

Procesos

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

13 de octubre de 2025



Este trabajo se entrega bajo la licencia “Atribución-CompartirIgual 4.0 Internacional”[1] (cc-by-sa).

Usted es libre de:

- ▶ **Compartir:** *Copiar y redistribuir el material en cualquier medio o formato.*
- ▶ **Adaptar:** *Remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.*
- ▶ **Se pueden dispensar estas restricciones si se obtiene el permiso de los autores.**
- ▶ *Las imágenes de terceros mantienen sus derechos originales.*

©2022 Gorka Guardiola y Enrique Soriano.

[1] Algunos derechos reservados. “Atribución-CompartirIgual 4.0 Internacional” (cc-by-sa). Para obtener la licencia completa, véase <https://creativecommons.org/licenses/by-sa/4.0/deed.es>. También puede solicitarse a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Recuerda:

- ▶ El proceso es un programa en ejecución, un flujo de control.
- ▶ El sistema operativo proporciona esa abstracción.
- ▶ Cada proceso piensa que está solo en la máquina, que tiene su propia CPU y su propia memoria.
- ▶ Los procesos tienen distintos datos asociados.

Llamadas al sistema y funciones de libc

- ▶ En este tema, comenzaremos a ver **llamadas al sistema** y funciones de la libc.
- ▶ Ojo: para algunas funciones de librería, existen versiones *thread safe* para cuando usemos hilos. Siempre debemos intentar usar esas funciones.

Llamadas al sistema y funciones de libc

- ▶ **Hay que comprobar los errores**
- ▶ Las llamadas al sistema suelen retornar un valor negativo en caso de error (NULL si retornan un puntero). Así se detecta si hay error.
- ▶ Si lo hay, puedes consultar una variable llamada `errno`, que contendrá el valor que describe el error que se ha producido (descrito en la página de manual correspondiente).

- ▶ `perror`: imprime la cadena asociada a `errno`. Si se pasa como argumento una cadena, escribe primero la cadena y después el error.
- ▶ `strerror`: devuelve la cadena asociada al error que se le pasa. La función `strerror_r` es la versión *thread safe*.
- ▶ `warn`: escribe el error de la última llamada concatenado después de la cadena con formato que se le pasa.
- ▶ `warnx`: lo mismo pero sin el error de la última llamada.

```
void perror(const char *s);  
char * strerror(int errnum);  
void warn(const char *fmt, ...);  
void warnx(const char *fmt, ...);
```

Trazando llamadas

- ▶ El comando `strace` nos permite trazar las llamadas al sistema que hace un proceso. P. ej.:

```
strace -p 3234
```

- ▶ El comando `ltrace` nos permite trazar las llamadas a funciones de biblioteca. P. ej.:

```
ltrace -p 3234
```

Carga de procesos

- ▶ Fuente → Compilación → Enlazado → Binario (ejecutable).
- ▶ El fichero binario contiene la información para crear un proceso, organizadas en secciones. Hay distintos formatos: ELF o Extensible Linking Format, PE (Windows), a.out (Unix antiguos), etc.
- ▶ El binario es simplemente una receta para crear un proceso para ejecutar el programa.
- ▶ En su cabecera se indica la arquitectura, el tamaño/**offset** de las tablas y secciones, y el punto de entrada (dirección para comenzar a ejecutar).

El enlazado puede ser:

- ▶ Estático: cuando se genera el binario ejecutable, se incluye todo el código necesario en el binario (el del programa y el código que usa de todas las bibliotecas).
- ▶ Dinámico: el binario no incluye el código de las bibliotecas. En *tiempo de ejecución*, el sistema operativo carga/enlaza las bibliotecas que intenta usar el binario.
 - ▶ Son bibliotecas compartidas (shared library) cuando los distintos procesos que usan una misma biblioteca la comparten (esto es, esa biblioteca sólo se carga una vez en memoria).

ELF: cabecera†

```
#define EI_NIDENT 16
typedef struct{
    unsigned char  e_ident[EI_NIDENT]; //magic, 32/64bit, encoding...
    Elf32_Half     e_type;              // executable, relocal, shared object...
    Elf32_Half     e_machine;          // intel, sparc, M68K, mips...
    Elf32_Word     e_version;          // version of elf
    Elf32_Addr     e_entry;            // entry point (virtual addr)
    Elf32_Off      e_phoff;            // offset for PH table (process segments)
    Elf32_Off      e_shoff;            // offset for SH table (elf sections)
    Elf32_Word     e_flags;            // flags for CPU
    Elf32_Half     e_ehsize;           // size of this header
    Elf32_Half     e_phentsize;        // size of the PH table
    Elf32_Half     e_phnum;           // entries in the PH table
    Elf32_Half     e_shentsize;        // size of the SH table
    Elf32_Half     e_shnum;           // entries in the SH table
    Elf32_Half     e_shstrndx;        // index of the string table in SH table
} Elf32_Ehdr;
```

- ▶ Section Header Table (SH): describe las secciones del fichero ELF.
- ▶ Program Header Table (PH): describe los segmentos del proceso que ejecute el binario.

ELF: secciones†

Secciones: tienen la información necesaria para **enlazar un objeto con el fin de generar un binario ejecutable**. Tienen la información necesaria *en tiempo de enlazado*. Algunas importantes son:

- ▶ `.text`: código (instrucciones) del fichero.
- ▶ `.data`, `.rodata`: variables inicializadas.
- ▶ `.bss`: descripción de las variables sin inicializar.
- ▶ `.plt`, `.got`, `.got.plt`, `.dynamic`: información para el enlazado dinámico.
- ▶ `.symtab`, `.dynsym`: tabla de símbolos, que es información sobre las variables que es útil para depurar los programas.
- ▶ `.strtab`, `.dynstr`: tabla con las cadenas de texto de la tabla de símbolos y las secciones.

ELF: segmentos

- ▶ Un ELF contiene información para poder **cargar el binario en memoria** para ejecutarlo.
- ▶ Un segmento es en realidad una parte de la región de memoria de *un proceso* (es parte de la estructura de datos que describe al proceso en el kernel).
- ▶ ELF describe cómo serán los segmentos cuando se cargue el binario a memoria.
- ▶ Un segmento puede contener información descrita en 0 o más secciones.
- ▶ Un segmento tiene atributos (solo-lectura, lectura/escritura, ejecutable), indica dónde se tiene que cargar en memoria, alineación, etc.

ELF: símbolos

- ▶ El enlazador usa la tabla de símbolos para resolverlos: las referencias a los distintos objetos exportados por un fichero se traducen en direcciones relativas en el código generado.
- ▶ Los depuradores también usan la tabla de símbolos: facilita la depuración.
- ▶ La tabla de símbolos **no es necesaria** para ejecutar un programa. El sistema operativo no necesita esa información para cargar el ejecutable.
- ▶ El comando `strip` elimina la tabla de símbolos de un fichero.
- ▶ Para cada símbolo se guarda: nombre, tamaño, ámbito (local, global), tipo (función, variable, etc.), ...

ELF: símbolos

El comando `nm` muestra los símbolos de un fichero objeto / ejecutable y su dirección de **memoria virtual** cuando se ejecute el programa. Algunos de los tipos de símbolos son:

- ▶ T: indica que está en el segmento de texto, donde hay instrucciones (text). Son subprogramas, etiquetas, etc.
- ▶ D: indica que está en el segmento de datos (data). Normalmente variables globales inicializadas.
- ▶ R: indica que es una variable de solo-lectura (read-only). Por ejemplo, literales de tipo string.
- ▶ B: indica que está en el segmento de datos sin inicializar (bss). Normalmente variables globales sin inicializar.
- ▶ U: indica que es un símbolo que está pendiente de resolver (undefined). Ese símbolo es externo, y todavía no se sabe dónde estará. Se resolverá al ejecutar el programa.

Carga de un ejecutable

Cuando se ejecuta un fichero, el **cargador**¹ (que forma parte del kernel), hace lo siguiente:

1. Comprueba si se puede ejecutar (permisos, etc.).
2. Comprueba el tipo de ejecutable (interpretado o binario). Si el fichero empieza por los caracteres `#!`, es interpretado: se ejecutará el interprete indicado a continuación, con la ruta de este fichero como argumento.
3. Copia las secciones del ELF a memoria, que se organiza en *segmentos*.
4. Copia los argumentos del programa y variables de entorno (`argc`, `argv`, `envp`) a la pila del proceso.
5. Inicializa el contexto del proceso (registros, etc.) y sus atributos: se asigna **PID** (Process ID), etc.
6. Pone el proceso a ejecutar, comenzará por su *punto de entrada*.

¹No confundir con el *boot loader* que realiza una tarea similar pero en el arranque.

Memoria virtual

- ▶ El proceso maneja únicamente direcciones de memoria virtual.
- ▶ El sistema operativo usa **paginación en demanda**: el programa se va cargando en memoria poco a poco según se demanda el acceso a sus direcciones de memoria.
- ▶ Si se borra o se sobrescribe un binario, los procesos pueden fallar.
- ▶ Hay segmentos que se pueden *compartir* entre procesos (texto, bibliotecas).

Enlazado dinámico

- ▶ Si el ejecutable fue enlazado estáticamente, tiene dentro todo lo necesario para ejecutar...
- ▶ ... pero cuando el enlazado es dinámico, el **enlazador dinámico**² debe resolver los símbolos que están sin definir: relocalizaciones.
- ▶ Básicamente, una relocalización es:
“reemplaza el valor de X bytes que está en el offset Y por la dirección del símbolo externo S”
- ▶ **Lazy binding**: la resolución no se hace de golpe a la hora de comenzar, se hace a medida que se van usando los símbolos.
- ▶ No siempre es así: se puede forzar a que se resuelvan todos los símbolos al principio (RELRO o relocation read-only).

²En linux, `/lib/ld-linux.so`

Lazy binding†

¿Cómo funciona?

- ▶ GOT (Global Offset Table): contiene las direcciones de los símbolos, esto es, la dirección a la que tiene que saltar un *trampolín*.
- ▶ PLT (Procedure Linkage Table): contiene el código (stubs) de los *trampolines*.
- ▶ Las relocalizaciones se realizan en la GOT, no en el texto (código)
→ se *parchea* la GOT, esto es, se escriben las direcciones allí según se van resolviendo.
- ▶ El texto siempre llama a la entrada en la PLT para la función a la que desea llamar (esto es, al *trampolín*).
- ▶ La primera vez que se llama a la entrada de la PLT para una función, se llama al enlazador dinámico para que resuelva el símbolo y *parchee* la entrada de la GOT. Después se llama a la función.
- ▶ En sucesivas llamadas, la GOT ya tiene *parcheada* la entrada y el trampolín salta a la función destino directamente.

Lazy binding†

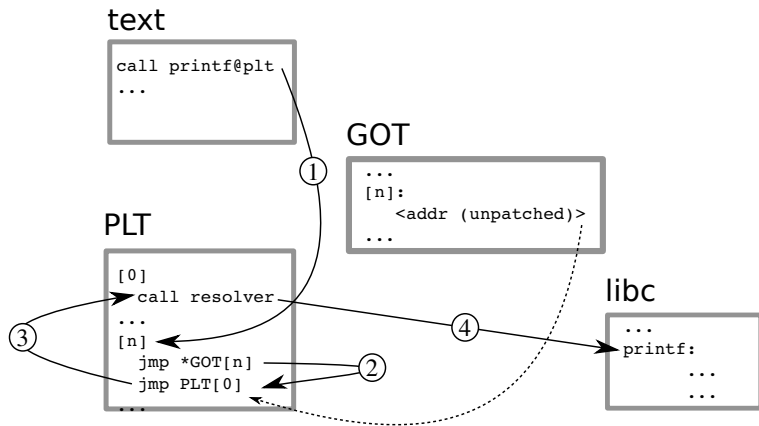
¿Cómo funciona? Supongamos que el proceso llama a `printf` **por primera vez**:

1. El texto en realidad llama a la función `printf@plt`.
2. La función `printf@plt` salta a la dirección de la entrada de `printf` en la GOT. Dicha entrada de la GOT está sin parchear, y está inicializada a la dirección siguiente instrucción (del propio código de `printf@plt`).
3. Dicha función llama al enlazador dinámico para resolver el símbolo y parchear la entrada de la GOT. Pasa como argumento el índice del símbolo en la tabla.
4. Después de resolver, se salta a la función `printf`.

Las llamadas **posteriores** saltarán, en el paso 2, a la función `printf` de la `libc`, porque la GOT ya está *parcheada*.

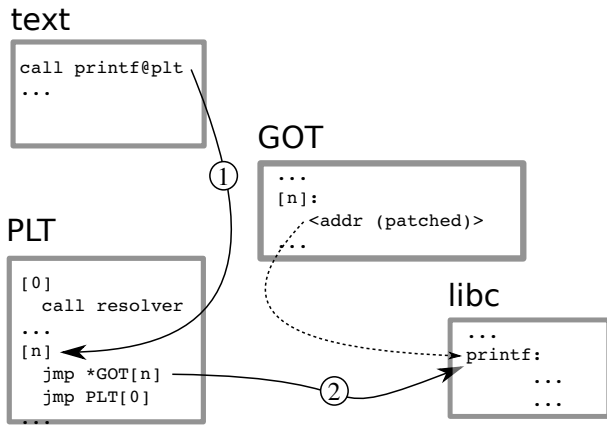
Lazy binding†

Primera llamada:



Lazy binding†

Siguientes llamadas:



- ▶ El comando `ldd` muestra las bibliotecas dinámicas que necesita un programa, junto con la dirección de memoria donde serán cargadas.
- ▶ Una biblioteca puede cargarse en la misma dirección de memoria para todos los procesos que la utilizan, pero no siempre es así. Si tenemos PIC (código independiente de posición), no ocurrirá de esa forma.

¿De dónde se saca la biblioteca? Depende del sistema y de su configuración. Por ejemplo:

1. Directorios de la variable de entorno `LD_PRELOAD`
2. Si el símbolo contiene barras ("`/`"), se carga de esa ruta (relativa o absoluta)
3. Directorios de la sección `DT_RPATH` del binario o del atributo `DT_RUNPATH` del binario
4. Directorios de la variable de entorno `LD_LIBRARY_PATH`
5. En el directorio `/lib`
6. En el directorio `/usr/lib`

Muerte de un proceso

- ▶ Cuando `main` retorna, la función que la llamó realiza una llamada al sistema `exit` para acabar. Se puede llamar desde cualquier parte del programa para terminar:
- ▶ El sistema terminará con el proceso y liberará los recursos asociados.
- ▶ El parámetro que se le pasa a `exit` determina el estado del proceso al acabar:
 - ▶ éxito (`EXIT_SUCCESS`, valor 0)
 - ▶ fallo (`EXIT_FAILURE`, valor distinto de 0).


```
void exit(int status);
```

Muerte de un proceso

- ▶ `err`: escribe el error de la última llamada concatenado después de la cadena con formato que se le pasa, y después termina el proceso con el estatus indicado.
- ▶ `errx`: escribe un error con la cadena con formato que se le pasa, y termina el proceso con el estatus indicado.

```
void err(int eval, const char *fmt, ...);  
void errx(int eval, const char *fmt, ...);
```

- ▶ Las variables de entorno son strings.
- ▶ En Linux y en general, se almacenan en área de usuario³.
- ▶ El proceso hereda de su creador una **copia** de sus variables de entorno.
- ▶ El tercer parámetro de `main` es un array de cadenas con las variables de entorno, terminado en `NULL`.
- ▶ La `libc` mantiene una variable global llamada `environ` que apunta a las variables de entorno.
- ▶ Al crear el proceso, se meten en la pila. Si se definen nuevas variables de entorno durante su vida, se mueven al heap.

³En algunos sistemas se almacenan en área de kernel, p. ej. Plan 9. 

- ▶ `getenv`: devuelve el valor de una variable de entorno. No devuelve memoria dinámica.
- ▶ `setenv`: escribe el valor de una variable de entorno. Si existe, la puede sobrescribir o no (tercer parámetro). Internamente, hace una copia de la cadena.
- ▶ `unsetenv`: elimina una variable de entorno.

```
char * getenv(const char *name);  
int  setenv(const char *name, const char *val, int o);  
int  unsetenv(const char *name);
```

Propiedades

- ▶ El kernel mantiene la información de los procesos en tabla de procesos.
- ▶ Para cada proceso, mantiene una estructura de datos con:
 - ▶ PID: número que identifica el proceso.
 - ▶ PPID (PID de su creador)
 - ▶ Estado
 - ▶ Prioridad
 - ▶ Registros (para cambios de contexto).
 - ▶ Directorio de trabajo
 - ▶ UID, GID: credenciales del proceso (usuario y grupo).
 - ▶ Segmentos: text, data, bss, stack
 - ▶ Descriptores de fichero (ficheros abiertos)
 - ▶ ...
- ▶ El comando `ps` nos da información sobre los procesos.

- ▶ En linux podemos inspeccionar las propiedades de los procesos en `/proc`⁴.
- ▶ Hay un directorio por PID. Dentro tenemos ficheros con información sobre cada proceso. Ejemplos:
 - ▶ `stat`: propiedades del proceso (PID, PPID, estado, ...).
 - ▶ `cmline`: línea que se usó para ejecutar.
 - ▶ `exe`: el programa que está ejecutando.
 - ▶ `fd`, `fdinfo`: información sobre ficheros abiertos.
 - ▶ `maps`: regiones de memoria del proceso.
 - ▶ `mem`: memoria del proceso.
 - ▶ ...

Credenciales

- ▶ El proceso ejecuta a nombre de un usuario y un grupo.
- ▶ El sistema aplica el **control de acceso** en base a las credenciales del proceso.
- ▶ La información sobre usuarios y grupos se almacena en los ficheros `/etc/passwd`, `/etc/shadow` y `/etc/group`.
- ▶ El UID es un número que identifica a un usuario. El GID identifica a un grupo. Un usuario puede pertenecer a múltiples grupos.
- ▶ Durante la vida del proceso las credenciales pueden cambiar.

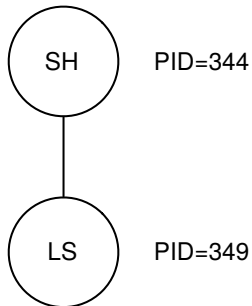
Llamadas al sistema

- ▶ getpid: devuelve el PID del proceso.
- ▶ getppid: devuelve el PID del creador.
- ▶ getuid: devuelve el UID.
- ▶ getgid: devuelve el GID.
- ▶ getcwd: devuelve el directorio de trabajo. Lo copia en el buffer que pasamos. Si buff es NULL, reserva memoria dinámica.
- ▶ chdir: cambia el directorio de trabajo. Retorna -1 en error.

```
pid_t getpid(void);  
pid_t getppid(void);  
gid_t getgid(void);  
uid_t getuid(void);  
char * getcwd(char *buf, size_t size);  
int chdir(const char *path);
```

Hasta ahora...

- ... sólo hemos creado procesos con la shell.
P. ej.
\$> ls



Llamadas al sistema

- ▶ Una para crear un proceso (fork) y otra para ejecutar un programa (exec).
- ▶ Podemos crear un nuevo flujo de control, para ejecutar el mismo programa o para ejecutar otro.
- ▶ Podemos configurar el nuevo proceso antes de que ejecute otro programa.
- ▶ Se podría hacer todo en una función, pero tendría demasiados parámetros.

Fork

- ▶ `fork` crea un nuevo proceso. Se crea un proceso con nuevo PID. En el padre, retorna el PID del hijo. En el hijo, la llamada retorna 0. En caso de error, retorna -1.
- ▶ En Linux, la función `fork` en realidad realiza la llamada al sistema `clone`.

```
int fork(void);
```

- ▶ El hijo es un clon exacto del padre.
- ▶ Padre e hijo ejecutan concurrentemente.
- ▶ El hijo es totalmente independiente del padre: abstracción de proceso.
- ▶ Se “*copia*” la memoria: TEXT, DATA, STACK. ¿Qué valor tienen las variables?

Condiciones de carrera

- ▶ No se sabe el orden en el que se van a ejecutar las instrucciones de los dos procesos.
- ▶ Cuando el resultado final depende de la carrera entre los dos procesos: condición de carrera.
- ▶ En este caso, los procesos no comparten memoria, pero sí comparten otros recursos del sistema (p. ej.: ficheros).
- ▶ Programación concurrente: evitar condiciones de carrera.

- ▶ `execv`: ejecuta un programa. `path` es la ruta del fichero ejecutable (binario o interpretado) que queremos ejecutar. `argv[]` es un array de strings con los argumentos para el programa, que tiene que terminar en un `NULL`. El primer elemento del array es el nombre del programa.

```
int execv(const char *path, char *const argv[]);
```

- ▶ `exec1`: similar, pero pasando los argumentos de otra forma. Ideal para cuando se saben los argumentos de antemano. El último argumento tiene que ser siempre `NULL`.

```
int exec1(const char *path, const char *arg, ...);
```

- ▶ Las dos funciones juntas nos permiten crear procesos para ejecutar cualquier programa.
- ▶ Desde `init`, todos los procesos se crean así.

Wait

- ▶ `wait`: retorna cada vez que un hijo cambia de estado. Es la forma de esperar a que terminen los hijos. No notifica el cambio de estado de los procesos nietos, etc.
- ▶ Retorna -1 si no hay hijos por los que esperar o en caso de ser interrumpida. En otro caso, retorna el PID del hijo.
- ▶ Hay que usar las macros descritas en la página de manual para discriminar entre los estados y conseguir el status de salida del hijo.

```
pid_t wait(int *wstatus);
```

Macros para ver qué ha pasado:

- ▶ `WIFEXITED(status)`
Verdadero si el proceso ha terminado llamando a `exit()`.
- ▶ `WEXITSTATUS(status)`
Evalúa a los 8 bits que describen el estado de salida del proceso (valor que el hijo pasó a `exit()`).
- ▶ `WIFSIGNALED(status)`
Verdadero si el proceso ha terminado por una señal.
- ▶ `WIFSTOPPED(status)`
Verdadero si el proceso no ha terminado pero ha sido parado.

Waitpid

- ▶ `waitpid`: espera hasta que el proceso con el PID indicado en el primer argumento acabe.

```
pid_t waitpid(pid_t pid, int *wstatus, int op);
```

Procesos muertos

- ▶ Cuando un proceso muere, no se puede eliminar su estructura de la tabla de procesos hasta que el padre recoja su estatus.
- ▶ Un proceso muerto pendiente de que se recoja su status es un *zombie*.
- ▶ Si el padre muere antes que los hijos, estos pasan a ser huérfanos (*orphan*).
- ▶ Alguien se tiene que hacer cargo de esperar por los huérfanos. Por eso, los huérfanos pasan a ser hijos de *init*.
- ▶ *init* llama a *wait* por todos sus hijos para evitar que se queden *zombies* para siempre.



En la shell

- ▶ comando `&`
el comando se ejecuta de forma asíncrona (segundo plano o *background*), la shell no espera a que termine ese comando para continuar.
- ▶ `$$`
variable con el PID de la shell.
- ▶ `$!`
variable con el PID del último comando ejecutado en segundo plano.
- ▶ `$?`
variable con el estatus del último comando.

▶ &&

se ejecuta el segundo comando si el primero termina con éxito.

Ejemplo:

```
test -f /tmp/a && echo el fichero existe
```

▶ ||

se ejecuta el segundo comando si el primero termina con fallo.

Ejemplo:

```
test -f /tmp/a || echo el fichero no existe
```

En la shell: Job Control

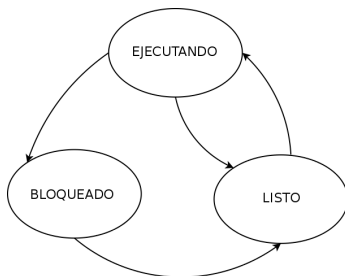
Cada comando que está ejecutando es un trabajo (*job*):

- ▶ `jobs`
Lista los trabajos actuales.
- ▶ `Ctrl + c`
Interrumpe el trabajo que está en primer plano.
- ▶ `Ctrl + z`
Para el trabajo que está en primer plano.
- ▶ `bg`
Reanuda, en segundo plano, un trabajo parado.
- ▶ `fg`
Reanuda, en primer plano, un trabajo parado.

Planificación

Para la planificación, podemos resumir los posibles estados de un proceso en:

- ▶ Ejecutando
- ▶ Listo para ejecutar
- ▶ Bloqueado



- ▶ El **planificador (scheduler)** del kernel se encarga de gestionar qué proceso ejecuta en la CPU (o CPUs).
- ▶ **Política vs Mecanismo**
- ▶ **Concurrencia vs paralelismo.**

► Cambio de contexto (simplificado)

1. Salvar el estado de los registros del procesador en la estructura de datos que representa al proceso en el kernel.
2. Cargar el estado de los registros del proceso entrante en el procesador El contador de programa al final.

Los **cambios de contexto** son costosos.

Planificación: políticas

- ▶ Ya sé sacar un proceso y meter otro... ¿Cómo reparto las CPUs?
- ▶ Los procesos se pueden ver como ráfagas de operaciones de CPUs y operaciones de entrada/salida.
- ▶ Algunos procesos están dominados por CPU y otros procesos están dominados por entrada/salida.

Criterios, ¡no se puede todo a la vez!

- ▶ **Justicia (fairness):** todos los procesos tienen su tiempo de CPU.
- ▶ **Eficiencia (efficiency):** sacar máximo partido a la CPU.
- ▶ **Respuesta interactiva:** es importante el tiempo de espera en la cola hasta que se empieza con él.
- ▶ **Respuesta (turnaround):** tiempo de entrega del resultado de un cómputo.
- ▶ **Rendimiento (throughput):** número de trabajos acabados por unidad de tiempo.

La planificación puede ser:

- ▶ No expulsiva (colaborativa, non-preemptive): el proceso abandona la CPU porque ha terminado su ráfaga de CPU y se bloquea haciendo E/S, o por decisión propia.
- ▶ Expulsiva (preemptive): el planificador puede expulsar a los procesos cuando lo decida, aunque no hayan terminado su ráfaga de CPU.

FCFS: First Come First Serve

- ▶ Los procesos listos para ejecutar se ponen en un FIFO.
- ▶ Efecto *convoy*: llega un proceso dominado por CPU y la acapara. Deja esperando mucho tiempo a los procesos dominados por E/S que están listos para ejecutar ráfagas cortas de CPU → se malgasta capacidad de E/S.
- ▶ Tiempo promedio de espera alto.
- ▶ Sencillo y fácil de implementar.

Planificación no expulsiva: SJF

SJF: Shortest Job First.

- ▶ En la cola tienen prioridad los procesos con ráfaga más corta.
- ▶ Tenemos que saber el volumen del trabajo de antemano.
- ▶ Mejora el tiempo de espera promedio.
- ▶ Problema: hambruna. ¿Qué pasa con las ráfagas largas?

SRTF: Shortest Remaining Time First.

- ▶ Es como SJF, pero expulsivo.
- ▶ Cuando entra en la cola de listos para ejecutar un proceso con una ráfaga de CPU más corta que lo que le queda al proceso actual, se expulsa al actual.
- ▶ Tiene el mismo problema: hambruna.

Round Robin:

- ▶ Se asigna la CPU en *cuantos*: tiempo máximo que el proceso puede estar usando la CPU.
- ▶ Se rota por los procesos que están listos para ejecutar.
- ▶ Cuando se agota el *cuanto*, se expulsa al proceso.
- ▶ El proceso puede dejar la CPU antes de que acabe su *cuanto* (p. ej. si se queda bloqueado).
- ▶ Los procesos nuevos entran por el final de la cola.

Planificación de procesos: Round-Robin

Pros y contras:

- ▶ Aumenta la respuesta interactiva.
- ▶ Reduce el rendimiento (throughput) por los cambios de contexto.

Problema: ¿Cómo se elige el *cuanto*?

- ▶ Si es pequeño, se desperdicia mucha CPU en los cambios de contexto.
- ▶ Si es grande, las aplicaciones interactivas sufren.

Planificación de procesos: prioridades

- ▶ No todos los procesos tienen la misma importancia.
 - ▶ P. ej. un reproductor de video vs. un cliente de correo.
- ▶ En general, la prioridad pueden ser **estática** o **dinámica**.
- ▶ Problema: inanición (*starvation*). Solución: tener en cuenta la edad del proceso (*aging*).

Planificación de procesos: colas multinivel con retroalimentación

En general:

- ▶ Número de colas.
- ▶ Algoritmo para cada cola.
- ▶ Método para subir el proceso a una cola de mayor prioridad.
- ▶ Método para bajar el proceso a una cola de menor prioridad.
- ▶ Método para determinar la cola en la que empieza un proceso.

Planificación de procesos: colas multinivel con retroalimentación

Ejemplo particular: Round-Robin con prioridades dinámicas.

- ▶ Múltiples colas, según prioridad.
- ▶ R-R con los procesos de cada cola.
- ▶ Si hay procesos listos en una cola, no se atienden las colas de menor prioridad.
- ▶ Cuando un proceso agota su *cuanto*, se baja su prioridad.
- ▶ Cuando un proceso no agota su *cuanto*, se sube su prioridad.

Sirven para representar la planificación. Por ejemplo:



Planificador para multiprocesadores

Cuando tenemos múltiples CPUs, el planificador debe especializarse para el tipo de arquitectura:

- ▶ SMP (Symmetric Multi-Processor): todas las CPUs son iguales y el acceso a las partes de la memoria física es el mismo para todas (UMA).
- ▶ NUMA (Non-Uniform Memory Access): algunas partes de la memoria y ciertos dispositivos están más cerca de unas CPUs que de otras.
- ▶ CPUs heterogéneas: algunas CPUs son menos potentes que otras o con distintas ISA que ofrecen instrucciones más apropiadas para ciertos trabajos (CPUs, GPUs, etc.).

Prioridades en Linux

- ▶ La prioridad (*nice*) va del valor -20 (máxima prioridad) al 19 (mínima prioridad).
- ▶ La prioridad se hereda del proceso padre.
- ▶ El comando `nice` sirve para ejecutar un programa indicando el nivel de prioridad. P. ej.:

```
nice -n 5 $HOME/myprogram
```

- ▶ El comando `renice` sirve para cambiar la prioridad de un proceso. P. ej.:

```
renice -5 5232
```