SISTEMAS OPERATIVOS
PROCESOS

Pedro de Miguel Anasagasti

ÍNDICE

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

PROCESOS
CONCEPTOS GENERALES

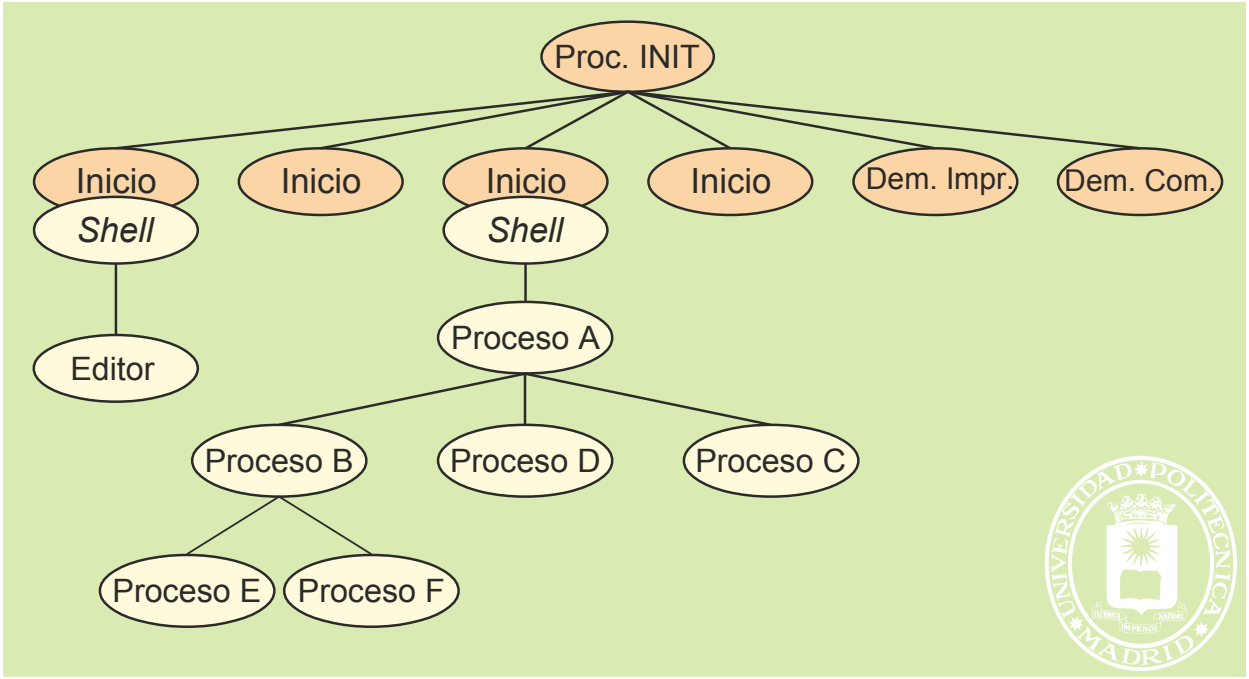
JERARQUÍA DE PROCESOS

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

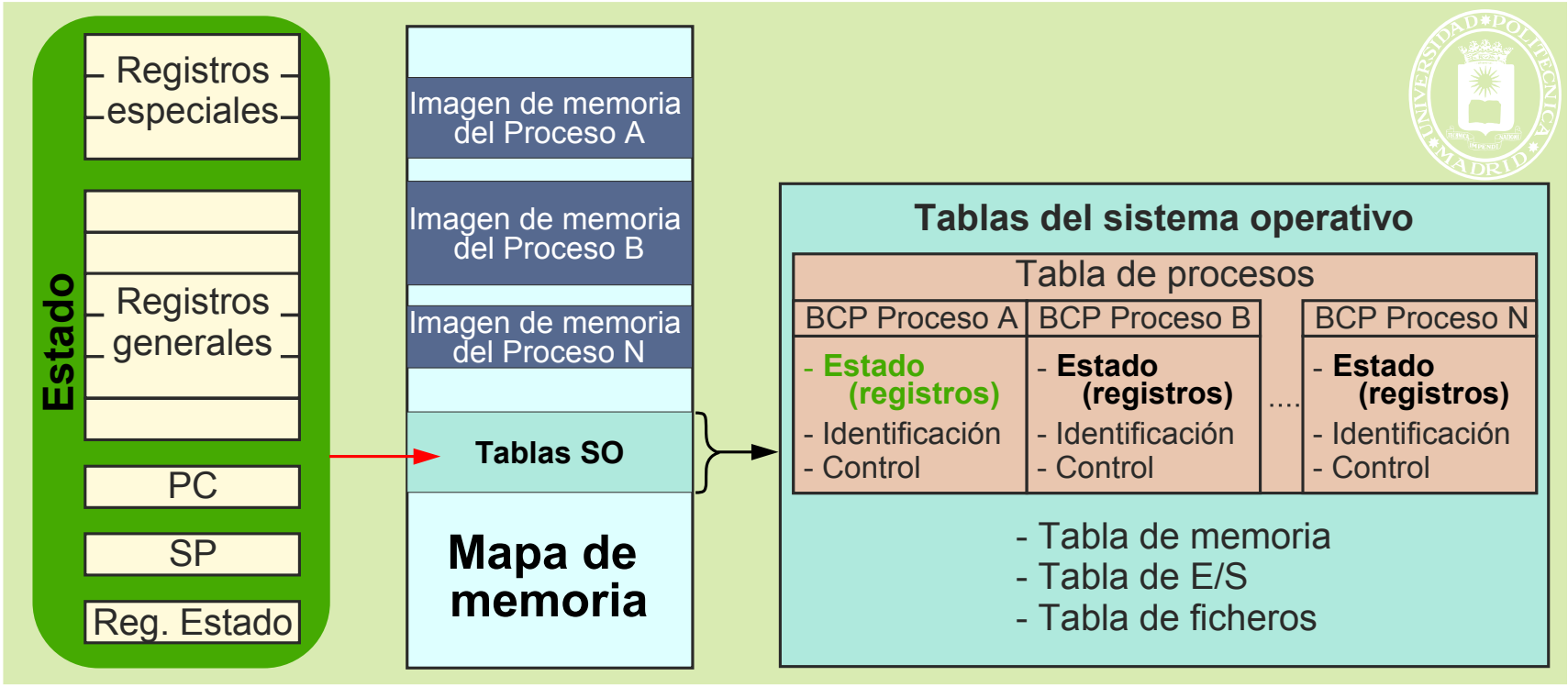
- Grupos de procesos dependientes de cada inicio.

INFORMACIÓN DEL PROCESO

Proceso: **Programa en ejecución**
Unidad de procesamiento gestionada por el SO

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 detener la ejecución de un proceso (por una interrupción), se debe copiar el estado del procesador en su BCP correspondiente. Realizado por la rutina de tratamiento de interrupciones

INFORMACIÓN DEL BCP I

Información de identificación (identifica al usuario y al proceso)

- pid del proceso.
- pid del padre.
- Identificador de usuario real: uid real.
- Identificador de grupo real: gid real.
- Identificador de usuario efectivo: uid efectivo.
- Identificador de grupo 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 de control del proceso I

En esta sección se incluye diversa información que permite gestionar al proceso. Se destacan los siguientes datos:

- **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).**
  - **Puertos asignados.**
- **Comunicación entre procesos. Espacio para almacenar señales y algún mensaje enviado al proceso.**

Información de control del proceso II

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

INFORMACIÓN FUERA DEL BCP

Justificación de información fuera del BCP.

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

Tabla de páginas: Se pone fuera.

- **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.**

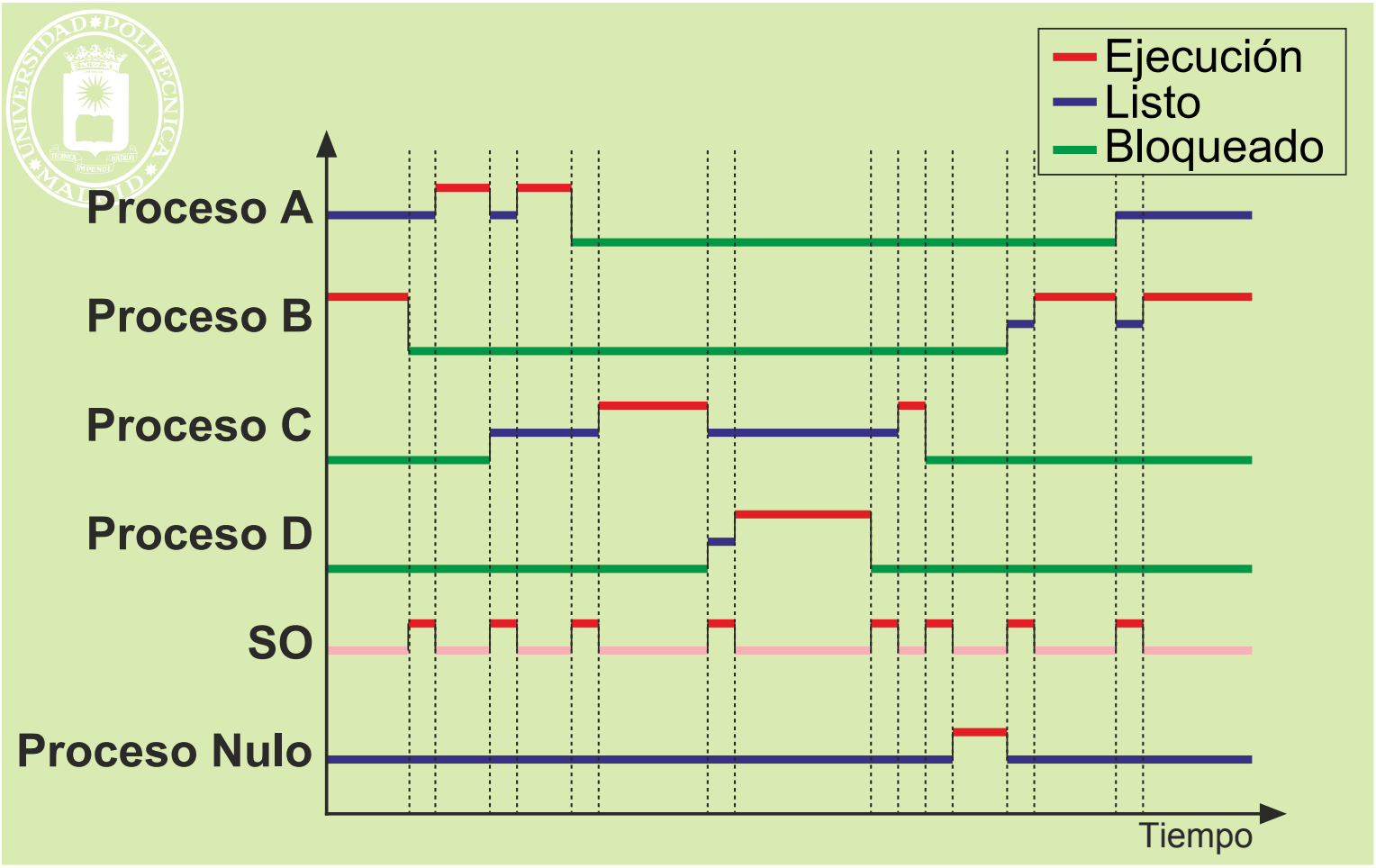
Punteros de posición de los ficheros.

- **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.**


UNIVERSIDAD POLITECNICA
MADRID
TECHNICA
NATIONI
IMPENDI

MULTITAREA

CONCEPTO DE MULTITAREA I



Proceso nulo

CONCEPTO DE MULTITAREA II

Ventajas de la multiprogramación

- Facilita la programación, dividiendo los programas en procesos(modularidad).
- Permite el servicio interactivo simultáneo de varios usuarios deforma 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

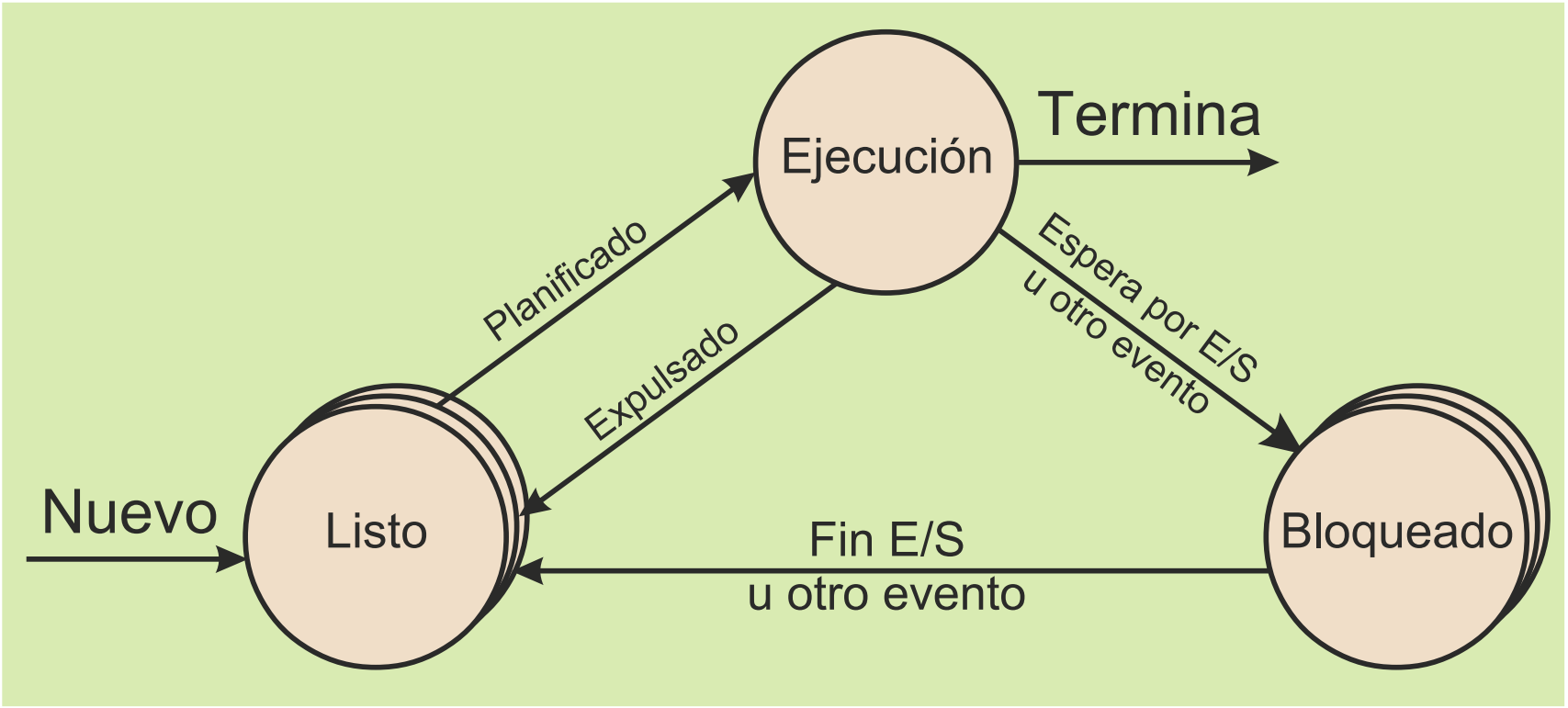
- Nº de procesos en memoria (procesos activos).

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.

ESTADOS DEL PROCESO

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

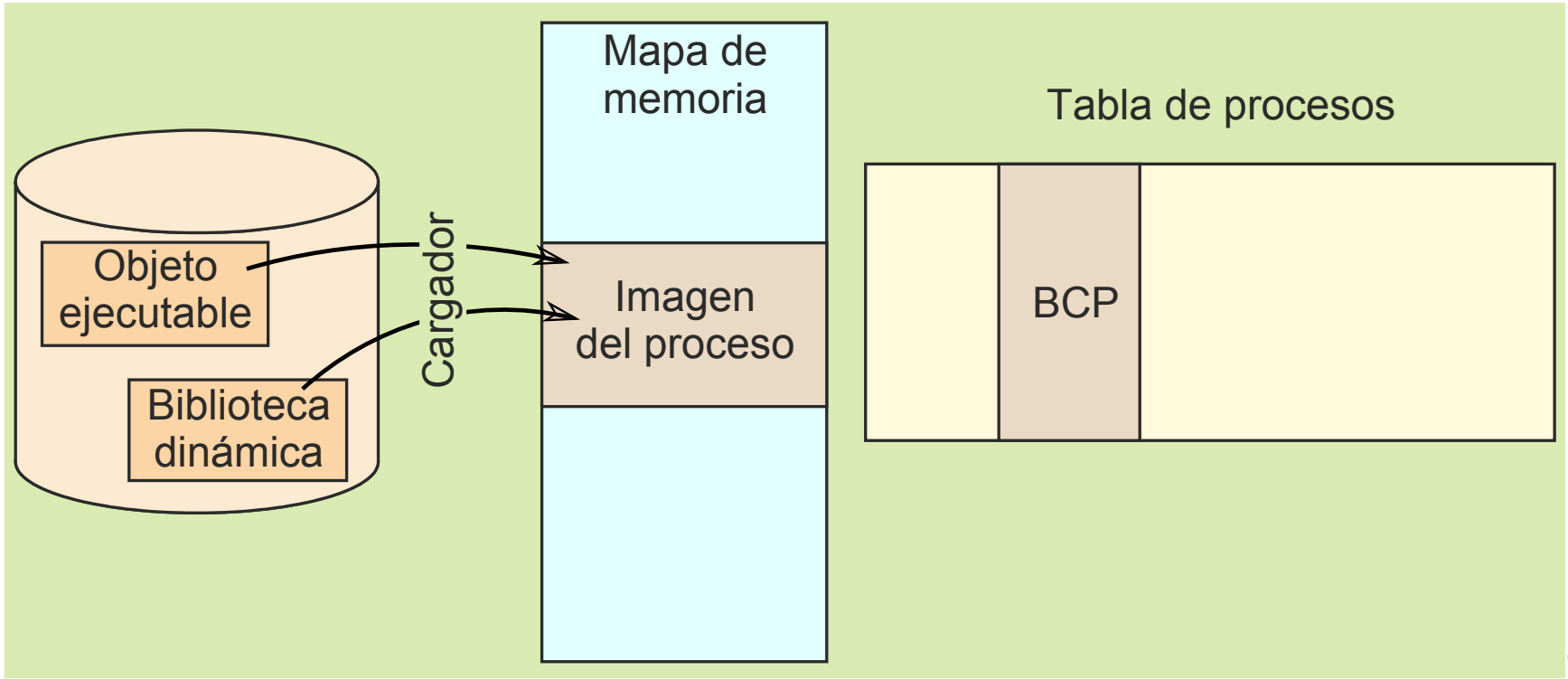


Planificador: Módulo del SO que decide qué proceso listo se ejecuta.

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

Creación del proceso (listo)

- 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 II

Interrupción del proceso (paso a bloqueado/listo)

- Causas: Interrupción, excepción o solicitud de servicio
- Se salva el estado del procesador

Activación del proceso (paso a ejecución)

- Lo selecciona el planificador
- Se restituye el estado del procesador
  - Se termina con el registro de estado seguido del contador de programa

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

CAMBIO DE CONTEXTO I

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 → Cambio de modo: de proceso a SO

- Una interrupción hace que se pase de ejecutar el proceso a ejecutar el SO, lo que significa un cambio de modo de ejecución.
- Necesidad de salvar el estado. Ejemplo:

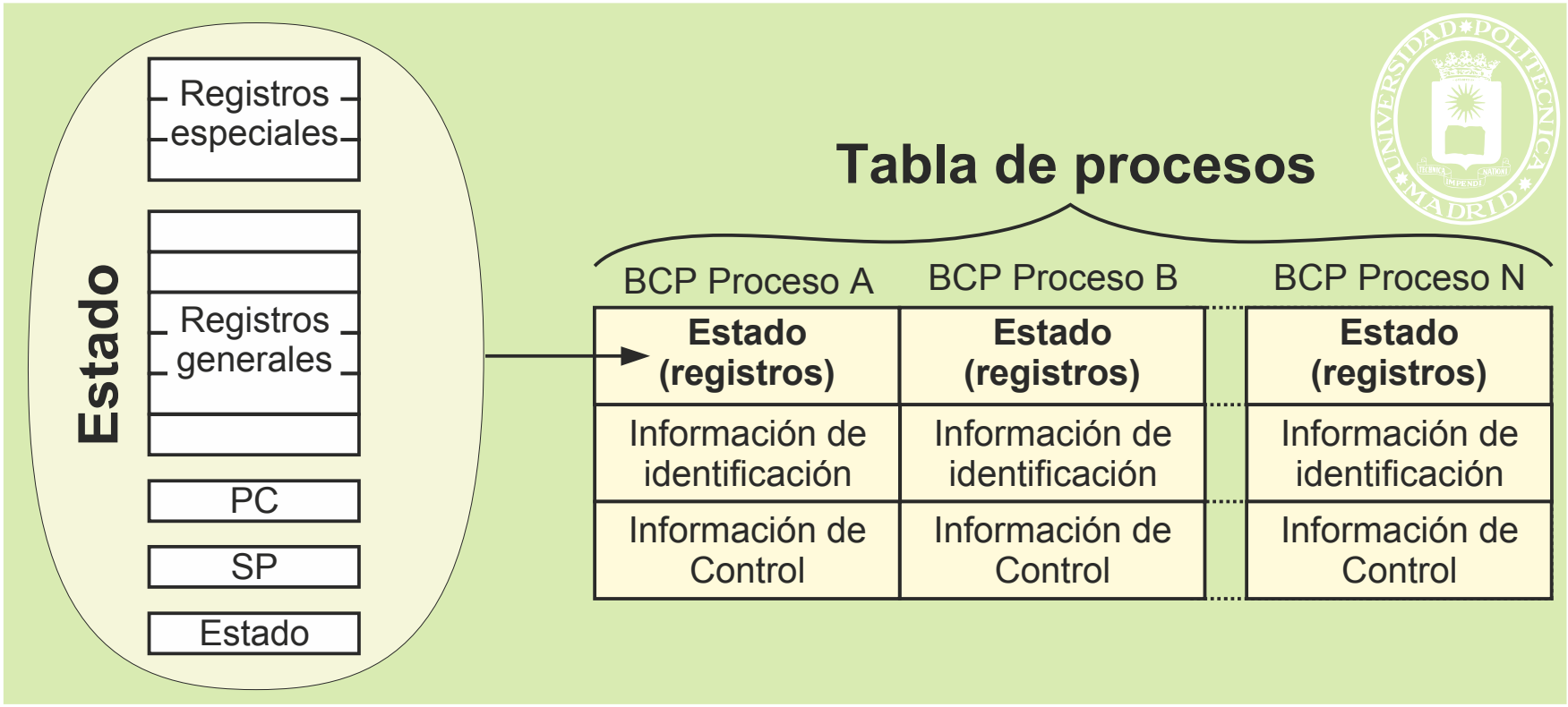
```
LD      .5,#CANT
```

⟶ 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 II



Activación → Cambio de modo: de SO a proceso

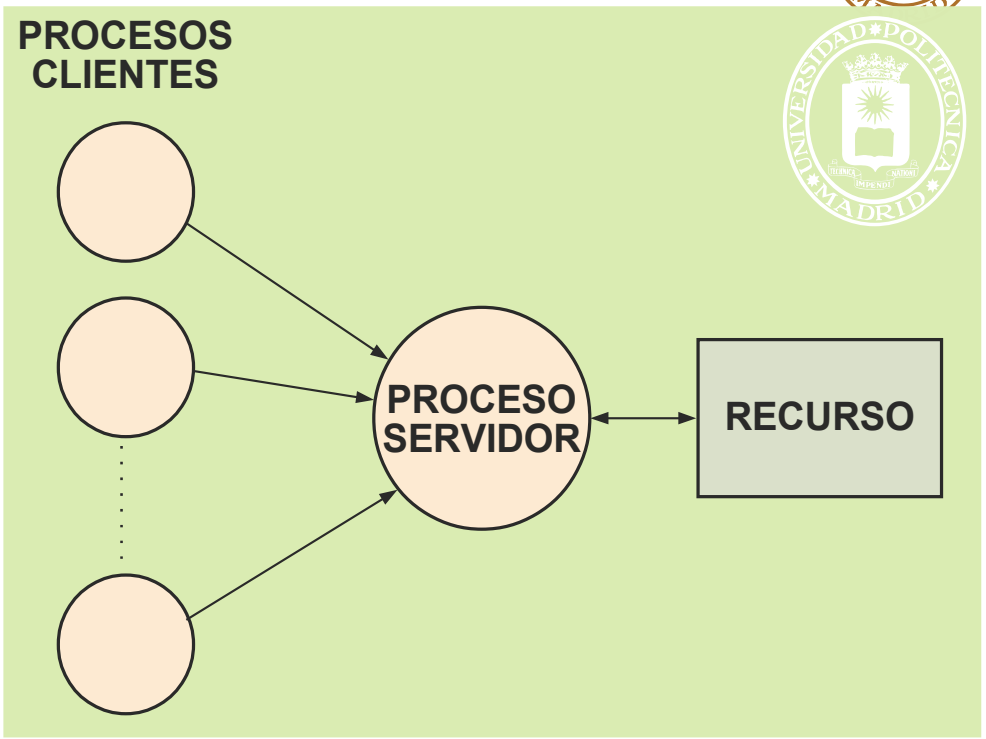
- **Se pasa de ejecutar el SO a ejecutar un proceso.**
- **Hay que restaurar el estado del proceso que está almacenado en el BCP.**

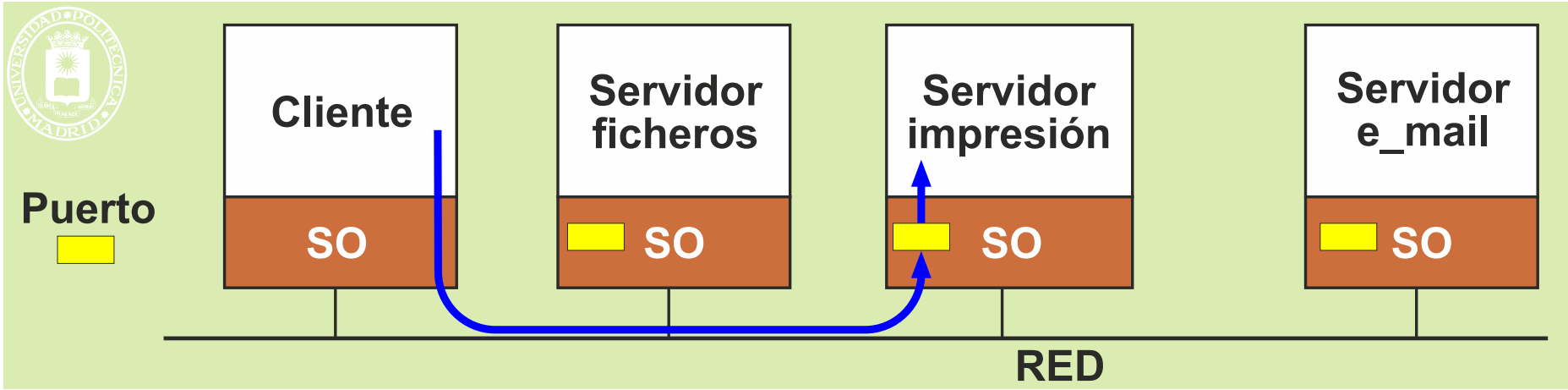
PROCESOS ESPECIALES

PROCESO SERVIDOR

Servidor

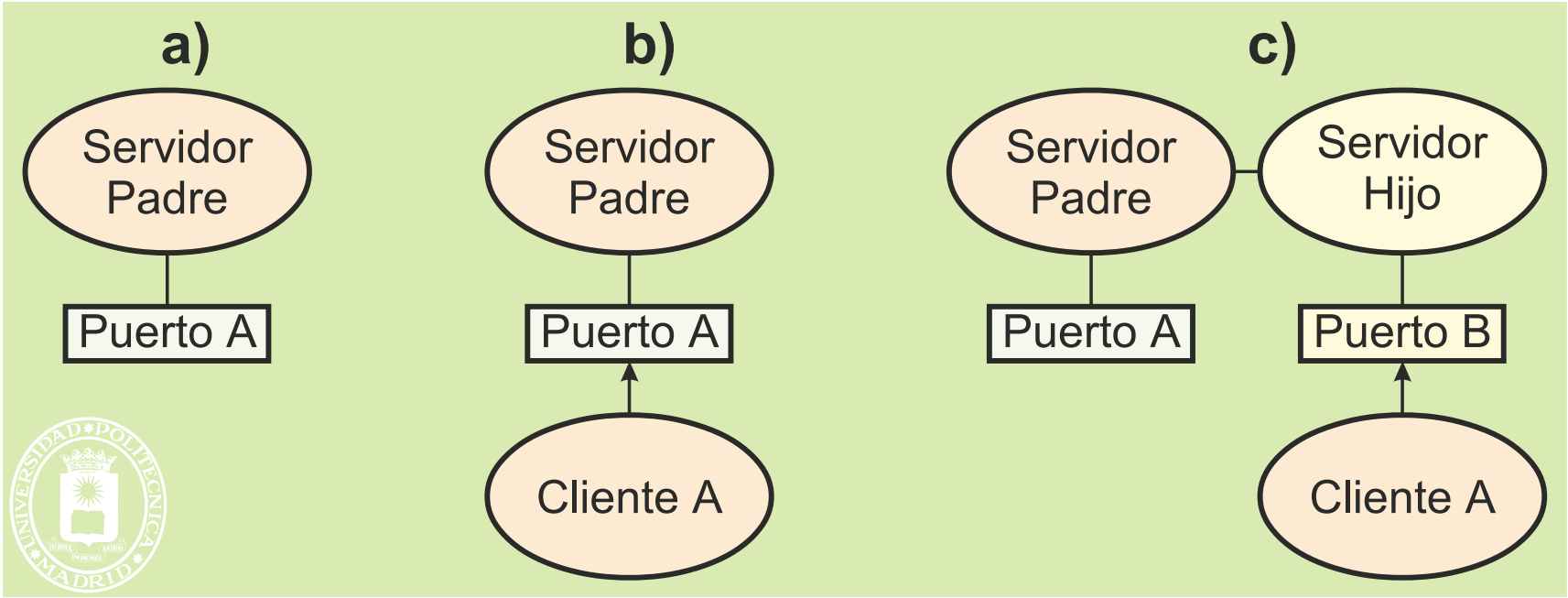
- Atiende peticiones.
- Realiza el bucle:
  - Lectura de orden.
  - Ejecución de la orden.
  - Contestación al cliente.





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.

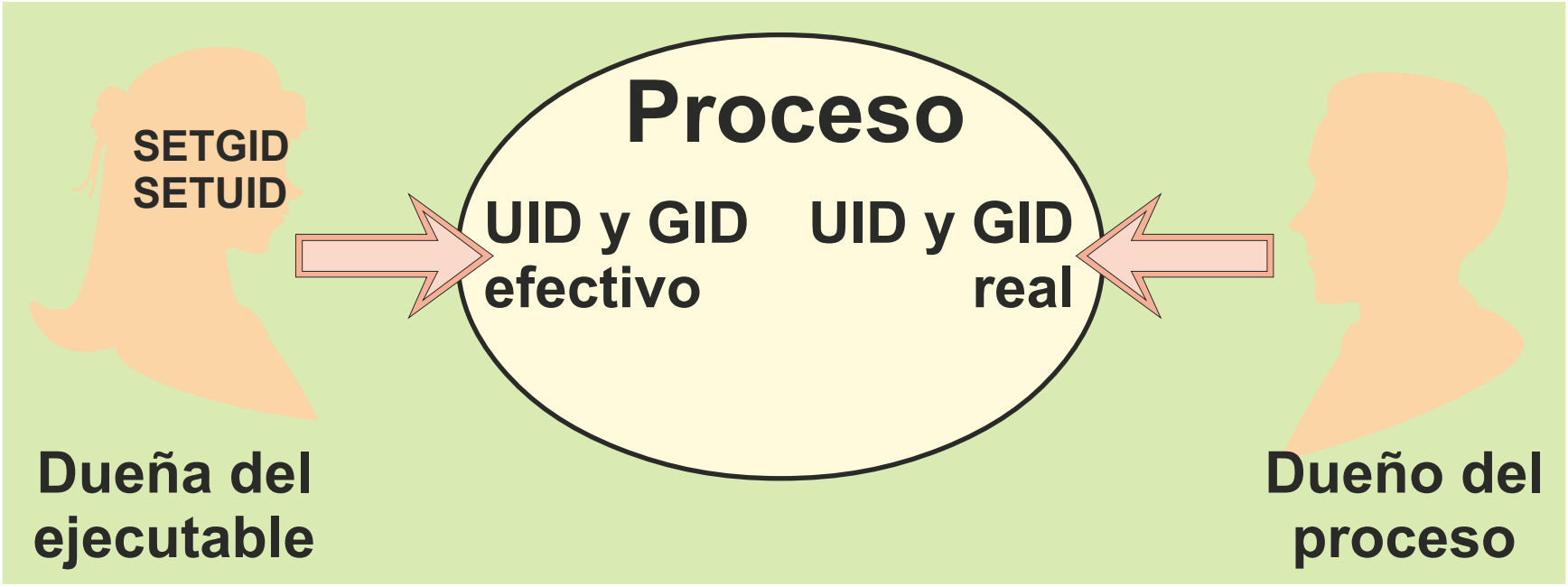
Características

- 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.

SERVICIOS UNIX
GESTIÓN DE PROCESOS



El SO utiliza el UID y GID efectivos para determinar los privilegios del proceso.

El fichero ejecutable puede tener activos los bits SETGID y SETUID.

- Si SETUID activo se pone como UID efectivo el UID del fichero ejecutable.
- Si SETGID activo se pone como GID efectivo el GID del fichero 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 getgid(void);`
  - Devuelven el identificador de usuario efectivo y del grupo efectivo.
- `int setuid(uid_t uid);`
  - Si el proceso es privilegiado se cambian el real y el efectivo. Si no es privilegiado es igual al seteuid.
- `int seteuid(uid_t euid);`
  - Establece el usuario efectivo. Si no es privilegiado solamente puede poner como efectivo su real, por ejemplo:

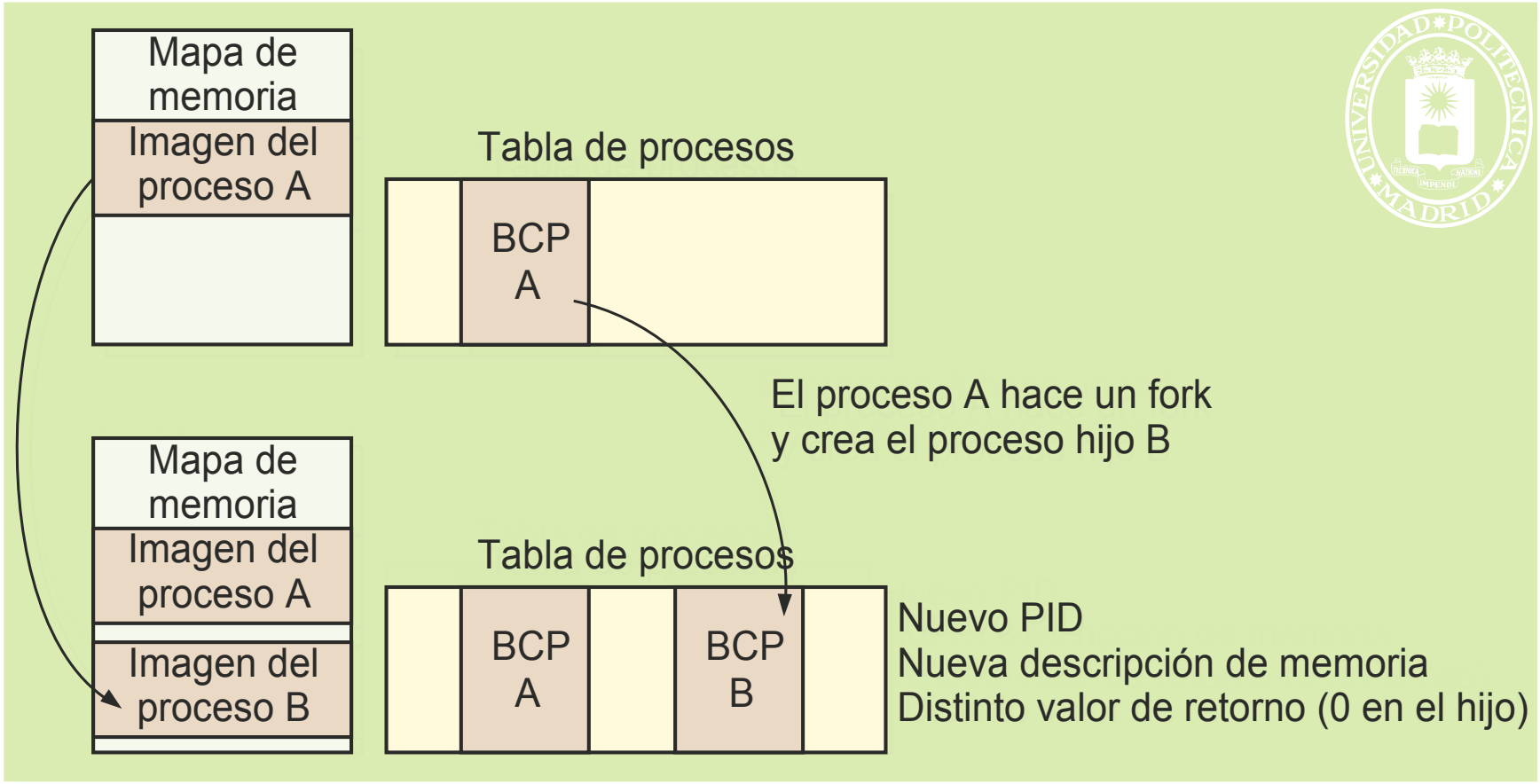
```
seteuid (getuid);
```

SERVICIOS UNIX DE GESTIÓN DE PROCESOS (Entorno)

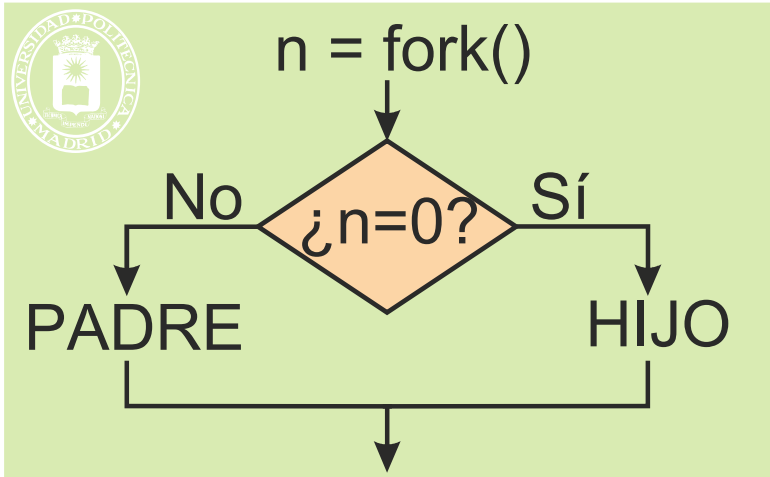
- `extern char *environ[];`
  - **Variable que apunta al entorno.**
- **Funciones de biblioteca:**
  - `char *getenv(const char *name);`
    - Devuelve el valor de la variable de entorno **name**.
      - » HOME, directorio de trabajo inicial del usuario.
      - » LOGNAME, nombre del usuario asociado 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)

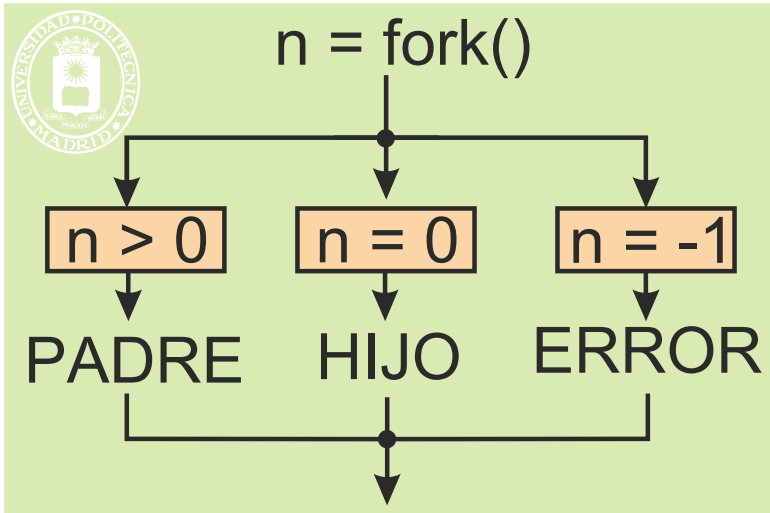
- `pid_t fork(void);`
  - **Crea un proceso hijo. Devuelve 0 al proceso hijo y el pid del hijo al**
  - **proceso padre.**
  - **El hijo es un clon del padre pero con nuevo pid, imagen de memoria y valor de retorno del fork.**



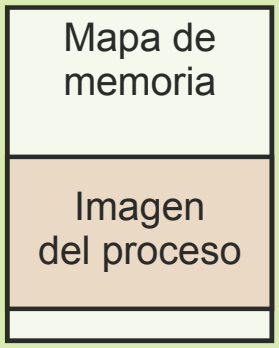
```
int n;
if ((n = fork()) == -1) {
   perror("Error al llamar a fork");
   /* perror: Función que imprime el texto más el
    tipo de error del SO. */
} else if (n == 0){    //Código del HIJO
      .... código del hijo ....
} else {               //Código del PADRE
      .... código del padre ....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



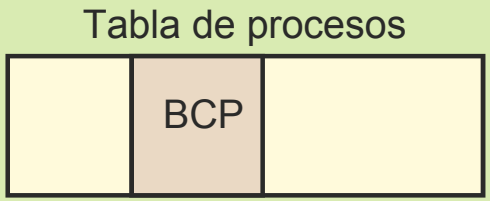

```
switch(fork()) {
case -1:
   perror("Error al llamar a fork");
   break;
case  0: //Código del HIJO
     .... código del hijo ....
   break;
default: //Código del PADRE
   .... código del padre ....
}
```



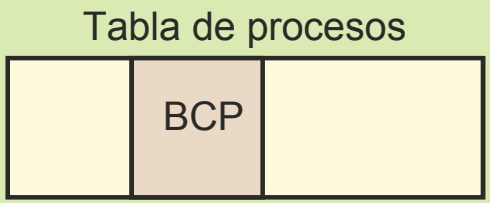
EL EXEC CAMBIA EL PROGRAMA DE UN PROCESO



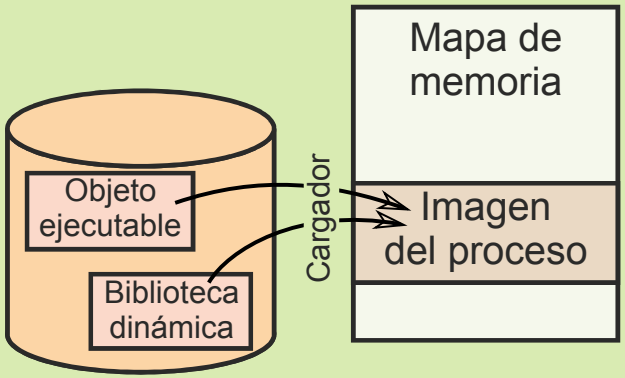
Ejecución de exec



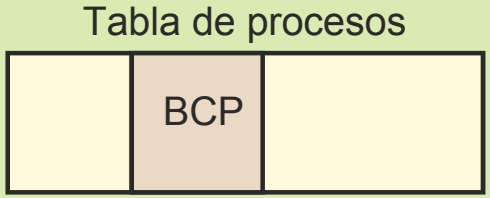
Vaciado del proceso



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)

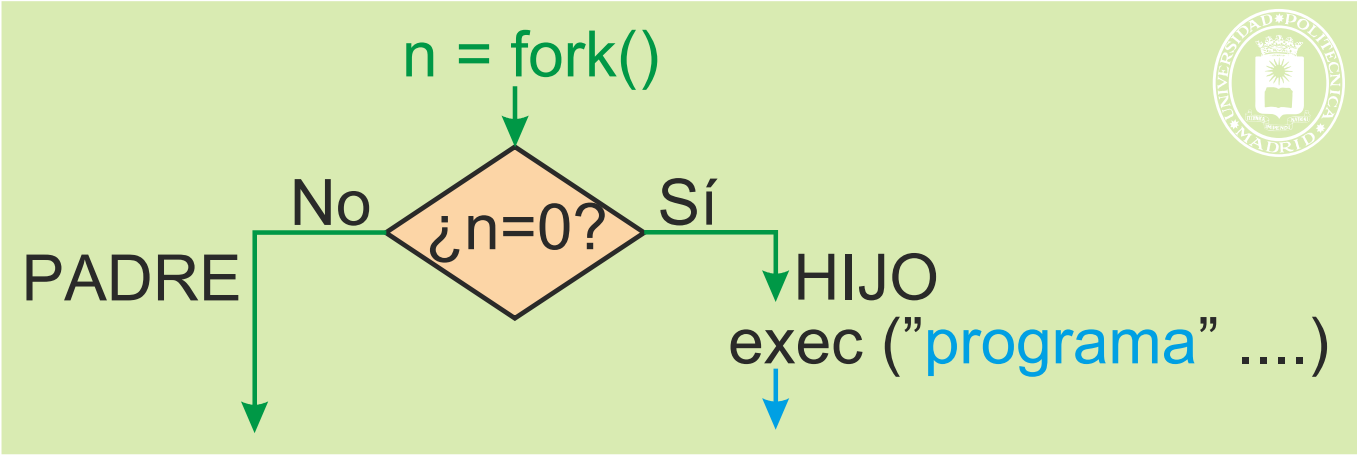

```
int execve(const char *path, char *const argv[], char
*const envp[]);
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
```

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 (puede 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 ....
```



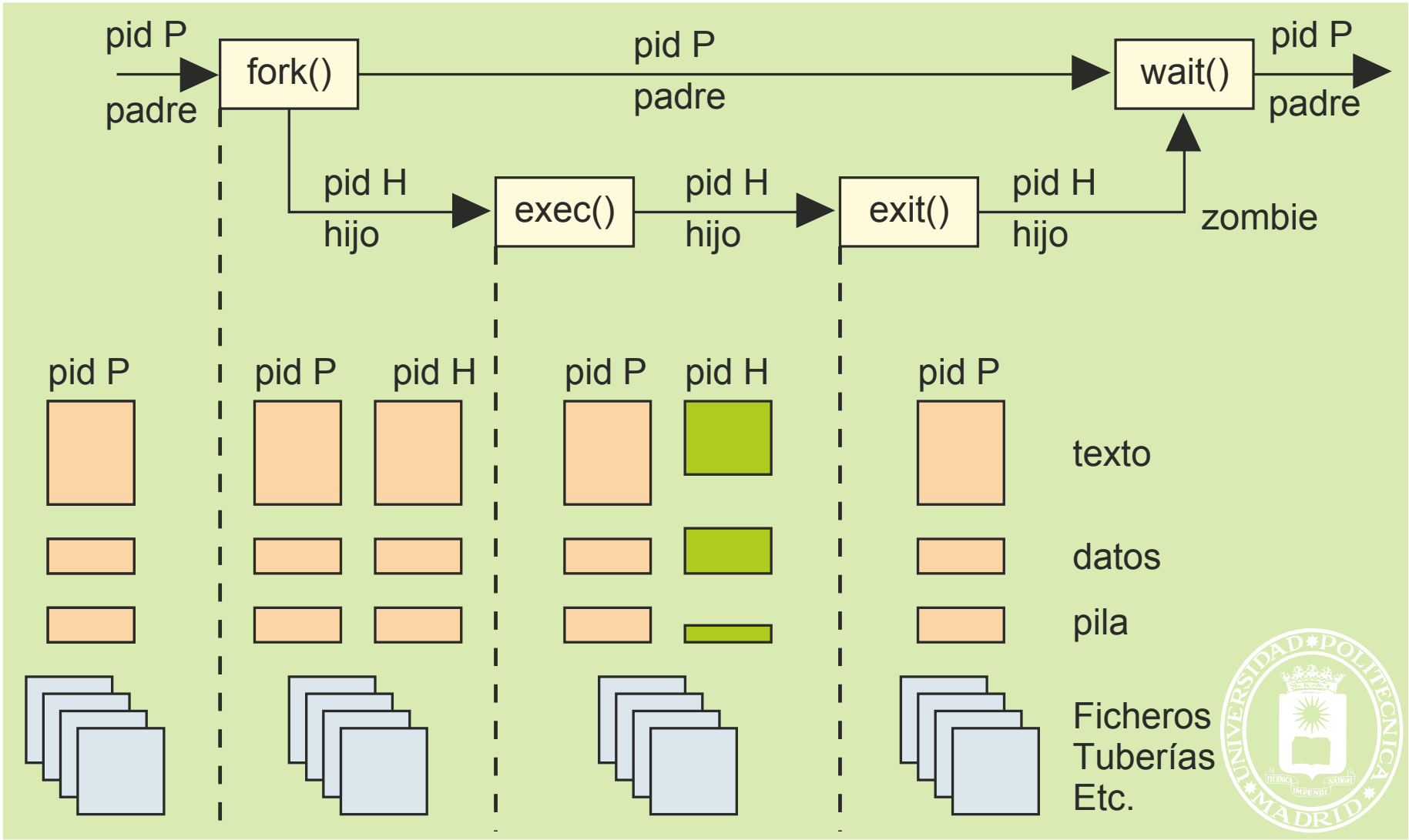
SERVICIOS GESTIÓN DE PROCESOS (Terminación I)

- `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.
  - Es más elegante terminar el `main` con un `return` que con `exit`
- `clock_t times(struct tms *buf);`
  - Retorna el tiempo real de ejecución y en buf suministra el tiempo de procesador consumido en modo usuario y el tiempo de procesador consumido en modo sistema por el proceso, así como el acumulado de los hijos terminados.

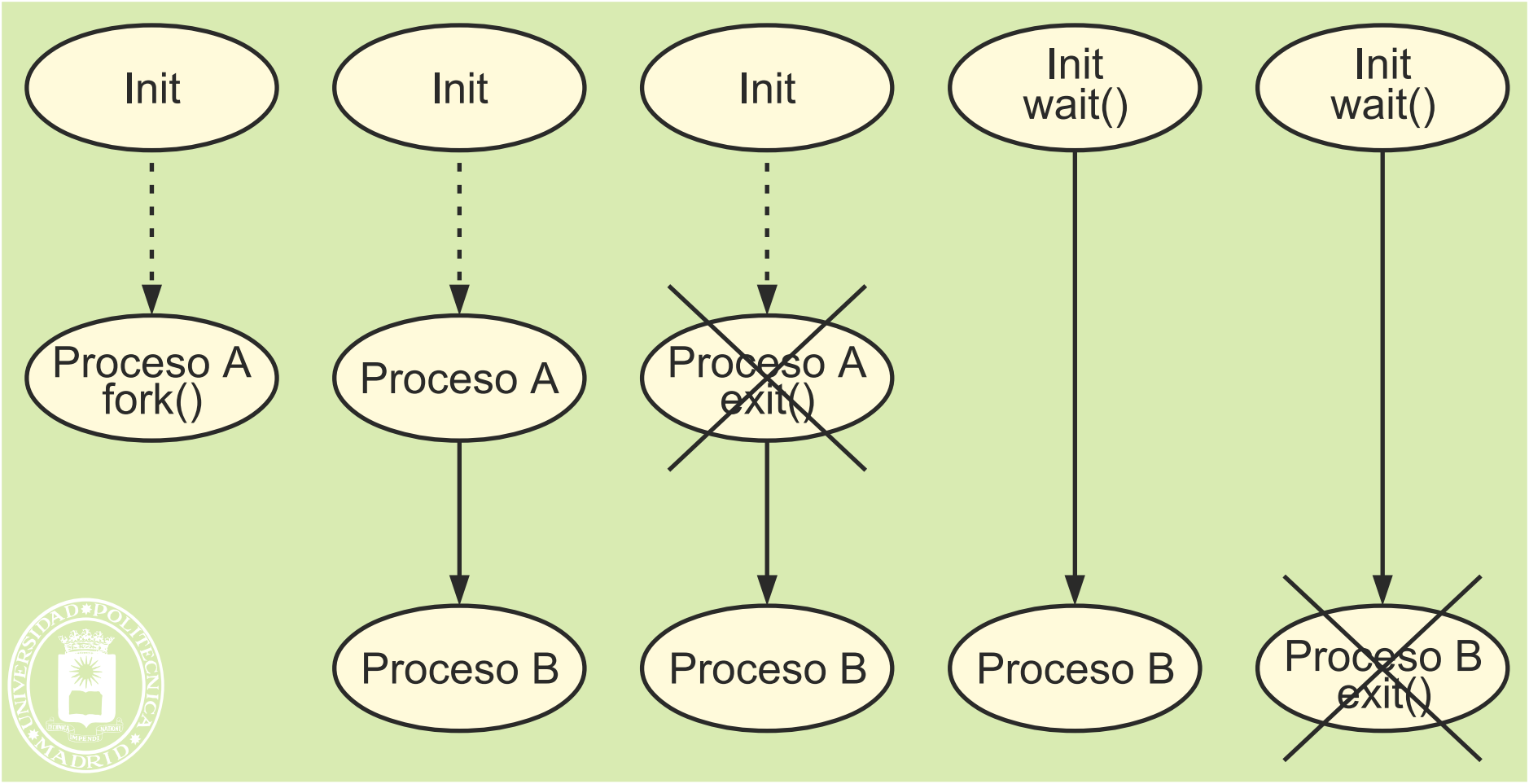
La variable `status` en `wait` y `waitpid`

- 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`

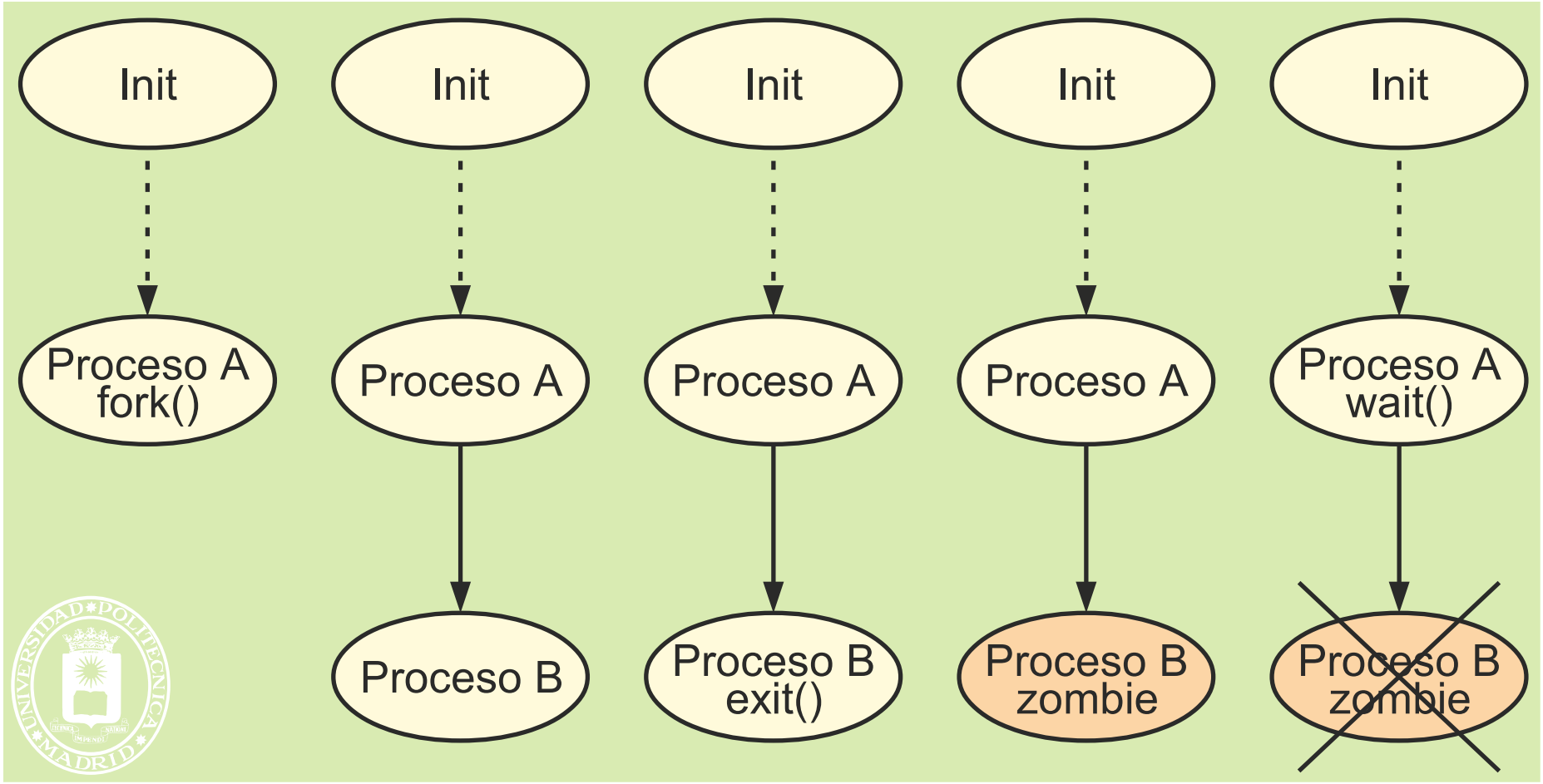
SERVICIOS GESTIÓN DE PROCESOS (UNIX)



El padre muere: INIT acepta los hijos.



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



PROGRAMA DE EJEMPLO I


```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;
}
```

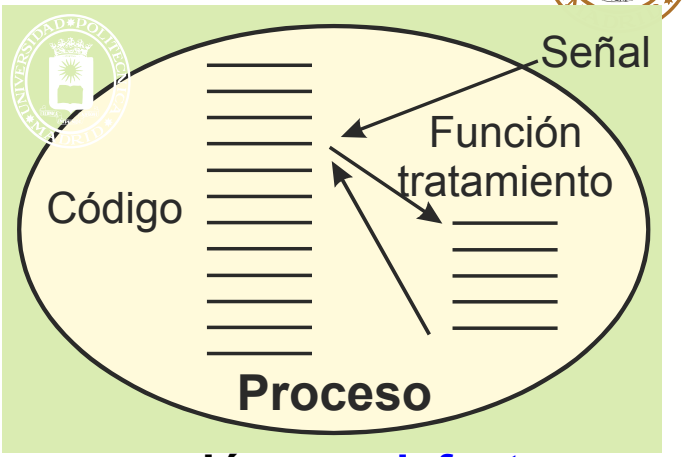
SEÑALES Y TEMPORIZADORES

SEÑALES

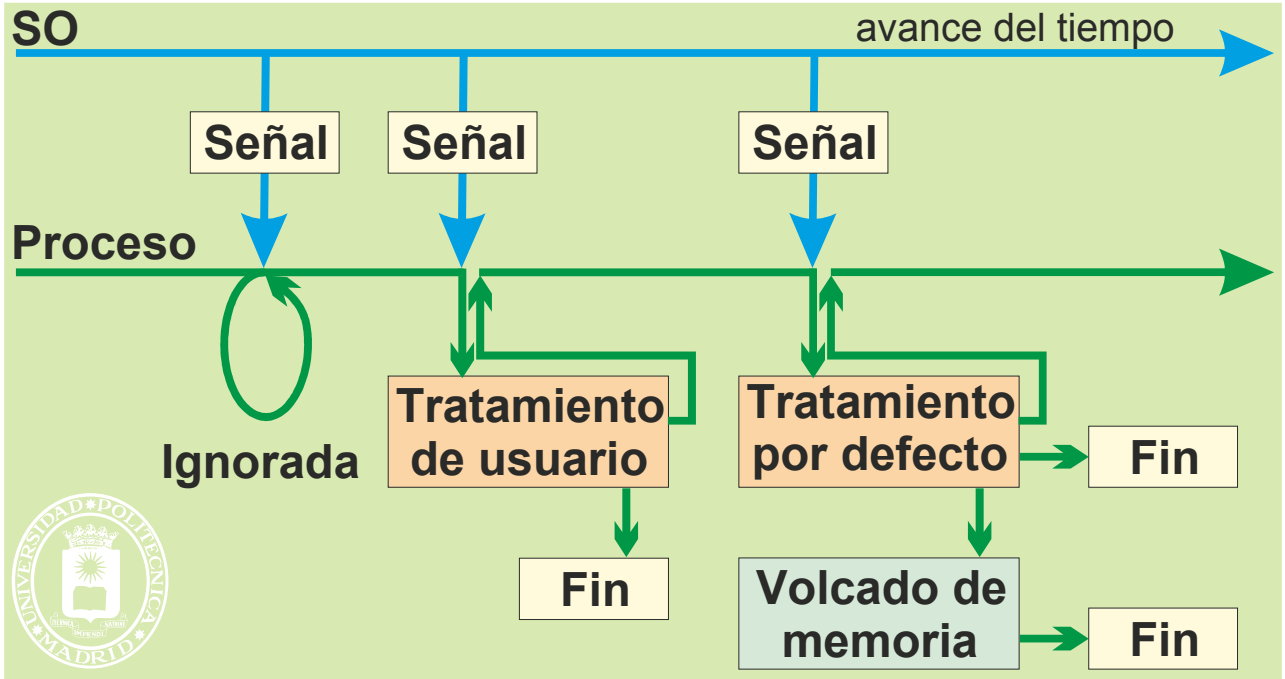
- **Desde el punto de vista del proceso, una señal:**
  - **Es un evento que recibe (a través del SO)**
  - **Interrumpe al proceso**
  - **Le transmite información muy limitada (un número, que identifica el tipo de señal)**
  - **Un proceso también puede enviar señales a otros procesos (del mismo grupo),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



- **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**
- **Servicios de espera de señales:**
  - **El servicio 'pause' detiene el proceso hasta que recibe una señal.**
  - **El servicio 'sleep' despierta al proceso cuando ha transcurrido el tiempo establecido o cuando se recibe una señal.**
- **Máscara**
  - **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 `sigprogmask`.**

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.

TIPOS DE SEÑALES

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

- **SIGABRT, terminación anormal.**
- **SIGALRM, señal de fin de temporización.**
- **SIGFPE, operación aritmética errónea.**
- **SIGHUP, colgado del terminal de control.**
- **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).**

TEMPORIZADORES

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

SERVICIOS UNIX
SEÑALES Y TEMPORIZADORES

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 señal se produce estando en llamada bloqueante, después de tratamiento sigue en la llamada
  - Por defecto, llamada termina con error y errno = EINTR

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.

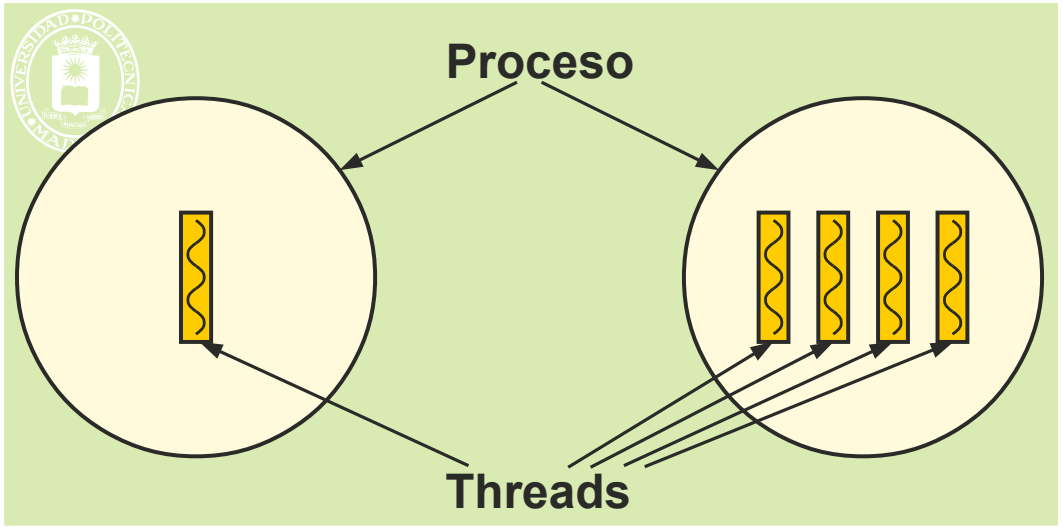
PROGRAMA DE EJEMPLO II


```
/* 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;
}
```

THREADS (PROCESOS LIGEROS)

- 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.

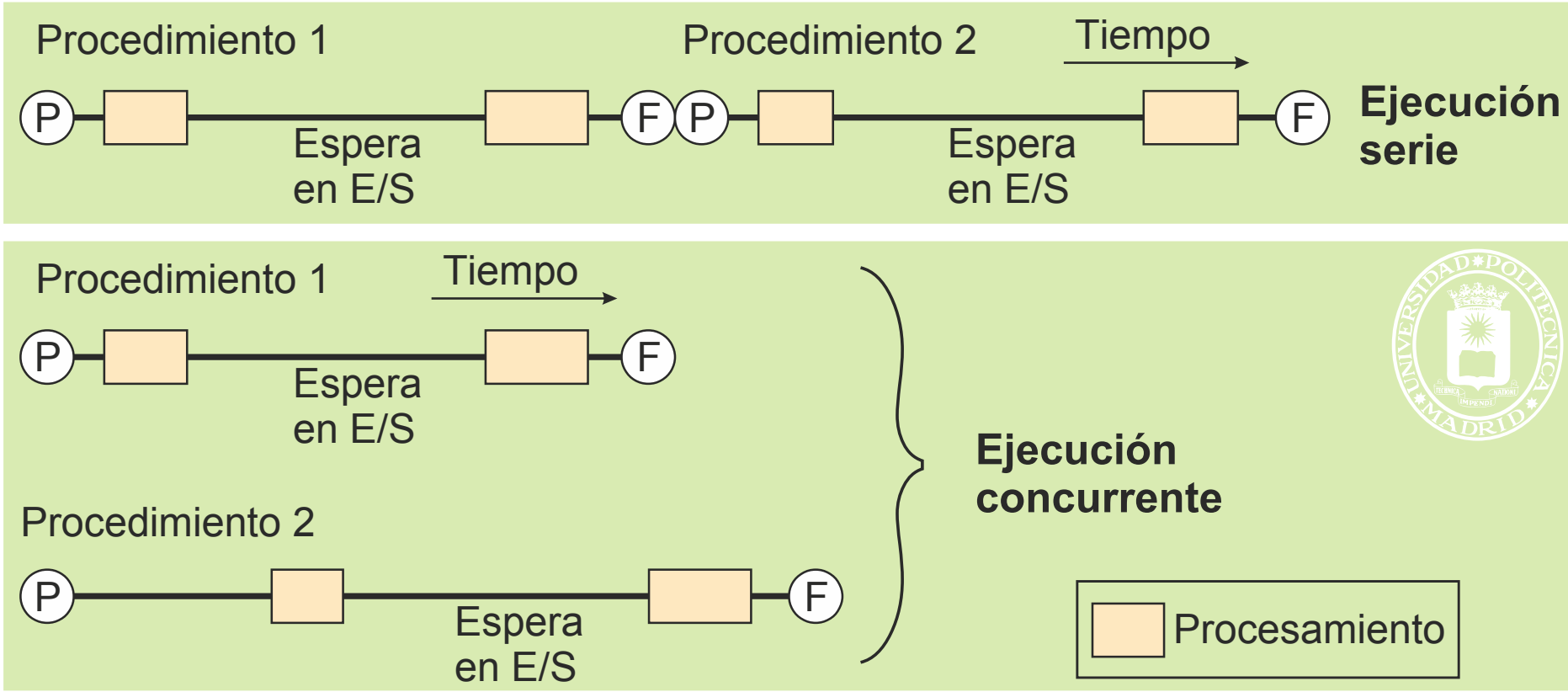


UNIX 1003.1.

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.**



ALTERNATIVAS DE DISEÑO

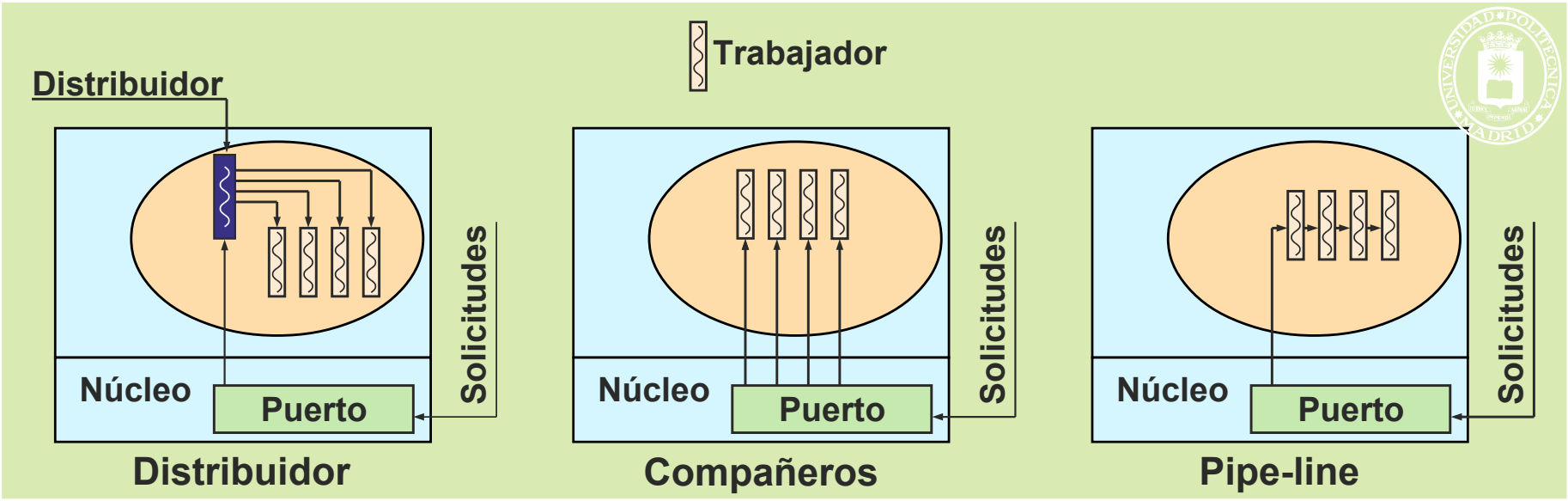
- Proceso con un único thread.
  - No hay concurrencia.
  - Llamadas al sistema bloqueantes.
- Proceso estructurado en máquina de estados finitos.
  - Concurrencia.
  - Llamadas al sistema no bloqueantes.
- Múltiples procesos convencionales cooperando.
  - Permite concurrencia.
  - No comparten variables.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

DISEÑO CON THREADS

- 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) [simplicidad vs. exclusión en el acceso]
  - 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)**



VENTAJAS E INCONVENIENTES DE LOS THREADS

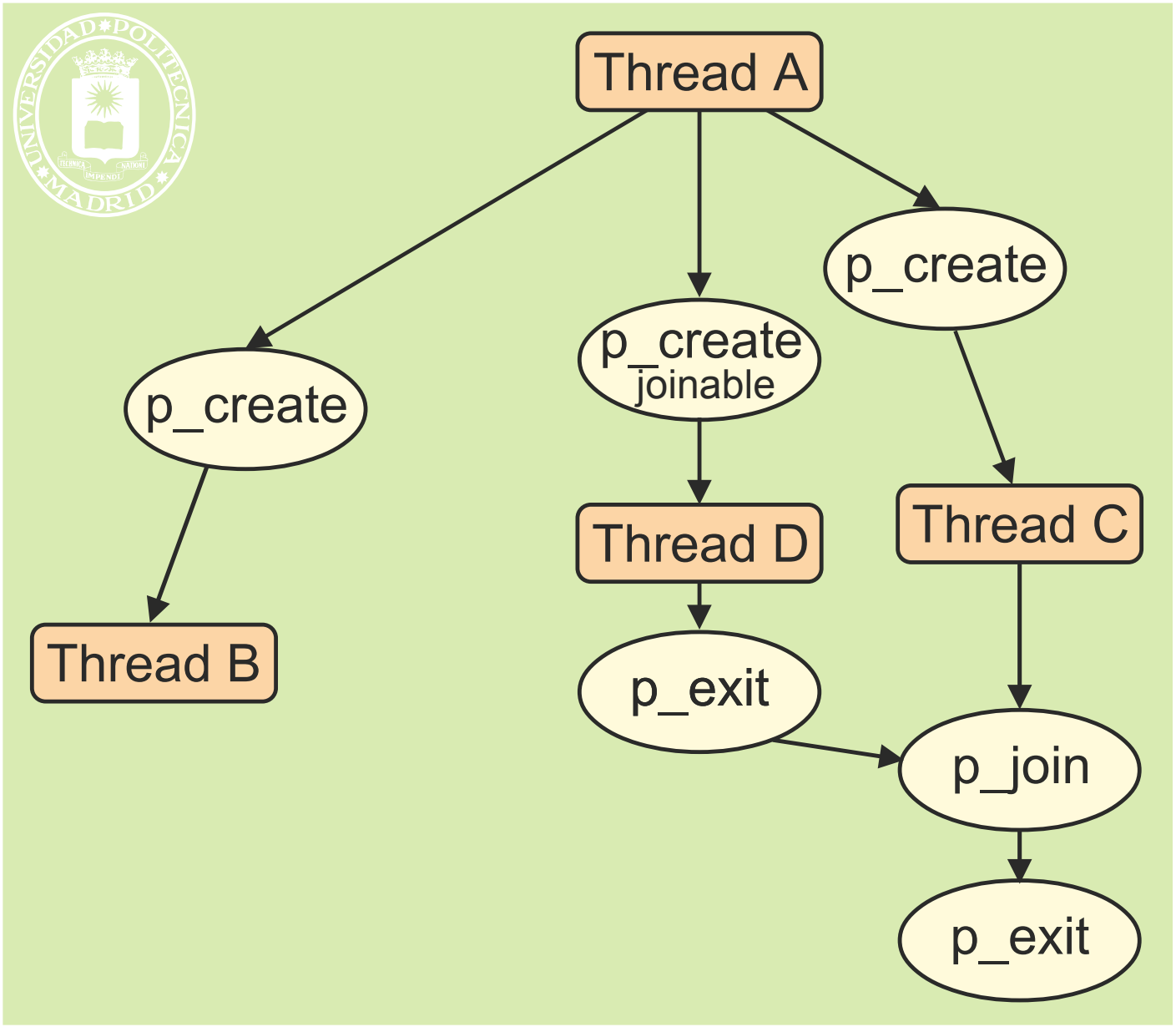
- Ventajas inherentes a la programación concurrente.
  - Separación de tareas.
  - Facilita la modularidad.
  - Aumenta el aprovechamiento de la CPU por una tarea.
- Mejores prestaciones que con procesos.
  - Los threads comparten memoria → variables comunes.
  - Sincronización entre procesos ligeros más sencilla.
- Menor número de cambios de contexto del proceso.
  - Sólo se bloquea el proceso ligero que hace la llamada al SO (cierto solamente para KLT)
- 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.

SERVICIOS UNIX
THREADS

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`.**
  - **Los atributos permiten especificar: tamaño de la pila, prioridad, política de planificación, etc.**
    - *joinable*: thread no desaparece hasta que otro haga pthread_join de él
    - *detached*: thread desaparece al terminar
  - **Existen diversas llamadas para modificar los atributos.**
- `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.**
- `int sched_yield(void)`
  - **El thread que la ejecuta cede el control a otro thread listo para ejecutar.**
- `int pthread_yield(void)`
  - **Cesión voluntaria del procesador**

EVOLUCIÓN DE PROCESOS LIGEROS



PROGRAMA DE EJEMPLO III


```
//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 IV


```
//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;
}
```