Simulador de S.O.

Práctica 5

Asignación Continua

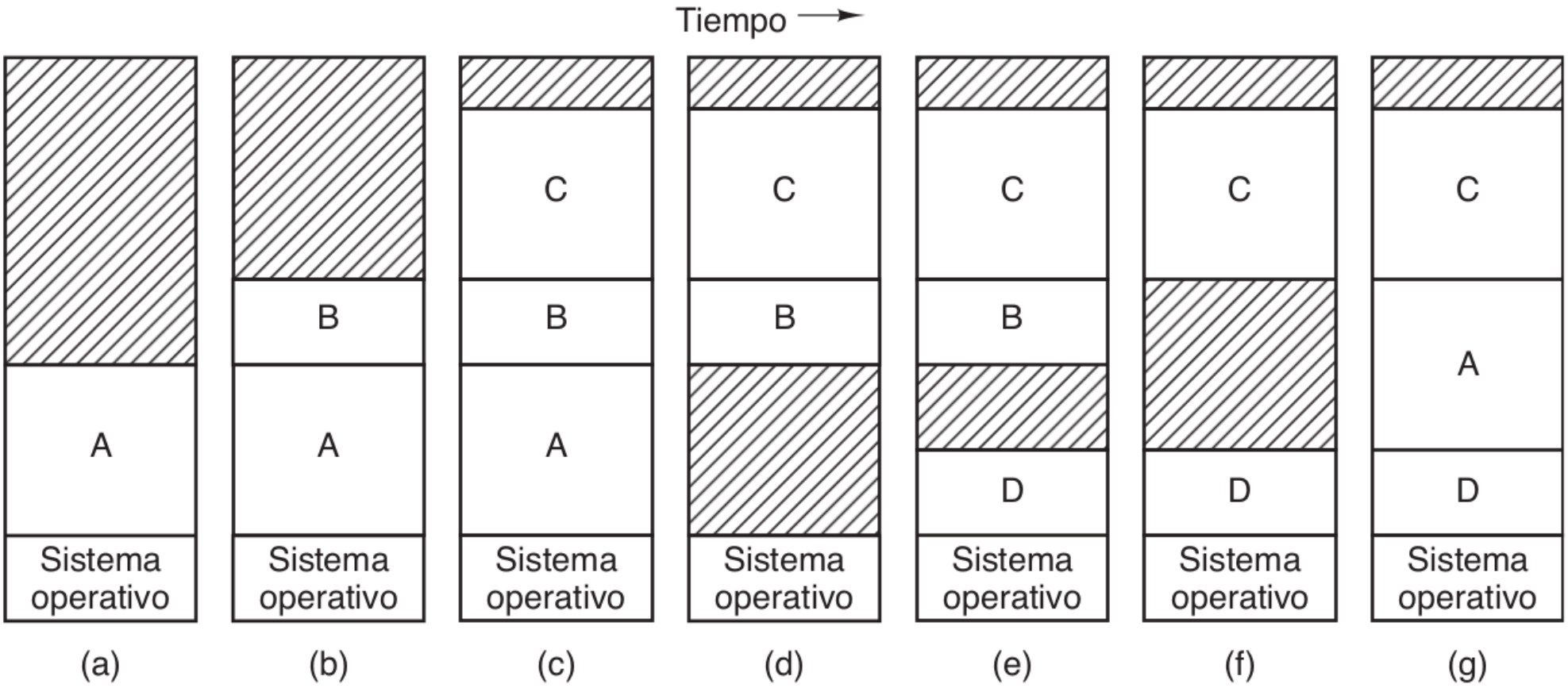
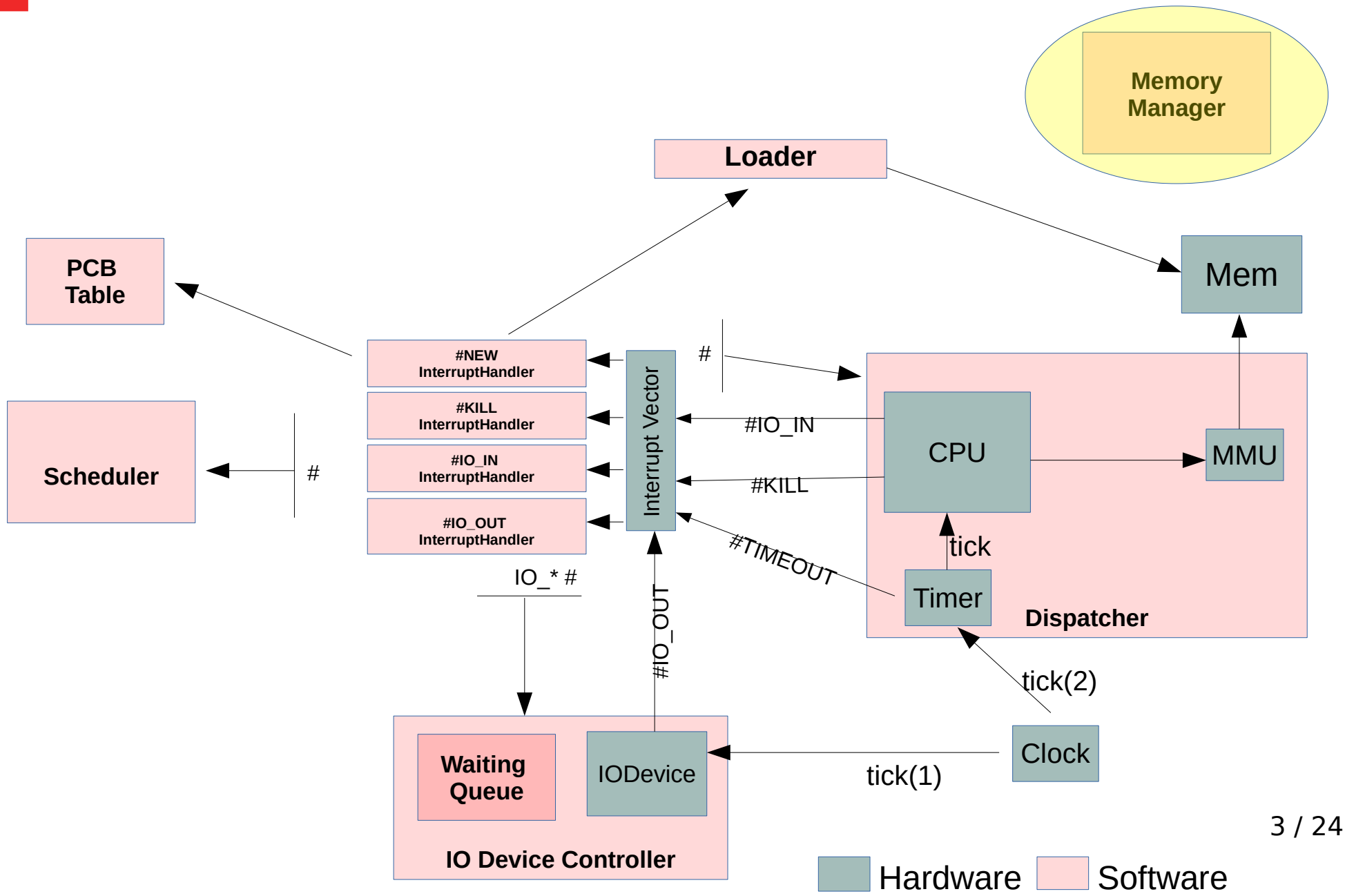
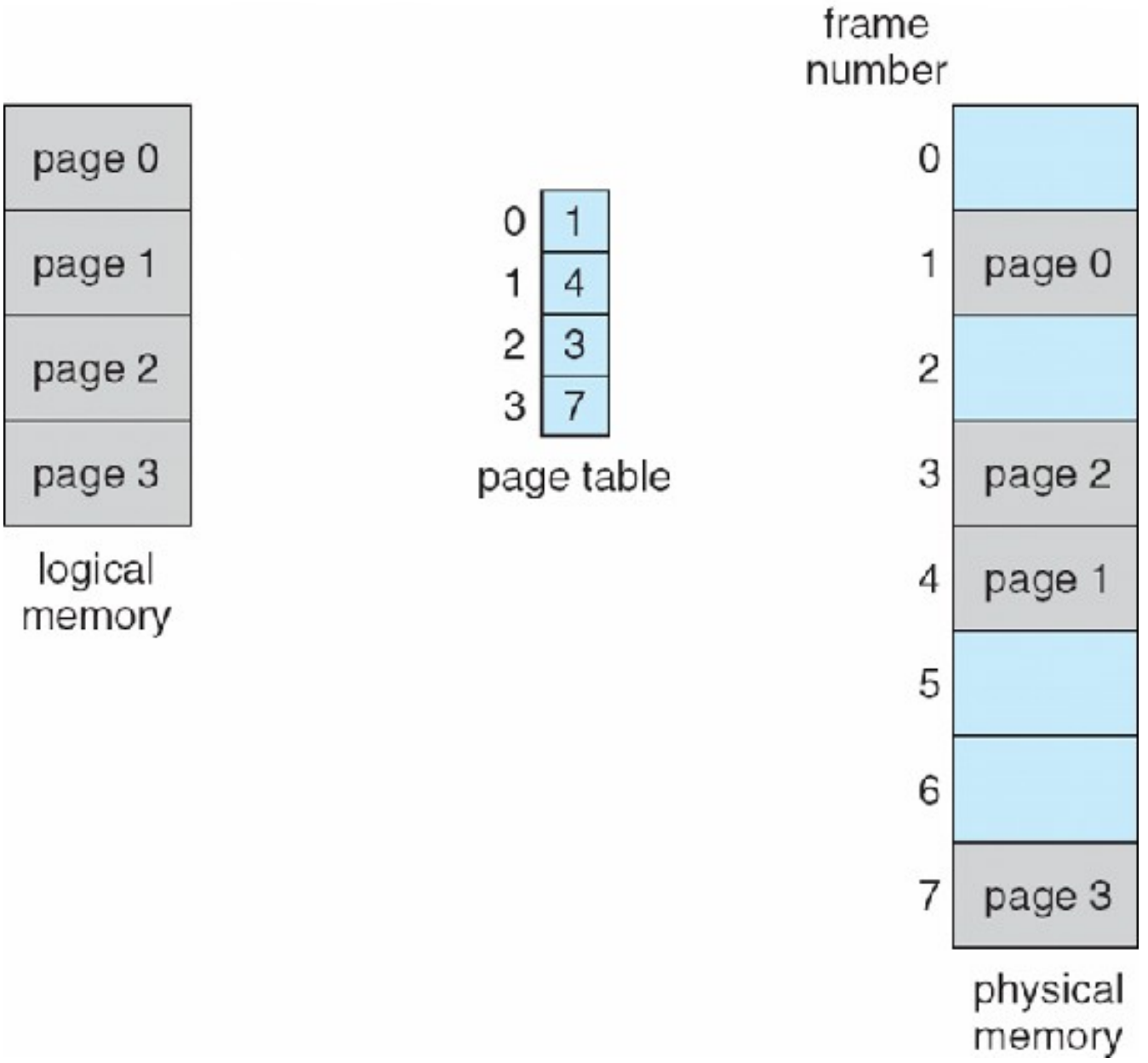


Figura 3-4. La asignación de la memoria cambia a medida que llegan procesos a la memoria y salen de ésta. Las regiones sombreadas son la memoria sin usar.

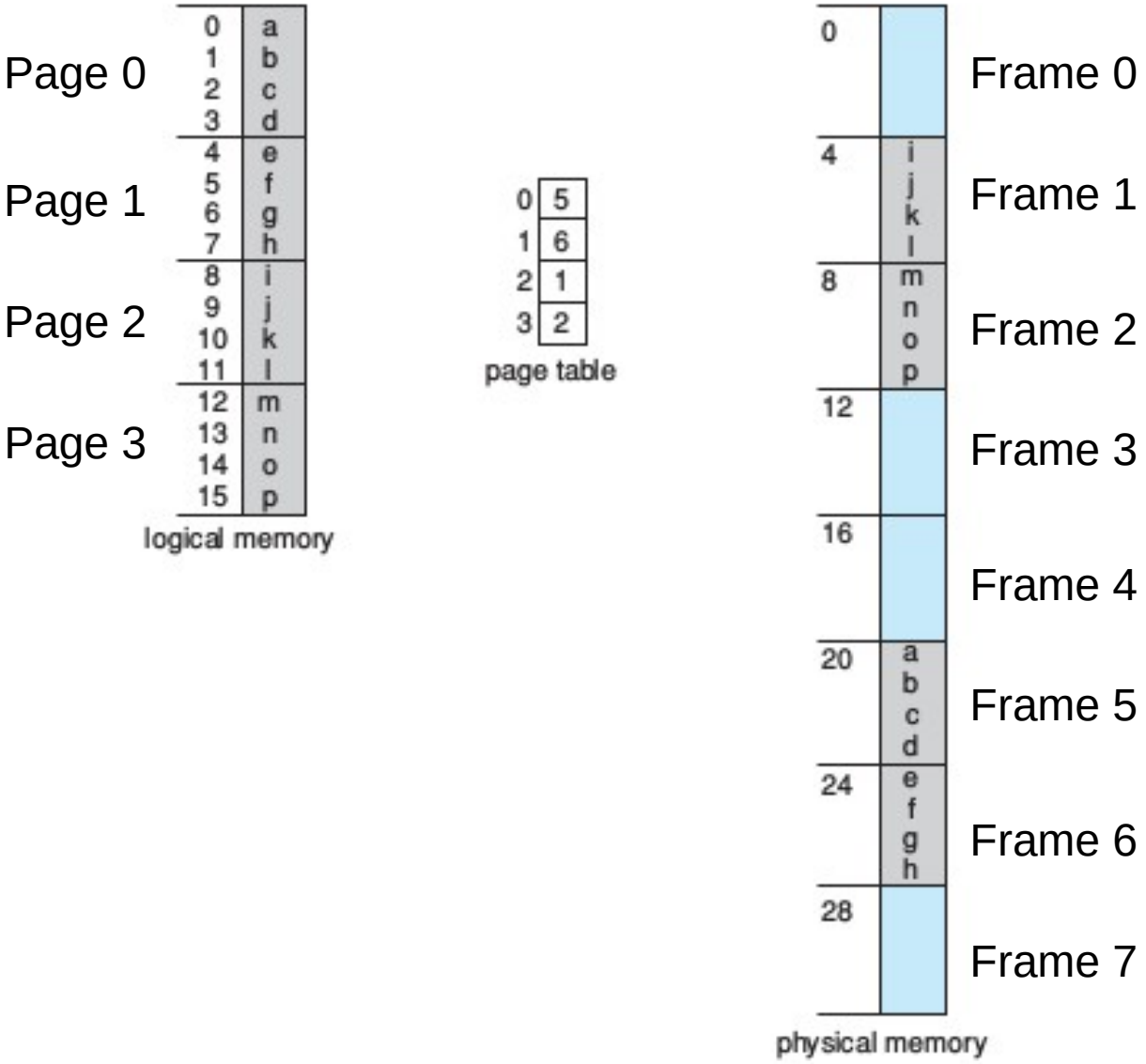
Practica 5: Memory Manager



Memoria Lógica y Física



Paginación de memoria de 32-bytes con “frame size” = 4-bytes

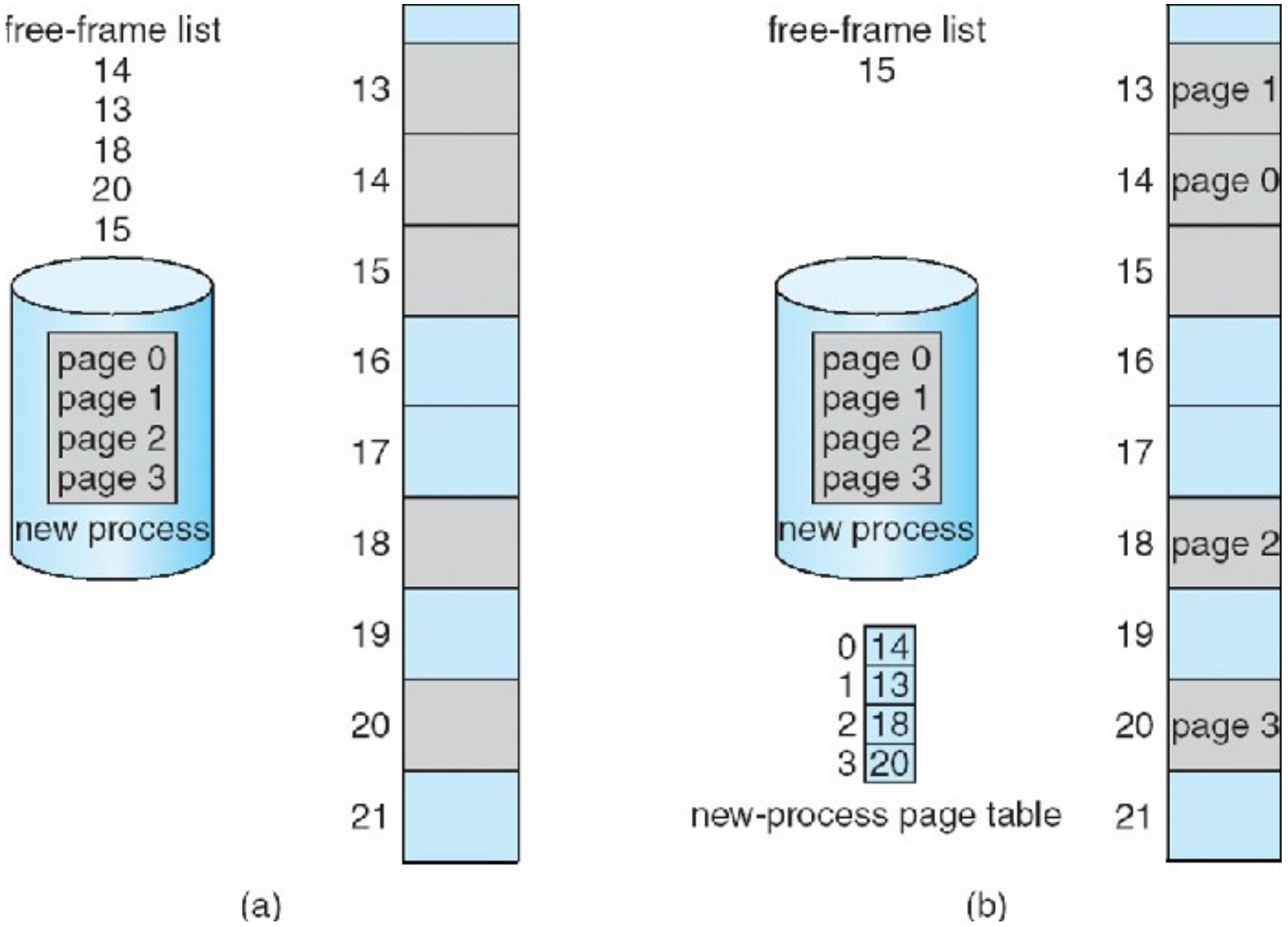


Memory Manager

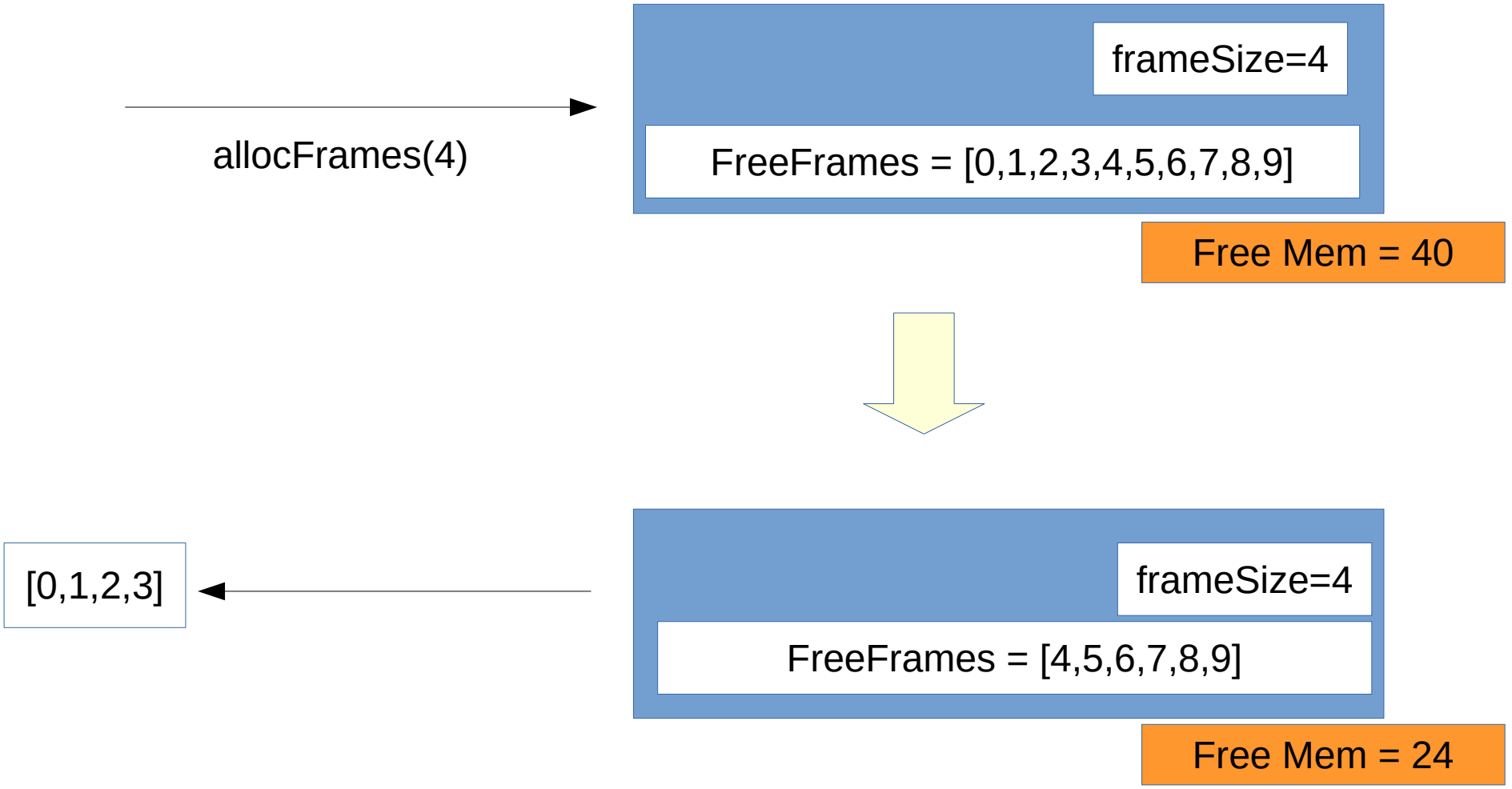
- Administra la memoria “lógica”
- Lleva la “cuenta” de los **frames** libres (y usados) en la memoria física.

frame = marco

Frames Libres



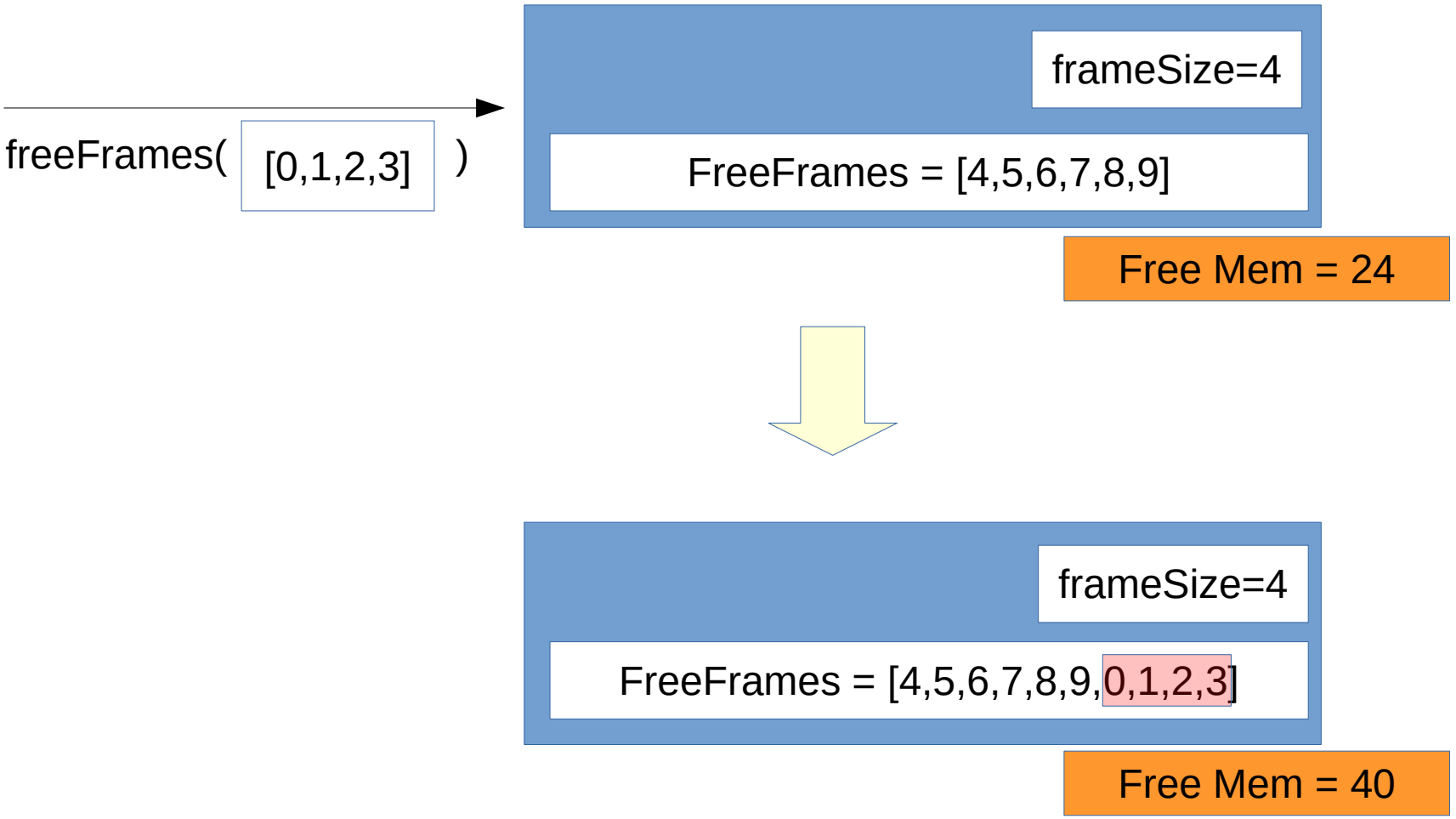
Alocación de Memoria



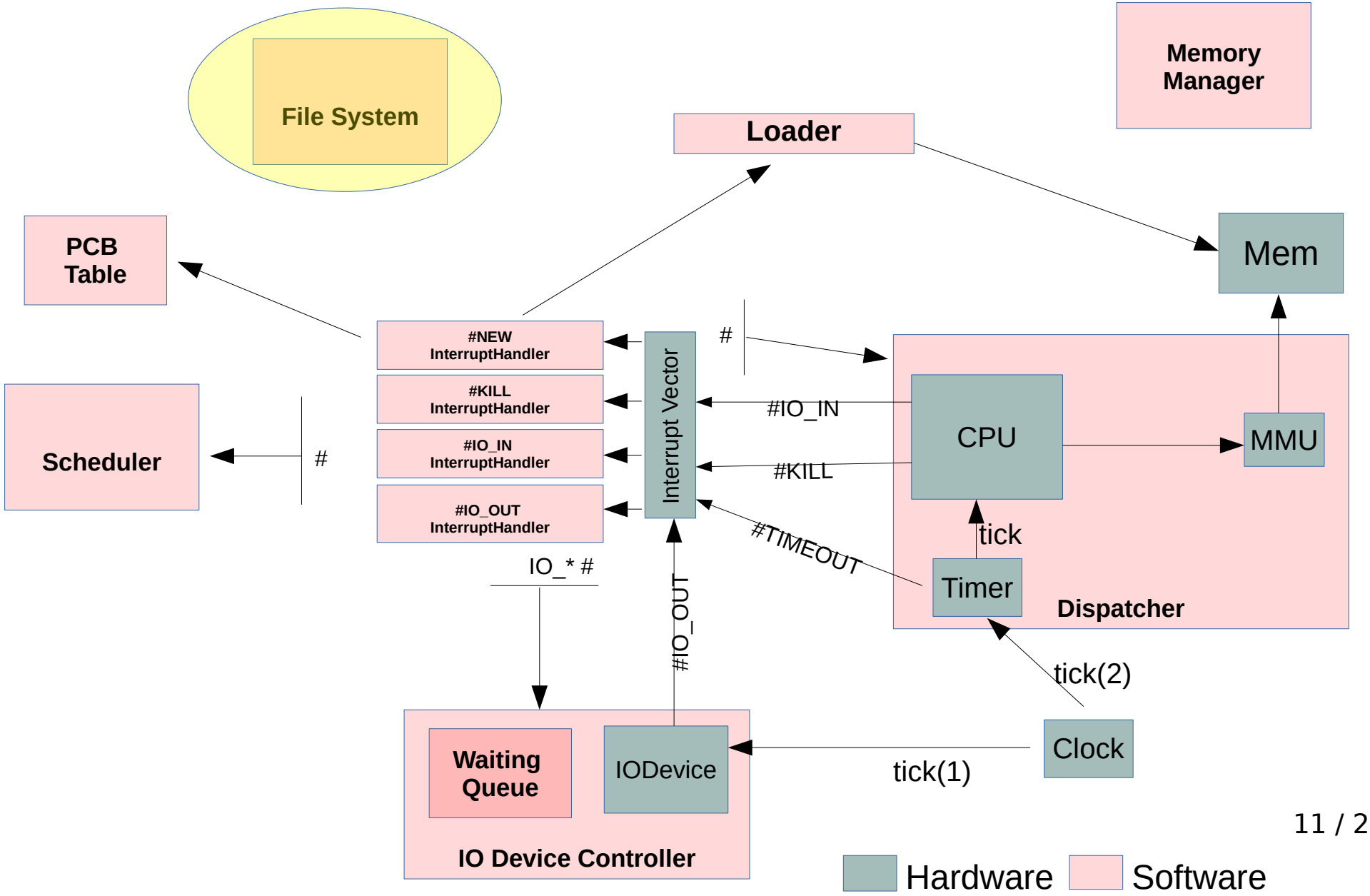
Memory Manager

- Cuando un proceso termine su ejecución, el S.O. debe liberar la memoria usada por éste.

Liberación de Memoria



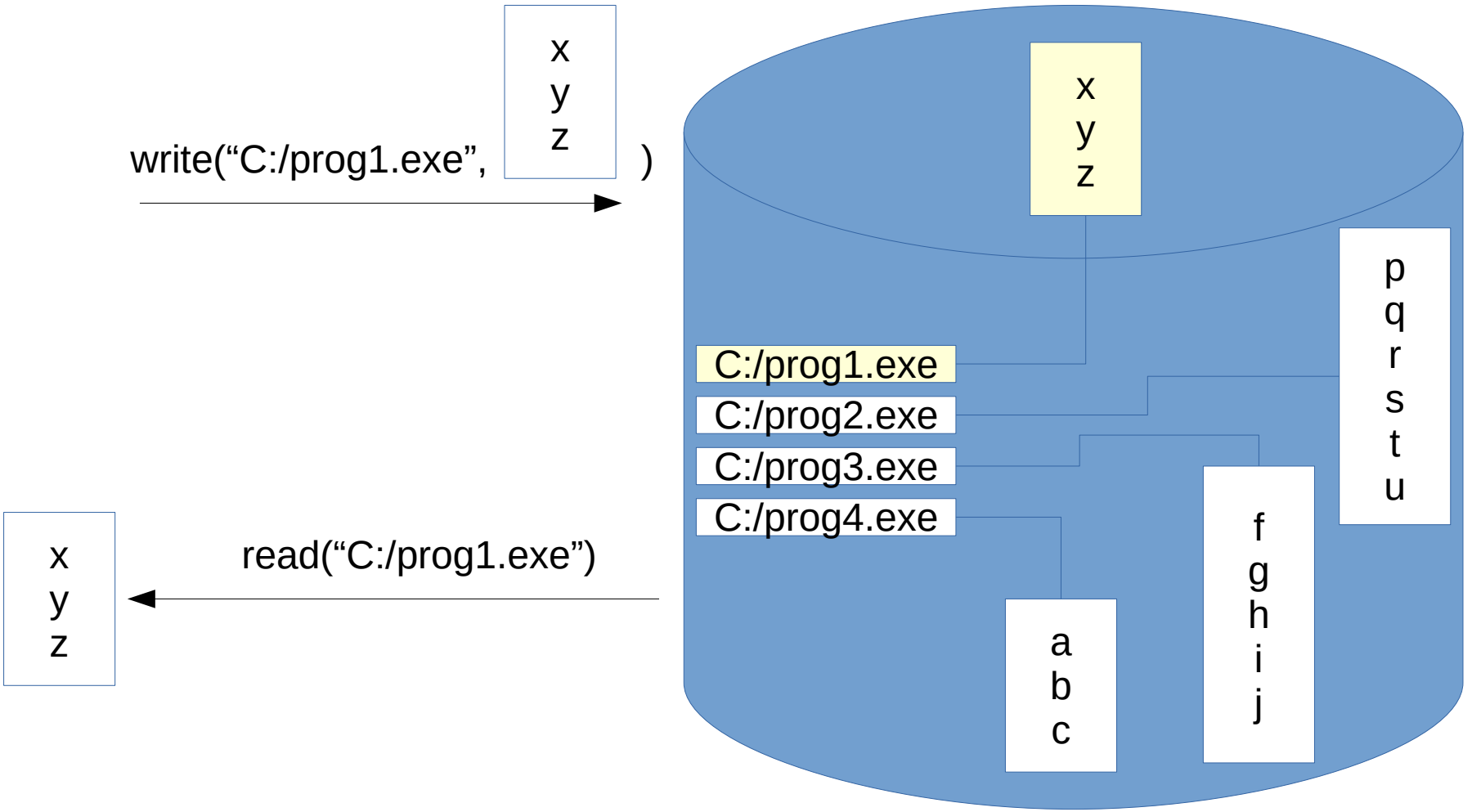
Practica 5: File System



File System

- Almacenamiento permantente donde residen los programas, son accedidos a partir de un “Path”
  - (ej: /home/user/prog1)
- Por el momento no manejamos la estructura real del File System

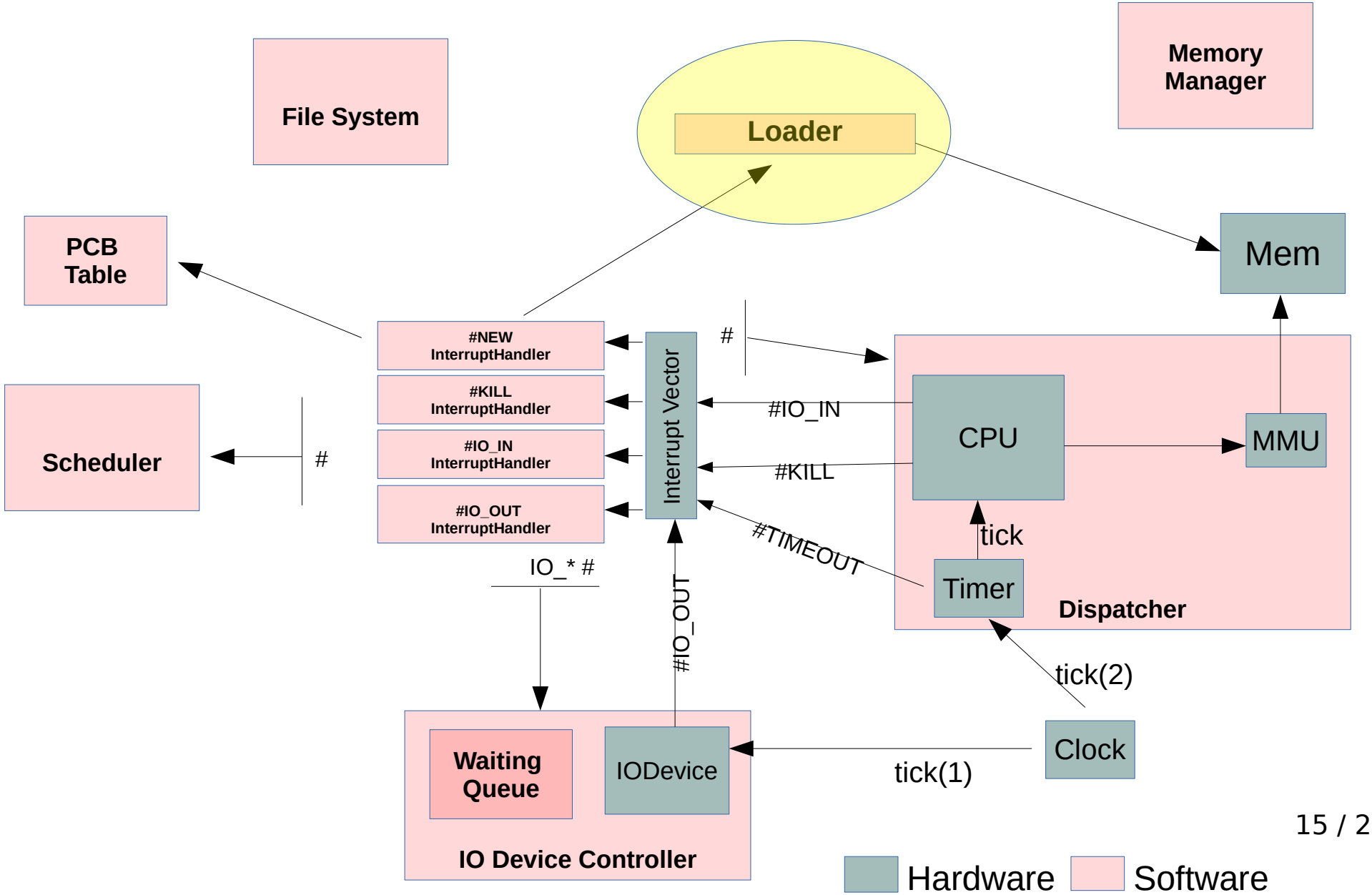
File System


x
y
z
write(“C:/prog1.exe”,
)
x
y
z
C:/prog1.exe
C:/prog2.exe
C:/prog3.exe
C:/prog4.exe
p
q
r
s
t
u
f
g
h
i
j
a
b
c
read(“C:/prog1.exe”)
x
y
z

Main con File System

```
##
##  MAIN
##
if __name__ == '__main__':
....

   # Ahora vamos a guardar los programas en el FileSystem
   ######################
   prg1 = Program([ASM.CPU(2), ASM.IO(), ASM.CPU(3), ASM.IO(), ASM.CPU(2)])
   prg2 = Program([ASM.CPU(7)])
   prg3 = Program([ASM.CPU(4), ASM.IO(), ASM.CPU(1)])

   kernel.fileSystem.write("c:/prg1.exe", prg1)
   kernel.fileSystem.write("c:/prg2.exe", prg2)
   kernel.fileSystem.write("c:/prg3.exe", prg3)

   # ejecutamos los programas a partir de un "path" (con una prioridad x)
   kernel.run("c:/prog1.exe", 0)
   kernel.run("c:/prog2.exe", 2)
   kernel.run("c:/prog3.exe", 1)
```

Practica 5: Loader

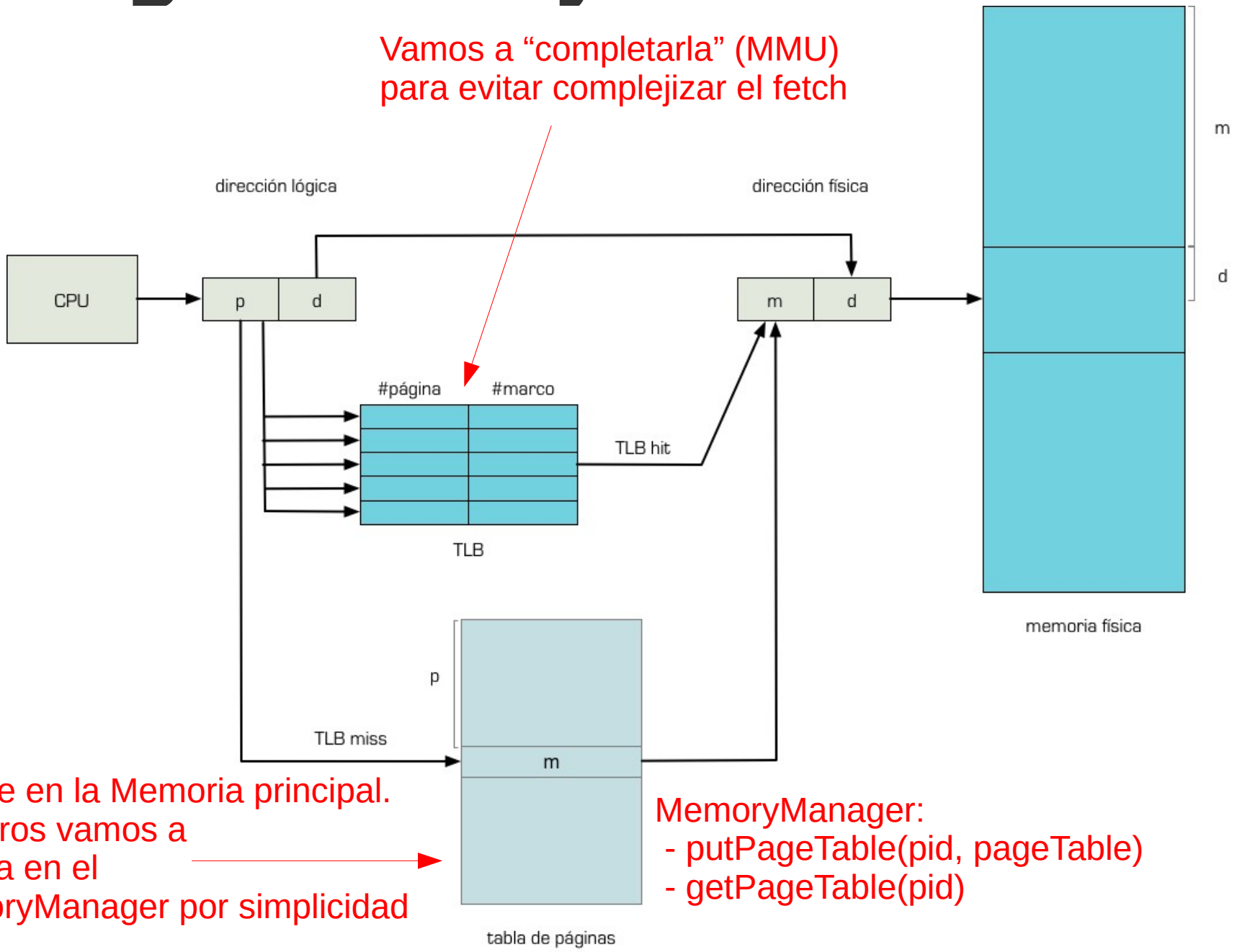


Loader (soft)

- Se encarga de “Cargar” el programa, que esta en el disco (File System), a la memoria.
- Debe alocar las **páginas** del programa en los **frames** “libres” de la memoria.
- Crea el PageTable del proceso y lo guarda en memoria

Page Table y TLB

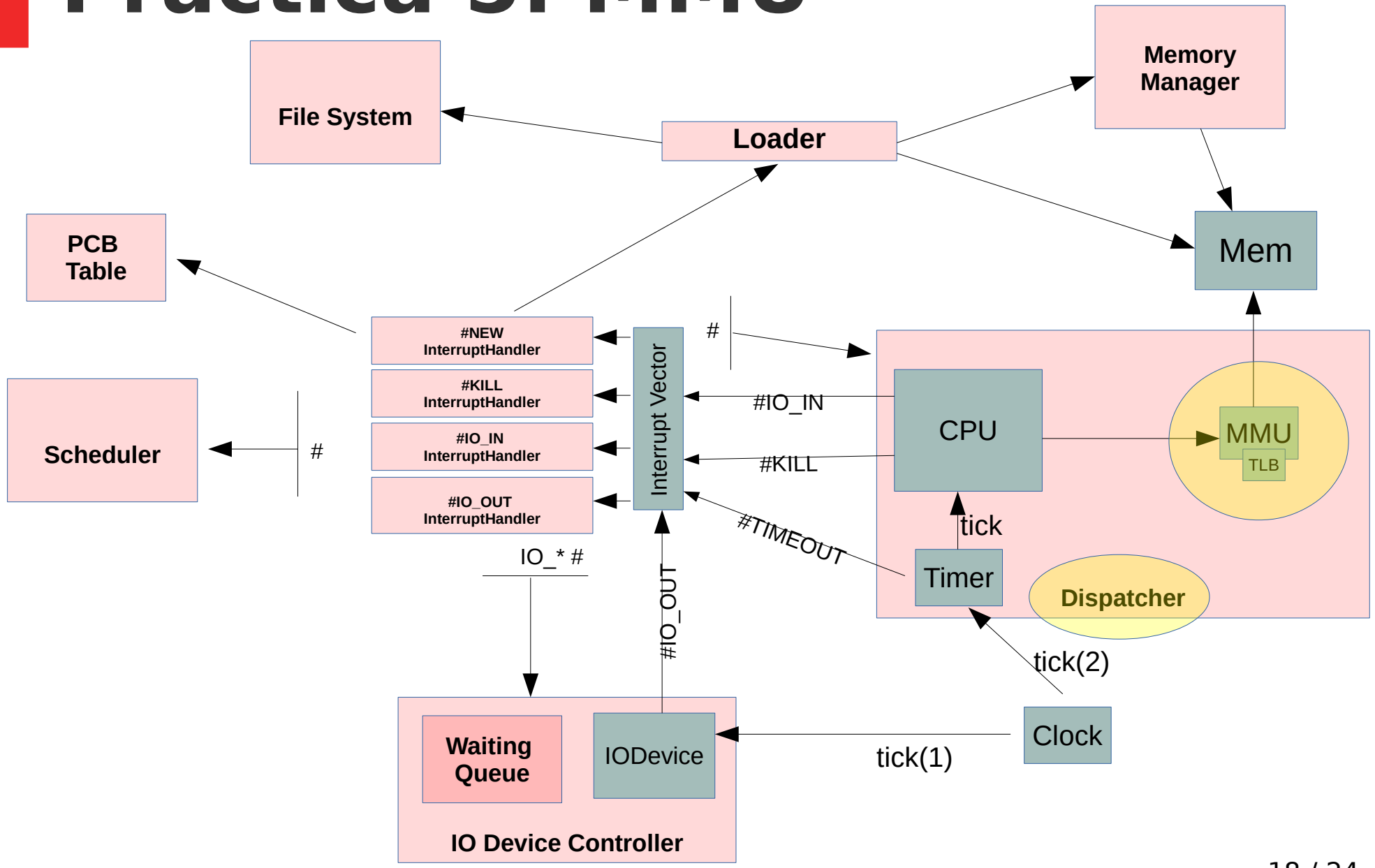
Vamos a “completarla” (MMU)
para evitar complejizar el fetch



Reside en la Memoria principal.
Nosotros vamos a
tenerla en el
MemoryManager por simplicidad

MemoryManager:
- putPageTable(pid, pageTable)
- getPageTable(pid)

Practica 5: MMU



MMU

- El MMU necesita el frameSize

Boot del S.O.

```
class Kernel():

    def __init__(self):
    HARDWARE.mmu.frameSize = 4
```

Tenemos que setearle el frameSize al MMU

Page Table

- Al momento de hacer context switch, debemos cargar la pageTable del proceso al MMU.
- El InterruptorHandler debe sacarlo del MemoryManager para pasarselo al dispatcher

MMU - Dispatcher

Dispatcher

```
def load(self, pcb, pageTableDelPCB)
   ...
  ## al hacer un context switch
   HARDWARE.mmu.resetTLB()
   HARDWARE.mmu.setPageFrame(0, 8)
   HARDWARE.mmu.setPageFrame(1, 3)
   HARDWARE.mmu.setPageFrame(2, 5)
```

Cargamos la PageTable del proceso actual en la TLB del MMU

P	Frame
0	8
1	3
2	5

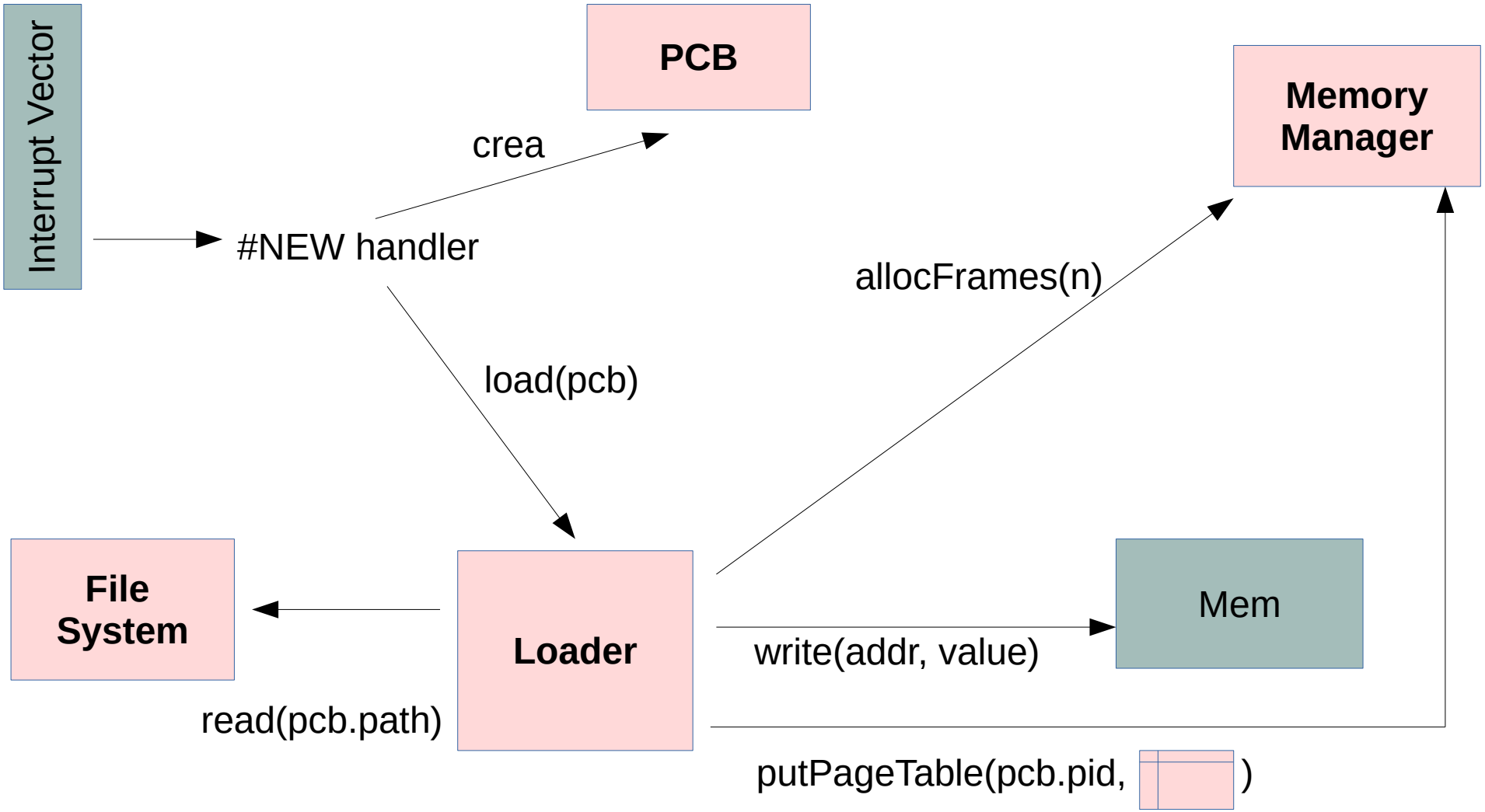
PageTable

Context Switch

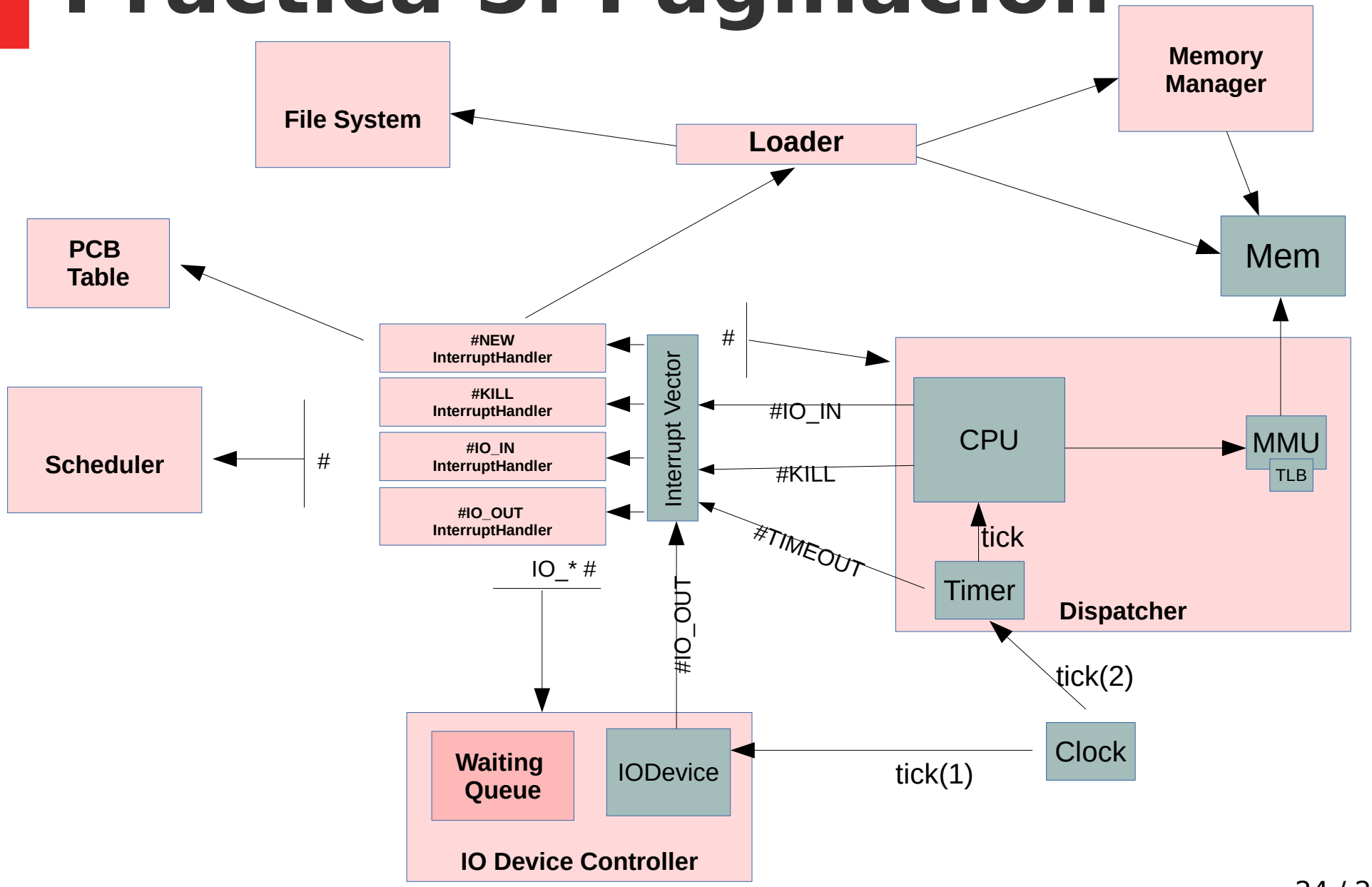
MMU - Fetch

```
def resetTLB(self):
    self._tlb = dict()

def setPageFrame(self, pageId, frameId):
    self._tlb[pageId] = frameId

def fetch(self,  logicalAddress):
...
    # calculamos la pagina y el offset correspondiente a la direccion logica recibida
    pageId = logicalAddress // self._frameSize
    offset = logicalAddress % self._frameSize

    # buscamos la direccion Base del frame donde esta almacenada la pagina
    try:
        frameId = self._tlb[pageId]
    except:
        raise Exception("\n*\n* ERROR \n*\n Error en el MMU\nNo se cargo la pagina  {pageId}")

    ##calculamos la direccion fisica resultante
    frameBaseDir  = self._frameSize * frameId
    physicalAddress = frameBaseDir + offset

    # obtenemos la instrucción alocada en esa direccion
    return self._memory.read(physicalAddress)
```

El #New queda así



Practica 5: Paginación



Hardware Software