
UNIVERSIDAD
POLITÉCNICA
DE MADRID

# SISTEMAS OPERATIVOS
# PROCESOS

**Pedro de Miguel Anasagasti**
**Víctor Robles Forcada**
**María S. Pérez Hernández**
**Raquel Cedazo León**

# ÍNDICE


- Conceptos generales de procesos
- Multitarea
- Implementación de procesos
- Servicios UNIX de gestión de procesos
- Señales y temporizadores
- Servicios UNIX de señales y temporizadores
- Servidores y demonios
- Procesos ligeros
- Servicios UNIX de procesos ligeros

# PROCESOS
# CONCEPTOS GENERALES

# CONCEPTO DE PROCESO


- Definición simplificada de proceso: programa en ejecución
- SO hace creer a cada programa en ejecución que tiene su propia máquina:
  - Proceso: abstracción de un procesador (de una máquina)
  - Contexto de ejecución independiente y estanco
    - Excepto por mecanismos de comunicación proporcionados por SO
- Proceso vs programa: programa (pasivo)!= proceso (activo)
  - Múltiples procesos ejecutando el mismo programa
- Definición más precisa de proceso: unidad de procesamiento gestionada por el SO
- En mayoría de SSOO: proceso asociado a programa "de por vida"
  - Windows CreateProcess: Nuevo proceso – nuevo programa
- En UNIX, proceso puede ejecutar varios programas durante su vida
  - Fork: nuevo proc – mismo prog
  - Exec: Mismo proc – nuevo prog

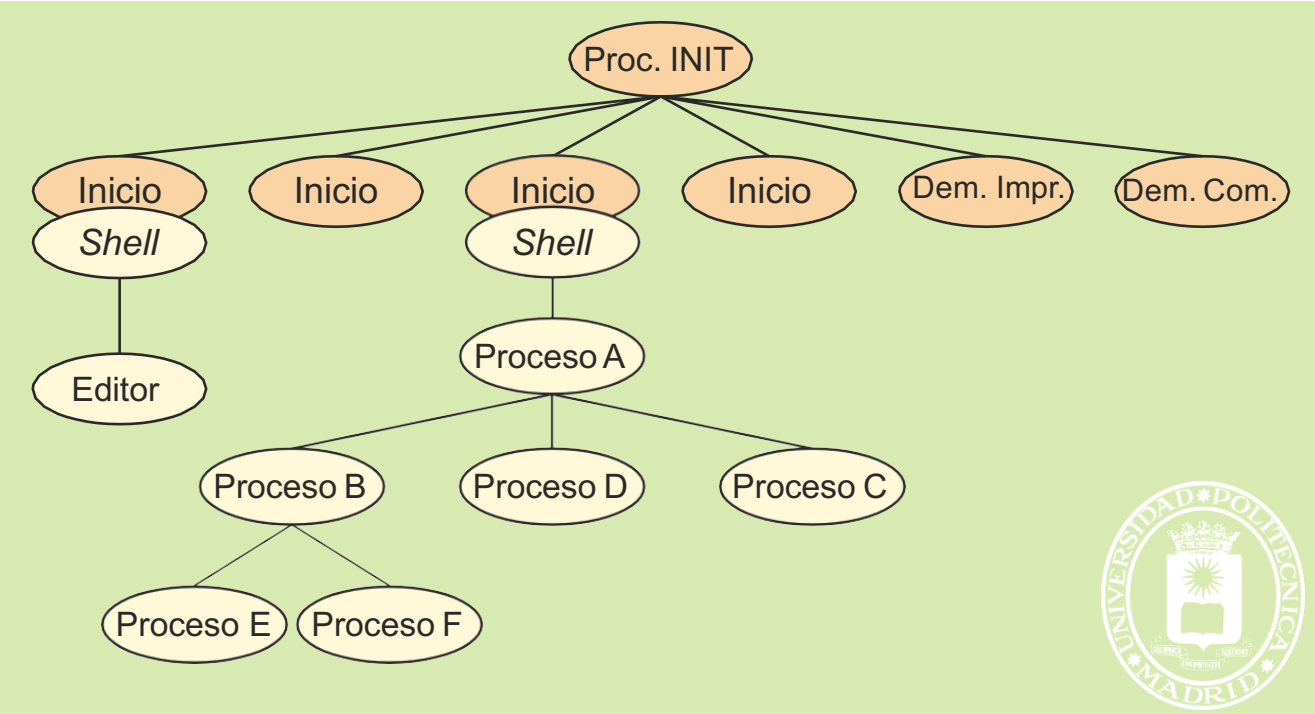
# JERARQUÍA DE PROCESOS

**Familia de procesos**

- Proceso hijo.
  - Proceso padre.
  - (Proceso hermano).
- (Proceso abuelo).

**Vida de un proceso**

  - Crea.
  - Ejecuta
- Muere o termina.




**Ejecución del proceso**

- No interactivo (Batch y segundo plano)
- Interactivo o primer plano.

**Grupo de procesos**

  - Grupos de procesos dependientes de cada inicio.

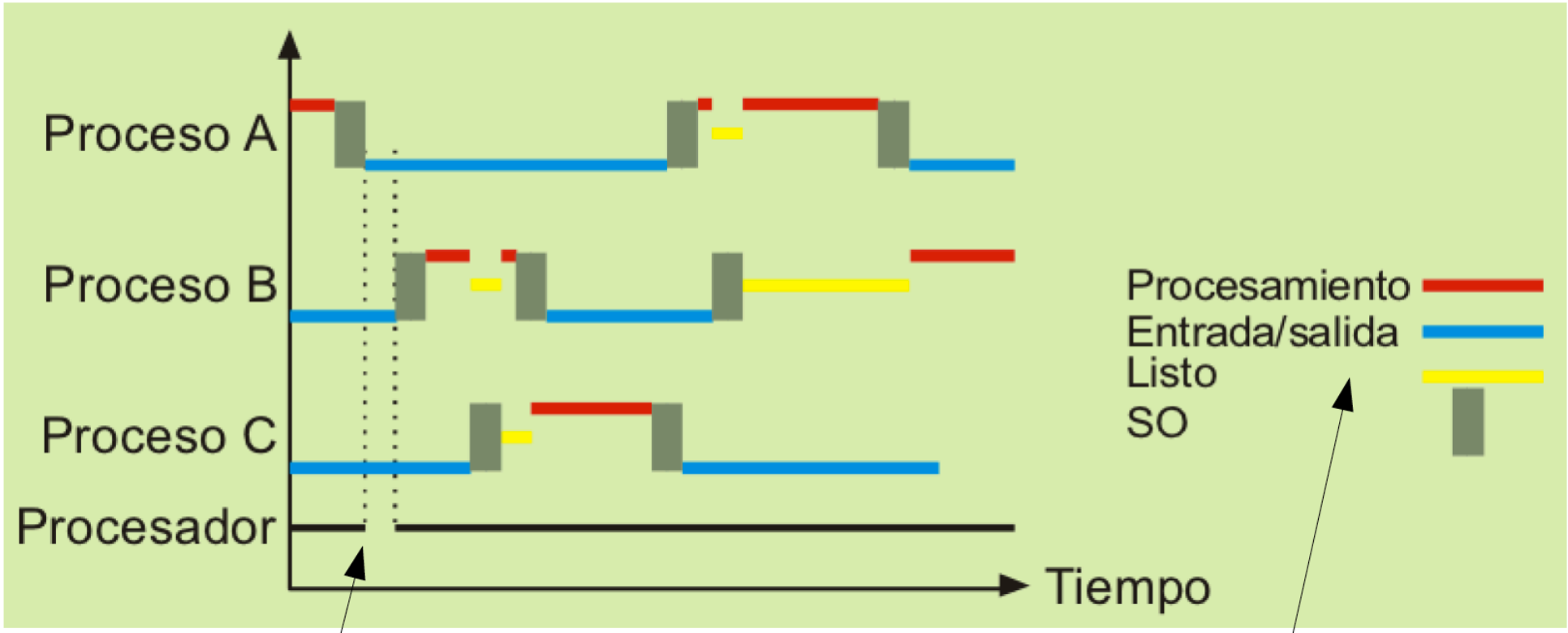
# ENTORNO DE PROCESO

- **Definición:**
  - **Tabla NOMBRE-VALOR que se pasa al proceso en su creación**
- **Ubicación:**
  - **Pila del proceso**
- **Servicios del SO para leer, modificar, añadir o eliminar variables de entorno**
- **Ejemplo de uso de las variables de entorno desde la interfaz en sistemas UNIX**
  - **Listar variables de entorno:**
    - **env**
    - **Ejemplo: echo $PATH**
  - **Modificar variables de entorno:**
    - **Ejemplo: $PATH = $PATH:XXXX**

HOME, directorio de trabajo inicial del usuario.
LOGNAME, nombre del usuario asociado a un proceso.
PATH, prefijo de directorios para encontrar ejecutables.
TERM, tipo de terminal.
TZ, información de la zona horaria

# MULTITAREA

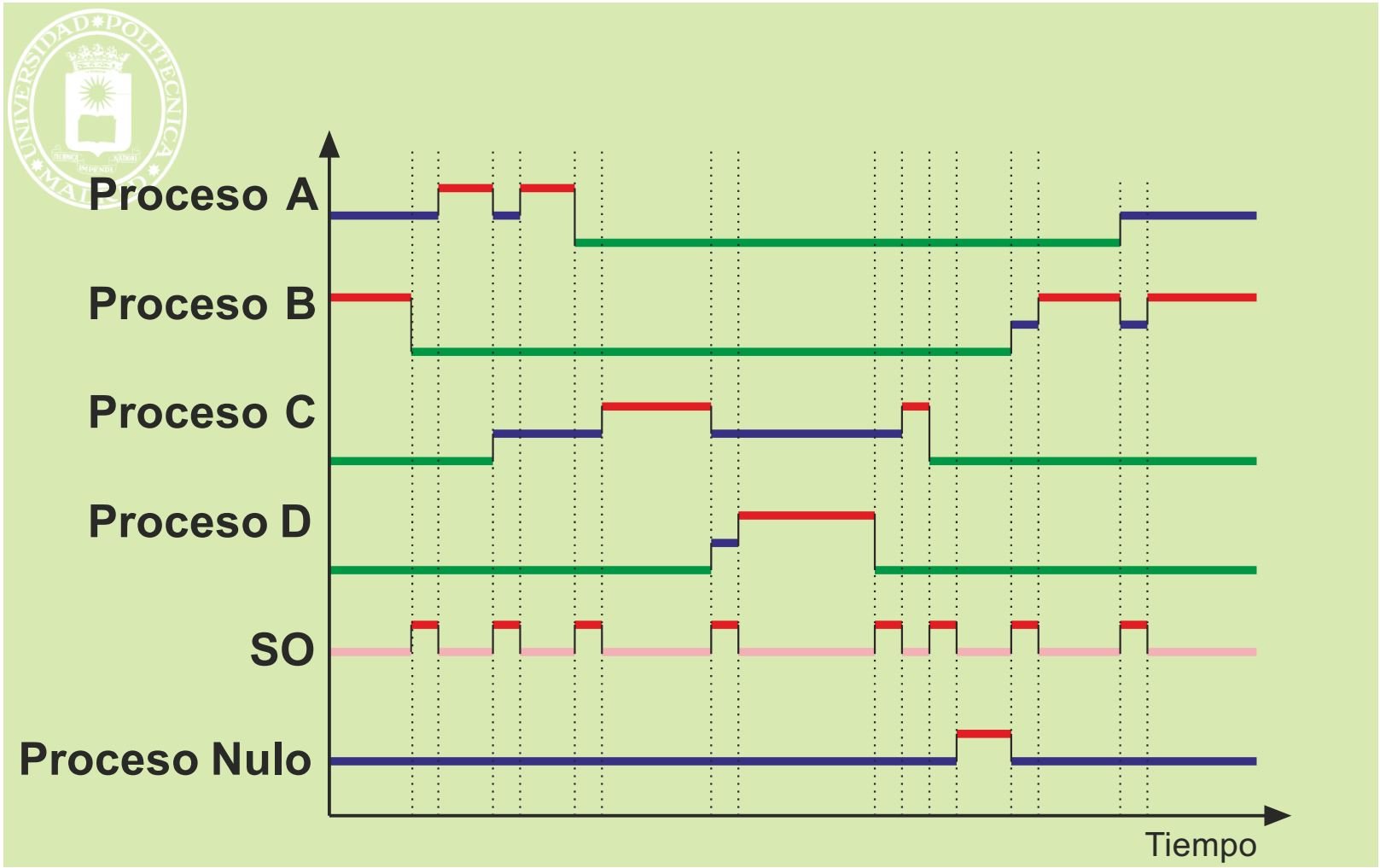
# CONCEPTO DE MULTITAREA




**Proceso nulo**

**Un proceso puede estar en diferentes estados: ejecución, listo y bloqueado**

# CONCEPTO DE MULTITAREA




**Proceso nulo**

# CONCEPTO DE MULTITAREA

## Ventajas de la multiprogramación

- Facilita la programación, dividiendo los programas en procesos(modularidad).
- Permite el servicio interactivo simultáneo de varios usuarios de forma eficiente.
- Aprovecha los tiempos que los procesos pasan esperando a que secompleten sus operaciones de E/S.
- Aumenta el uso de la CPU.

## Grado de multiprogramación

- Nº de procesos en memoria (procesos activos).

## Necesidades de memoria

- Sistema sin memoria virtual. La memoria principal debe tener capacidad para almacenar el SO y todos los procesos.
- Sistema con memoria virtual. Los marcos de página disponibles se reparten entre el SO y los procesos.

# IMPLEMENTACIÓN DE PROCESOS

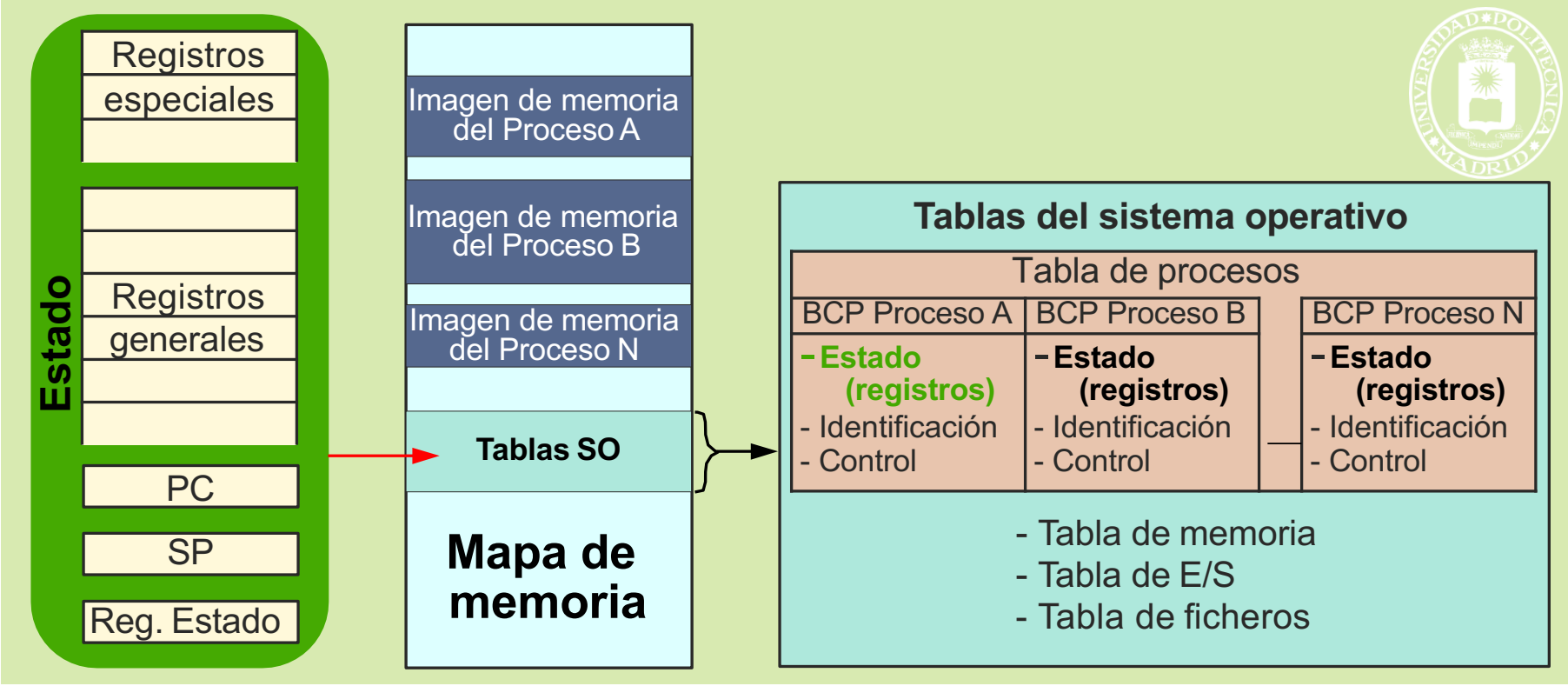
# MULTIPLEXACIÓN DEL PROCESADOR

- **SO reparte N procesadores entre M procesos existentes (M>N habitualmente)**
  - SO va asignando y revocando uso del procesador a cada proceso
- **SO debe asegurar que cuando reanuda la ejecución un proceso está "instalado" el contexto de ese proceso:**
  - Registros del procesador tal como los dejó en su última ejecución
  - Sus ficheros abiertos, su directorio actual, etc.
  - Imagen de memoria del proceso instalada
  - Recuerde: dirección X de proceso 1 != dirección X de proceso 2
- **SO debe almacenar y gestionar la información de cada proceso**
  - Contexto de un proceso

# CONTEXTO DEL PROCESO

**La información se organiza en tres grupos:**

- **Estado del procesador**
- **Imagen (mapa) de memoria**
- **Tablas del SO**



# ESTADO DEL PROCESADOR

- Está formado por el contenido de todos los registros del procesador
- Puede residir en:
  - Registros del procesador, cuando el proceso está en ejecución
  - En el BCP, cuando el proceso no está en ejecución
- Al bloquear o expulsar un proceso, el SO copia el estado del procesador en su BCP correspondiente
  - Realizado por la rutina de tratamiento de interrupciones
- Recordatorio: Ejecución del SO dirigida por eventos
  - Una vez activado el primer proceso, sólo ejecuta el SO cuando se produce un evento

# INFORMACIÓN DEL BCP

## Información de identificación

- pid del proceso.
- pid del padre.
- Identificador de usuario (UNIX: real: uid real; efectivo: uid efectivo)
- Identificador de grupo (UNIX: real: gid real; efectivo: gid efectivo)
- Identificadores de grupos de procesos (el proceso pertenece a uno o más grupos de procesos)

## Estado del Procesador

- Contiene los valores iniciales del estado del procesador o su valor en el instante en que fue interrumpido el proceso

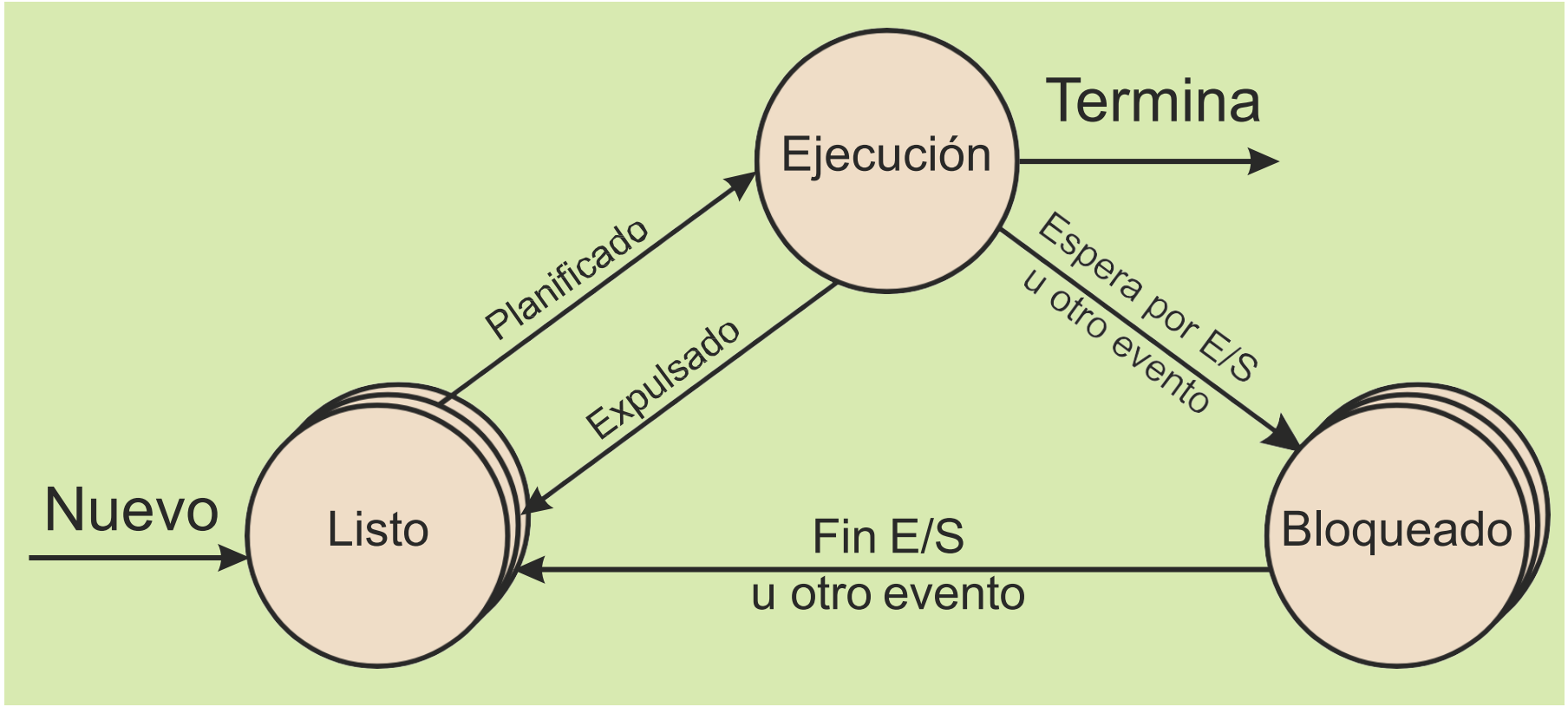
# INFORMACIÓN DEL BCP

## Información de control del proceso

- Información de planificación y estado.
  - Estado del proceso (bloqueado, listo o en ejecución).
  - Evento por el que espera el proceso cuando está bloqueado.
  - Información de planificación: prioridad y tiempo en espera.
- Descripción de las regiones de memoria asignadas al proceso.
- Recursos asignados, tales como:
  - Ficheros abiertos (tabla de descriptores)
  - Directorio actual, directorio raíz, (UNIX) máscara de ficheros
  - Puertos asignados
  - Recursos de sincronización (semáforos, cerrojos, etc.)
- Comunicación entre procesos. Espacio para almacenar señales y algún mensaje enviado al proceso.

# ESTADOS DEL PROCESO

- En **ejecución**: uno por procesador.
- **Bloqueado**: en espera de E/S o evento (p.e. pause).
- **Listo** para ejecutar.




**Planificador**: Módulo del SO que decide qué proceso listo se ejecuta.

**Un proceso es expulsado porque ha gastado un cierto tiempo de procesador o porque surge otro proceso más prioritario listo para ejecutar.**

# INFORMACIÓN DEL BCP

## Información de control del proceso

- Señales (UNIX)
  - Señales armadas
  - Máscara de señales
- Temporizador.
- Información de contabilidad (uso de recursos)
  - Tiempo de procesador consumido
  - Operaciones de E/S realizadas
  - Límites en el uso de recursos

# INFORMACIÓN FUERA DEL BCP

**Justificación:**

- **Por razones de implementación (eficiencia).**
- **Para compartirla con otros procesos.**

**Tabla de páginas:**

- **Describe la imagen de memoria del proceso.**
- **Tamaño muy variable.**
- **El BCP contiene el puntero a la tabla de páginas.**
- **Para compartir memoria se requiere que la tabla sea externa al BCP.**

**Punteros de posición de los ficheros:**

- **Si se añaden a la tabla de descriptores (en el BCP) no se pueden compartir.**
- **Si se asocian al nodo-i se comparte siempre.**
- **Se ponen en una estructura común a los procesos (tabla intermedia) y se asigna un nuevo puntero en cada servicio OPEN**

# CAMBIO DE CONTEXTO

## Cambio de contexto

- Se pasa de ejecutar el proceso A a ejecutar el proceso B
- Requiere dos cambios de modo: de proceso A al SO y del SO al proceso B

## Interrupción | Excepción | Llamada → Cambio de modo: de proceso a SO

- Tratamiento de ese evento requiere bloqueo o expulsión de proceso A
- Necesidad de salvar el estado del proceso A en BCP. Ejemplo:

```
LD      .5,#CANT
```

⟶ llega una interrupción y se pasa al SO.

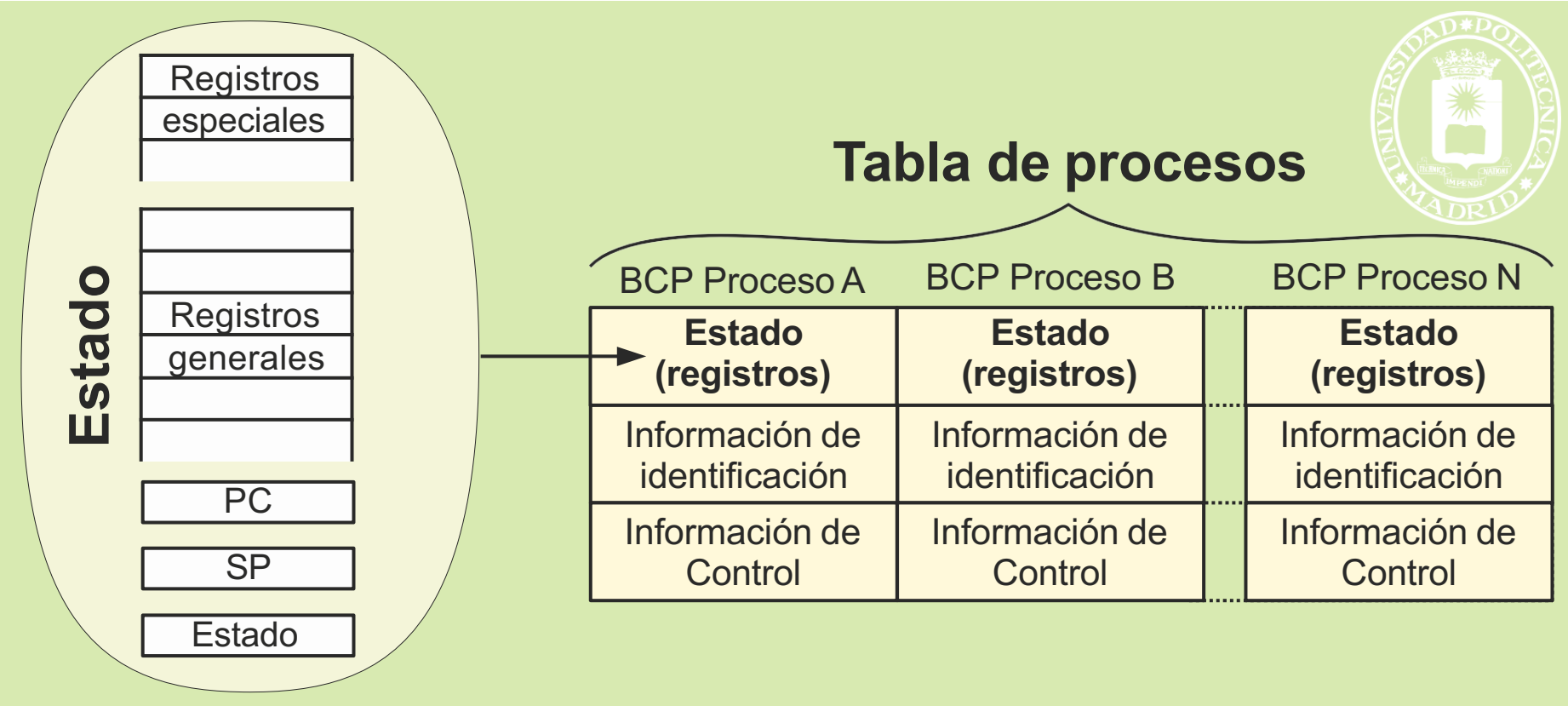
```
LD      .1,[.5]
```

¿Qué valor tiene el registro 5 al reiniciar la ejecución del proceso?

# CAMBIO DE CONTEXTO

## Activación → Cambio de modo: de SO a proceso

- **Se pasa de ejecutar el SO a ejecutar el proceso B**
- **Hay que restaurar el estado del proceso B que está almacenado en el BCP**



# OPERACIONES SOBRE LOS PROCESOS

## Operación de creación de un proceso

- **UNIX: fork(): Nuevo proceso – mismo programa; exec(): mismo proceso- nuevo programa**

## Operación de terminación de un proceso

- **Proceso usa una llamada para terminar (UNIX: _exit())**
- **Parámetro con valor de terminación**

## Operación de esperar fin de otro proceso y recoger valor de terminación

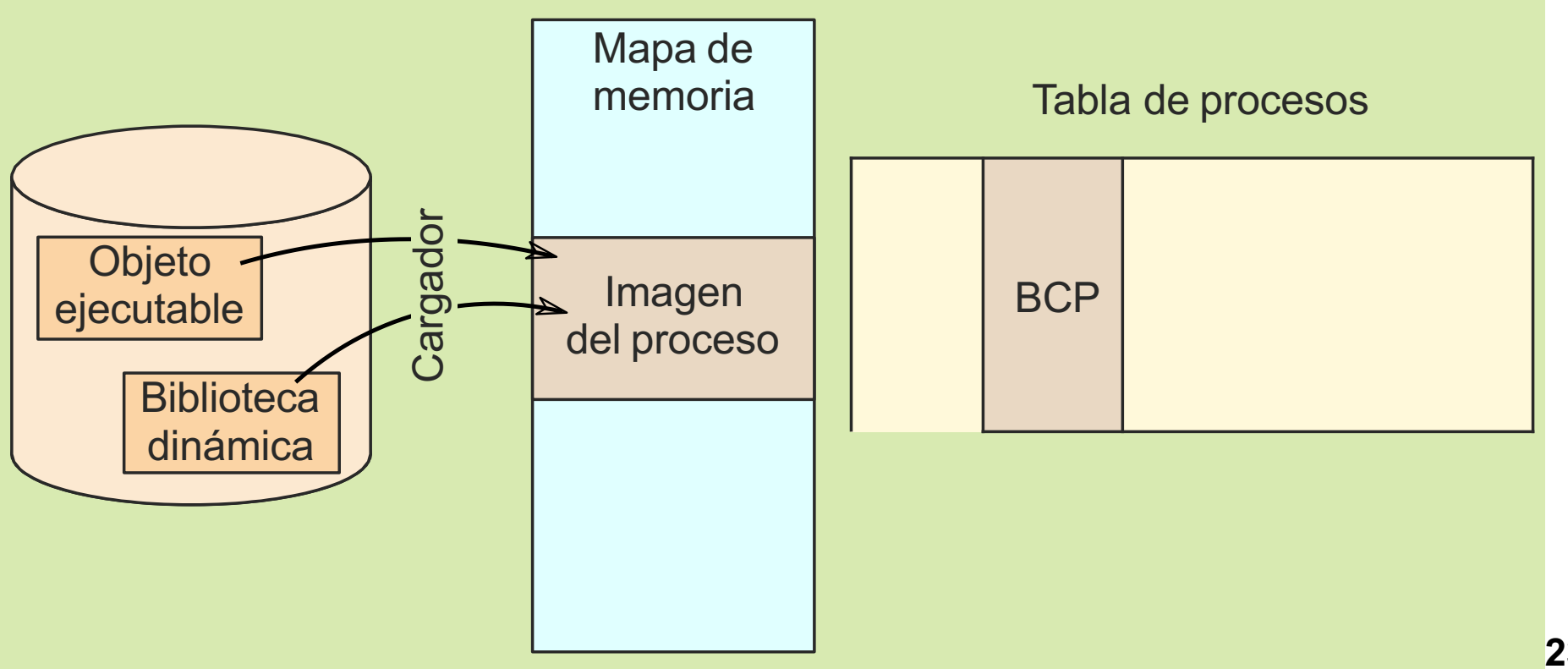
- **UNIX: wait() realizado por el proceso padre**

## Operación de abortar otro proceso

- **En UNIX integrada en el mecanismo de señales**

# OPERACIÓN DE CREACIÓN DEL PROCESO

- Crear la imagen de memoria
- Seleccionar BCP libre
- Rellenar el BCP
- Cargar segmento de texto y segmento de datos
- Crear la pila inicial, en el segmento de pila, con el entorno del proceso y los parámetros de invocación



# VIDA DEL PROCESO

**Proceso padre lo crea**

- **Se inicia en estado listo**

**Activación del proceso (paso a ejecución)**

- **Proceso en ejecución se bloquea, expulsa o termina**
- **Lo selecciona el planificador y se restituye el estado del procesador**

**Proceso en ejecución pasa a bloqueado (se salva el estado del procesador)**

- **Proceso solicita llamada bloqueante o excepción por fallo de página**

**Proceso en ejecución pasa a listo (se salva el estado del procesador)**

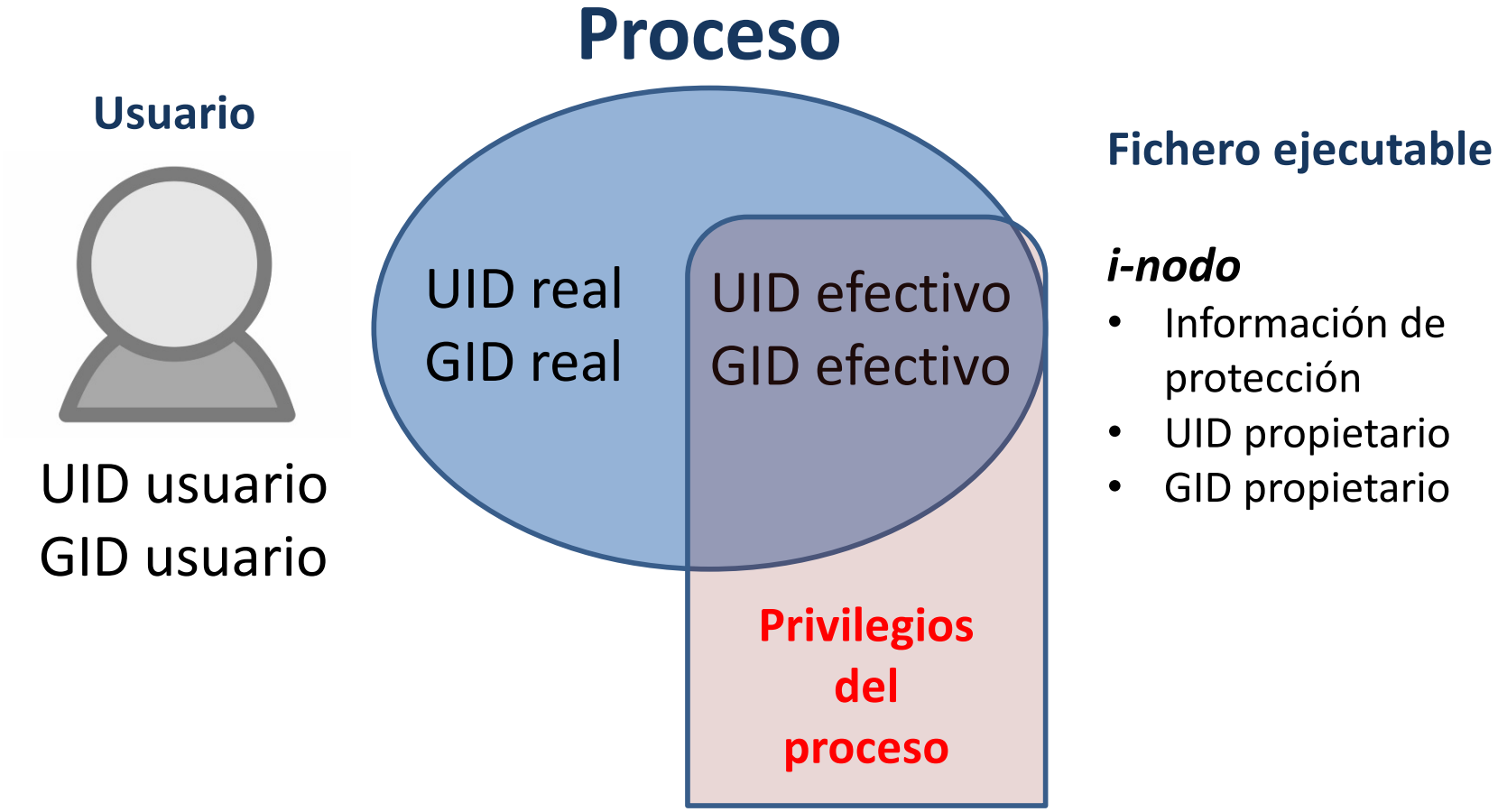
- **Llamada al sistema o interrupción provoca expulsión**
  - **Ha gastado su turno o surge otro proceso más prioritario listo**

## Terminación del proceso

- Tipos de terminación
  - Voluntaria: invoca llamada al sistema para tal fin (o fin del programa)
  - Involuntaria: Error de ejecución (excepción) o abortado por un usuario u otro proceso
- En cualquier caso, se recuperan los recursos asignados al proceso
  - Si la asignación es exclusiva, se libera el recurso
  - Si la asignación es compartida el SO llevará un contador de usuarios. Cuando el contador llega a 0 se libera el recurso

# SERVICIOS UNIX
# GESTIÓN DE PROCESOS

# IDENTIFICADORES DE SEGURIDAD DEL PROCESO




| | |
|---|---|
| UID real = UID usuario | UID efectivo = UID real |
| GID real = GID usuario | GID efectivo = GID real |

# IDENTIFICADORES DE SEGURIDAD DEL PROCESO II

**Fichero ejecutable**
**SETUID activo**

| | |
|---|---|
| UID real = UID usuario | UID efectivo = UID propietario |
| GID real = GID usuario | GID efectivo = GID real |

***i-nodo***

- Información de protección
- UID propietario
- GID propietario

Añadir permisos SETUID
$ chmod u+s /home/usuario/fich_ejecutable
Eliminar permisos SETUID
$ chmod u-s /home/usuario/fich_ejecutable

**Fichero ejecutable**
**SETGID activo**

| | |
|---|---|
| UID real = UID usuario | UID efectivo = UID real |
| GID real = GID usuario | GID efectivo = GID propietario |

***i-nodo***

- Información de protección
- UID propietario
- GID propietario

Añadir permisos SETGID
$ chmod g+s /home/usuario/fich_ejecutable
Eliminar permisos SETGID
$ chmod g-s /home/usuario/fich_ejecutable

# SERVICIOS UNIX DE GESTIÓN DE PROCESOS (Identificación)


- `pid_t getpid(void);`
  - Devuelve el identificador del proceso.
- `pid_t getppid(void);`
  - Devuelve el identificador del proceso padre.
- `uid_t getuid(void);` `gid_t getgid(void);`
  - Devuelven el identificador de usuario real y del grupo real.
- `uid_t geteuid(void);` `gid_t getegid(void);`
  - Devuelven el identificador de usuario efectivo y del grupo efectivo.
- `int setuid(uid_t uid);`
  - Si el proceso es privilegiado (*el identificador de usuario efectivo del proceso que efectúa la llamada es el de root*) se cambian el uid real y el uid efectivo.
  - Si el proceso no es privilegiado solo se cambia el uid real.
- `int seteuid(uid_t euid);`
  - Si el proceso es privilegiado se establece el usuario efectivo.
  - Si el proceso no es privilegiado solamente puede poner como efectivo su real, por ejemplo: `seteuid (getuid);`

## EJEMPLO

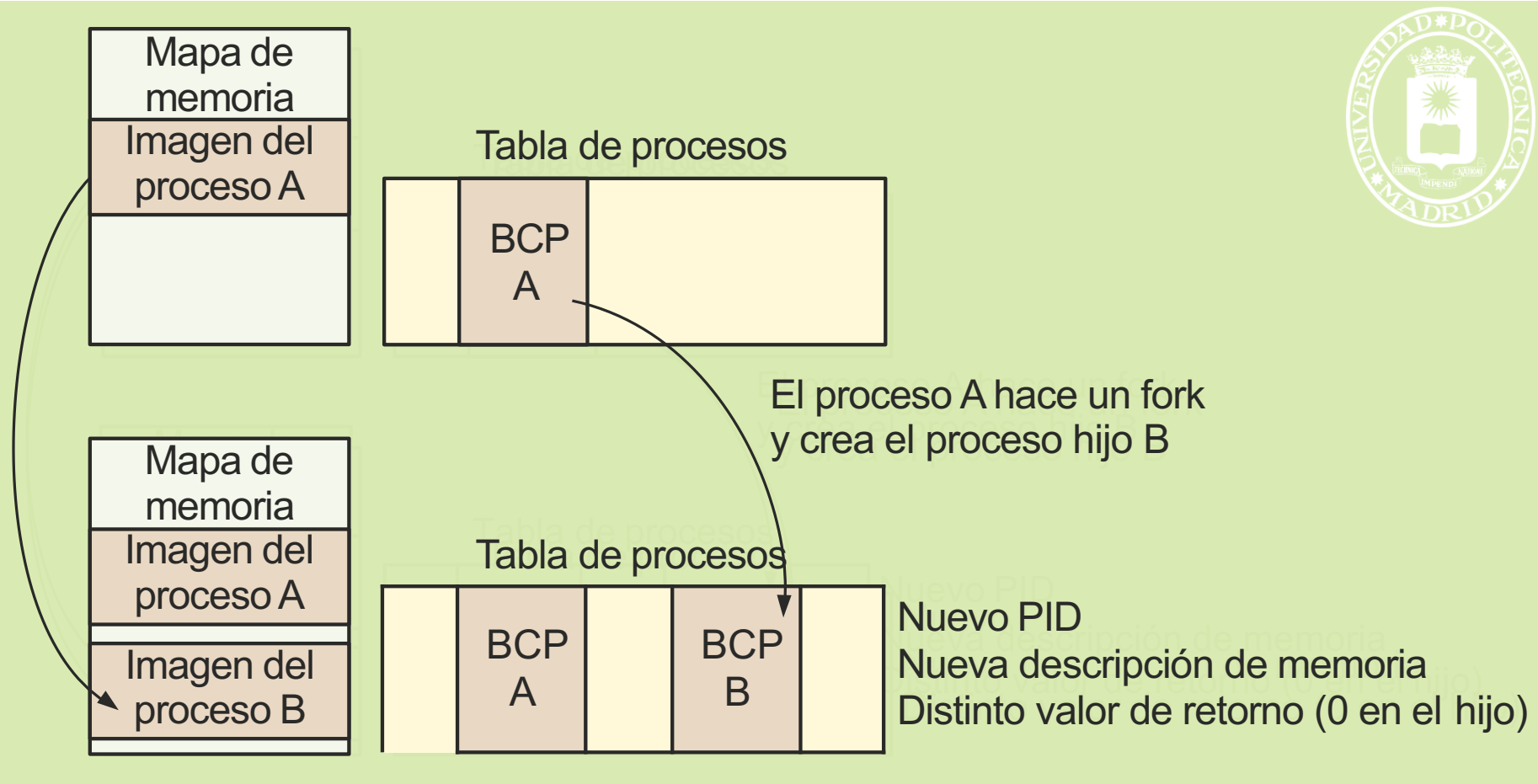
```
#include <unistd.h>
#include <stdio.h>
int main()
{
        int i;
        pid_t id_proceso;
        pid_t id_padre;

        id_proceso = getpid();
        id_padre = getppid();
        printf ("Id del proceso: %d\n", id_proceso);
        printf ("Id del proceso padre: %d\n", id_padre);
        sleep(10);
        return 0;
}
```

# SERVICIOS UNIX DE GESTIÓN DE PROCESOS (Entorno)

- `extern char *environ[];`
  - **Variable que apunta al entorno.**
- **Funciones de biblioteca:**
  - `char *getenv(const char *name);`
    - Devuelve el valor de la variable de entorno **name**.
      - » HOME, directorio de trabajo inicial del usuario.
      - » LOGNAME, nombre del usuario asociado a un proceso.
      - » PATH, prefijo de directorios para encontrar ejecutables.
      - » TERM, tipo de terminal.
      - » TZ, información de la zona horaria.
  - `int putenv(char *string);`
    - Establece el valor de las variables de entorno.

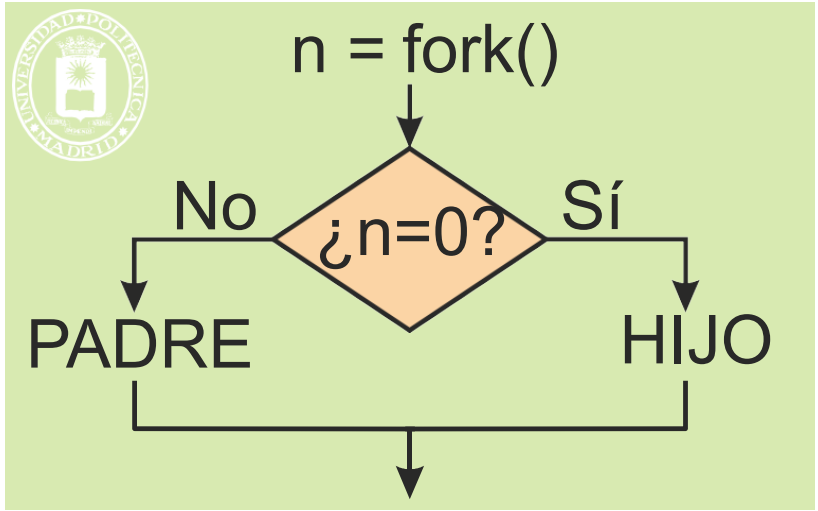
# SERVICIOS UNIX DE GESTIÓN DE PROCESOS (Creación I)

- `pid_t fork(void);`
  - **Crea un proceso hijo. Devuelve 0 al proceso hijo y el pid del hijo al proceso padre.**
  - **El hijo es un clon del padre pero con nuevo pid, imagen de memoria y valor de retorno del fork.**

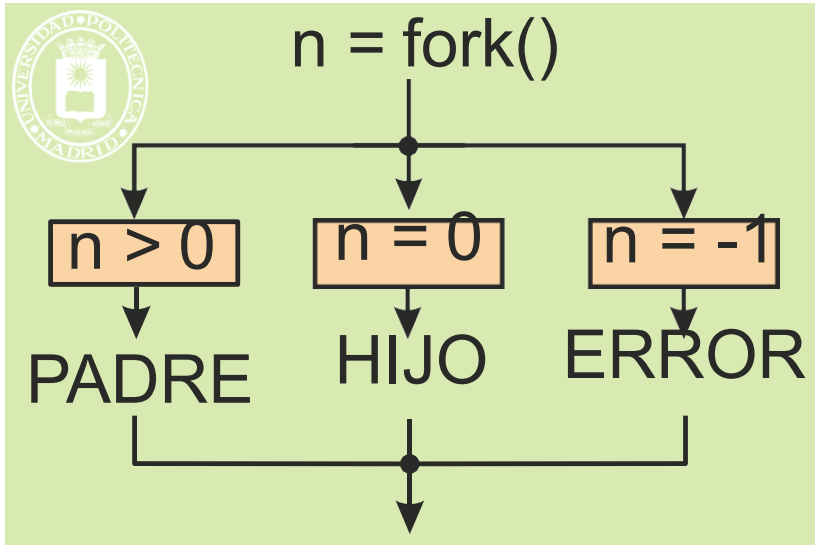


# SERVICIOS UNIX DE GESTIÓN DE PROCESOS (Creación II)

```
int n;
if ((n = fork()) == -1) {
   perror("Error al llamar a fork");
   /* perror: Función que imprime el texto más el
    tipo de error del SO. */
} else if (n == 0){     //Código del HIJO
      .... código del hijo ....
} else {                //Código del PADRE
      .... código del padre ....
}
```




```
switch(fork()) {
case -1:
   perror("Error al llamar a fork");
   break;
case  0: //Código del HIJO
     .... código del hijo ....
   break;
default: //Código del PADRE
   .... código del padre ....
}
```



# Ejemplos

Pinta la jerarquía de procesos

```
switch(fork()) {  case -1:
   perror("Error al llamar a fork");  break;
case   0: //Código del HIJO
   fork();
   break;
default: //Código del PADRE
.... código del        padre ....
}
```

Pinta la jerarquía de procesos

```
switch(fork()) {  case -1:
   perror("Error al llamar a fork");  break;
case   0: //Código del HIJO
   break;
default: //Código del PADRE
        fork();
.... código del        padre ....
}
```

# Ejemplos II

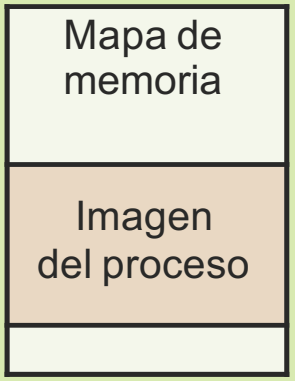
**¿Qué jerarquía de procesos queda en este caso?**
**¿Qué valores se imprimen por pantalla? ¿En qué orden?**

```
a=1;
fork();
a=a+1;
fork();
a=a+1;
printf("%d",a);
```

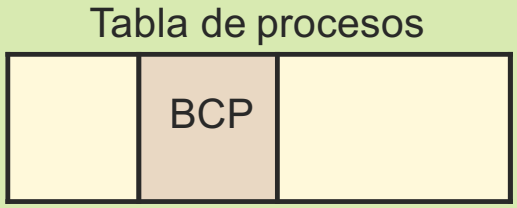
**¿Qué se imprime por pantalla?**

```
int a=1;
switch(fork()) {
case -1:
   perror("Error al llamar a fork");  break;
case   0: //Código del HIJO
     .... código del   hijo ....
   a=a+5;
   printf("%d",a);
   break;
default: //Código del PADRE
    a=a+1;
    printf("%d",a);
    .... código del   padre ....
}
```

# EL EXEC CAMBIA EL PROGRAMA DE UN PROCESO

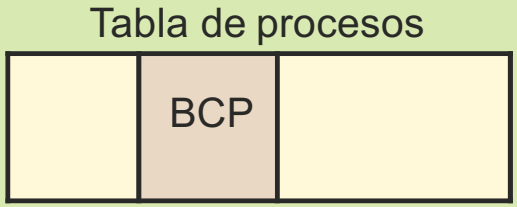



## Ejecución de exec

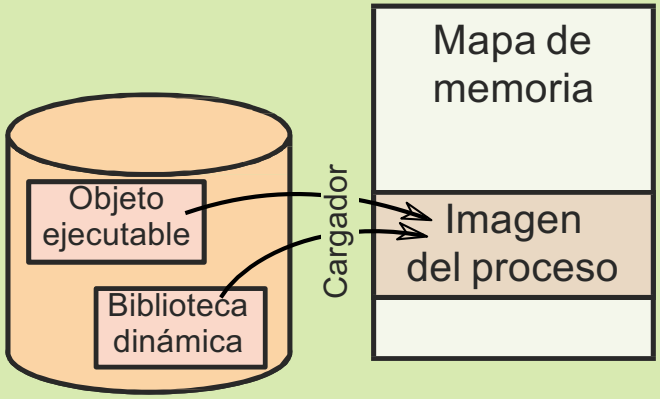




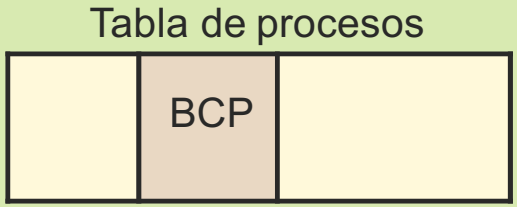
## Vaciado del proceso




**Se borra la imagen de memoria**
**Se borra la descripción de la memoria**
**Se borra el estado (registros)**
**Se conserva el PID, PID padre, GID, etc.**
**Se conservan los descriptores fd.**




## Carga de nueva imagen




**Se carga la nueva imagen**
**Se carga el nuevo estado con una nueva dirección de arranque**

# SERVICIOS GESTIÓN DE PROCESOS (Creación III)

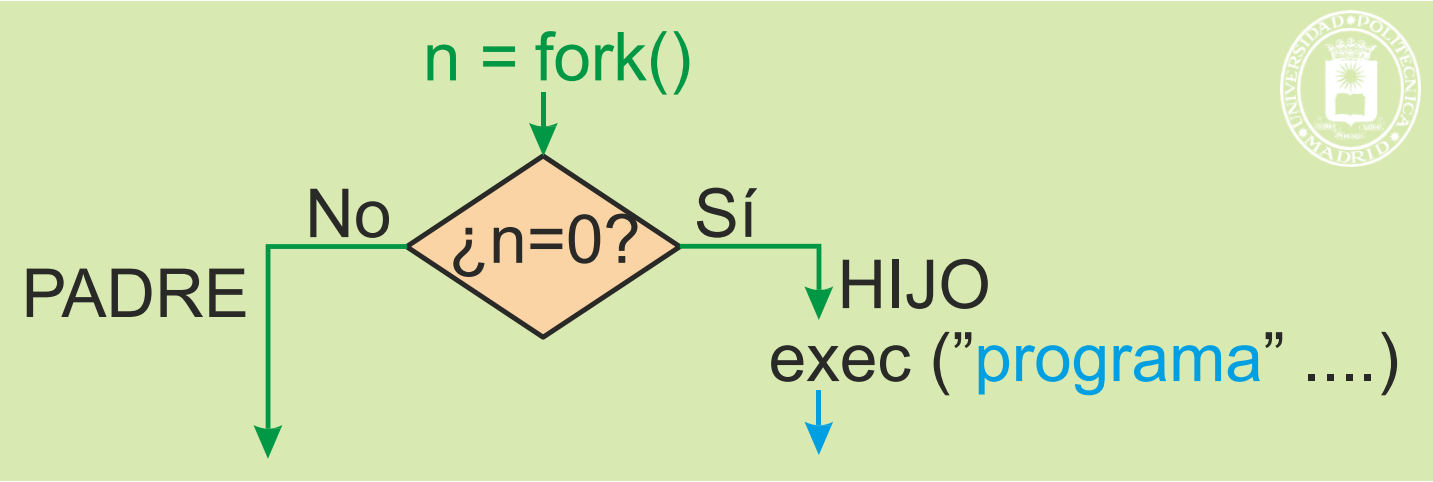

- `int execve(const char *path, char *const argv[], char *const envp[]);`
  `int execlp(const char *file, const char *arg, ...);`
  `int execvp(const char *file, char *const argv[]);`

| Función | pathname | filename | Arg List | argv[] | environ | envp[] |
|---|---|---|---|---|---|---|
| execl | X | | X | | X | |
| execlp | | X | X | | X | |
| execle | X | | X | | | X |
| execv | X | | | X | X | |
| execvp | | X | | X | X | |
| execve | X | | | X | | X |
| execvpe | | X | | X | | X |
| Letra | | p | l | v | | e |

- **Permite a un proceso pasar a ejecutar otro programa (código). El pid no cambia**
- **Cambia la imagen de memoria del proceso. El fichero ejecutable se especifica con nombre completo (path) o relativo (file).**
- **El entorno se mantiene o se cambia mediante envp.**
- **Mantiene en el BCP todas las informaciones que no son dependientes del programa.**
  - **Información de identificación, pero podrían cambiar el UID y GID efectivo.**
  - **Descriptores abiertos.**
- **Se crea una nueva pila del proceso con el entorno y los parámetros y las variables locales del main.**

# SECUENCIA TÍPICA DEL FORK CON EXEC

```
int n;
if ((n = fork()) == -1)
   perror("Error al llamar a fork");
   .... Tratamiento del error ....
else if (n == 0){ //Código del HIJO
   execl("programa",...); //Ojo, exec no retorna
   perror("programa");
   exit(1); //si error en el exec
}
//El PADRE continua
.... código del padre ....
```



# SERVICIOS GESTIÓN DE PROCESOS (Terminación I)

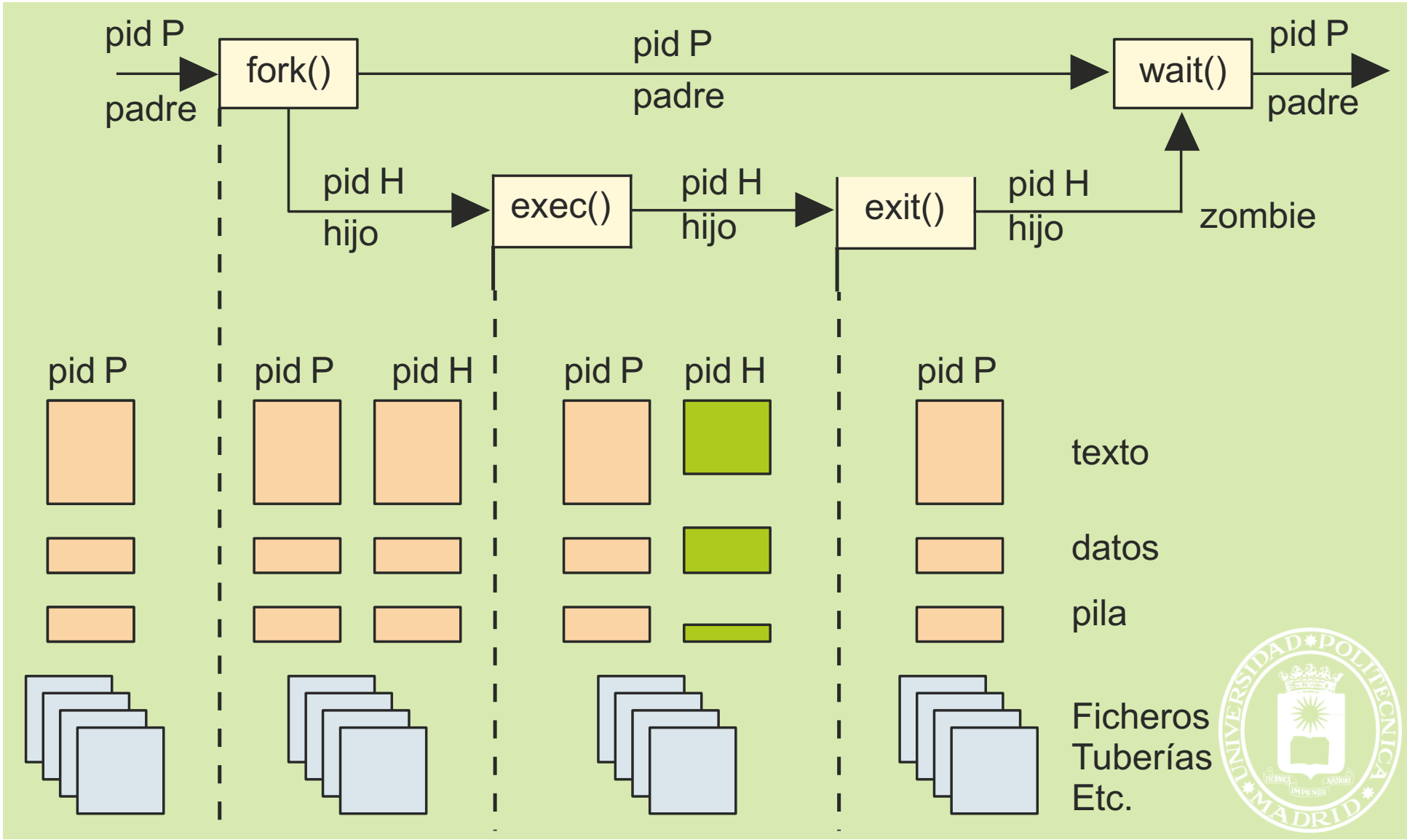

- `pid_t wait(int *status);`
  - Espera hasta que termina un proceso hijo (el primero que termine). Retorna el identificador del proceso hijo y el estado de terminación del mismo.
- `pid_t waitpid(pid_t pid, int *status, int options);`
  - Espera hasta que termina el proceso pid.
- `void exit(int status);`
  - Finaliza la ejecución de un proceso indicando el estado (`status & 0377`) de terminación del mismo.
    - Se manda la información de estado y 8 bits indicando modo de finalización
  - Es más elegante terminar el `main` con un `return` que con `exit`

# SERVICIOS GESTIÓN DE PROCESOS (Terminación II)

La variable `status` en `wait` y `waitpid`

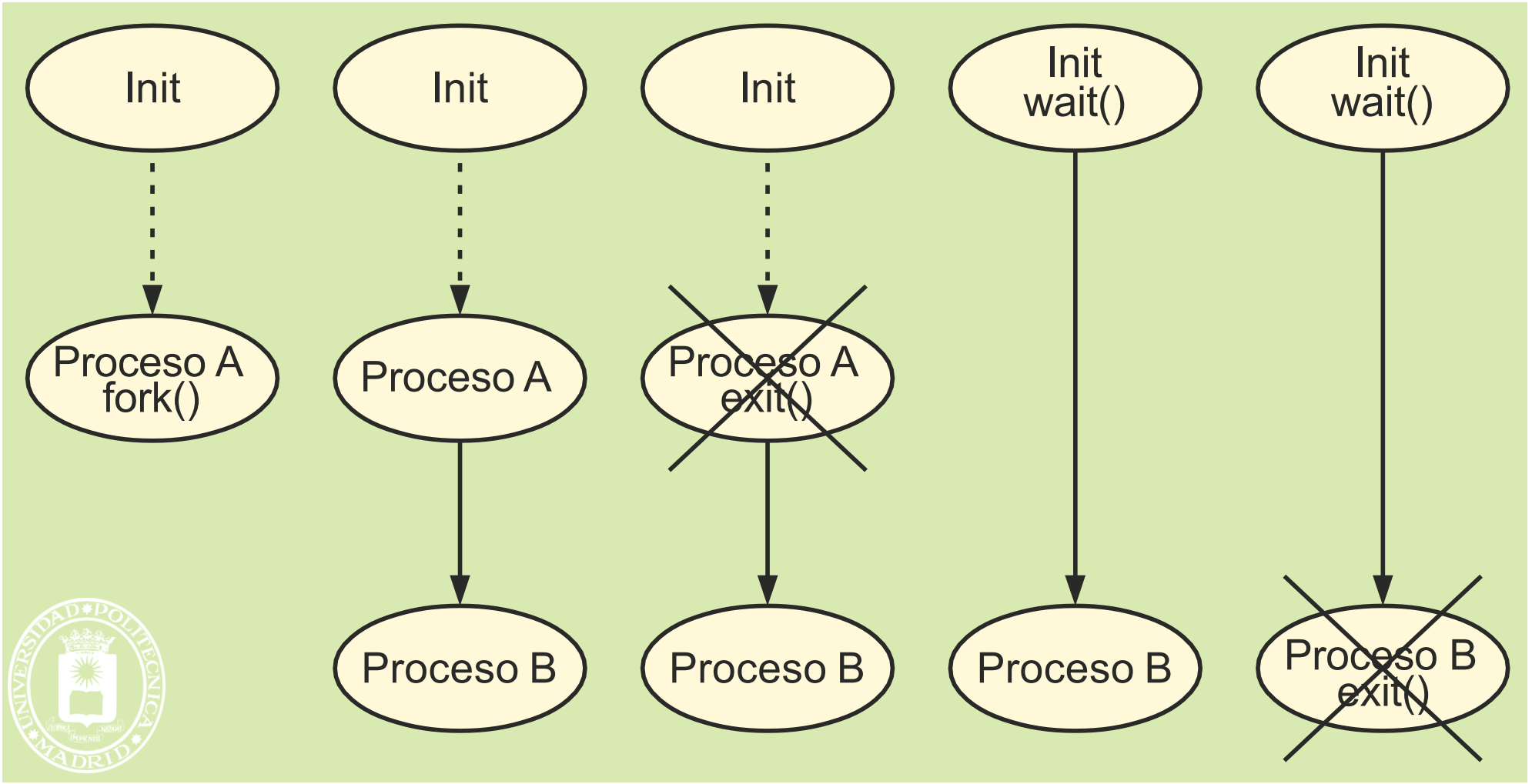
- status contiene dos valores interesantes:
  - Qué pasó con el proceso hijo: finalizó correctamente o por la recepción de una señal
  - Cómo finalizó el proceso hijo: valor de finalización del proceso o número de señal que provocó la finalización
- Macros definidas sobre la variable `status`
  - `WIFEXITED(status)`: valor positivo si el hijo finalizó normalmente
  - `WEXITSTATUS(status)`: valor devuelto por el proceso hijo (`exit` o `return`) si finalizó normalmente
  - `WIFSIGNALED(status)`: valor positivo si el proceso finalizó por la recepción de una señal
  - `WTERMSIG(status)`: número de señal que provocó la finalización de proceso
  - Ubicadas en `sys/wait.h`

# SERVICIOS GESTIÓN DE PROCESOS (UNIX)



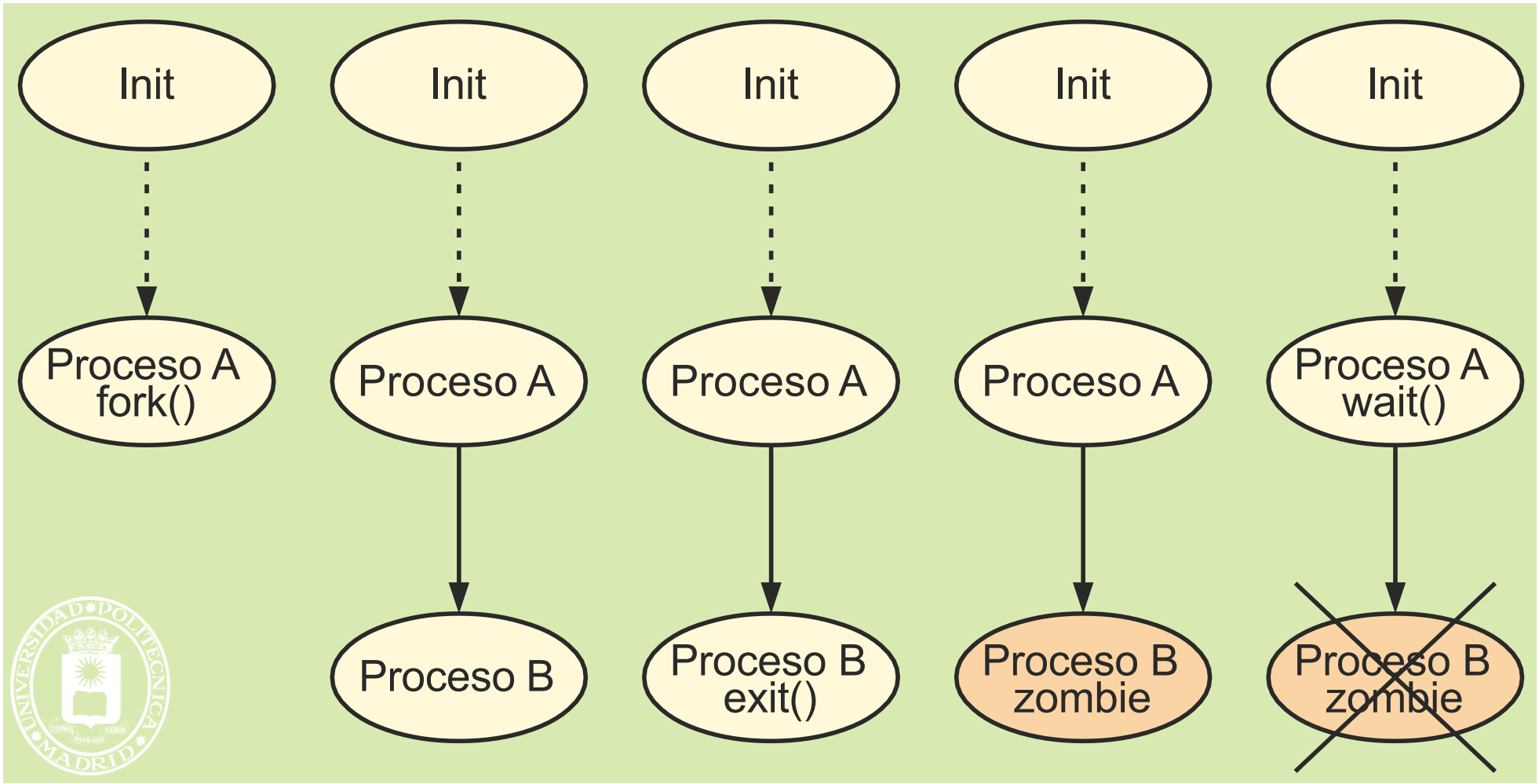
# EVOLUCIÓN DE PROCESOS I

**El padre muere: INIT acepta los hijos.**



# EVOLUCIÓN DE PROCESOS II

**Zombie: el hijo muere y el padre no hace wait.**



# EJEMPLO




```c
/* Ejecución de ls -l con paso de argumentos */
int main(int argc, char* argv[]) {
   pid_t pid;
   int status;
   char *argumentos[3];
   argumentos[0] = "ls"; /* define los argumentos */
   argumentos[1] = "-l";
   argumentos[2] = NULL;
   pid = fork();
   if (pid == 0) { /* proceso hijo */
      execvp(argumentos[0], argumentos);
      perror(argumentos[0]);
      exit(1);
   } else if (pid < 0) { /* error */
      perror("fork");
      exit(1);
   }
   /* proceso padre */
   while (pid != wait(&status))
      continue;
   return 0;
}
```

# SEÑALES Y TEMPORIZADORES

# SEÑALES


- **El SO notifica a un proceso la ocurrencia de un determinado evento por medio del mecanismo de las señales (UNIX)**

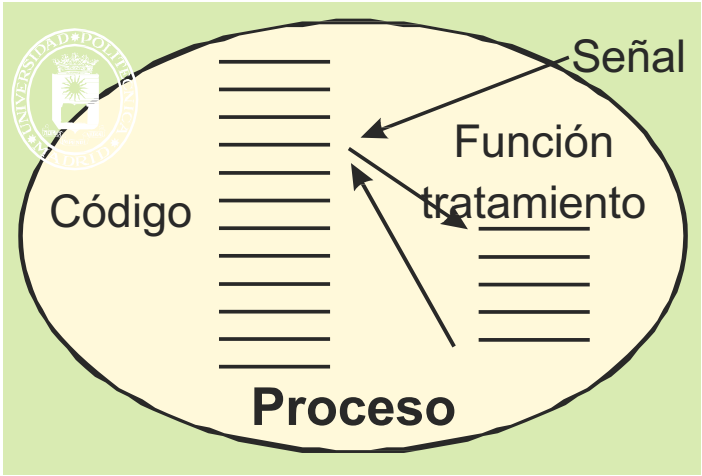
- **Desde el punto de vista del proceso, una señal:**
  - **Es un evento que recibe (a través del SO)**
  - **Interrumpe al proceso**
  - **Le transmite información muy limitada (un número, que identifica el tipo de señal)**
  - **Un proceso también puede enviar señales a otros procesos (del mismo grupo), mediante el servicio kill().**
- **Desde el punto de vista del SO:**
  - **Una señal se envía a un único proceso**
  - **Origen:**
    - SO → proceso
    - proceso → proceso

# SEÑALES

- **Hay muchos tipos de señales, según su origen.**
  - **Originadas por excepciones de HW, por ejemplo:**
    - Instrucción ilegal.
    - Violación de memoria.
    - Desbordamiento en operación aritmética.
  - **Originadas por Interrupciones, por ejemplo:**
    - Vence el temporizador.
    - Abortar proceso desde teclado.
  - **Originadas por otro proceso, mediante el servicio de envío *kill***

## SEÑALES: Acción en el proceso

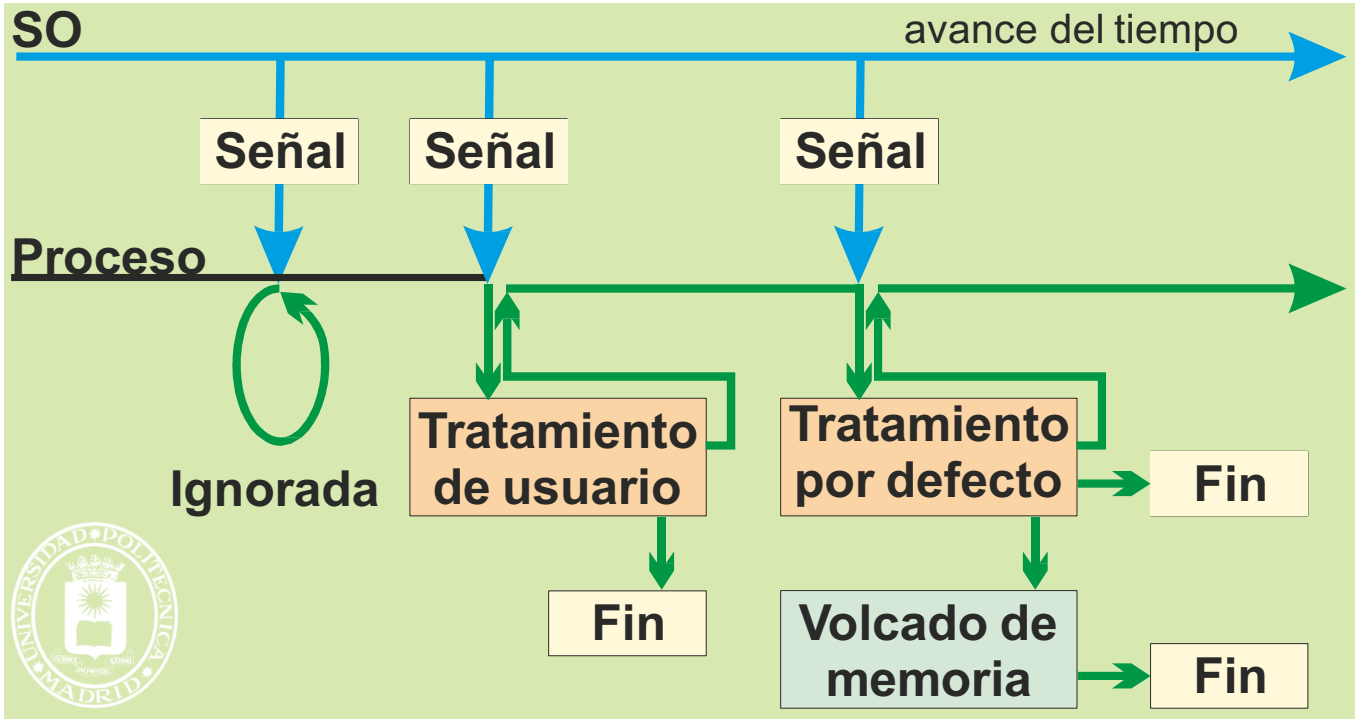
- **Ignorar**: El proceso puede haberle indicado al SO que ignore ese tipo de señal (servicio `sigaction`).
- **Armado** de la señal: El proceso le indica al SO la función a ejecutar para ese tipo de señal (servicio `sigaction`). El SO emula para el proceso una interrupción cuyo tratamiento es esa función.
- Si el proceso no ha indicado nada, se produce una acción por **defecto**:
  - El proceso, en general, muere.
  - Hay algunas señales que se ignoran o tienen otro efecto.



# SEÑALES: Acción en el proceso

- **Máscara de bloqueo de señales**: El proceso tiene una máscara que permite enmascarar diversos tipos de señales

  - Contiene un bit por tipo de señal. Si el bit de máscara está activo las señales de ese tipo no son enviadas al proceso, quedando pendientes
  - Servicio `sigprocmask`



# FORK Y EXEC

## FORK

- El hijo hereda el armado de señales del padre.
- El hijo hereda las señales ignoradas.
- El hijo hereda la máscara de señales.
- La alarma se cancela en el hijo.
- Las señales pendientes no son heredadas.

## EXEC

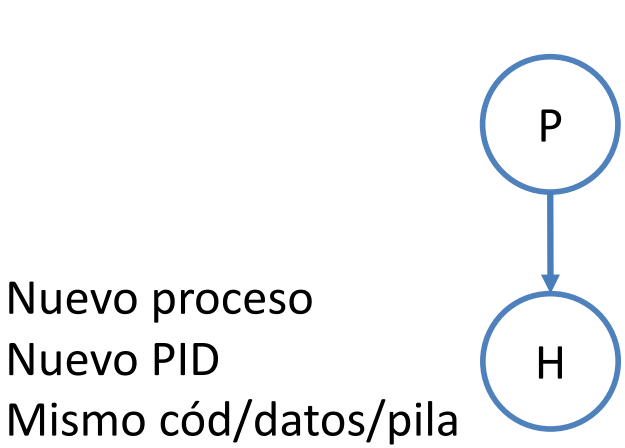
- El armado desaparece pasándose a la acción por defecto (ya no existe la función de armado).
- Las señales ignoradas se mantienen.
- La máscara de señales se mantiene.
- La alarma se mantiene.
- Las señales pendientes siguen pendientes.

# TEMPORIZADORES

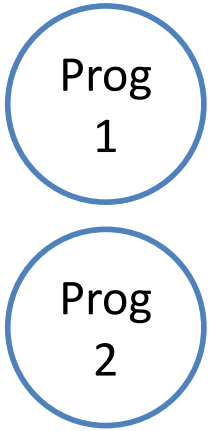

- El SO mantiene uno o varios temporizadores por proceso (en su BCP)
  - El proceso activa el temporizador (UNIX: `alarm`).
- El SO envía una señal al proceso cuando vence su temporizador (UNIX: SIGALRM).
- fork: el proceso hijo NO hereda los temporizadores.
- exec: Después del exec SI se conservan los temporizadores.

# FORK Y EXEC

| | FORK | EXEC |
|---|---|---|
| Armado | ✓ | ✗ |
| Ignorado | ✓ | ✓ |
| Máscara | ✓ | ✓ |
| Alarma | ✗ | ✓ |
| Señales pendientes | ✗ | ✓ |




Nuevo proceso
Nuevo PID
Mismo cód/datos/pila




Mismo proceso
Mismo PID
Diferente cód/datos/pila

# SERVICIOS UNIX
# SEÑALES Y TEMPORIZADORES

# TIPOS DE SEÑALES


**El fichero de cabecera `signal.h` declara la lista de señales. Algunos ejemplos son:**

- **SIGALRM, señal de fin de temporización.**
- **SIGFPE, operación aritmética errónea.**
- **SIGILL, instrucción hardware inválida.**
- **SIGINT, señal de atención interactiva (ctrl + C).**
- **SIGKILL, señal de terminación (no se puede ignorar ni armar) (`kill -9`).**
- **SIGPIPE, escritura en un pipe sin lectores.**
- **SIGQUIT, señal de terminación interactiva (ctrl + \).**
- **SIGSEGV, referencia a memoria inválida.**
- **SIGTERM, señal de terminación (señal por defecto de `kill`).**
- **SIGUSR1, señal definida por la aplicación.**
- **SIGUSR2, señal definida por la aplicación.**
- **SIGCHLD, indica la terminación del proceso hijo.**
- **SIGCONT, continuar si está bloqueado el proceso.**
- **SIGSTOP, señal de bloqueo (no se puede armar ni ignorar) (ctrl + Z).**

# CONJUNTO DE SEÑALES


- Existe una serie de señales que un proceso puede recibir durante su ejecución
- Un proceso puede realizar operaciones sobre conjuntos de señales
  - `int sigemptyset(sigset_t *set);`
    - Lo inicializa con un conjunto de señales vacío
  - `int sigfillset(sigset_t *set);`
    - Lo inicializa con todas las señales disponibles en el sistema
  - `int sigaddset(sigset_t *set, int signo);`
    - Añade al conjunto la señal con número signo
  - `int sigdelset(sigset_t *set, int signo);`
    - Borra del conjunto la señal con número signo
  - `int sigismember(sigset_t *set, int signo);`
    - Determina si una señal pertenece a un conjunto

# SERVICIOS DE SEÑALES

## Envío de señales

- `int kill(pid_t pid, int sig);`
  - **Envía al proceso pid la señal sig.**

## Armado de señal

- `int sigaction(int sig, struct sigaction *act, struct sigaction *oact);`

**Acción a realizar como tratamiento de sig. Campos de sigaction:**

- **sa_handler: función de armado, SIG_IGN o SIG_DFL**
- **sa_mask: señales a bloquear durante ejecución de función de armado**
  - Además de la propia sig
- **sa_flags: Opciones diversas**
  - **P. e. SA_RESTART**
  - Si la señal se produce estando en una llamada bloqueante, después del tratamiento sigue en la llamada
  - Si no se activa, por defecto, la llamada termina con error y errno = E_INTR


- `int sigprocmask (int how, const sigset_t *set, sigset_t *o_set);`

Examina o modifica la máscara de señales activa para un proceso
Si una señal está bloqueada por la máscara, no es procesada hasta el momento en que deja de estar bloqueada por la máscara (se memoriza que llegó)
Parámetro **how**

- **SIG_BLOCK**: añade un conjunto de señales a las que se encuentran bloqueadas en la máscara actual
- **SIG_UNBLOCK**: elimina un conjunto de señales de las que se encuentran bloqueadas en la máscara actual
- **SIG_SETMASK**: especifica un conjunto de señales que serán bloqueados

Parámetro **set**

Conjunto de señales que serán utilizadas para la modificación (puede ser NULL)

Parámetro **oset**

Máscara previa a la modificación (puede ser NULL)

# SERVICIOS DE SEÑALES


**Espera de señales**

- `int pause(void);`
  - Bloquea al proceso hasta la recepción de una señal.

**Temporización**

- `unsigned int alarm(unsigned int seconds);`
  - Genera la recepción de la señal SIGALRM pasados *seconds* segundos.
- `int sleep(unsigned int seconds);`
  - El proceso despierta cuando ha transcurrido el tiempo establecido o cuando se recibe una señal.

# Imprimir mensaje cada 10 segundos

```
#include <signal.h>
#include <stdio.h>
void tratar_alarma(void) {
  printf("Activada \n");
}

int main()  {
  struct sigaction act;

  /* establece el manejador para SIGALRM */
  act.sa_handler = tratar_alarma;
  act.sa_flags = 0;      /* ninguna acción especifica */
  sigaction(SIGALRM, &act, NULL);

  act.sa_handler = SIG_IGN;              /* ignora SIGINT */
  sigaction(SIGINT, &act, NULL);
  for(;;){       /* recibe SIGALRM cada 10 segundos */
    alarm(10);
    pause();
  }
}
```

# Temporización de proceso hijo

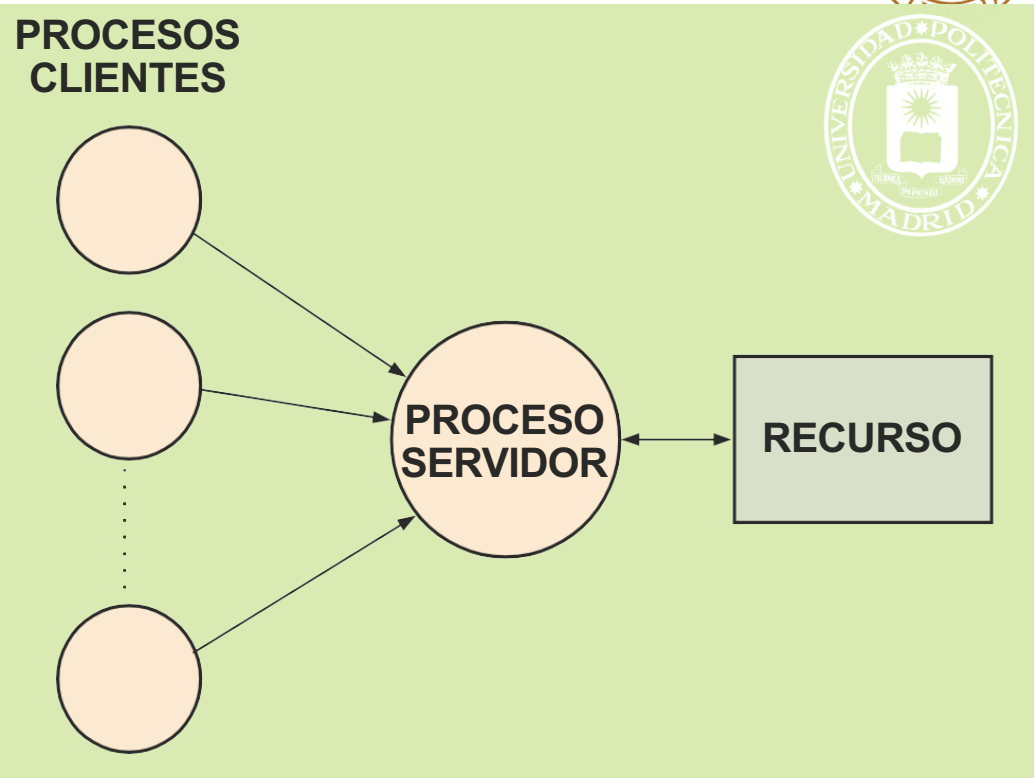
```
/* programa que temporiza a su hijo, y le mata al cabo de N segundos.*/
pid_t pid;
void tratar_alarma(int n) { kill(pid, SIGKILL); }

int main(int argc, char *argv[]) {
   int status;
   char **argumentos = &argv[1];
   struct sigaction act;
   switch(pid = fork()) {
      case -1: perror ("fork"); exit(1);
      case  0: /* proceso hijo */
         execvp(argumentos[0], argumentos);
         perror("exec"); exit(1);
      default: /* padre */
            /* establece el manejador */
         act.sa_handler = &tratar_alarma; /* función a ejecutar */
         act.sa_flags = SA_RESTART; /* evita E_INTR en el wait */
         sigaction(SIGALRM, &act, NULL);
         alarm(5);
         wait(&status);
   }
   return 0;
}
```

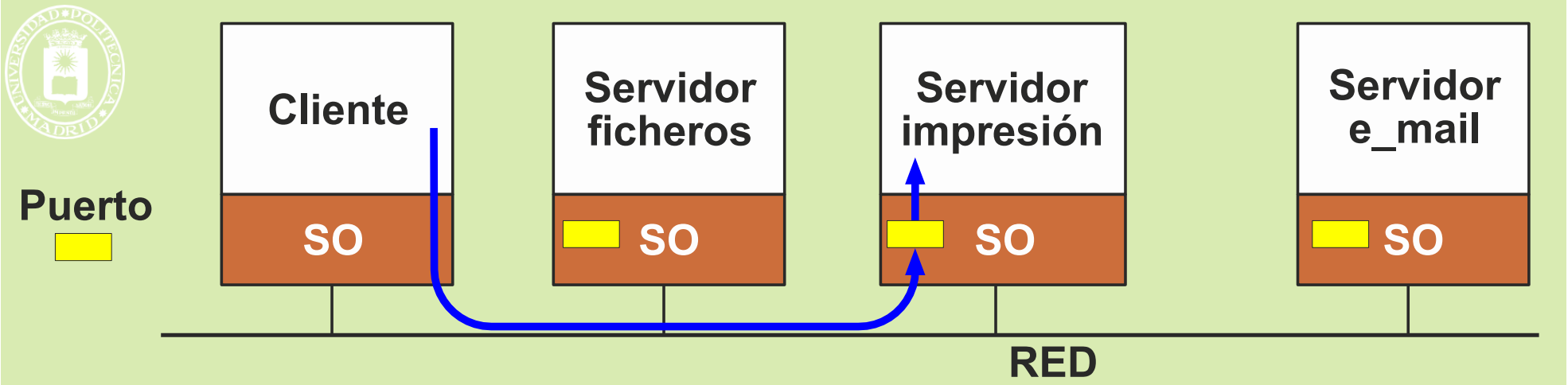
# PROCESOS ESPECIALES

# PROCESO SERVIDOR

## Servidor

- Atiende peticiones.
- Realiza el bucle:
  - Lectura de orden.
  - Ejecución de la orden.
  - Contestación al cliente.

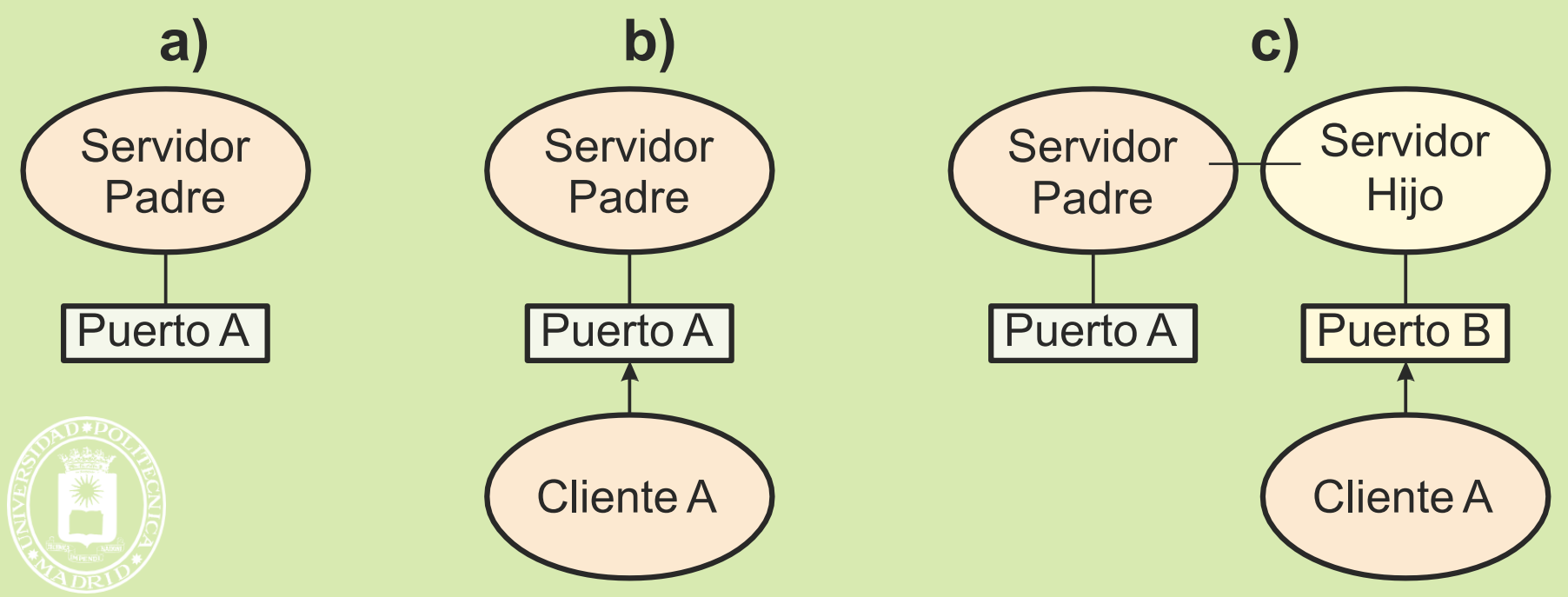


# PROCESO SERVIDOR PARALELO

## Bucle

- Lectura de orden.
- Asignación de un nuevo puerto y lanza hijo.
- Proceso hijo que atiende al cliente.
- Vuelta al comienzo.



# PROCESO DEMONIO

## Es un proceso:

- Que ejecuta en background (su padre no le espera).
  - No asociado a un terminal o proceso login.
  - Que espera a que ocurra un evento.
- O que debe realizar una tarea de forma periódica.

## Características

- Se arrancan al inicializar el sistema.
- No mueren.
- Están normalmente en espera de evento.
- No hacen el trabajo, lanzan otros procesos o procesos ligeros.

# EJEMPLOS DE SERVIDORES Y DEMONIOS UNIX


- lpd — line printer daemon.
- inetd — arranca servidores de red: ftp, telnet, http, finger,talk, etc.
- smbd — demonio de samba.
- atd — ejecución de tareas a ciertas horas.
- crond — ejecución de tareas periódicas.
- nfsd — servidor NFS.
- httpd — servidor WEB.

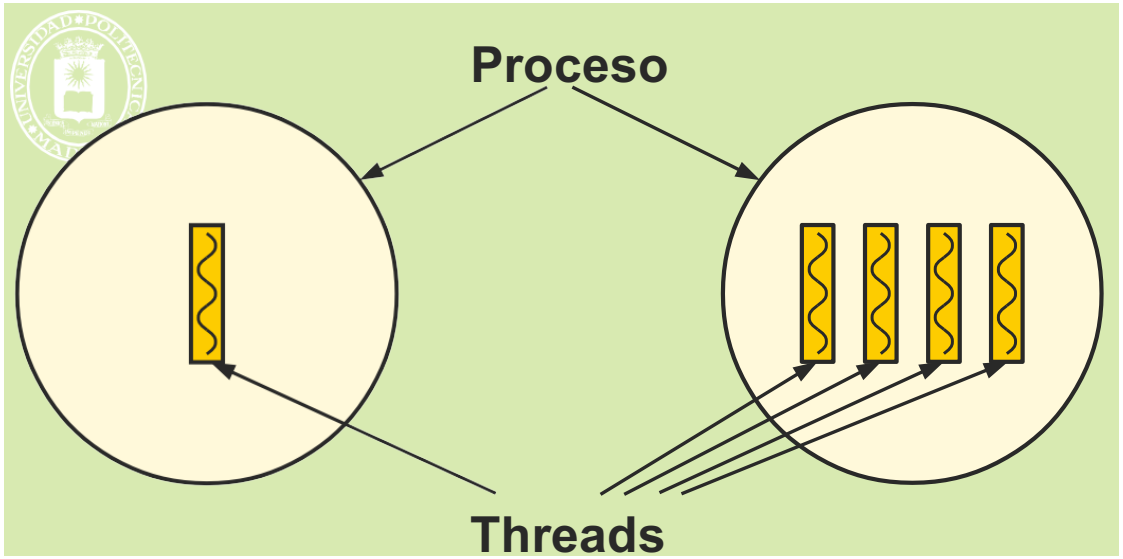
# THREADS (PROCESOS LIGEROS)

# THREADS




- Problema: proceso utilizado para 2 objetivos casi contradictorios
  - Aislar actividades independientes vs
  - Expresar paralelismo de aplicación
- Este aislamiento puede ser inconveniente para expresar el paralelismo:
  - Creación de proceso y c. contexto consumen tiempo considerable
  - Actividades paralelas de una aplicación requiere fuerte acoplamiento
- Solución: Crear una nueva abstracción (thread) y redefinir la de proceso
  - Thread (hilo); flujo de ejecución dentro de un proceso
  - Proceso: contexto de ejecución con múltiples threads
- Ventajas de los threads
  - Son más ligeros
  - Más fuertemente acoplados
- Aplicación de las dos abstracciones
  - Proceso: Expresar actividades independientes
  - Thread: Expresar paralelismo de la aplicación

- Por thread (BCT).
  - Registros ( especialmente el contador de programa).
  - Pila.
  - Estado (ejecutando, listo o bloqueado).
  - Máscara de señales
- Por proceso (BCP).
- Espacio de memoria.
  - Variables globales.
  - Ficheros abiertos.
  - Temporizadores.
  - Señales y semáforos.
  - Contabilidad
  - Threads

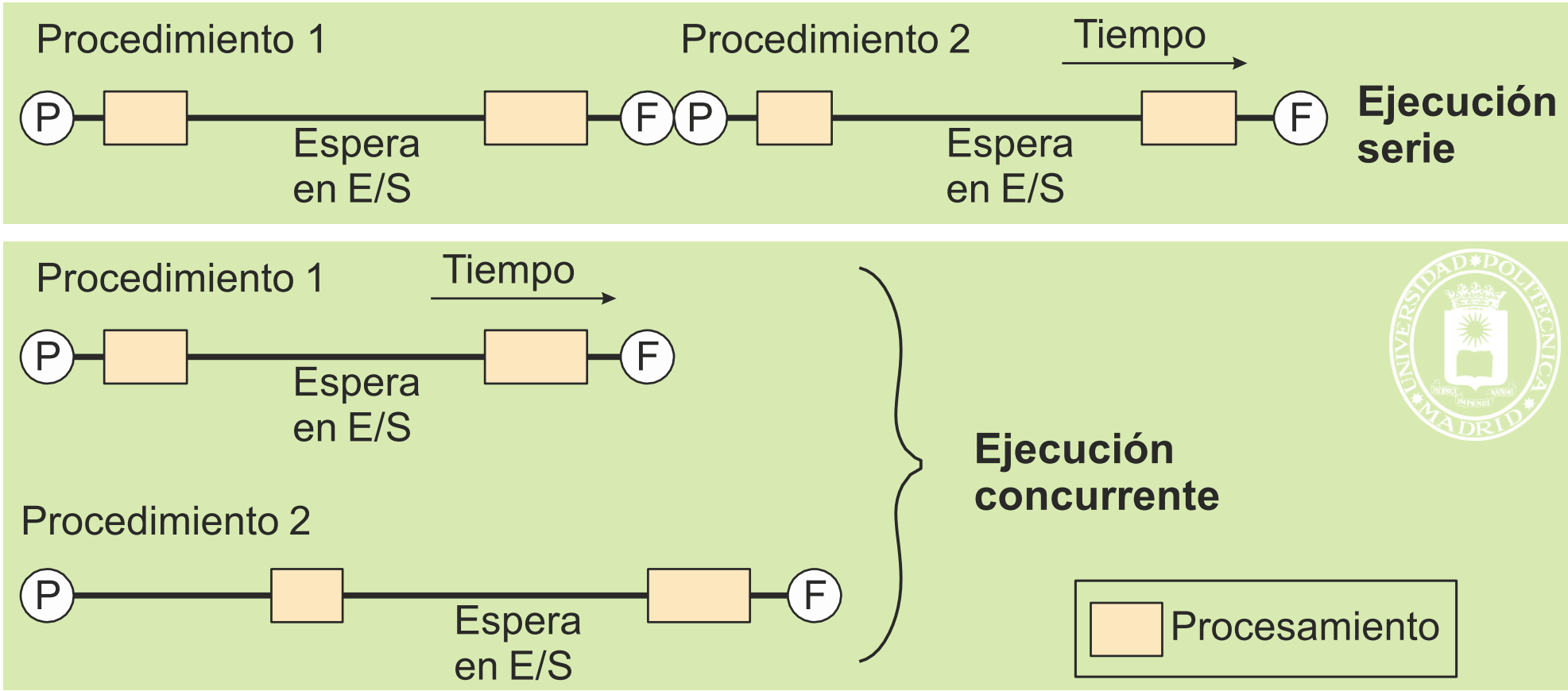



El SO contará con una estructura BCP extendida para soportar las informaciones de los distintos threads.

# VENTAJA DE LA EJECUCIÓN CONCURRENTE



- **Mediante la ejecución concurrente el tiempo total en ejecutar una tarea compuesta por varios procedimientos independientes se acorta.**



# VENTAJAS E INCONVENIENTES DE LOS THREADS




- Ventajas inherentes a la programación concurrente.
  - Separación de tareas.
  - Facilita la modularidad.
  - Aumenta el aprovechamiento de la CPU por una tarea.
- Mejores prestaciones que con procesos.
  - Los threads comparten memoria → variables comunes.
  - Sincronización entre procesos ligeros más sencilla.
- Inconvenientes inherentes a la programación concurrente.
  - Programación compleja.
  - Hay que sincronizar el acceso a los datos compartidos.
  - Se organiza el acceso a zonas compartidas mediante ‘mutex’.
- Si produce un error grave un thread muere todo el proceso, lo que puede suponer un problema.

# ALTERNATIVAS DE DISEÑO

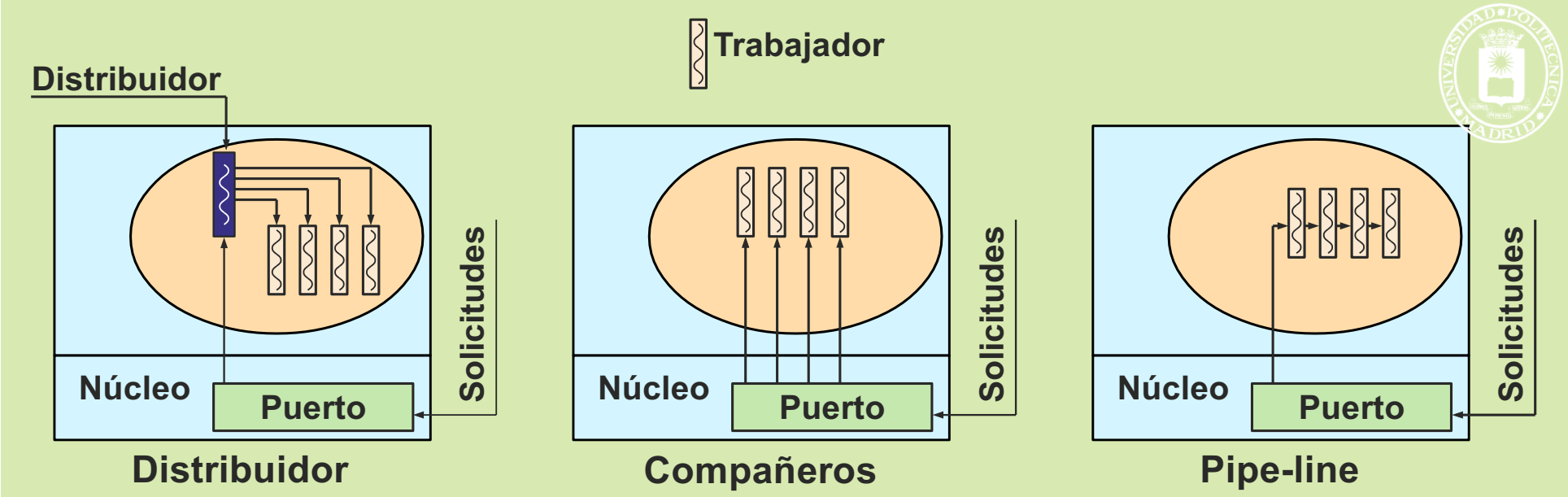



- Proceso con un único thread.
  - No hay concurrencia.
  - Llamadas al sistema bloqueantes.
- Múltiples procesos convencionales cooperando.
  - Permite concurrencia.
  - Llamadas al sistema bloqueantes.
  - No comparten variables.
  - Mayor sobrecarga de ejecución.
- Threads.
  - Permite concurrencia.
  - Comparten variables.
  - Llamadas al sistema bloqueantes.
  - Menor sobrecarga de ejecución que varios procesos normales.


- Thread.
  - Concurrencia y variables compartidas.
  - Las llamadas al sistema bloquean solamente al proceso ligero que las emite (KLT).
- Permite división de tareas.
- Aumenta la velocidad de ejecución del trabajo.
- Principios básicos en la programación concurrente.
  - Variables globales compartidas entre procesos ligeros
  - Mutex (p.ej, semáforos)
  - Imaginar otras llamadas al mismo código al mismo tiempo

Ejemplos de arquitecturas software basado en threads.

- Distribuidor con varios threads trabajadores (creados por cada solicitud de servicio o previamente creados)
- Threads 'compañeros'
- Esquema segmentado o *pipe-line* (tipo cadena de fabricación)



# SERVICIOS UNIX
# THREADS

# GESTIÓN DE THREADS (Atributos)




- `int pthread_attr_init(pthread_attr_t *attr)`
  - Inicializa un objeto atributo para crear nuevos threads
- `int pthread_attr_destroy(pthread_attr_t *attr)`
  - Destruye el objeto atributo
- `int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)`
  - Define el tamaño de la pila
- `int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate)`
  - PTHREAD_CREATE_JOINABLE: thread no desaparece hasta que otro espere por su finalización (mediante pthread_join)
  - PTHREAD_CREATE_DETACHED: thread independiente, liberará sus recursos al terminar
- `int pthread_attr_getdetachstate (pthread_attr_t *attr, int *detachstate)`
  - Lee si es JOINABLE o DETACHED

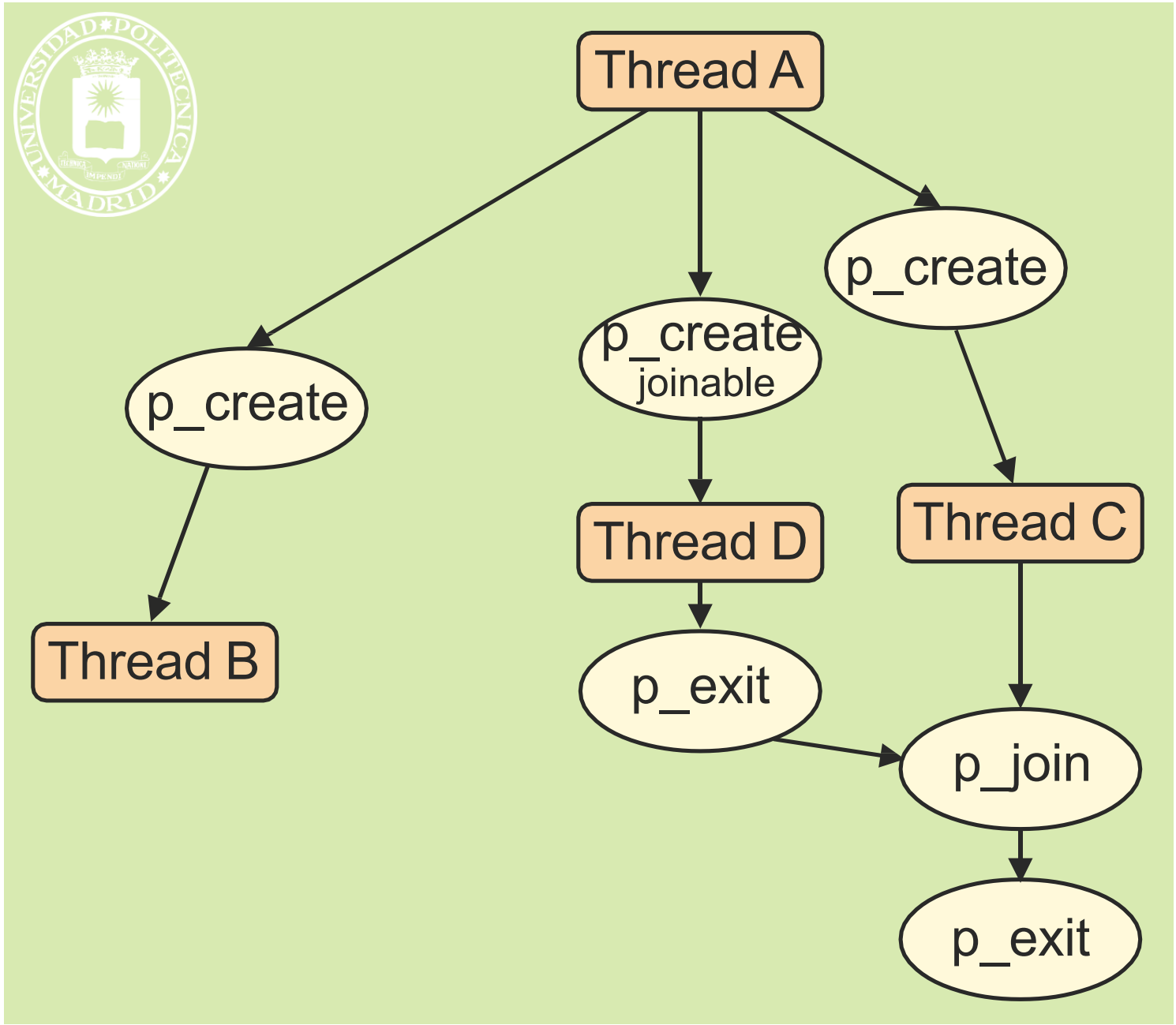
# GESTIÓN DE THREADS (Creación y terminación)




- `int pthread_create(pthread_t *thread, const thread_attr_t *attr, void *(*func)(void *),void *arg)`
  - Crea un thread que ejecuta func con argumento arg y atributos `attr` (por defecto: JOINABLE)
- `Pthread_t pthread_self()`
  - Devuelve el identificador del thread que la llama
- `int pthread_join(pthread_t thid, void **value)`
  - Suspende la ejecución de un thread hasta que termina el thread con identificador thid.
  - Devuelve el estado de terminación del thread.
- `int pthread_exit(void *value)`
  - Permite a un thread finalizar su ejecución, indicando el estado de terminación del mismo.

# EVOLUCIÓN DE PROCESOS LIGEROS





# PROGRAMA DE EJEMPLO




```
//Crea 10 threads modo joinnable (en bucle for)
#define MAX_THREADS 10
void *func(void *p) {
   printf("Thread %d \n", pthread_self());
   pthread_exit(0);
}
int main(void) {
   int j;
   pthread_attr_t attr;
   pthread_t thid[MAX_THREADS];
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
   for(j = 0; j < MAX_THREADS; j++)
      pthread_create(&thid[j], &attr, &func, NULL);
   for(j = 0; j < MAX_THREADS; j++)
      pthread_join(thid[j], NULL);
   pthread_attr_destroy (&attr);
   return 0;
}
```

# PROGRAMA DE EJEMPLO




```c
//Crea 10 threads modo detached
#define MAX_THREADS 10
void *func(void *p) {
   printf("Thread %d \n", pthread_self());
   pthread_exit(0);
}
int main(void) {
   int j;
   pthread_attr_t attr;
   pthread_t thid[MAX_THREADS];
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
   for(j = 0; j < MAX_THREADS; j ++)
      pthread_create(&thid[j], &attr, &func, NULL);
   pthread_attr_destroy (&attr);
   sleep(5); //Si no se espera, los threads mueren al morir el proceso
   return 0;
}
```