



# SISTEMAS OPERATIVOS

*Grado en Desarrollo de Videojuegos*  
*Universidad Complutense de Madrid*

---

## TEMA 3. Procesos

# Tema 3. Procesos

---

## 3.1 Introducción

- Procesos y Programas
- Ciclo de vida de un proceso

## 3.2 Representación en el Sistema Operativo

- Bloque de Control de Procesos (PCB)
- El sistema de ficheros /proc

## 3.3 Control de procesos

- Creación y terminación
- Sincronización y códigos de salida
- Señales

## 3.5 Grupos de Procesos

- El proceso init
- Sesiones y grupos
- Ejecución en primer y segundo plano

# Tema 3. Procesos

---

## 3.6 Planificación

- Multiprogramación
- Planificador: Niveles de planificación, Activación y objetivos.
- Modelos de Planificación
- Algoritmos no-expropiativos
  - First come first serve
  - Shortest job first
- Algoritmos expropiativos
  - Round Robin
  - Shortest Remaining First
  - Colas Multinivel con Retroalimentación
- Planificador de Linux



# SISTEMAS OPERATIVOS

*Grado en Desarrollo de Videojuegos*  
*Universidad Complutense de Madrid*

---

## TEMA 3.1 Introducción

# Introducción. Programas

Un **programa** es un conjunto de instrucciones máquina y datos, almacenados en una **imagen ejecutable en disco** (entidad pasiva)

*ELF: Executable & Linking Format*

Cabecera ELF
Tabla de Programa
Otra Información
Segmento de Texto
.text
Segmento de Datos
.data
.bss
Otra Información

Información para cargar las diferentes partes (secciones) del programa en memoria.

Tipo	Offset	Tamaño	Dir. Virtual	Tamaño	Flags
LOAD	0x02dd0	0x0258	0x03dd0	0x0268	R W -
Secciones: .data, .bss, ...					

Algunas secciones importantes de un ejecutable:

**Sección**    **Propósito**

.text	Instrucciones
.rodata	Constantes (p.ej. cadenas literales)
.data	Variables globales o static, inicializadas
.bss	Variables globales o static, sin inicializar

# Introducción. Procesos

```
int numero = 21; ///.data  
  
int resultado; ///.bss  
  
const char *msg = "Resultado:\n"; ///.rodata  
  
int main(void) {  
    static int factor = 2; ///.data  
  
    ///.text  
    resultado = numero * factor;  
    printf("%s", msg);  
    return 0;  
}
```

Un **proceso** es una abstracción del sistema operativo que representa un programa en ejecución, incluye:

- Código del programa
- Datos
- Recursos
- Estructuras de control

compilación y  
enlazado



Cargador

0x08048100

**Memoria Virtual**

**Texto (.text)**

0x08073eff  
0x08074f00

**Datos**

(.bss, .data, .rodata)

0x08079cff

**Heap ↓**

Memoria mapeada  
Librerías dinámicas

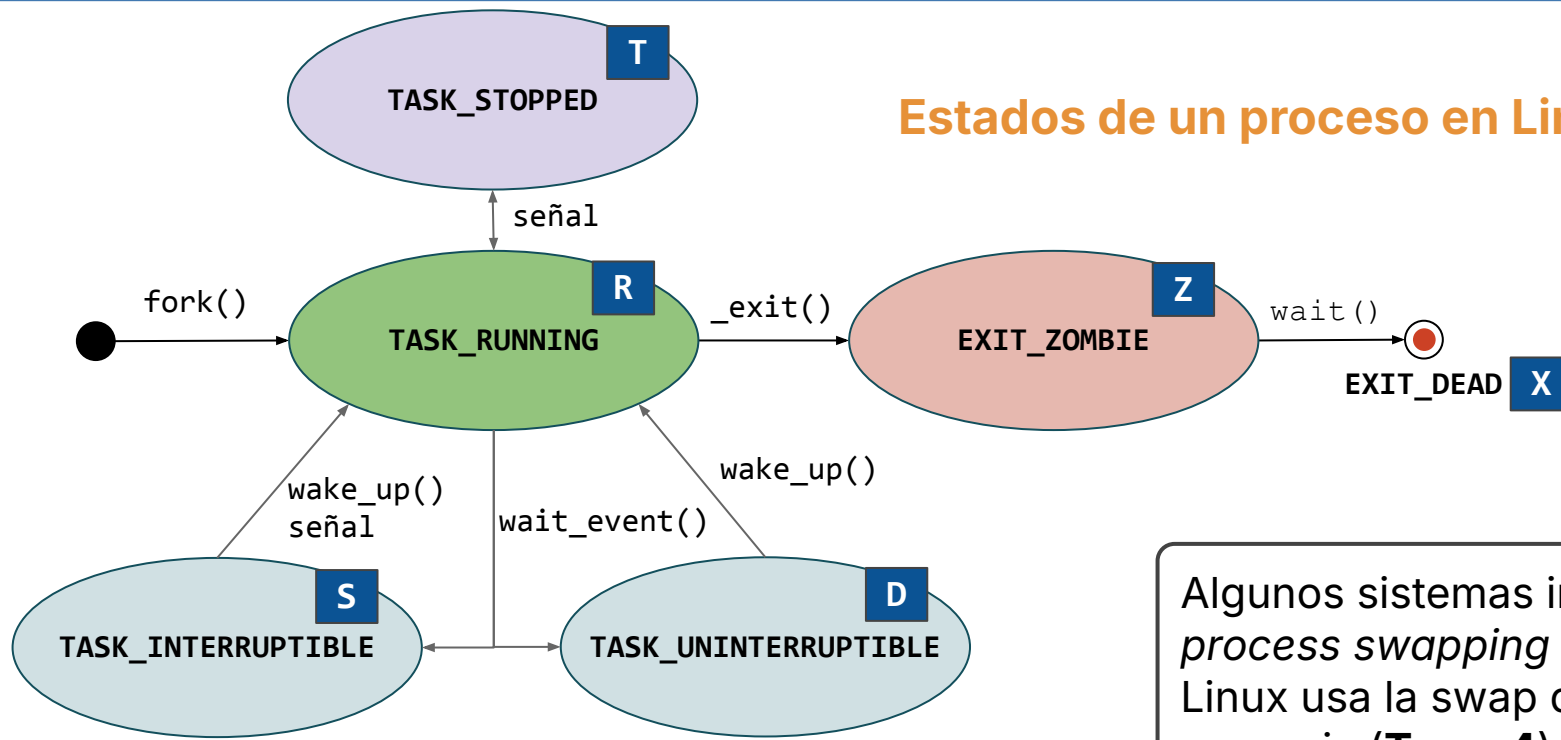
**Memoria Virtual**  
(Tema 4)

0xbfc9650c

**Pila ↑**

# Introducción. Ciclo de Vida

## Estados de un proceso en Linux



Algunos sistemas implementan *process swapping* (ineficiente). Linux usa la swap de páginas de memoria (**Tema 4**).

### Estado del Proceso

### Descripción

TASK_RUNNING	En ejecución o preparado (cola de ejecución, ver 3.6 Planificación)
TASK_INTERRUPTIBLE	Esperando o realizando E/S interrumpible por señal
TASK_UNINTERRUPTIBLE	Esperando E/S no interrumpible (ej. lectura bloque de disco)
TASK_STOPPED	Parado por una señal puede reanudarse (también TASK_TRACED)
EXIT_ZOMBIE	Proceso terminado, en el sistema para sincronizar su finalización



# SISTEMAS OPERATIVOS

*Grado en Desarrollo de Videojuegos*  
*Universidad Complutense de Madrid*

---

## TEMA 3.2 Representación en el Sistema Operativo



# Bloque de Control de Proceso (PCB)

- El **bloque de control de proceso** (PCB) es una estructura de datos que contiene toda la información necesaria para gestionar el proceso.
- En Linux la PCB está implementada por la estructura `task_struct`, incluye:
  - Identificadores del proceso y propietario.
  - Estado del procesador (registros CPU, contador de programa, puntero de pila...)
  - Planificación
  - Memoria virtual
  - Entrada/Salida (descriptor de ficheros...)
  - Contabilidad

# Identificadores del Proceso (I)

## Identificadores

- **Process ID** (PID). Identificador único que se asigna en la creación. Se utiliza para identificar al proceso en las llamadas al sistema.
- **Parent Process ID** (PPID). Identificador del proceso que lo creó.
- Identificadores de **Grupo de Procesos** (GID) y **Sesión** (SID). Son abstracciones que permiten a la shell implementar control de trabajos (Tema 3.5)

## Llamadas al Sistema

`<unistd.h>`

- Los identificadores del proceso se pueden obtener con:

```
pid_t getpid(void);
```

```
pid_t getppid(void);
```

**Más información `credentials(7)`**

# Identificadores del Proceso (II)

## Propietario

- Identificador del **usuario** (UID) y **grupo** (GID). Identifican al propietario del proceso.
- Identificadores **efectivos** del **usuario** (EUID) y **grupo** (EGID). Estos identificadores son los que usa el SO para el control de acceso.
  - Ejecutables con los bits setuid y setgid fijan los identificadores efectivos a los del archivo.

## Llamadas al Sistema

<unistd.h>

- Los identificadores del propietario proceso se pueden obtener y fijar con:

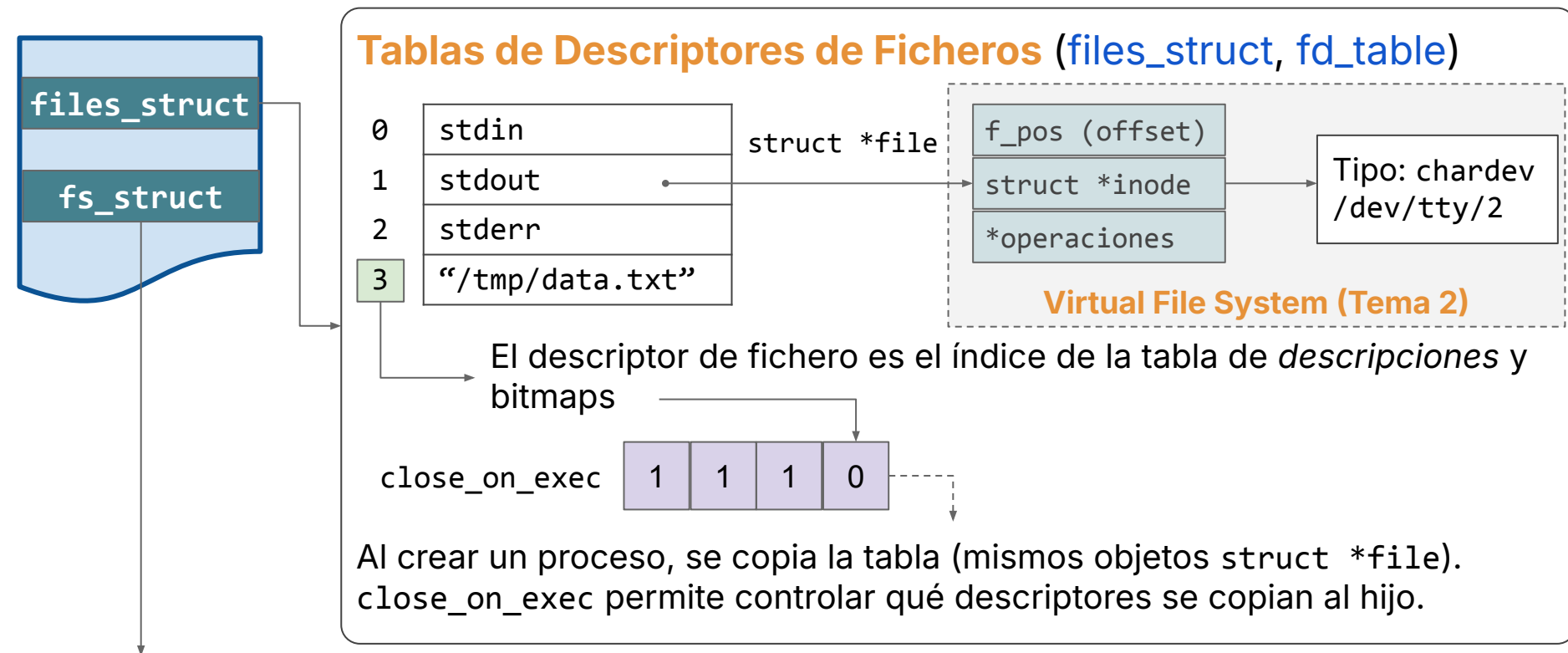
```
uid_t getuid(void);
```

```
gid_t getgid(void);
```

```
uid_t geteuid(void);      int setuid(uid_t euid);
```

```
gid_t getegid(void);      int setgid(gid_t egid);
```

# Tabla de Descriptores



## Contexto del Sistema de Ficheros

- **Directorio de trabajo**, `pwd(1)` usado para resolver toda **ruta relativa** en el proceso:  

```
char *getcwd(char *buffer, size_t size);  
int chdir(const char *path);
```
- **Directorio raíz (root)**, `chroot(1,2)` usado para interpretar **rutas absolutas**. Limita el espacio de nombres del SF del proceso (p.ej. contenedores).

# Contexto de Ejecución y Memoria

## Contexto de Ejecución

- Registros de la CPU:
  - Contador de programa
  - Puntero de pila
  - Registros de propósito general
  - Registros específicos del procesador (p.ej. SIMD o FPU)
- En linux se almacenan en `struct thread_struct` en `task_struct` con implementaciones específicas para cada arquitectura.

## Gestión de Memoria

- **Tema 4.** El PCB contiene el perfil completo de memoria virtual del proceso (`mm_struct`)
- Las páginas de memoria incluyen los segmentos de código, datos, *heap*, pila, librerías dinámicas...
- Además incluye los argumentos del programa (**argv**) y **entorno (Tema 1)**. Ver además `getenv(3)`, `setenv(3)`, `unsetenv(3)` y `environ(7)`

# Sistema de Ficheros /proc

## Sistema de Ficheros /proc/

- /proc expone las estructuras del *kernel* en un árbol de directorios
- La PCB de cada proceso se encuentra en una subdirectorio de la forma [/proc/<PID>](#)
- El comando ps(1) y top(1) muestran la información de los procesos del sistema

### Ejemplo de algunos ficheros del directorio proc de un proceso

```
...
|-- cmdline
|-- cwd -> /home/ruben
|-- environ
|-- exe -> /usr/bin/bash
|-- fd
|   |-- 0 -> /dev/pts/1
|   |-- 1 -> /dev/pts/1
|   |-- 2 -> /dev/pts/1
|   `-- 255 -> /dev/pts/1
|-- map_files
|   |-- 560f03ad6000-560f03b0d000 -> /usr/bin/bash
|   |-- 560f03b0d000-560f03bdf000 -> /usr/bin/bash
|   ...
|-- maps
|-- root -> /
|-- status
```

**fs\_struct**

**files\_struct**

**mm\_struct**

**fs\_struct**



# SISTEMAS OPERATIVOS

*Grado en Desarrollo de Videojuegos*  
*Universidad Complutense de Madrid*

---

## TEMA 3.3 Control de Procesos

# Creación de Procesos (I)

`<unistd.h>`

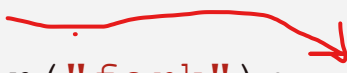


## Crear (clonar) un proceso

```
pid_t fork(void);
```

- Devuelve:
  - En el proceso **hijo**: 0
  - En el proceso **padre**: PID del proceso hijo
  - -1 en caso de **fallo**, no se crea el proceso hijo y se establece `errno`
- El *kernel* crea una copia del proceso padre (`task_struct`) nuevo proceso:
  - Memoria virtual (*copy-on-write*). Incluido los segmentos de código y datos, y el entorno.
  - Contexto de ejecución (contador de programa)
  - Tabla de descriptores (`close_on_exec`)
  - Manejadores de señales.



# Creación de Procesos (II)

```
int main() {  
    pid_t pid;  
  
    pid = fork();  
  
    switch (pid) {  
        case -1:  ERROR  
            perror("fork");  
            exit(1);  
  
        case 0:  0 IDENTIFICA AL HIJO  
            printf("Hijo: %i (padre: %i)\n", getpid(), getppid());  
            break;  
  
        default:  CUALQUIER OTRO NÚMERO IDENTIFICA AL PADRE  
            printf("Padre: %i (hijo: %i)\n", getpid(), pid);  
            break;  
    }  
  
    return 0;  
}
```

# Creación de Procesos (II)

## Ejecutar un programa

Reemplaza la imagen del proceso actual por una nueva

- path es la ruta a un ejecutable ELF (o un *script*)
- El primer elemento de argv es el nombre del programa y el último es NULL
- La familia de funciones exec(3) usa execve(2):

```
int execl(const char *path, const char *a0, ...);
int execv(const char *path, char *const argv[]);
int execlp(const char *file, const char *a0, ...);
int execvp(const char *file, char *const arg[]);
int execlenv(const char *path, const char *a0, ..., char *const env[]);
int execve(const char *path, char *const arg[], char *const env[]);
int execvpe(const char *file, char *const arg[], char *const env[]);
```

Busca el ejecutable en el **path (p)**, especifica el **entorno (e)**

Argumentos en modo **lista (l)** o **vector (v)**

# Terminación de Procesos (I)

- Un proceso puede finalizar:
  - Voluntariamente, llamando `_exit(2)` (o ejecutando `return` desde `main()`)
  - Al recibir una señal (múltiples causas)
- **Finalización** voluntaria el proceso:  
`void _exit(int status);`
  - `status` es el estado de salida, que debe ser un número menor que 255
    - Por convenio, 0 (`EXIT_SUCCESS`) indica éxito y 1 (`EXIT_FAILURE`), error
    - No devolver nunca `errno` ni -1
    - Accesible en la *shell* vía `$?` o en el proceso padre vía `wait(2)`
  - Los **descriptores de fichero** abiertos por el proceso se cierran
  - Los **hijos del proceso** quedan huérfanos y son adoptados por un proceso antecesor (`init(1)` o `systemd(1)`)
  - El proceso padre recibe la señal `SIGCHLD`
- La función **`exit(3)`** llama a las funciones registradas con `atexit(3)` y `on_exit(3)`, vacía y cierra los *streams* abiertos de `stdio(3)`, elimina los ficheros temporales creados por `tmpfile(3)`, y finalmente ejecuta a `_exit(2)`

# Terminación de Procesos (II)

- Estado **Zombi**: Un proceso termina y su padre no sincroniza la finalización
- Esperar a que un proceso hijo termine:

```
pid_t wait(int *wstatus);  
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- **pid** especifica a qué procesos hijos esperar:
  - > 0 El proceso cuyo PID es pid
  - 0 Cualquier proceso hijo de su grupo
  - -1 Cualquier proceso hijo (igual que wait(2))
  - <-1 Un proceso hijo cuyo PGID es -pid
- **options** es una OR bit a bit de las siguientes opciones:
  - WNOHANG: retorna sin esperar si no hay hijos que hayan terminado
  - WUNTRACED: retorna si un proceso hijo ha sido detenido
  - WCONTINUED: retorna si un proceso hijo detenido ha sido reanudado
- **wstatus** contiene información de estado, que puede consultarse con macros:
  - WIFEXITED(s) indica si el hijo terminó normalmente con \_exit(2). En ese caso, WEXITSTATUS(s) devuelve el estado de salida
  - WIFSIGNALED(s) indica si el hijo terminó al recibir una señal. En ese caso, WTERMSIG(s) devuelve el número de la señal recibida
- Devuelve el PID del hijo terminado o -1 en caso de error

# Envío de Señales (I)

- Las señales son **interrupciones software**, que informan a un proceso de la ocurrencia de un evento de forma **asíncrona**
- Las genera un proceso o el núcleo del sistema
- Las opciones en la ocurrencia de un evento son:
  - Realizar la acción por defecto asociada a la señal.
  - Capturar la señal con una función manejador.
  - Bloquear la señal
  - Ignorar la señal
- Algunas **señales**:
  - **SIGINT**: Interrupción. Se puede generar con Ctrl+C (**F**=terminar proceso)
  - **SIGSTOP**: Parar proceso. No se puede capturar, bloquear ni ignorar (**P**=parar)
  - **SIGTSTP**: Parar proceso. Se puede generar con Ctrl+Z (**P**)
  - **SIGCONT**: Reanudar proceso parado (continuar)
  - **SIGKILL** (9): Terminación brusca. No se puede capturar, bloquear ni ignorar (**F**)
  - **SIGSEGV**: Violación de segmento *de datos* (**F** y **C**=core, volcado de memoria)
  - **SIGTERM**: Terminar proceso (**F**)
  - **SIGCHLD**: Terminación del proceso hijo (**I**, ignorar)

**signal(7)**

# Envío de Señales (II)

- Enviar una señal a un proceso:

```
int kill(pid_t pid, int signal);
```

<signal.h>

SV+BSD+POSIX

- **pid** indica a qué procesos se enviará la señal:
  - >0: Al proceso con ese PID
  - 0: A todos los procesos de su grupo
  - -1: A todos los procesos (de mayor a menor), excepto init y el proceso
  - <-1: A todos los procesos del grupo cuyo PGID es -pid
- **signal** es la señal que se enviará (si es 0, se simula el envío)
- El comando `kill(1)`, también *built-in* de la *shell*, expone esta funcionalidad
- Llamadas equivalentes:

```
int raise(int signal);
```

```
int abort(void);
```

- `raise(signal) ⇒ kill(getpid(), signal)`
- `abort() ⇒ kill(getpid(), SIGABRT)`

<stdlib.h>



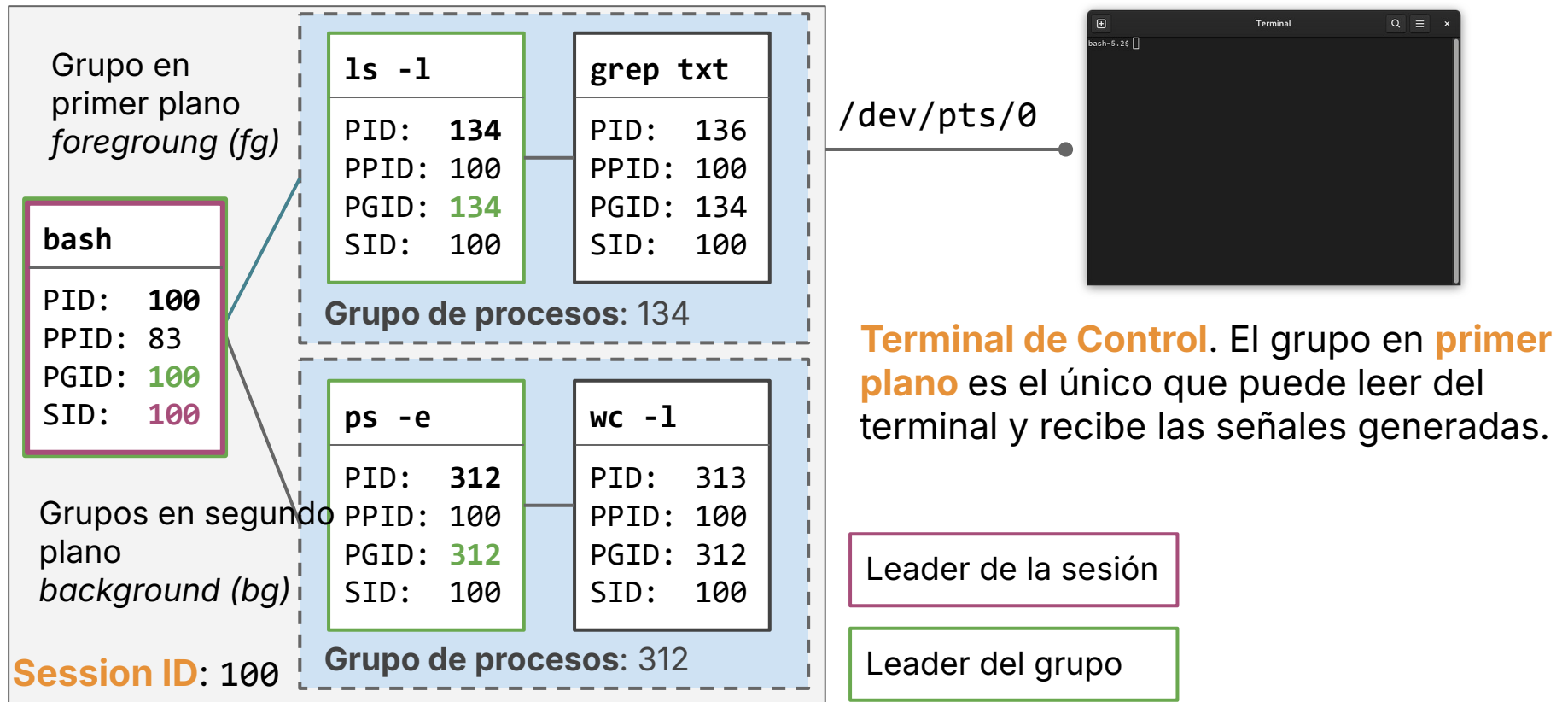
# SISTEMAS OPERATIVOS

*Grado en Desarrollo de Videojuegos*  
*Universidad Complutense de Madrid*

---

## TEMA 3.4 Grupos de Procesos

# Sesiones y Grupos



- **Comandos** para la gestión de trabajos (grupos): jobs, fg y bg
- **Llamadas al sistema** para la gestión de sesiones y grupos:

```
pid_t getpgid(pid_t pid);  
int setpgid(pid_t pid, pid_t pgid);  
pid_t getsid(pid_t pid);  
pid_t setsid(void);
```



# El Proceso `init`

---

- El **proceso `init`** es el primer proceso que arranca el *kernel* después de inicializar el sistema. Ejemplos: `systemd`, `OpenRC`, `sysvinit`...
- El proceso `init` se encarga de:
  - Gestionar los servicios del sistema (`red`, `login`, montaje de sistemas de ficheros, terminales de control...)
  - ***subreaper*** (`prctl(2)`). *Adopta* a los procesos huérfanos (proceso padre terminado) y ejecuta `wait()` para recoger los procesos zombie.



# SISTEMAS OPERATIVOS

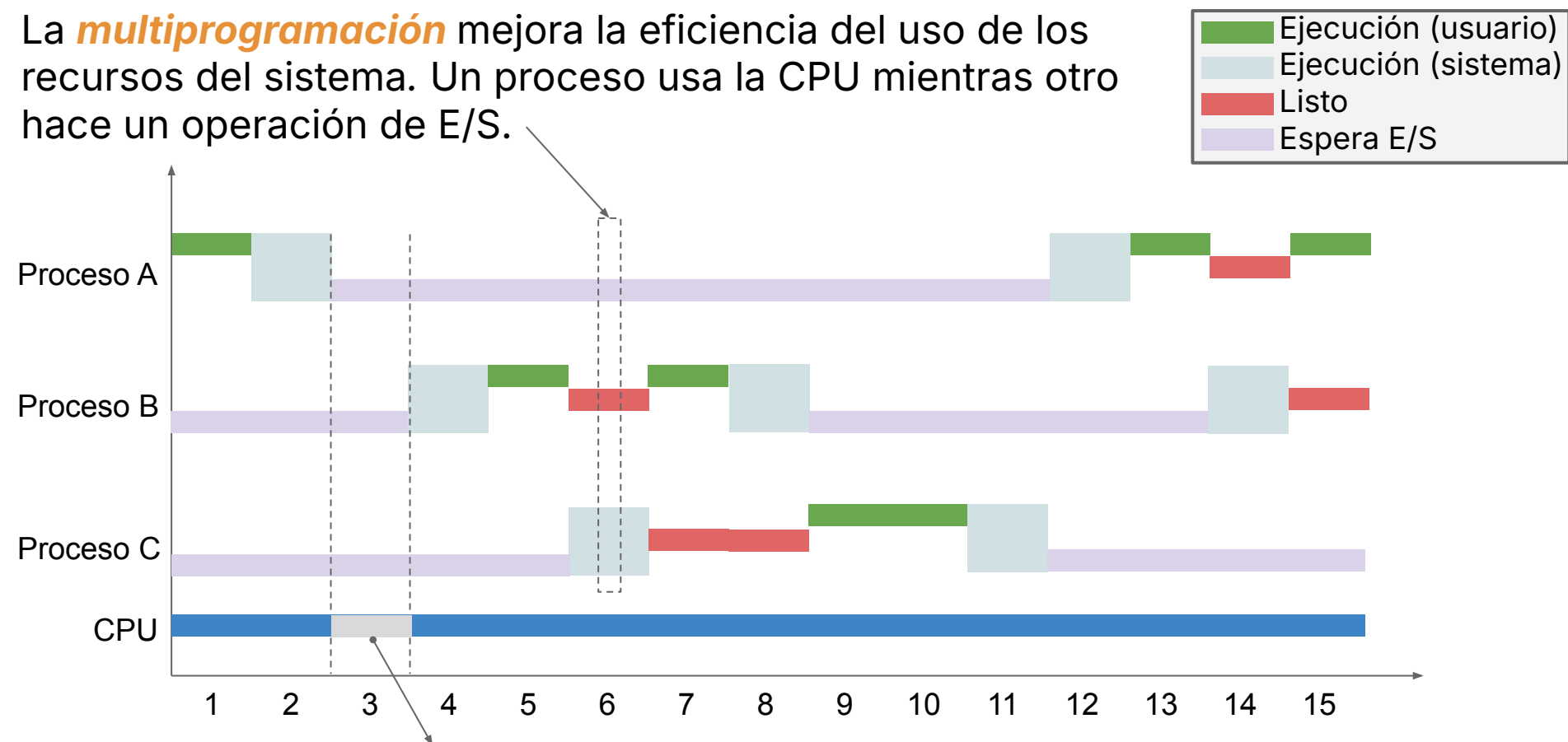
*Grado en Desarrollo de Videojuegos*  
*Universidad Complutense de Madrid*

---

## TEMA 3.6 Planificación

# MultiProgramación

La **multiprogramación** mejora la eficiencia del uso de los recursos del sistema. Un proceso usa la CPU mientras otro hace un operación de E/S.



Utilización de la CPU 14 de 15 ciclos (93%)

- ¿A qué proceso asignamos la CPU? (**Algoritmos de Planificación**)
- ¿Cómo cambiamos de un proceso a otro? (**Cambio de Contexto**)
- ¿Cuándo evaluamos el estado del sistema? (**Tick de reloj**)

# Niveles de Planificación

## Niveles de Planificación

Tradicionalmente los planificadores se clasifican en tres tipos:

- Planificador a **largo plazo**. Control de admisión de **trabajos** en el sistema (procesos que se pueden ejecutar)
- Planificador a **medio plazo**. Mueve los procesos de memoria principal a secundaria (*process swapping*).
- Planificador a **corto plazo**. Decide qué procesos, de los que están listos para ejecutarse, se ejecutarán a continuación en la CPU.

## Planificadores en el Sistema Operativo

- El único **planificador en los SO de propósito general** actuales es el planificador a **corto plazo**
- Los planificadores a largo plazo se implementan con aplicaciones específicas de gestión de trabajos (p.ej. Slurm o PBS)
- Los planificadores a medio plazo ha sido reemplazado por los mecanismos de paginación, Tema 4). OS para dispositivos móviles (Android) pueden suspender aplicaciones en segundo plano.

# Activación del Planificador

## Eventos de Planificación

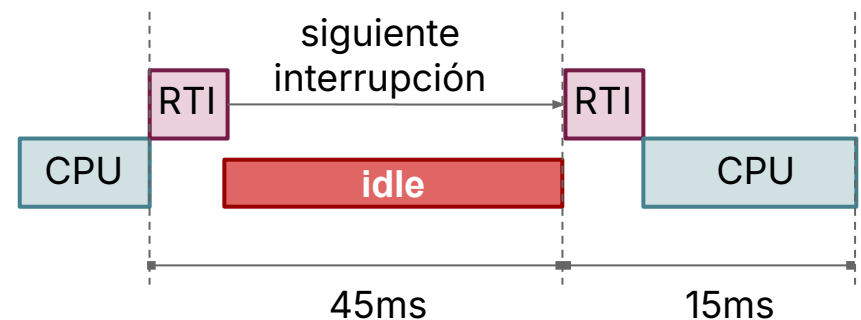
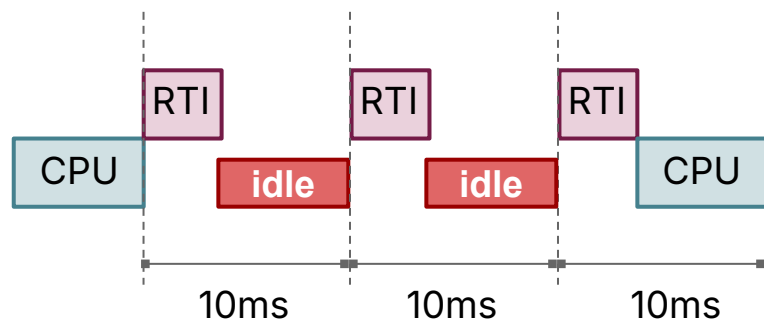
Determina cuándo se ejecuta el planificador para evaluar el estado del sistema:

- **Periódicamente.**

- El temporizador hardware genera una **interrupción** a intervalos regulares
- La frecuencia define el **tick rate** (p.ej. Linux 100Hz por defecto CONFIG\_HZ)
- La rutina de tratamiento de interrupción activa el evento de planificación

- **Tickless**

- La interrupción del temporizador se programa a demanda.
- Mejor perfil en ahorro de energía para sistemas empotrados o móviles.
- [Linux en general](#) usa un modelo híbrido *tickless* cuando no hay tareas



# Activación del Planificador (II)

## Eventos de Planificación

Determina cuándo se ejecuta el planificador para evaluar el estado del sistema:

- **Cambio estado del proceso**

- La tarea se bloquea (E/S, futex, sleep,...) `TASK_INTERRUPTIBLE` o `TASK_UNINTERRUPTIBLE`
- La tarea se desbloquea (E/S completada, futex desbloqueado...) `TASK_RUNNING`

- **Creación o terminación de un proceso**

- Después de `fork()` o `clone()` se crea una nueva tarea que se añade a la cola de tareas ejecutables.
- Cuando el proceso termina por cualquier causa: recepción de una señal (`SIGKILL`, `SIGSEV...`), de forma voluntaria (`_exit()`)

- **Cesión voluntaria**

- El proceso cede la CPU (`sched_yield()`)
- El proceso cambia su prioridad (ej. `nice()`)

# Cambios de Contexto

## Cambio de Contexto

Conjunto de acciones que realiza el SO para cambiar el proceso en ejecución:

- **Cambio de la memoria virtual**
  - Actualización de los registros de memoria al contexto de memoria del nuevo proceso (mm\_struct, ej. dirección de la tabla de páginas, caché asociadas... **Tema 4**)
- **Cambio del contexto ejecución de la CPU** (thread\_struct)
  - Guardar los registros de propósito general, contador de programa, pila
  - Guardar los registros específicos (e.j. fpu)
  - Actualizar el puntero de pila a la nueva tarea
  - Cargar los registros de la nueva tarea
  - Continuar con la ejecución de la nueva tarea (en modo usuario)

- La operación de **cambio de contexto** de un proceso es **costosa**
- El cambio de contexto entre **threads** es menor (memoria virtual) pero **no despreciable**.

# Objetivos de Planificación

El planificador es un algoritmo que asigna recursos a los procesos tratando de optimizar un objetivo:

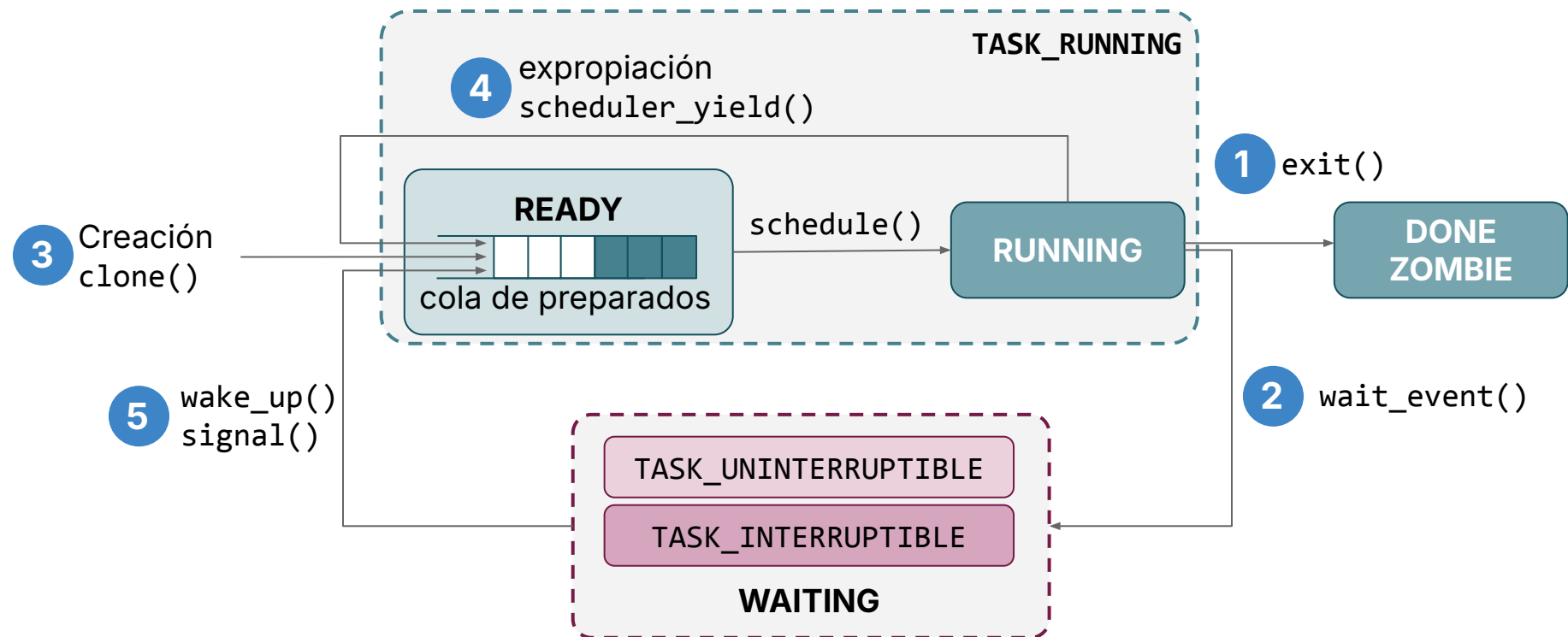
- **Utilización:** maximizar el tiempo que la CPU está en uso.
- **Productividad:** maximizar el número de procesos terminados por unidad de tiempo.
- **Tiempo de respuesta:** minimizar el tiempo desde que empieza el proceso hasta que produce la primera respuesta.
- **Turnaround:** minimizar el tiempo desde que entra el proceso en el sistema hasta que termina.
- **Tiempo de espera:** minimizar el tiempo total en la cola de espera.

$$\text{Tiempo de espera} = \text{Turnaround} - \text{Tiempo CPU} - \text{Tiempo E/S}$$

- **Fairness (justicia):** reparto **equitativo** de los recursos modulado por la prioridad y características de cada proceso.



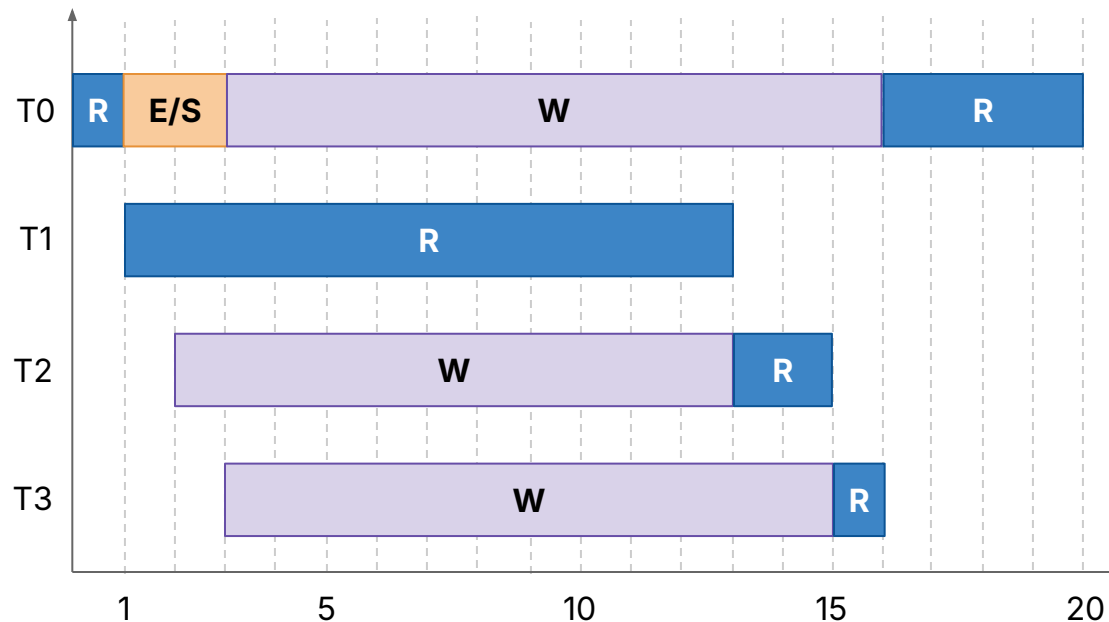
# Modelos de Planificación



- **No expropiativo:** El proceso usa la CPU hasta que termina, entra en espera de un evento o cede la CPU voluntariamente. 1 2 4
- **Expropiativo:** El proceso usa la CPU hasta que el planificador decide reemplazarlo por otro. 1 – 5

# First Come First Serve (FCFS)

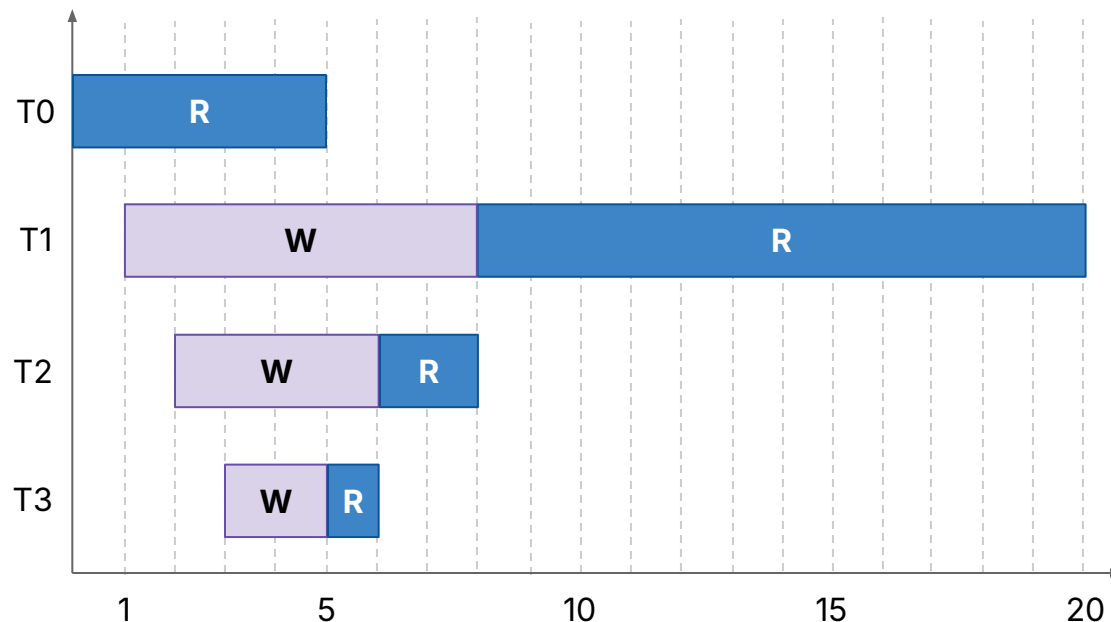
- Los procesos se planifican según el **orden de entrada** en la cola de preparados.
- Las tareas largas (T1) pueden bloquear otras más cortas (T2, T3) o aquellas que se interrumpen para E/S (T0)
- En general presenta **baja productividad**, y un **alto tiempo de respuesta y latencia**



Proceso	Entrada (ms)	CPU (ms)
T0	0	5
T1	1	12
T2	2	2
T3	3	1

# Shortest Job First (SJF)

- Se planifican **primero los procesos más cortos**
- Favorece tareas interactivas, la **productividad** (más número de tareas completadas) y el **tiempo medio de espera**
- Es necesario conocer el perfil de ejecución de las tareas (predicción basada en los ciclos anteriores)
- Problemas de **inanición** o injusticias (*unfairness*) para tareas largas (expropiativo)\*

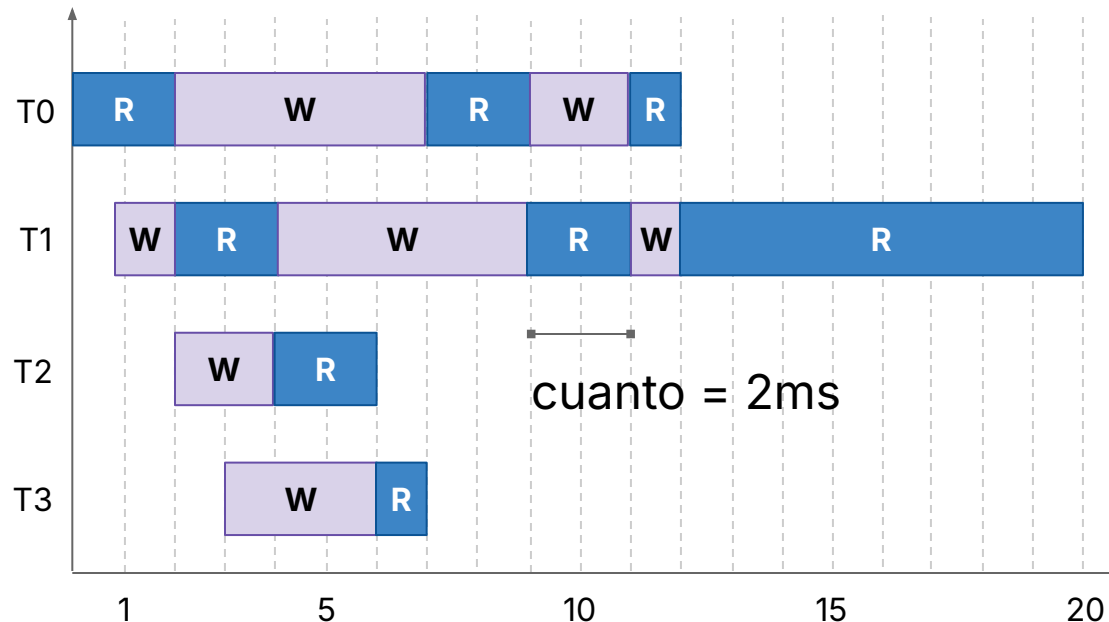


Proceso	Entrada (ms)	CPU (ms)
T0	0	5
T1	1	12
T2	2	2
T3	3	1

\* Los planificadores basados en prioridad presentan este problema

# Round Robin (RR)

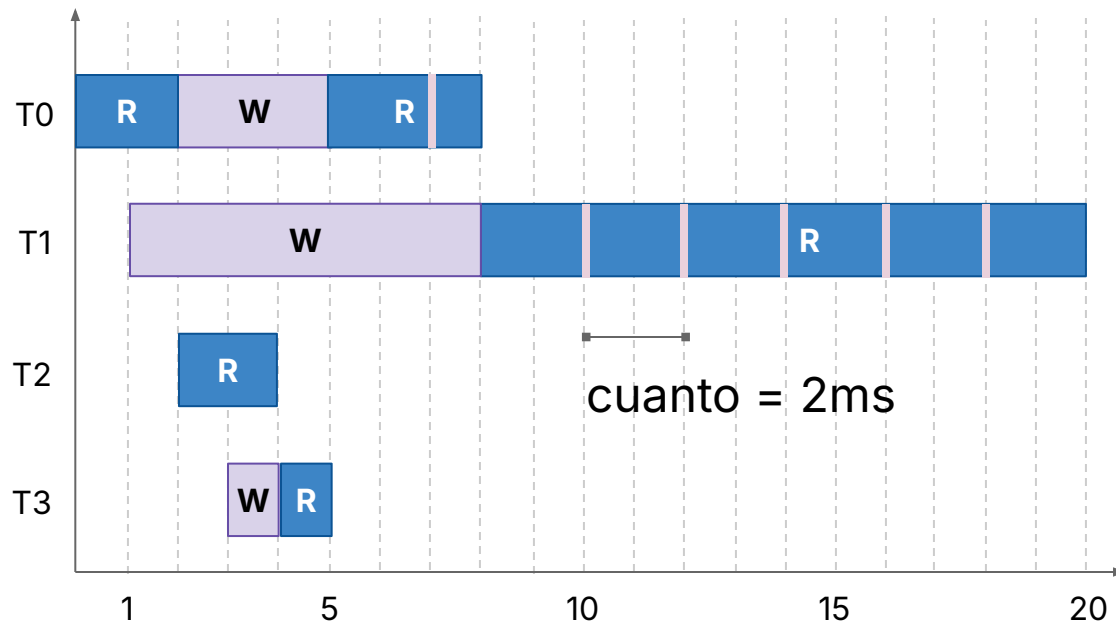
- Combina **FCFS** con **periodos fijos de tiempo (cuanto)**
- Un proceso se ejecuta hasta que termina, se bloquea, cede la CPU o termina su cuanto. La cola de espera se gestiona como una cola FIFO circular.
- Asignación equitativa (fair) y bajo tiempo de espera para procesos de distinta duración
- El tiempo de respuesta se degrada con muchos procesos o procesos de la misma duración
- Duración del cuanto: largos ( $\rightarrow$ FCFS) cortos ( $\uparrow$ cambios contexto)



Proceso	Entrada (ms)	CPU (ms)
T0	0	5
T1	1	12
T2	2	2
T3	3	1

# Shortest Remaining Time First (SRTF)

- Combina **SJF** con **periodos fijos de tiempo** (**cuanto**)
- El siguiente trabajo es el de menor tiempo restante para la finalización
- Es necesario conocer el perfil de ejecución de las tareas.
- Problemas de **inanición** o injusticias (*unfairness*) para tareas largas



Proceso	Entrada (ms)	CPU (ms)
T0	0	5
T1	1	12
T2	2	2
T3	3	1

# Colas Multinivel con Retroalimentación (I)

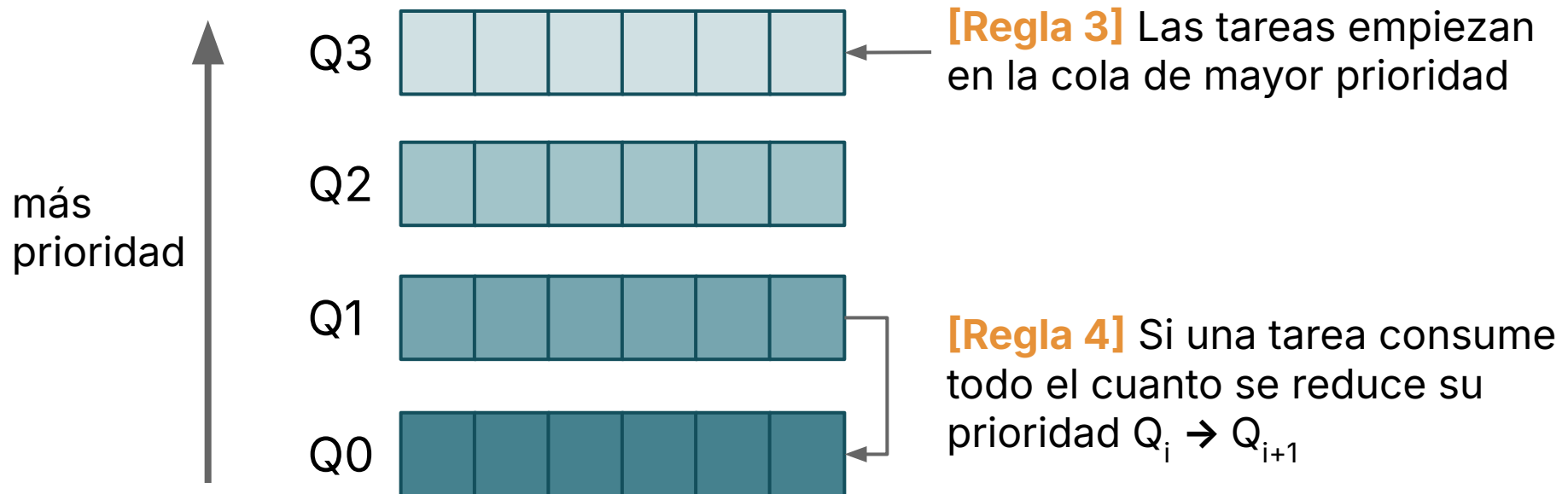
## Motivación y Objetivos

- **Diferentes tipos de trabajos**
  - Interactivos, intensivos en CPU, intensivos en E/S,...
  - Los algoritmos anteriores no pueden acomodar todos los tipos.
- **Combinar distintos algoritmos para perfiles diferentes**
  - **[Obj. 1] Turnaround**: comportamiento similar al SJF
  - **[Obj. 2] Tiempo de respuesta**: comportamiento similar a RR
  - Posibilidad de emplear algoritmos con objetivos son contradictorios
- **Desafíos**
  - No sabemos *a priori* de qué tipo es un trabajo o cuáles son sus ciclos de uso de CPU.
  - Permitir al administrador especificar la prioridad relativa de los trabajos

# Colas Multinivel con Retroalimentación (II)

## Estructura

- Los trabajos se clasifican en diferentes colas de preparados. Cada cola tiene asociada una prioridad.
- **[Regla 1]** Los procesos con mayor prioridad (en colas de mayor prioridad) se ejecutan primero
- **[Regla 2]** Los procesos con la misma prioridad se ejecutan usando RR

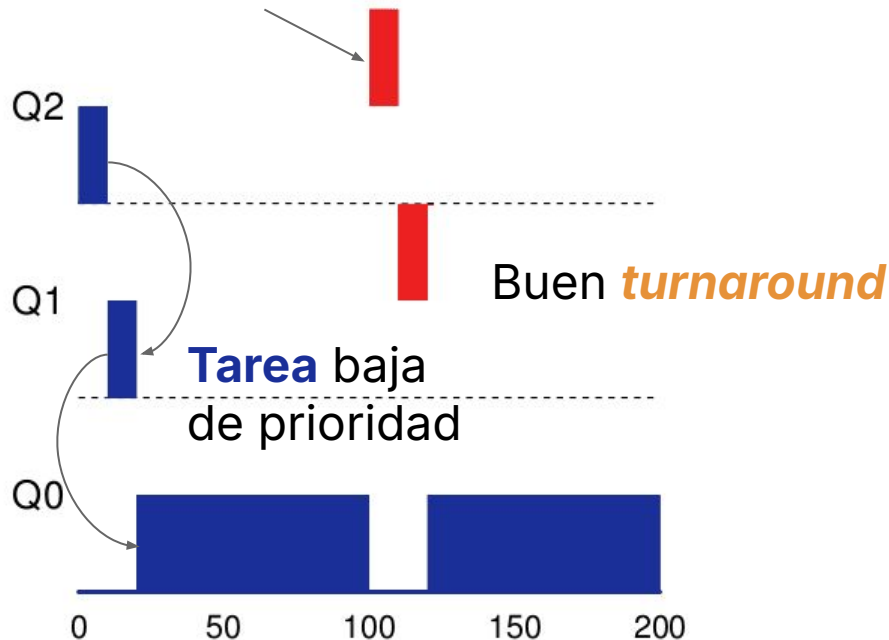


# Colas Multinivel con Retroalimentación (III)

## Ejemplo

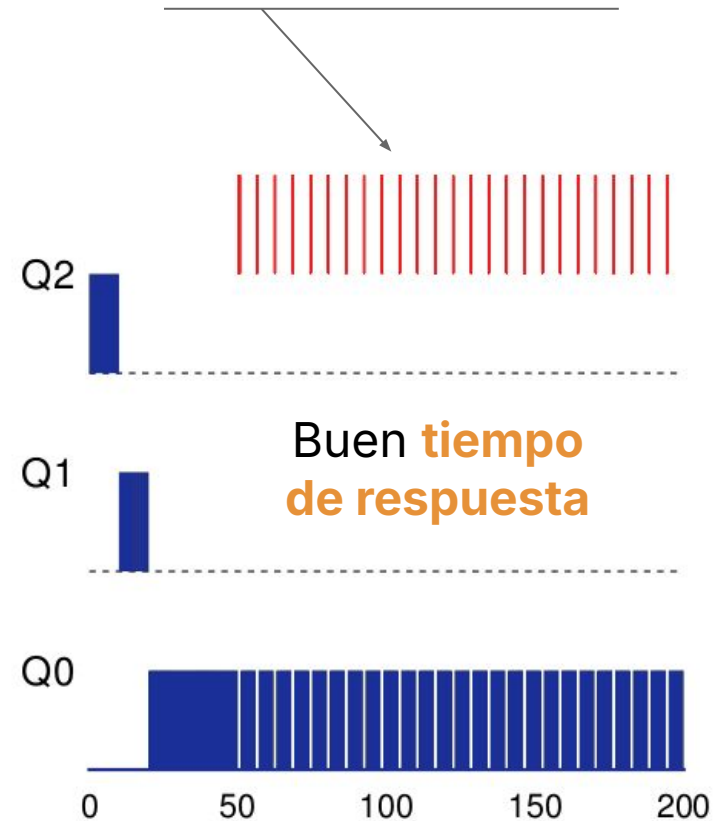
- Tarea de **larga duración (CPU)**
- Tarea de **corta duración**
- La tarea de corta duración se añade al sistema ( $t=100$ )

**Tarea** comienza con máx. prioridad



## Ejemplo

- Tarea de **larga duración (CPU)**
- Tarea **interactiva**
- Ciclos de CPU cortos menores que el cuánto (**mantiene prioridad**)





# Colas Multinivel con Retroalimentación (IV)

## Problemas

- **Inanición**

- Tareas intensivas en CPU (terminan en prioridad baja) con muchas tareas interactivas (prioridad alta)
- **Priority Boost**. Periódicamente aumentar la prioridad de todas las tareas (mover a la cola más prioritaria)

- **Abuso de las reglas de planificación**

- Si una tarea cede la CPU antes de que expire el cuanto no baja de prioridad
- **Accounting CPU**. Si el tiempo de ejecución total en la cola  $Q_i$  es mayor que el cuanto se reduce la prioridad.

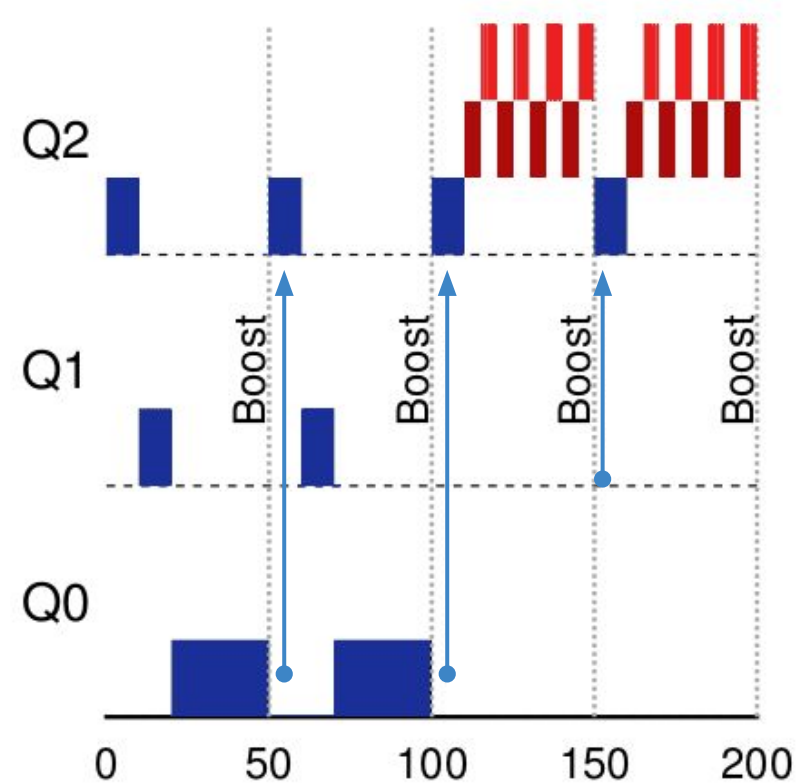
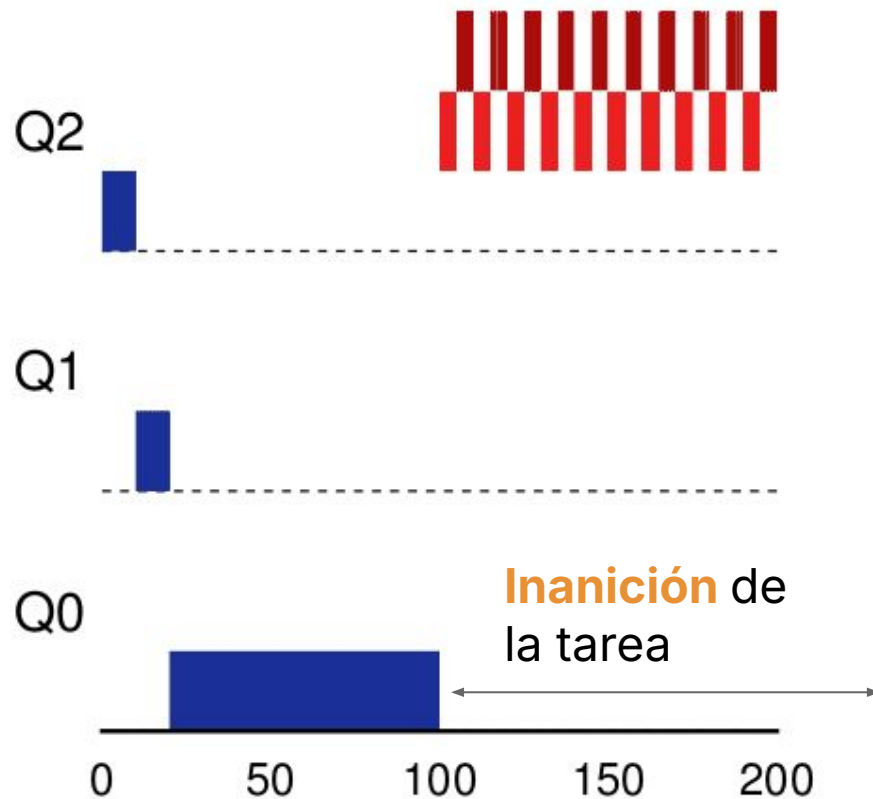
- **Las tareas no aumentan de prioridad**

- Una tarea intensiva en CPU seguirá en la mínima prioridad aunque pase a una fase interactiva.
- **Priority Boost**

# Colas Multinivel con Retroalimentación (V)

## Ejemplo

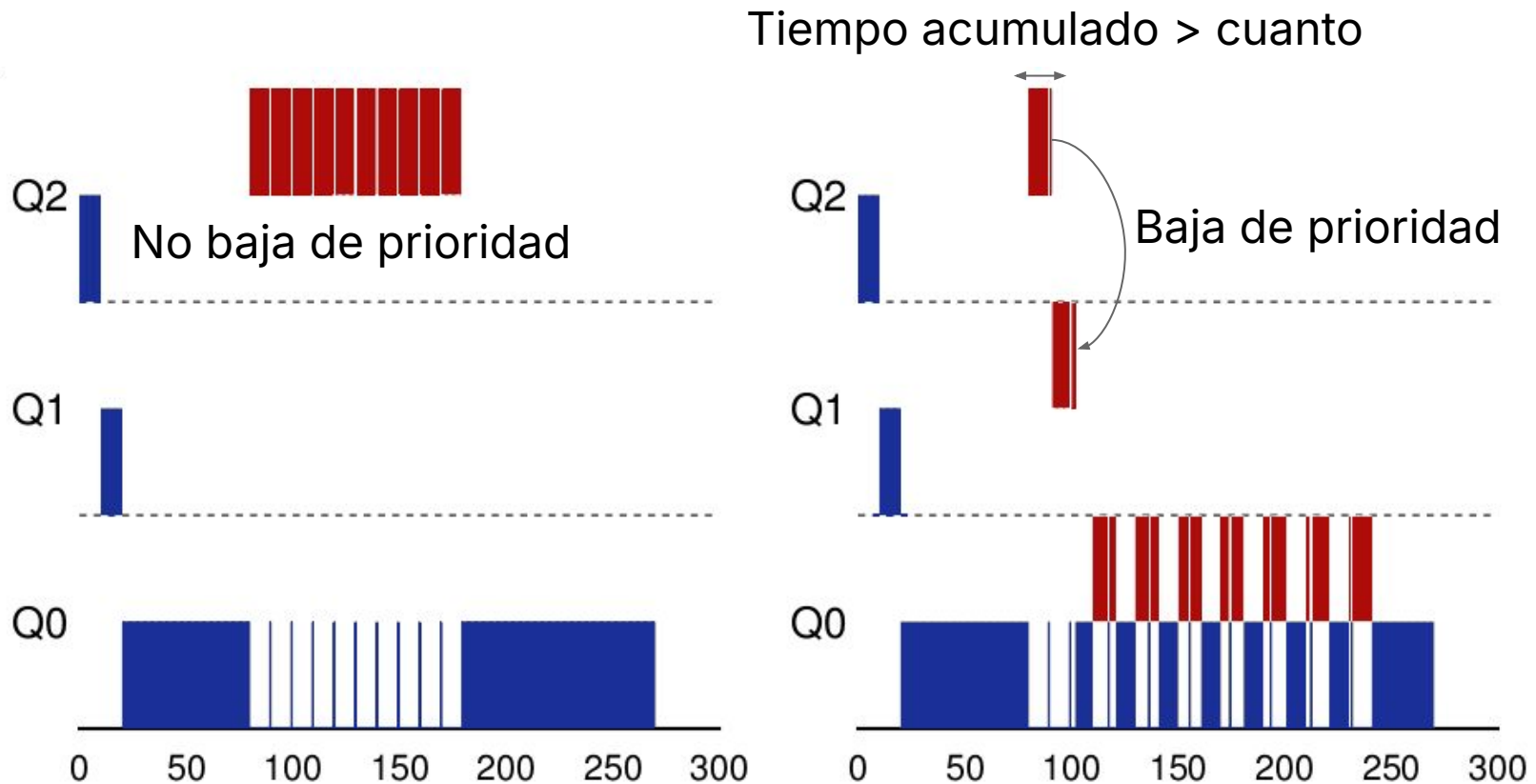
- Tarea de **larga duración (CPU)**
- Tarea de **corta duración** ( $t=100\text{ms}$ )
- Tarea de **corta duración** ( $t=110\text{ms}$ )



# Colas Multinivel con Retroalimentación (VI)

## Ejemplo

- Tarea de **larga duración (CPU)**
- Tarea de **larga duración (CPU)**, cede la CPU antes de que expire el cuanto



# Planificador en Linux

## Políticas de planificación

- **SCHED\_OTHER**: Política estándar de tiempo compartido con prioridad 0, que considera el valor de *nice* (entre -20 y 19, 0 por defecto) para repartir la CPU
  - Completely Fair Scheduler (**CSF**) hasta Linux 6.6
  - Earliest Eligible Virtual Deadline First (**EEVDF**)
- **SCHED\_FIFO**: Política de tiempo real FIFO con prioridades entre 1 y 99, donde las tareas se ejecutan hasta que se bloquean por E/S, son expropiadas por una tarea con mayor prioridad o ceden la CPU.
- **SCHED\_RR**: Como la anterior, pero las tareas con igual prioridad se ejecutan por turnos (*round-robin*) en porciones de tiempo (100 ms por defecto).

