# Sistemas Operativos

## Procesos

Marcelo Arroyo y Laura Tardivo

Dpto. de Computación - FCEFQyN - Universidad Nacional de Río Cuarto

2017

# Gestión de procesos/procesador

## Objetivos

- Carga de programas (archivos ejecutables) en memoria y control de su ejecución.
- Multitarea (multiplexado de CPU's):
  1. Paralelizar operaciones de E/S con uso de CPU.
  2. *Timesharing*: Evitar la monopolización del uso de CPUs por algún proceso.
- Protección:
  1. Gestión de modos de ejecución de código:
     1. Procesos en modo *user*.
     2. Código del *kernel* en modo *supervisor*.
  2. Confinamiento: Cada proceso debe acceder sólo a *sus propios* espacios de memoria asignados.

# Inicio del sistema (booting)

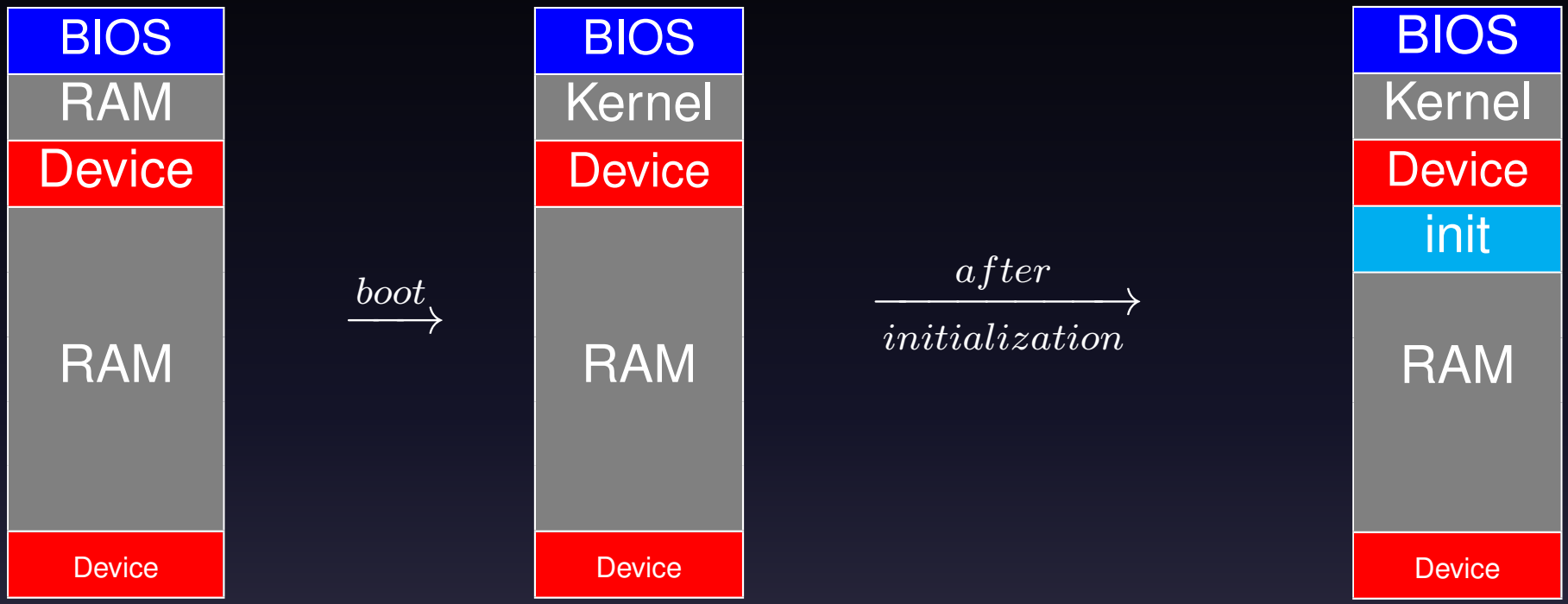
1. Al encenderse, el procesador comienza a ejecutar el código a partir de una dirección pre-establecida de memoria (usualmente cero).
2. Este código (*Basic Input/Output System* en la PC), se encuentra en una memoria `Read Only Memory (ROM)`.
3. Este pequeño programa carga el primer bloque del dispositivo de almacenamiento primario (ej: Hard disk) conocido como *master boot record (MBR)* a un área de memoria pre-establecida (`0x7C00` en la IBM-PC) y salta (`jmp`) a ésta dirección.
4. El programa contenido en el `MBR` *carga* el *kernel* del SO desde el dispositivo de almacenamiento en algún área de memoria.
5. Se transfiere el control `jmp` al punto de entrada del kernel.

# Inicialización

El *kernel* deberá inicializar el sistema:

1. Inicializar estructuras de datos básicas: *basic mem allocator, interrupts vector, . . .*
2. Establecer el modo de ejecución de la CPU (*kernel mode*).
3. Inicializar el subsistema de gestión de la memoria.
4. Inicializar dispositivos básicos.
5. Inicializar el sistema de archivos (*fs*).
6. Iniciar otras CPUs (en sistemas multiprocesadores).
7. Cargar el primer proceso del sistema (`init`).
8. Programar el *timer* y habilitar interrupciones.
9. Invocar el *planificador (scheduler)* de procesos.
    1. Ésta función *selecciona* un proceso para darle el control (iniciar o continuar) con su ejecución.

# Inicialización



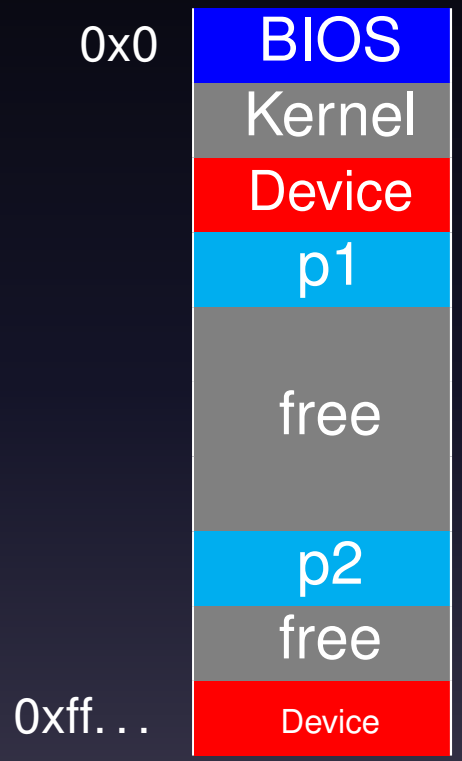

- BIOS : ROM.
- Device : *Memory mapped* device memory/ports.
- init : Initial (user mode) process.
- RAM : Free memory.

# Gestión de memoria

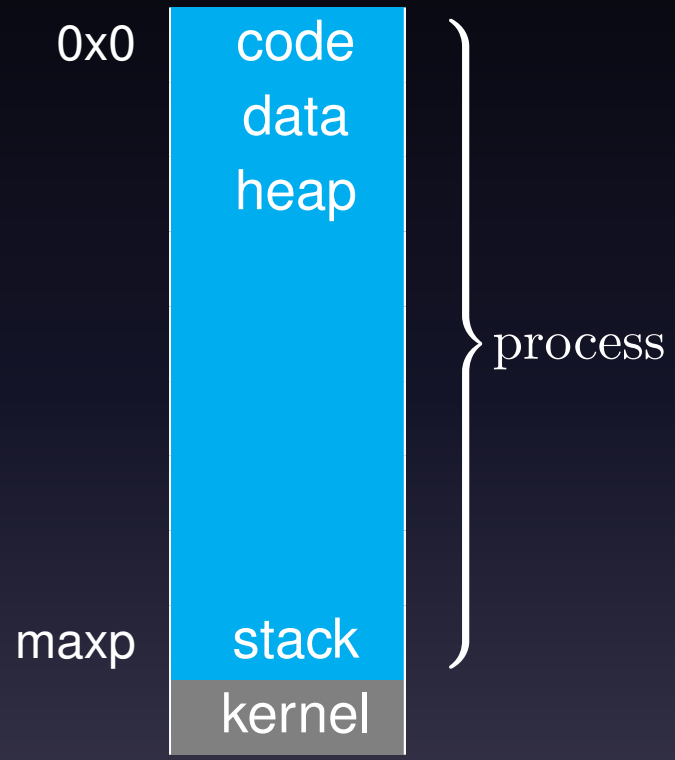
## Direcciones lógicas vs físicas

- Los programas se compilan para funcionar lógicamente a partir de una dirección *lógica* de memoria (por ejemplo, cero), independiente se su dirección de carga real (*física*).
- ¿Cómo pueden funcionar correctamente entonces?
  1. *Código reubicable (Position Independent Code)*: Todas las direcciones que referencia un programa son *desplazamientos* a una dirección base.
  2. Al darle el control a un proceso (*context switch)* el *kernel* configura la CPU para que uno (o más) registros de reubicación (o base) contengan las direcciones base (físicas).
- Ver tutorial sobre compilación y enlazado.

# Memoria: Espacios lógicos vs físicos




Vista física




Vista lógica

# Protección

## Modos de ejecución

- Un proceso de usuario no debería ejecutar instrucciones privilegiadas (le podría quitar el control al SO). Ejemplos:
  - Deshabilitación de interrupciones
  - Acceso a dispositivos de entrada-salida
  - Acceso a instrucciones y registros del sistema
- **x86**: 4 anillos (0..3) pensados para estructurar un SO:
  0. Más privilegiado (kernel)
  1. Device drivers (I/O. . . )
  2. Servicios del sistema (filesystem, gestor de memoria, . . . )
  3. menos privilegiado (user mode)

# Protección

## Modos de ejecución

- Un proceso de usuario no debería ejecutar instrucciones privilegiadas (le podría quitar el control al SO). Ejemplos:
  - Deshabilitación de interrupciones
  - Acceso a dispositivos de entrada-salida
  - Acceso a instrucciones y registros del sistema
- **x86**: 4 anillos (0..3) pensados para estructurar un SO:
  0. Más privilegiado (kernel)
  1. Device drivers (I/O...)
  2. Servicios del sistema (filesystem, gestor de memoria, ...)
  3. menos privilegiado (user mode)

  (Linux, xv6, Windows,... usan sólo los anillos 0 y 3)

# Confinamiento

## Espacios de memoria

- Un proceso debería ejecutarse confinado en su propio espacio de memoria.
- El sistema (hardware+SO) debería prevenir cualquier intento de violación de acceso.

# Confinamiento

## Espacios de memoria

- Un proceso debería ejecutarse confinado en su propio espacio de memoria.
- El sistema (hardware+SO) debería prevenir cualquier intento de violación de acceso.

## Mecanismos (mapas de memoria por proceso)

Por cada proceso se mantiene una lista de bloques:

- **Segmentación**: bloques de tamaño variable (dirección base y límite)

# Confinamiento

## Espacios de memoria

- Un proceso debería ejecutarse confinado en su propio espacio de memoria.
- El sistema (hardware+SO) debería prevenir cualquier intento de violación de acceso.

## Mecanismos (mapas de memoria por proceso)

Por cada proceso se mantiene una lista de bloques:

- **Segmentación**: bloques de tamaño variable (dirección base y límite)
  Mapping lógico con segmentos de programa (código, datos, stack, heap)

# Confinamiento

## Espacios de memoria

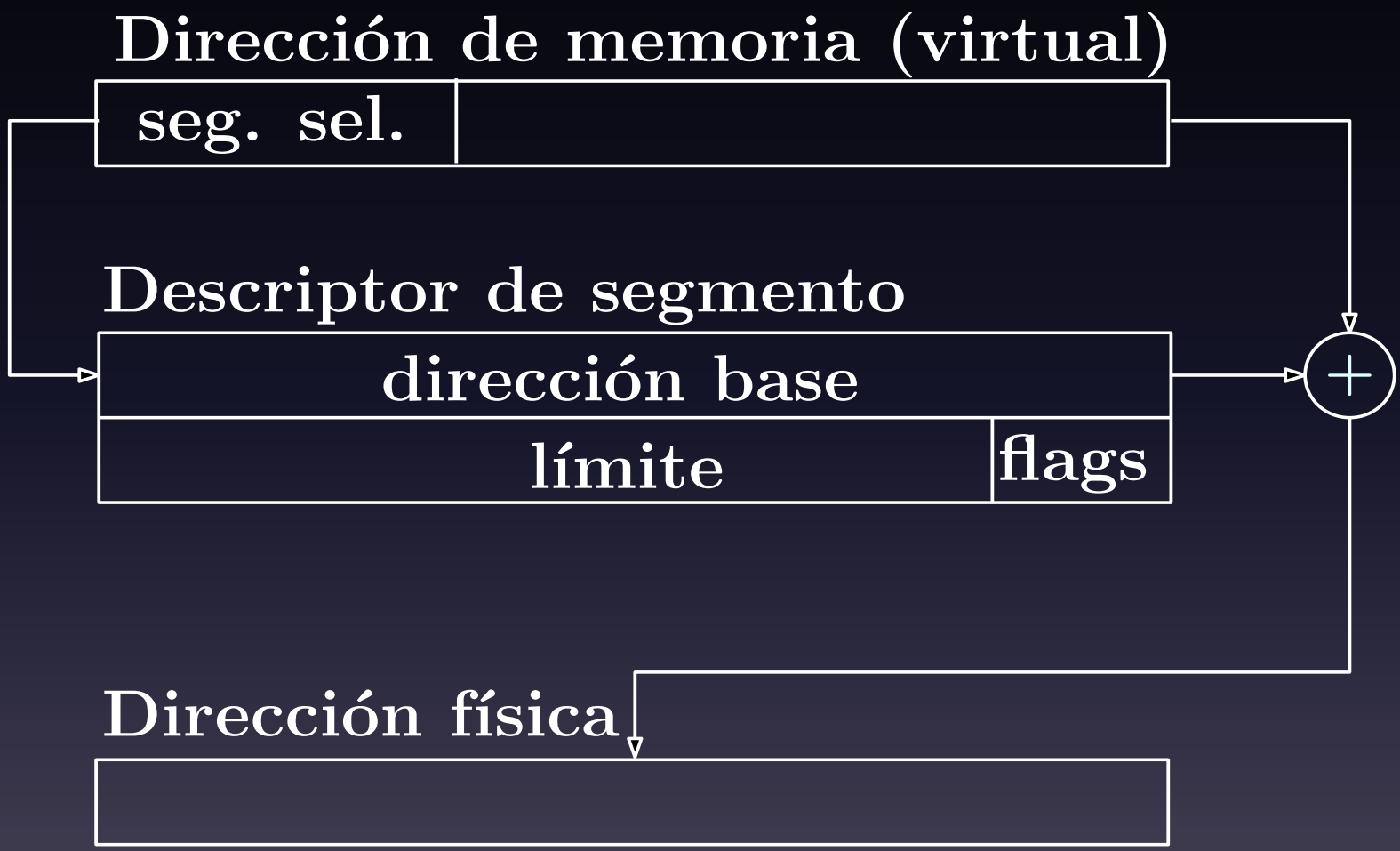
- Un proceso debería ejecutarse confinado en su propio espacio de memoria.
- El sistema (hardware+SO) debería prevenir cualquier intento de violación de acceso.

## Mecanismos (mapas de memoria por proceso)

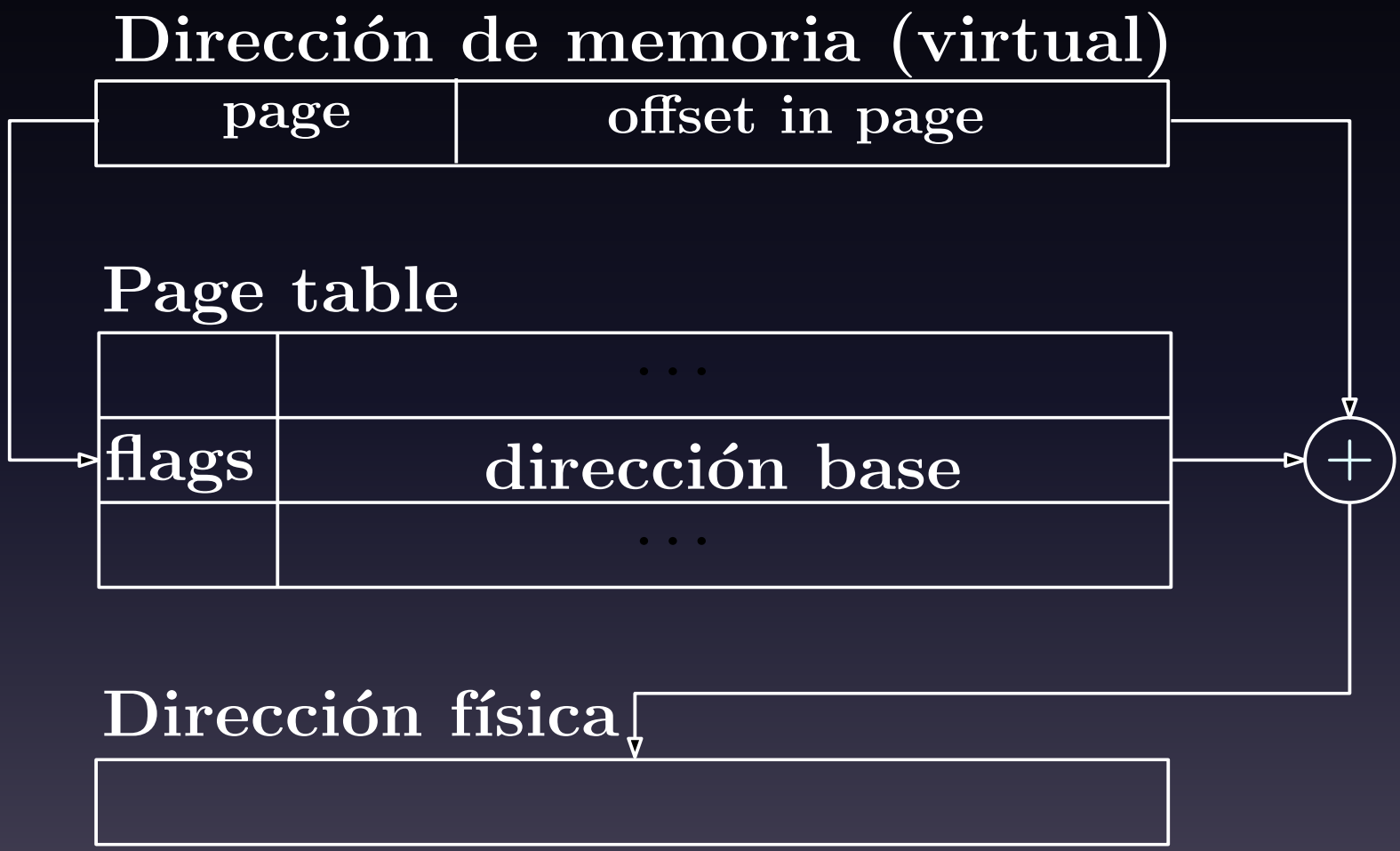
Por cada proceso se mantiene una lista de bloques:

- **Segmentación**: bloques de tamaño variable (dirección base y límite)
  Mapping lógico con segmentos de programa (código, datos, stack, heap)
- **Paginado**: bloques de tamaño fijo (ej: 4KB en x86)
  Ventaja: permite la implementación más eficiente de *Memoria virtual*

# Segmentación



# Paginado


Dirección de memoria (virtual)
page
offset in page
Page table
flags
dirección base
Dirección física

# Procesos

## Proceso

Un proceso es una instancia de un programa en ejecución.

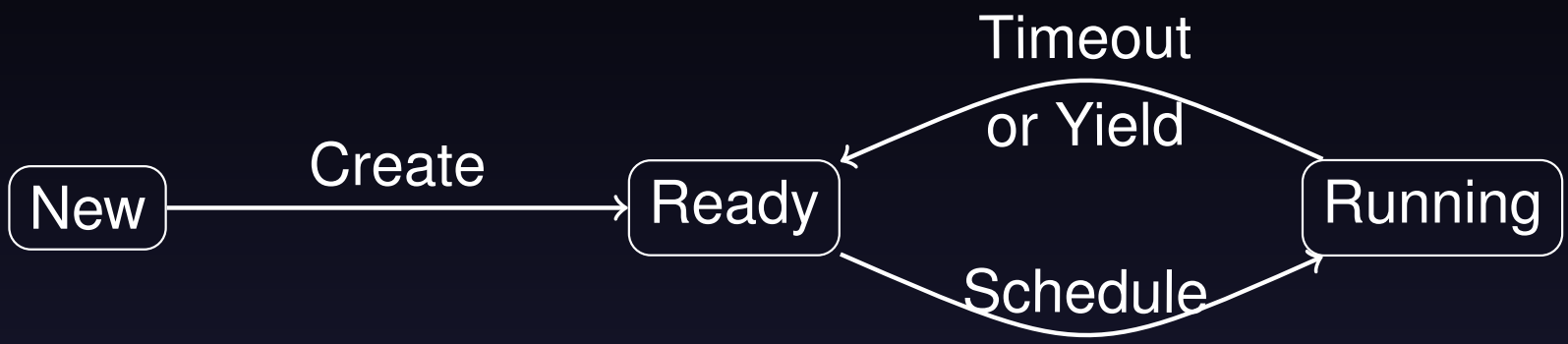
## Descriptor (Process Control Block o PCB)

- Identificador único (pid)
- Identificador del proceso padre
- Estado: running, ready, blocked (sleeping), terminated, ...
- Mapa de memoria:
  - segmentos y/o páginas
  - Código, datos, stack, heap
  - stack en modo kernel, ...
- Contabilidad: tiempos de uso de cpu, E/S, ...
- Recursos: files, ...

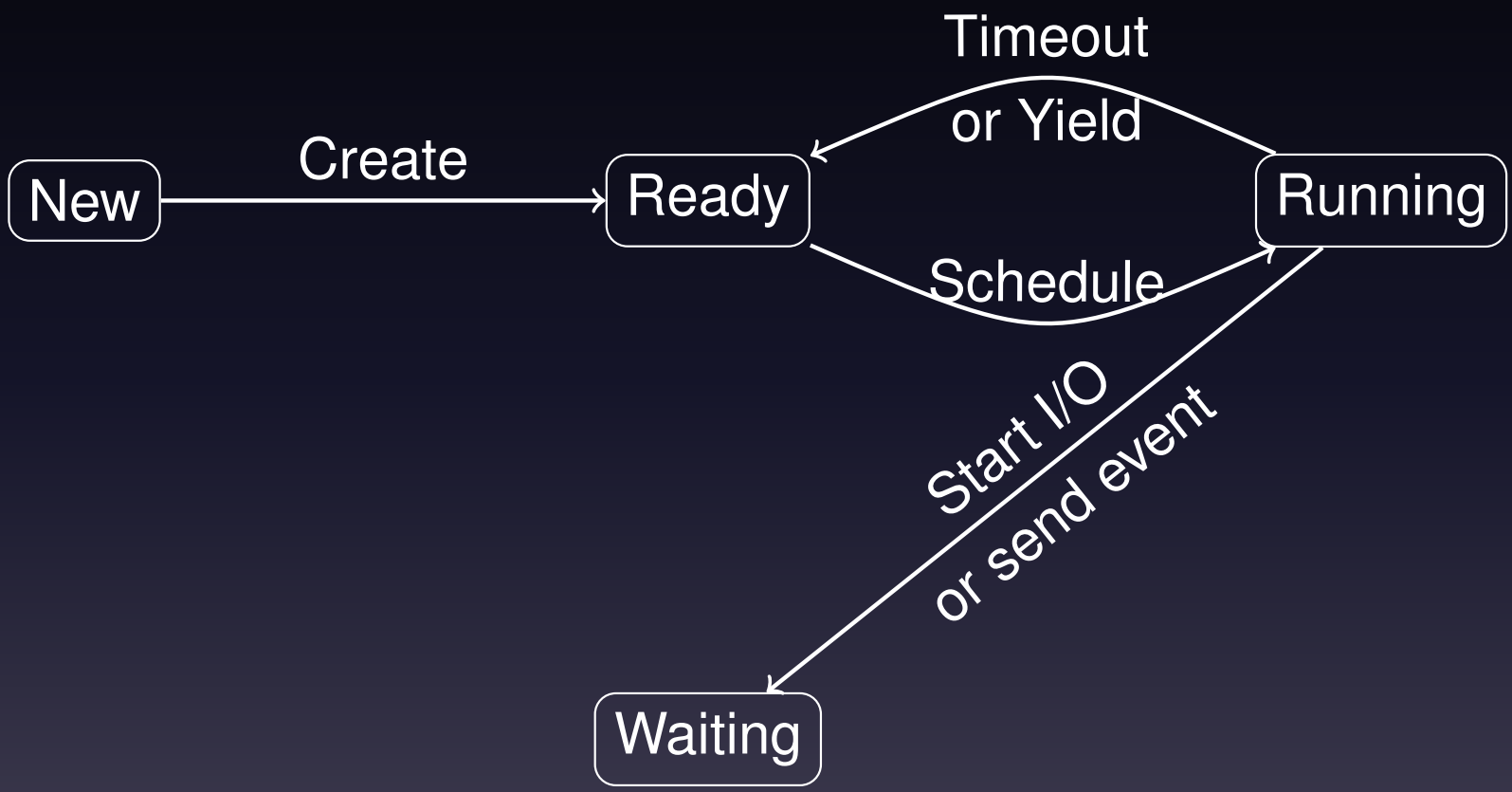
# Estados de un proceso

# Estados de un proceso

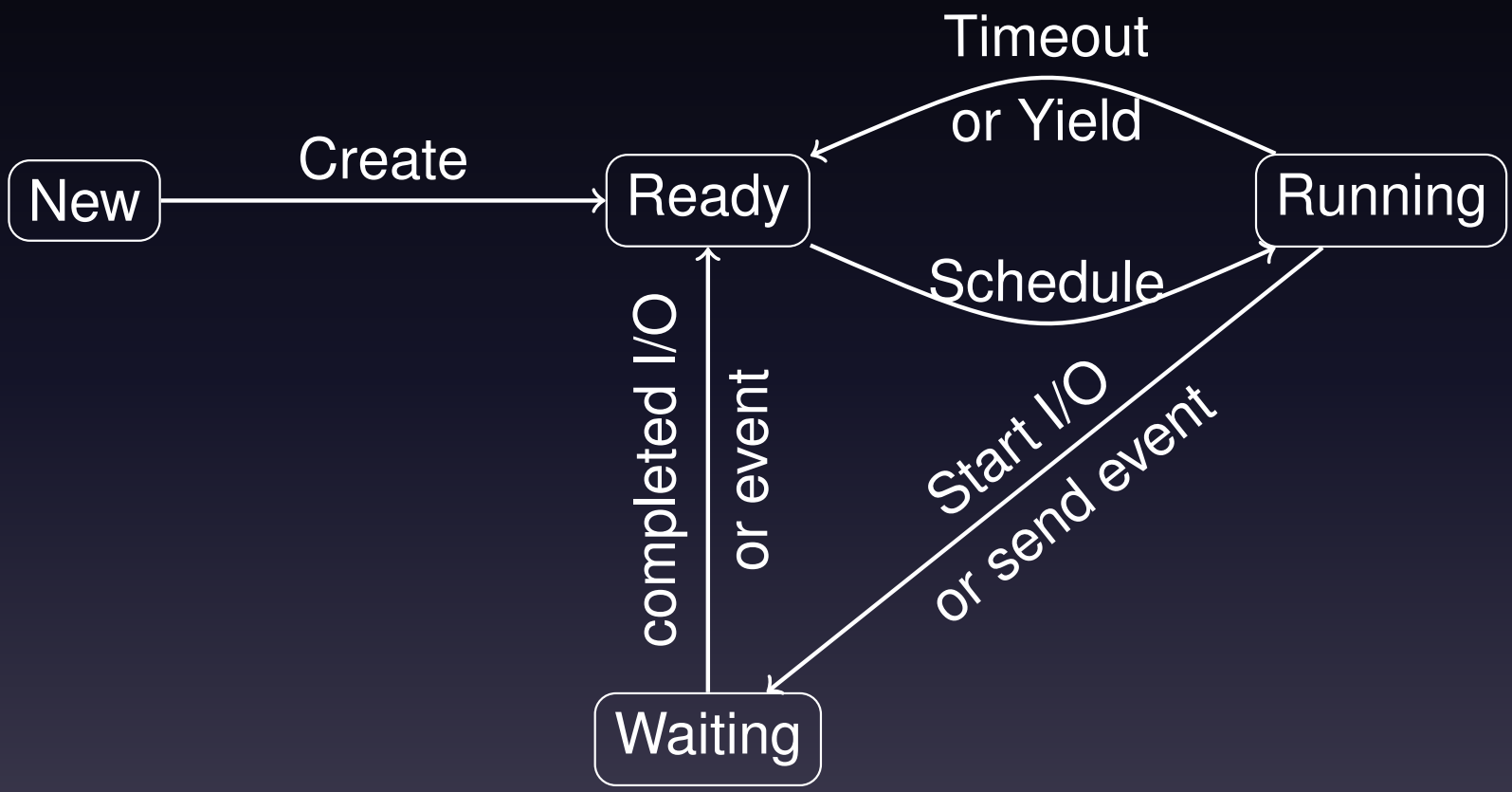
# Estados de un proceso



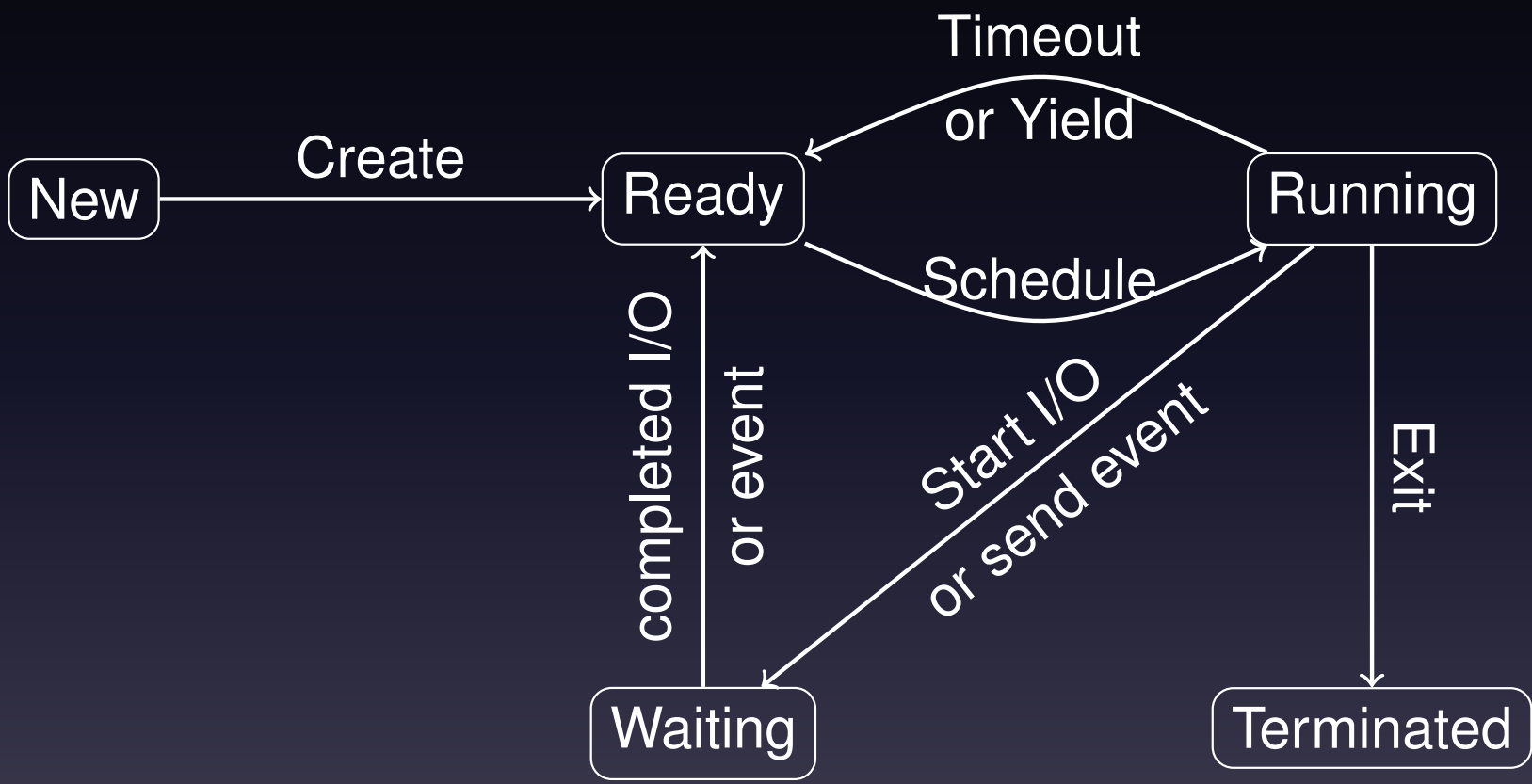
# Estados de un proceso



# Estados de un proceso



# Estados de un proceso



# Control de procesos

## Control de la CPU

- El SO toma el control cuando ocurre:
  1. Una interrupción (*IRQ*) de algún dispositivo (timer, disco, teclado, . . . )
     Esto hace que tome el control en forma asincrónica.
  2. Un proceso en ejecución ejecuta una *trampa (trap)*[a].
- En un sistema de *timesharing* se programa el *timer* para que genere interrupciones cada cierto intervalo de tiempo (*ms*).
- Cuando el kernel toma el control puede realizar un *cambio de contexto (context switch)*: Dar el control de ejecución a otro proceso.
- De esta forma, los procesos se ejecutan como si fuesen *corrutinas*.

[a] También llamadas interrupciones por software (*SWI*).

# Arquitectura x86(IA32)

Reg. de prop. generales (32 bits)

| |
|---|
| **E A X** |
| **E B X** |
| **E C X** |
| **E D X** |
| **E S P (Stack pointer)** |
| **E B P (Base pointer)** |
| **E S I** |
| **E D I** |

Selectores de segmentos (16 bits)

| |
|---|
| **CS (code segment)** |
| **DS (data segment)** |
| **SS (stack segment)** |
| **ES (extra segment)** |
| **FS** |
| **GS** |

Otros registros (32 bits)

| |
|---|
| EIP (Instruction pointer) |
| EFLAGS (status) |

# x86(IA32) Segmentación

## Tablas de descriptores de segmentos

Un selector de segmento es un índice a un elemento de una tabla de descriptores de segmentos.
Hay una tabla global (*GDT*, apuntada por el reg. GDTR) y puede haber una tabla local (*LDT*) por cada proceso (apuntada por el reg. LDTR).

# x86(IA32) Segmentación

## Tablas de descriptores de segmentos

Un selector de segmento es un índice a un elemento de una tabla de descriptores de segmentos.
Hay una tabla global (*GDT*, apuntada por el reg. GDTR) y puede haber una tabla local (*LDT*) por cada proceso (apuntada por el reg. LDTR).

Segment Selectors

| | *index* | T | *CPL* |
|---|---|---|---|
| CS | 0x00 | 1 | 11 |
| DS | 0x01 | 1 | 11 |
| SS | 0x02 | 1 | 11 |
| ... | | | |

Segment Descriptor Table

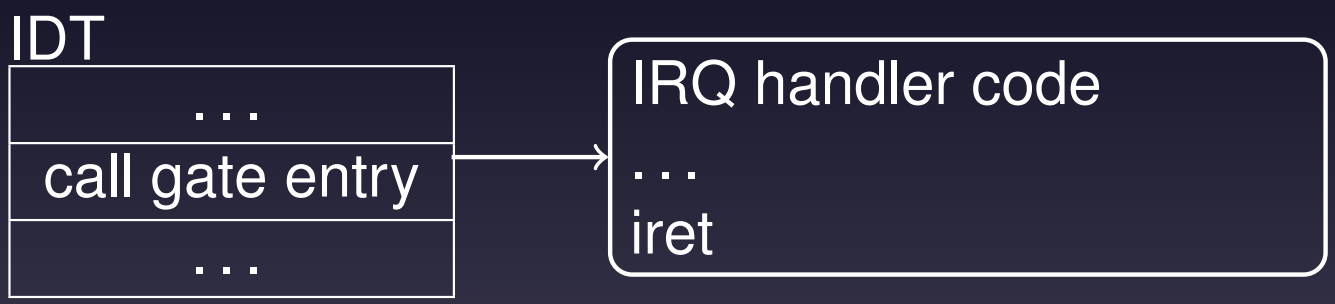
| | *base address* | *limit* | *flags* |
|---|---|---|---|
| CS | *base address* | *limit* | *flags* |
| DS | *base address* | *limit* | *flags* |
| SS | ... | ... | ... |

Ejemplo de una tabla (LDT) de descriptores de segmentos.

# x86(IA32) Interrupciones

## Tabla de descriptores de iterrupciones (IDT)

La IDT contiene descriptores de manejadores de interrupciones (call gates).
La *IDT* está apuntada por el registro IDTR.



# x86(IA32) Niveles de privilegio

## Cambio de nivel de privilegio

Una invocación a un procedimiento por medio de una *call gate* se puede realizar por:

- Una interrupción de hardware
- Trap (el proceso ejecuta *int n*)

El nuevo nivel de privilegio depende del campo DPL del registro `cs` del procedimiento invocado (que está en la entrada correspondiente de la IDT).

Process (PL=3)...
...
interrupt (ej: int n)
...
...

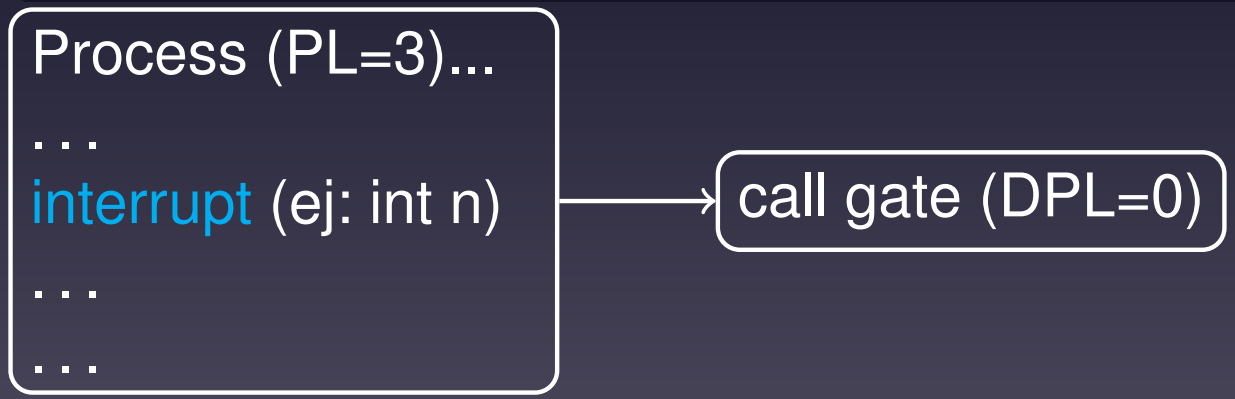
# x86(IA32) Niveles de privilegio

## Cambio de nivel de privilegio

Una invocación a un procedimiento por medio de una *call gate* se puede realizar por:

- Una interrupción de hardware
- Trap (el proceso ejecuta *int n*)

El nuevo nivel de privilegio depende del campo DPL del registro `cs` del procedimiento invocado (que está en la entrada correspondiente de la IDT).



# x86(IA32) Niveles de privilegio

## Cambio de nivel de privilegio

Una invocación a un procedimiento por medio de una *call gate* se puede realizar por:

- Una interrupción de hardware
- Trap (el proceso ejecuta *int n*)

El nuevo nivel de privilegio depende del campo DPL del registro `cs` del procedimiento invocado (que está en la entrada correspondiente de la IDT).

# x86(IA32) Task State Segment

## Soporte para task switching

Tabla de *Task States* (uno por proceso). El registro TR apunta a la TS corriente. En un cambio de privilegio (ej: interrupción), la CPU salva el estado en el TS y carga el nuevo automáticamente.
Una Task State contiene:

- Espacio para salvar el contenido de los registros.
- Permisos de acceso a puertos de E/S.
- Stack pointers para los niveles 0,1 y 2.
- Link al TSS previo (para iret).

El mecanismo no soporta reentrancia.