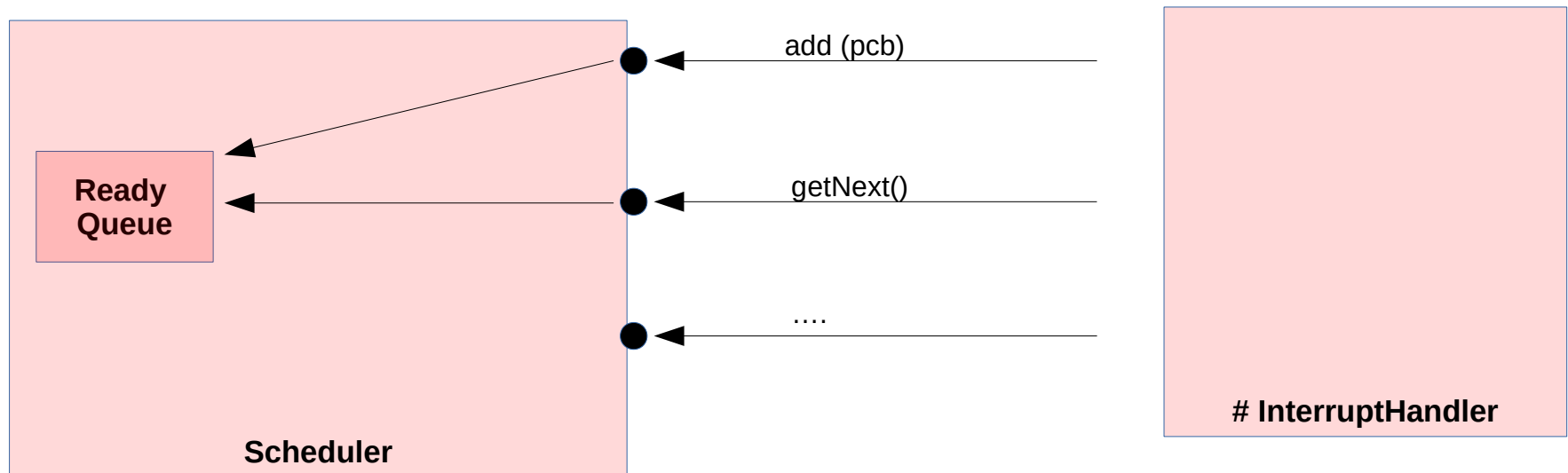
Práctica 4

Practica 4: Scheduler



Practica 4: Scheduler

- El Scheduler debe ser el único componente que acceda y modifique la **Ready Queue**
- Si definimos un “protocolo comun”, podremos intercambiar el “tipo” de protocolo sin afectar los demas componentes



Ocultamos la readyQueue dentro del Scheduler para no impactar en los demás componentes un cambio en el algoritmo de planificación (y ordenación de la readyQueue)



Preemptive Scheduler

- Cómo podemos manejar la expropiación?

Preemptive Scheduler

```
#### Cada vez que tengamos que agregar un pcb a la readyQueue  
#### tenemos que preguntarle a scheduler si debemos expropiar
```

```
    pcbToAdd # el pcb que acaba de “pasar” a ready  
    pcbInCPU = pcbTable.runningPCB
```

```
    If scheduler.mustExpropriate(pcbInCPU, pcbToAdd):
```

```
        ## hay que expropiar  
        pcbExpropiado = Sacar el PCB actual del CPU  
        agregar pcbExpropiado a la cola de ready  
        Cargar en el CPU el pcbToAdd
```

```
    else :
```

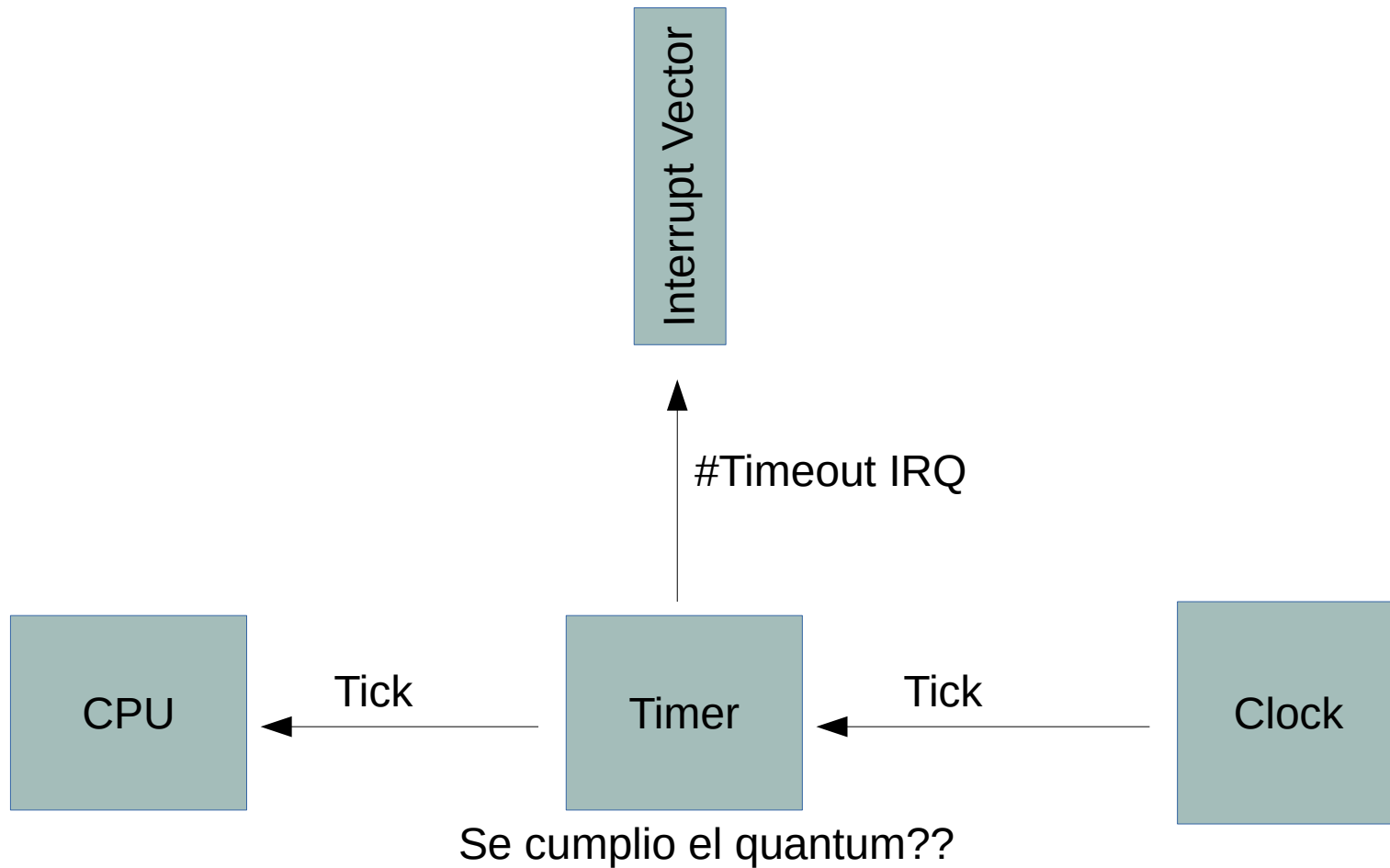
```
        agregar pcbToAdd a la cola de ready del scheduler
```



Round Robin

- Cómo podemos manejar el timeout de Round Robin?
- Tenemos control de cuantos ticks puede correr la cpu??

Timer



Timer

Class Timer

```
def __init__(self, cpu, interruptVector):
    self._cpu = cpu
    self._interruptVector = interruptVector
    self._tickCount = 0 # cantidad de de ciclos "ejecutados" por el proceso actual
    self._active = False # por default esta desactivado
    self._quantum = 0 # por default esta desactivado

def tick(self, tickNbr):
    # registro que el proceso en CPU corrio un ciclo mas
    self._tickCount += 1

    if self._active and (self._tickCount > self._quantum) :
        # se "cumplio" el limite de ejecuciones
        timeoutIRQ = IRQ(TIMEOUT_INTERRUPTION_TYPE)
        self._interruptVector.handle(timeoutIRQ)
    else:
        self._cpu.tick(tickNbr)
```

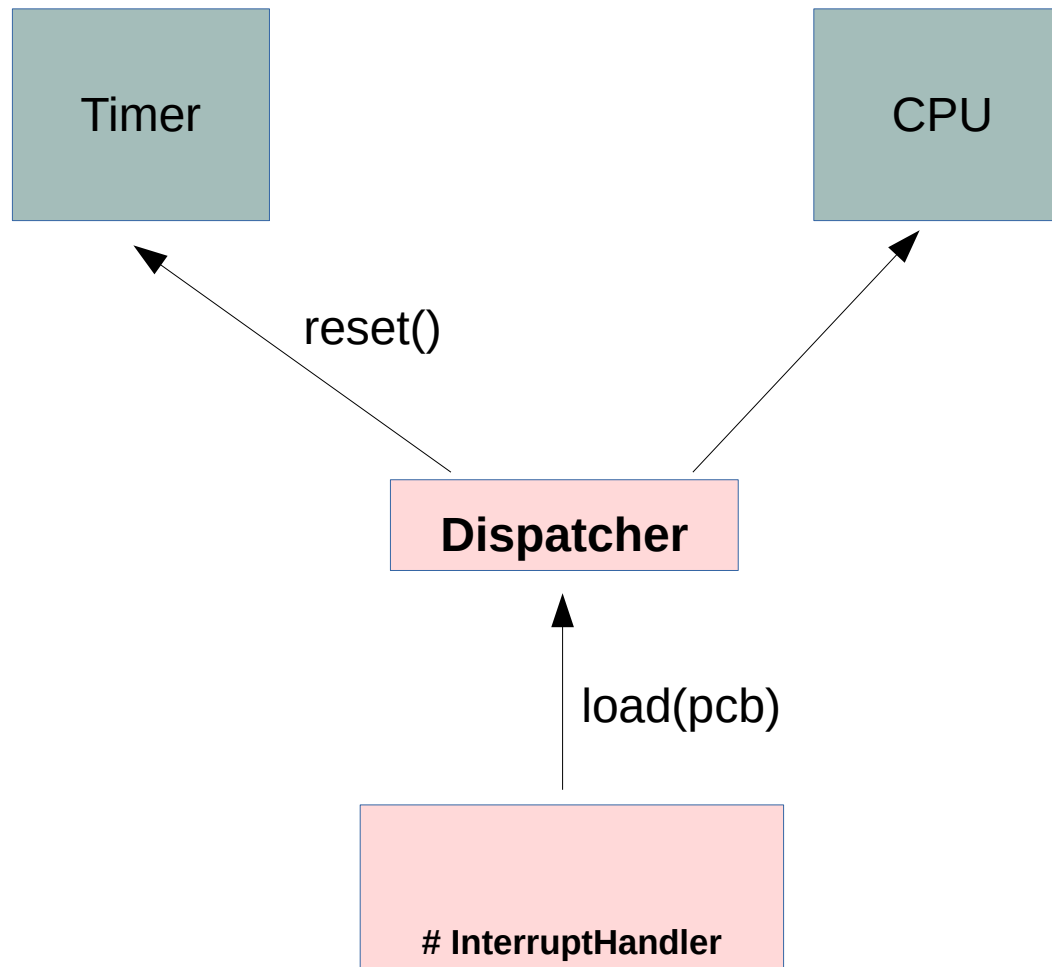



Timer

- Esta completo?
- Que pasa si el proceso de CPU se va a I/O antes de que se cumpla el quantum?

Context Switch

Volver a 0 el contador de ticks
a ejecutar antes de #timeout



Reset del Timer

Dispatcher

```
def load(self, pcb)
    ...
    ## al hacer un context switch
    HARDWARE.timer.reset()
```

Class **Timer**

```
def reset(self):
    self._tickCount = 0
```

Configuración Round Robin

- Para activar el Timer, solo debemos setearle un quantum.

```
@quantum.setter  
def quantum(self, quantum):  
    self._active = True  
    self._quantum = quantum
```

- También debemos implementar un handler para la interrupcion **#Timeout**

Configuración Round Robin

```
class Kernel():

    def __init__(self):
        ## setup interruption handlers
        ...
        timeoutHandler = TimeoutInterruptionHandler(self)
        HARDWARE.interruptVector.register(TIMEOUT_INTERRUPTION_TYPE, timeoutHandler)

        ## controls the Hardware's I/O Device
        self._ioDeviceController = IoDeviceController(HARDWARE.ioDevice)

        self._scheduler = SchedulerRoundRobin()

        ## Timer configuration
        HARDWARE.timer.quantum = 4 # podriamos hacer que lo configure el
                                   # el SchedulerRoundRobin directamente
```

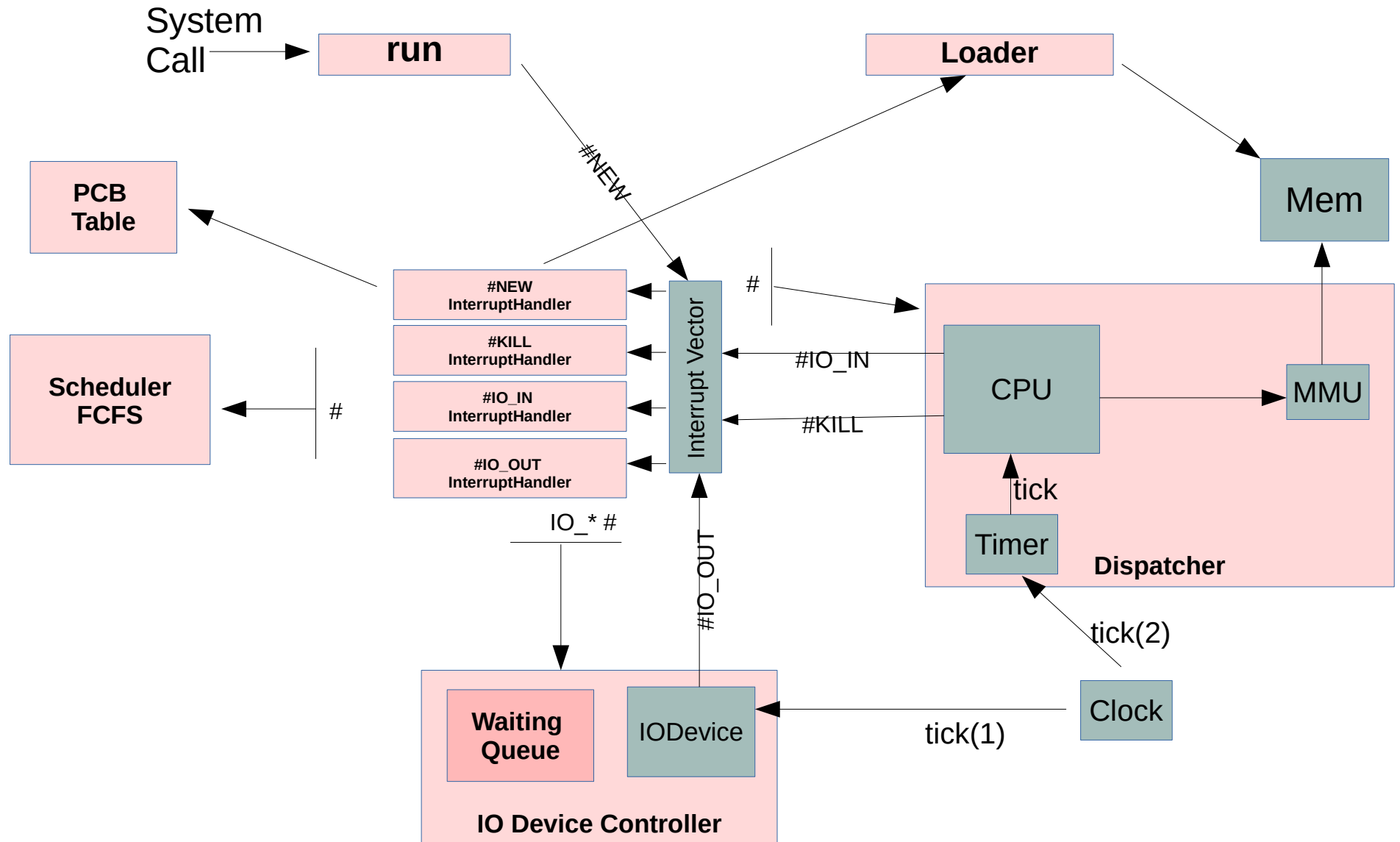
Para los otros Schedulers

- Si no seteamos el quantum del timer, el CPU va a ejecutar todos los tick (Timer desactivado).

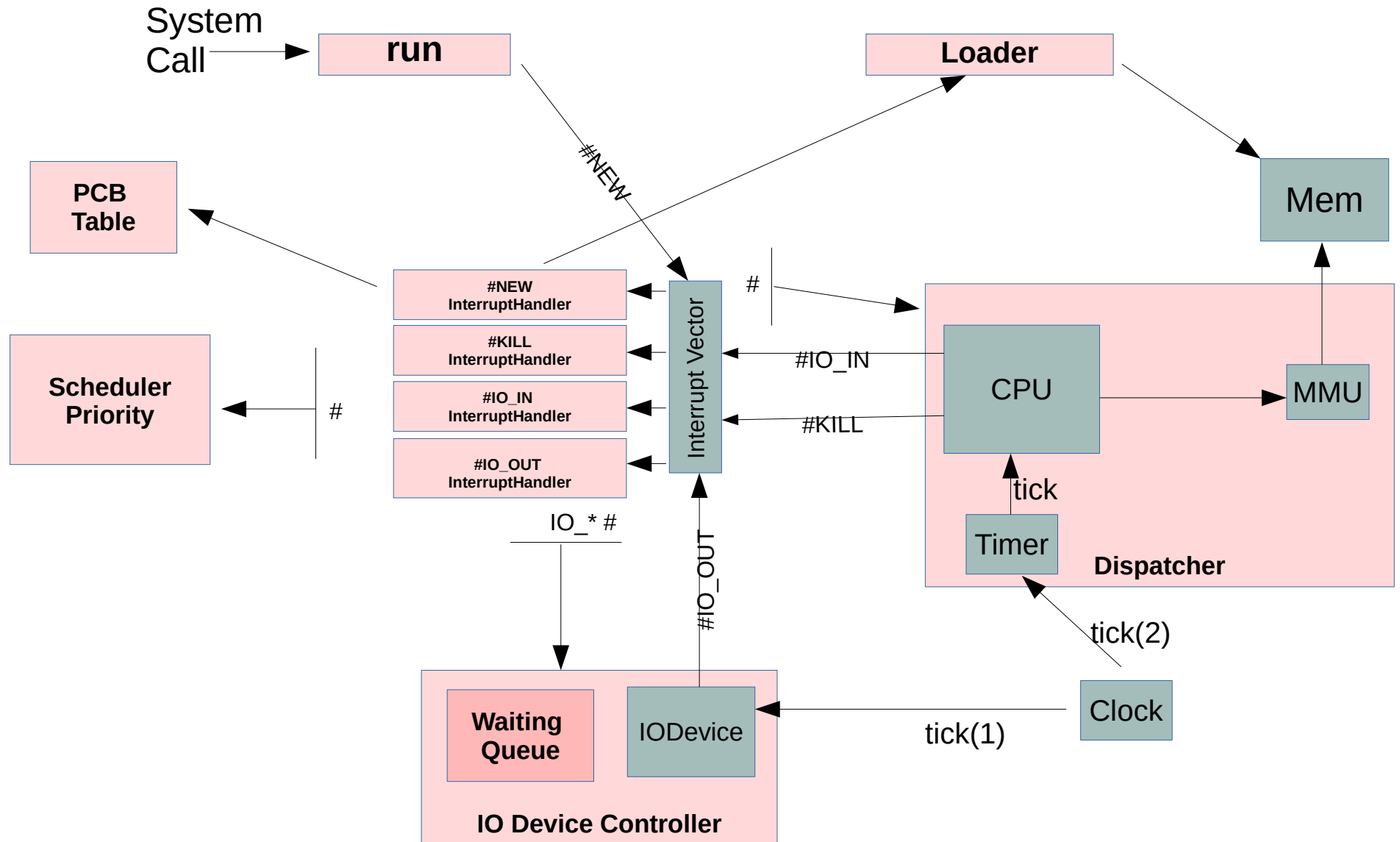
Class Timer

```
def __init__(self, cpu, interruptVector):  
    self._cpu = cpu  
    self._interruptVector = interruptVector  
    self._tickCount = 0    # cantidad de de ciclos “ejecutados” por el proceso actual  
    self._on = False    # por default esta desactivado  
    self._quantum = 0    # por default esta desactivado
```

Practica 4: Scheduler FCFS



Practica 4: Scheduler Prio *



* preemptive / noPreemptive

Hardware Software

Practica 4: Scheduler RR

