Tema 2. Procesos e hilos

UNIVERSITAT JAUME·I

Índice

- Concepto de proceso
- Información del proceso
- Estados del proceso
- Formación de un proceso
- Cambio de contexto
- Servicios POSIX para gestión de procesos
- Concepto de hilo de ejecución
- Servicios POSIX para gestión de hilos
- Planificación de procesos e hilos

Tema 2. Procesos e hilos

Bibliografía

- J. Carretero et al. *Sistemas Operativos: Una Visión Aplicada.* McGraw-Hill. 2007. Apartados 3.1-3.5, 3.8-3.9, 3.12, 3.13.1 y 3.13.2. Tema 4.
- José Manuel Badía, M. A. Castaño, Miguel Chóver, Javier Llach, R. Mayo. Introducción Práctica al Sistema Operativo UNIX. Col.lecciò "Material docent". Servicio de Publicaciones de la UJI. 1996. Apartado 16.3.1.

UNIVERSITAT JAUME·I

Resultados de aprendizaje

- Relacionar el concepto de proceso e hilo con el modelo de ejecución de la arquitectura y los problemas inherentes de planificación, comunicación y sincronización.
  - Definir qué es un proceso
  - Describir la información asociada a un proceso
  - Describir los estados de un proceso
  - Describir cuándo y cómo se producen las transiciones entre los estados de un proceso
  - Distinguir entre estado de un proceso y estado de procesador
  - Describir qué información de un proceso se guarda en el descriptor de un proceso, cuál no y por qué
  - Describir qué tablas tiene el SO para la gestión de procesos

UNIVERSITAT JAUME·I

Resultados de aprendizaje (cont.)

- Distinguir entre cambio de contexto y cambio de proceso
- Analizar cómo tiene lugar en un sistema un cambio de contexto que no implica cambio de proceso
- Analizar cómo cambia en un sistema el proceso que está en ejecución en el procesador en un instante dado
- Implementar sencillos programas escritos en C que utilicen llamadas al sistema relacionadas con la gestión básica de procesos
- Describir qué es un hilo y justificar las ventajas y desventajas de utilizar hilos frente a procesos
- Implementar sencillos programas escritos en C que utilicen llamadas al sistema para la gestión de hilos
- Explicar la necesidad de planificar el acceso a los diferentes recursos del sistema
- Explicar y aplicar diversos algoritmos de planificación del procesador

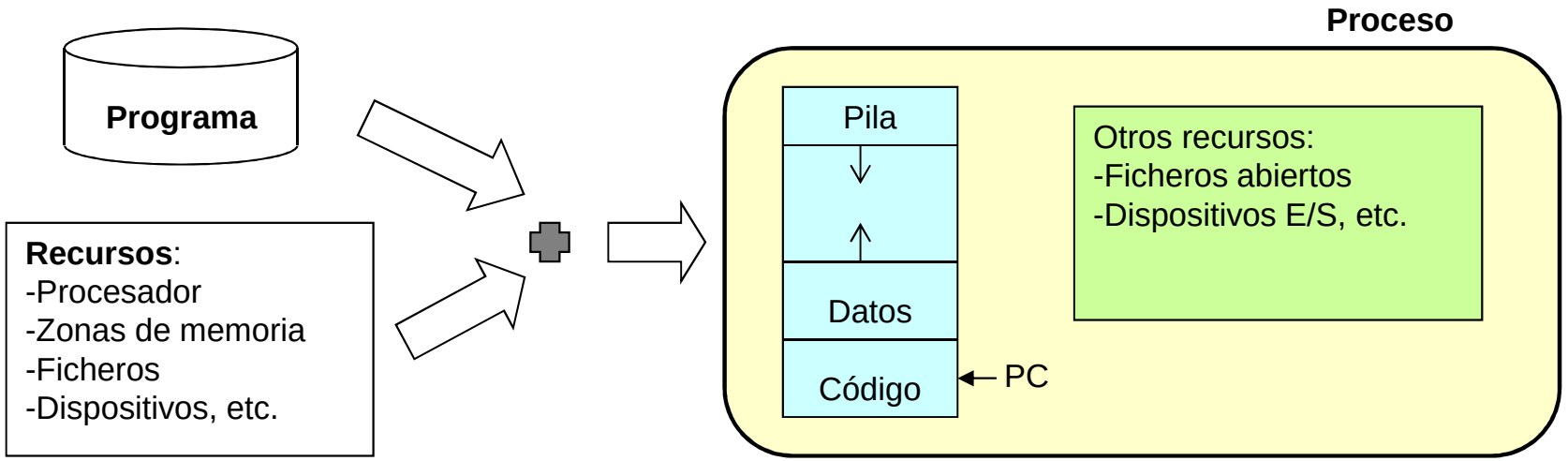
Tema 2. Procesos e hilos

UNIVERSITAT JAUME·I

Índice

Concepto de proceso

- **¿Qué es un proceso?**
  - Instancia de un programa en ejecución, requiriendo para ello unos recursos

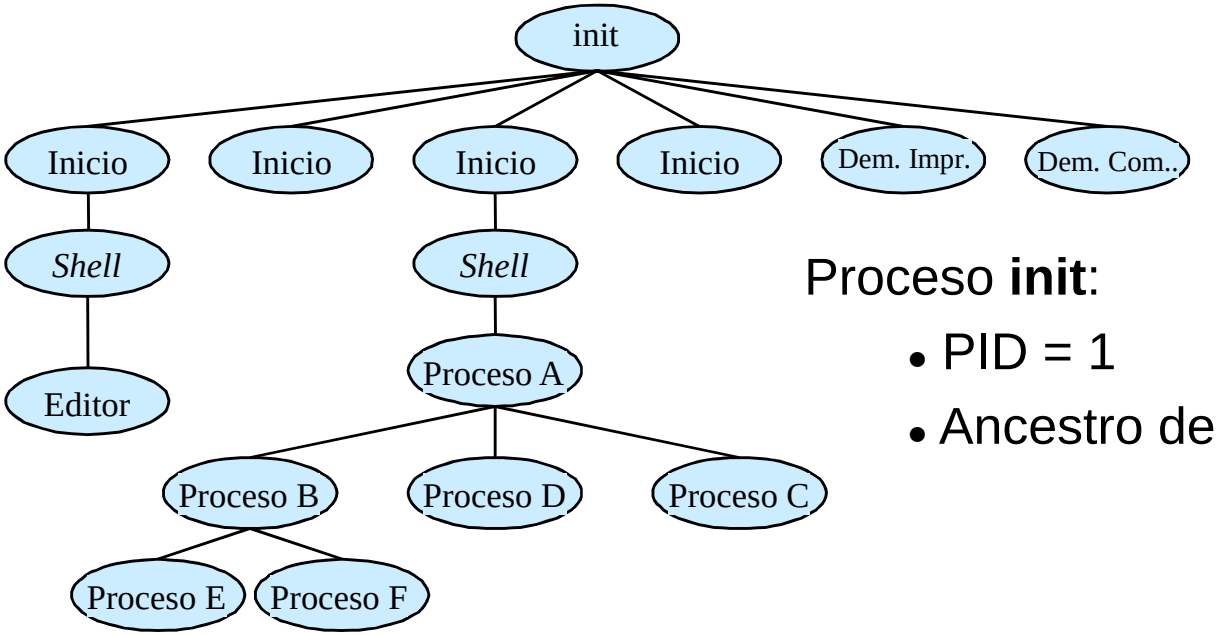


Programa: estructura pasiva

Proceso: estructura activa

Concepto de proceso

- **Jerarquía de procesos:**
  - Algunos SO, como Unix, mantienen una estructura jerárquica entre procesos



Proceso **init**:

- PID = 1
- Ancestro de todos los procesos

  - Otros, como Windows NT/2000 (en adelante WNT/2K), no la mantienen

Concepto de proceso

- **El proceso *nulo* (o la tarea *ociosa*):**
  - ¿Qué ocurre cuando el procesador está ocioso?
    - Se ejecuta el proceso nulo
  - ¿Qué hace el proceso nulo?
    - Ejecuta un bucle infinito que no realiza ninguna operación útil
    - En sistemas Unix suele tener PID=0
  - Objetivo:
    - Entretener al procesador cuando no hay ninguna otra tarea

Concepto de proceso

- **Grupos de procesos:**
  - Los procesos forman grupos de procesos con alguna característica común
    - Conjunto de procesos creados por el mismo padre
    - Conjunto de procesos creados a partir de un *shell*
    - Conjunto de procesos dependientes de un terminal
  - Se pueden realizar ciertas operaciones sobre un grupo de procesos
    - Matar todos los procesos de un grupo de procesos
    - Envío de señales a todos los procesos de un grupo de procesos

Tema 2. Procesos e hilos

UNIVERSITAT JAUME·I

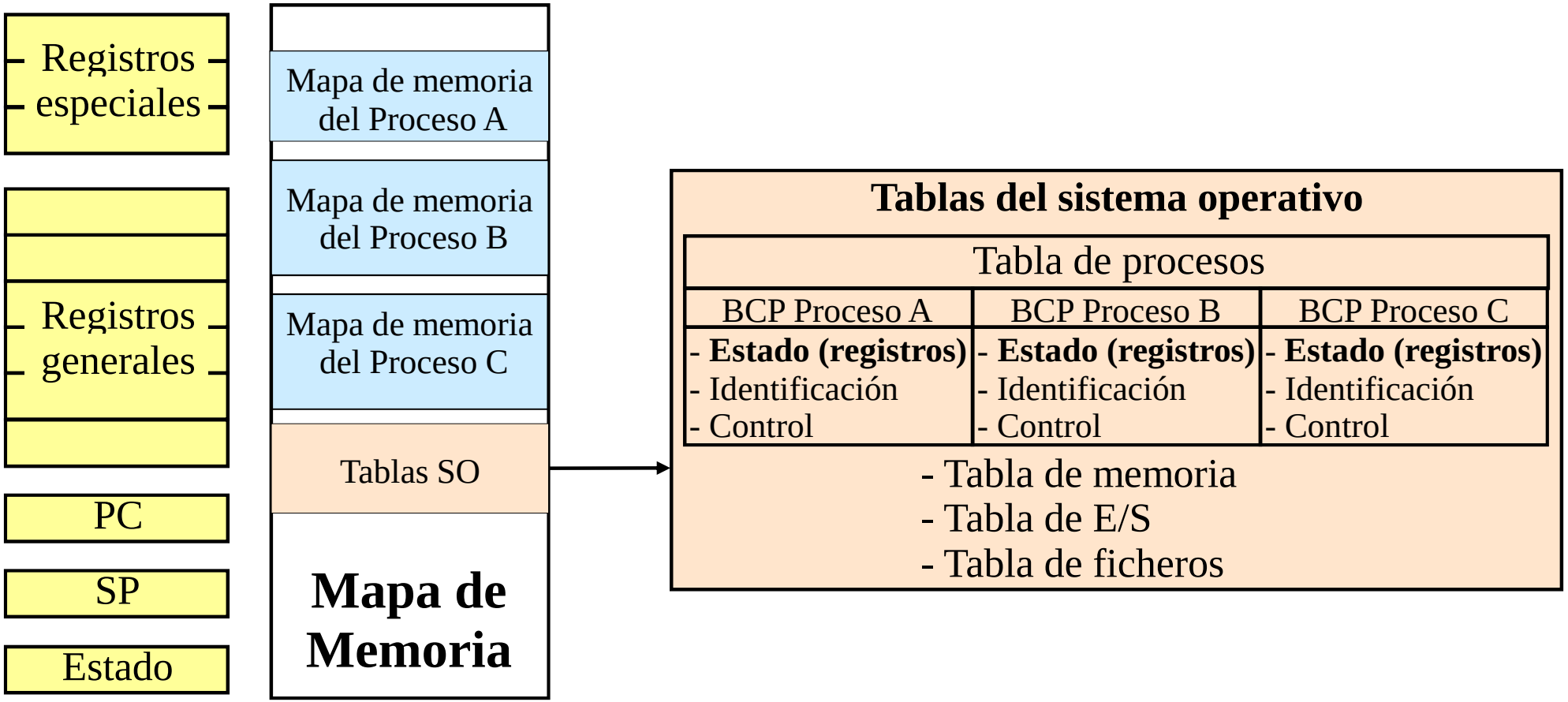
Índice

- Concepto de proceso
- Información del proceso
- Estados del proceso
- Formación de un proceso
- Cambio de contexto
- Servicios POSIX para gestión de procesos
- Concepto de hilo de ejecución
- Servicios POSIX para gestión de hilos
- Planificación de procesos e hilos

Información del proceso

- **Estado del procesador:**
  - Contenido de los registros del modelo de programación
- **Imagen de memoria:**
  - Contenido de los segmentos de memoria en los que reside el código y los datos del proceso
- **Bloque de control de proceso (BCP) o descriptor de proceso**

Información del proceso



Estado del procesador

- Formado por el contenido de todos los registros del procesador:
  - Registros generales
  - Contador de programa
  - Puntero de pila
  - Registro de estado
  - Registros especiales
- Cuando un proceso está ejecutando su estado del procesador reside en los registros del computador
- Cuando un proceso no se está ejecutando su estado del procesador reside en el BCP

Preparación del código de un proceso

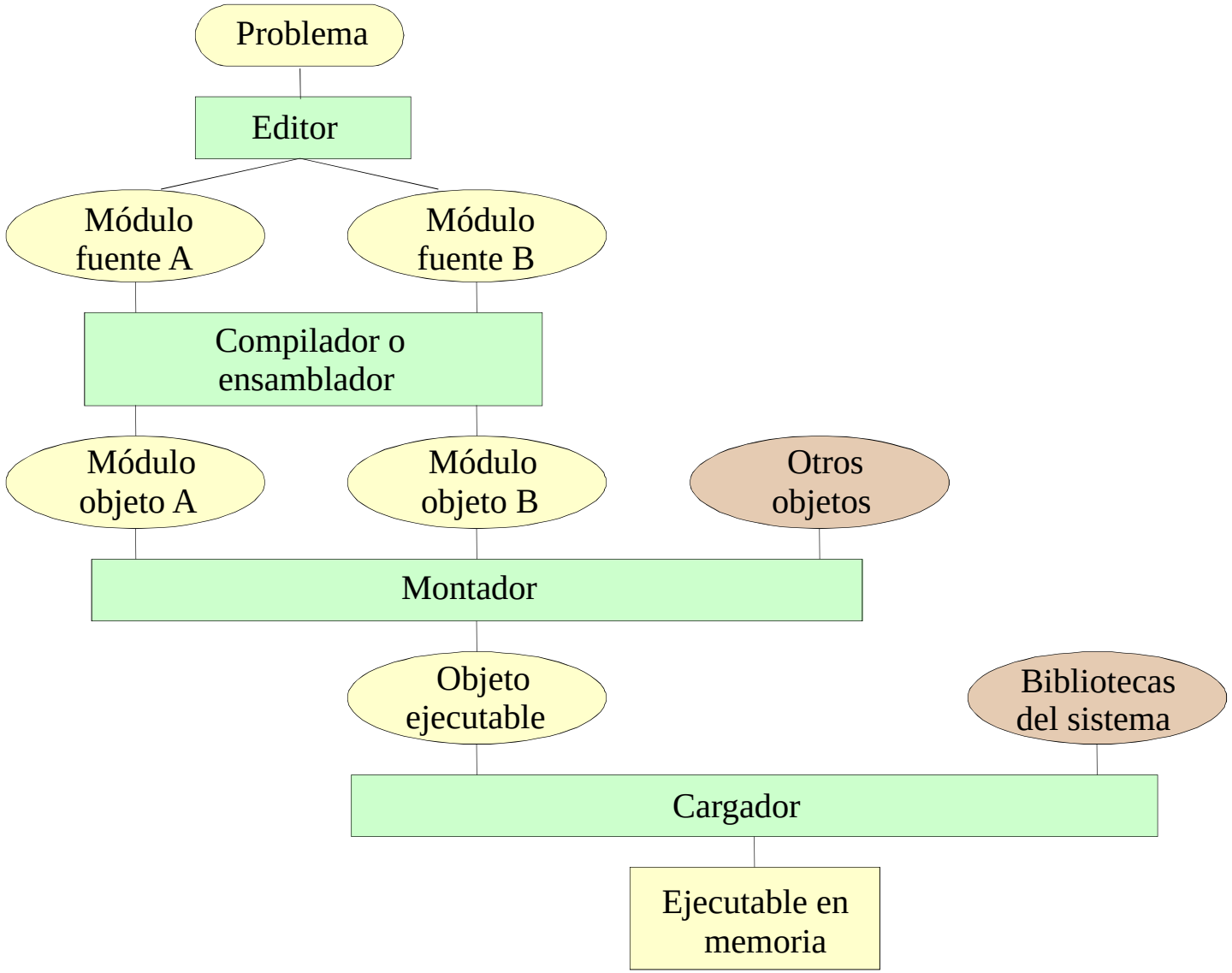
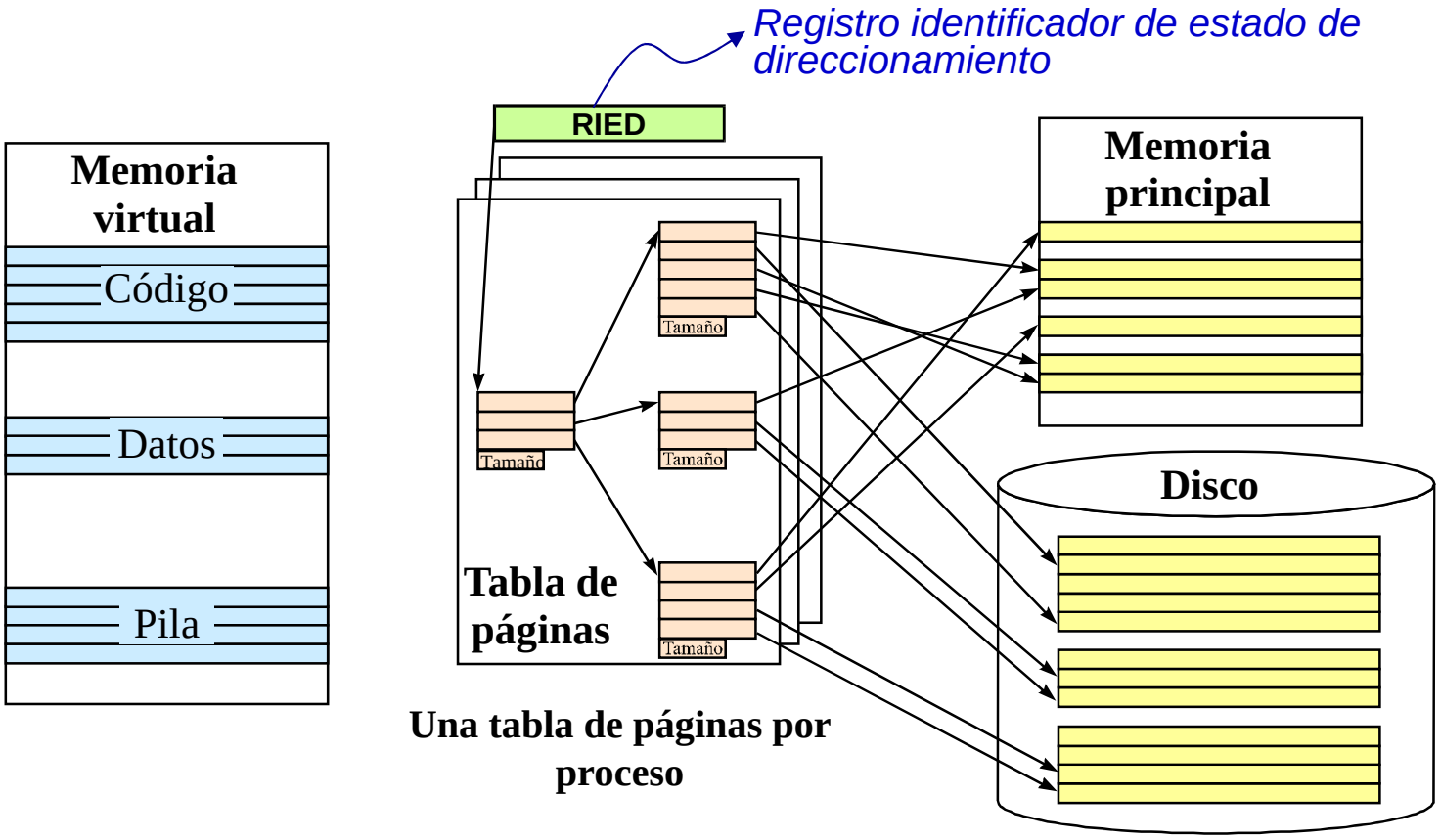


Imagen de memoria

- Formada por los espacios de memoria que un proceso está autorizado a utilizar
- La memoria del proceso la asigna el gestor de memoria del SO
- Si un proceso genera una dirección que esta fuera del espacio de direcciones el HW genera una interrupción HW interna
- La imagen de memoria, dependiendo del computador, puede estar referida a memoria virtual o memoria física

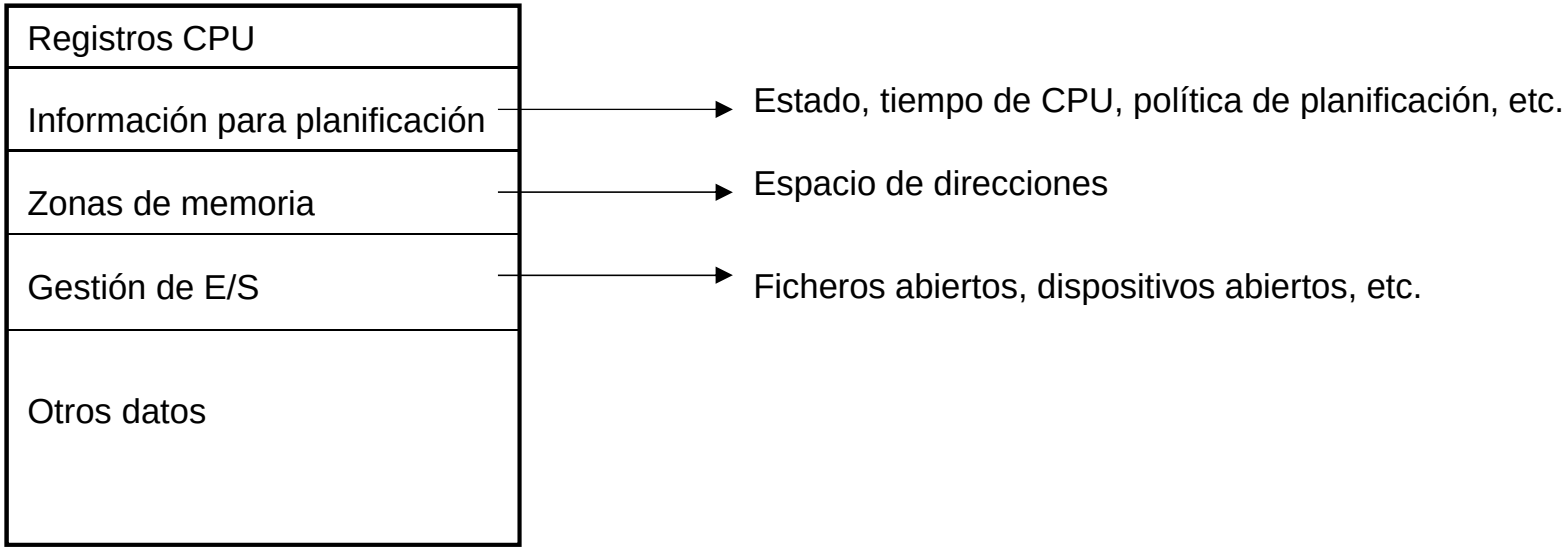
Imagen de memoria

- **Imagen de memoria de un proceso en un sistema con memoria virtual:**



Descriptor de proceso

- **¿Qué es el descriptor de un proceso?**
  - Representación de un proceso en el SO
  - Contiene información asociada al proceso
  - También llamado ***bloque de control del proceso*** **(BCP)**



Información del BCP

- **Información de identificación:**
  - PID del proceso, PID del padre
  - ID de usuario real (*uid real*)
  - ID de grupo real (*gid real*)
  - ID de usuario efectivo (*uid efectivo*)
  - ID de grupo efectivo (*gid efectivo*)
- **Estado del procesador**
- **Información de control del proceso:**
  - Información de planificación y estado
  - Descripción de los segmentos de memoria del proceso
  - Recursos asignados (ficheros abiertos, ...)
  - Comunicación entre procesos (señales, ...)
  - Punteros para estructurar los procesos en listas o colas
  - Información de auditoría (tiempo de CPU consumido, ...)

Tabla de ficheros abiertos por el proceso

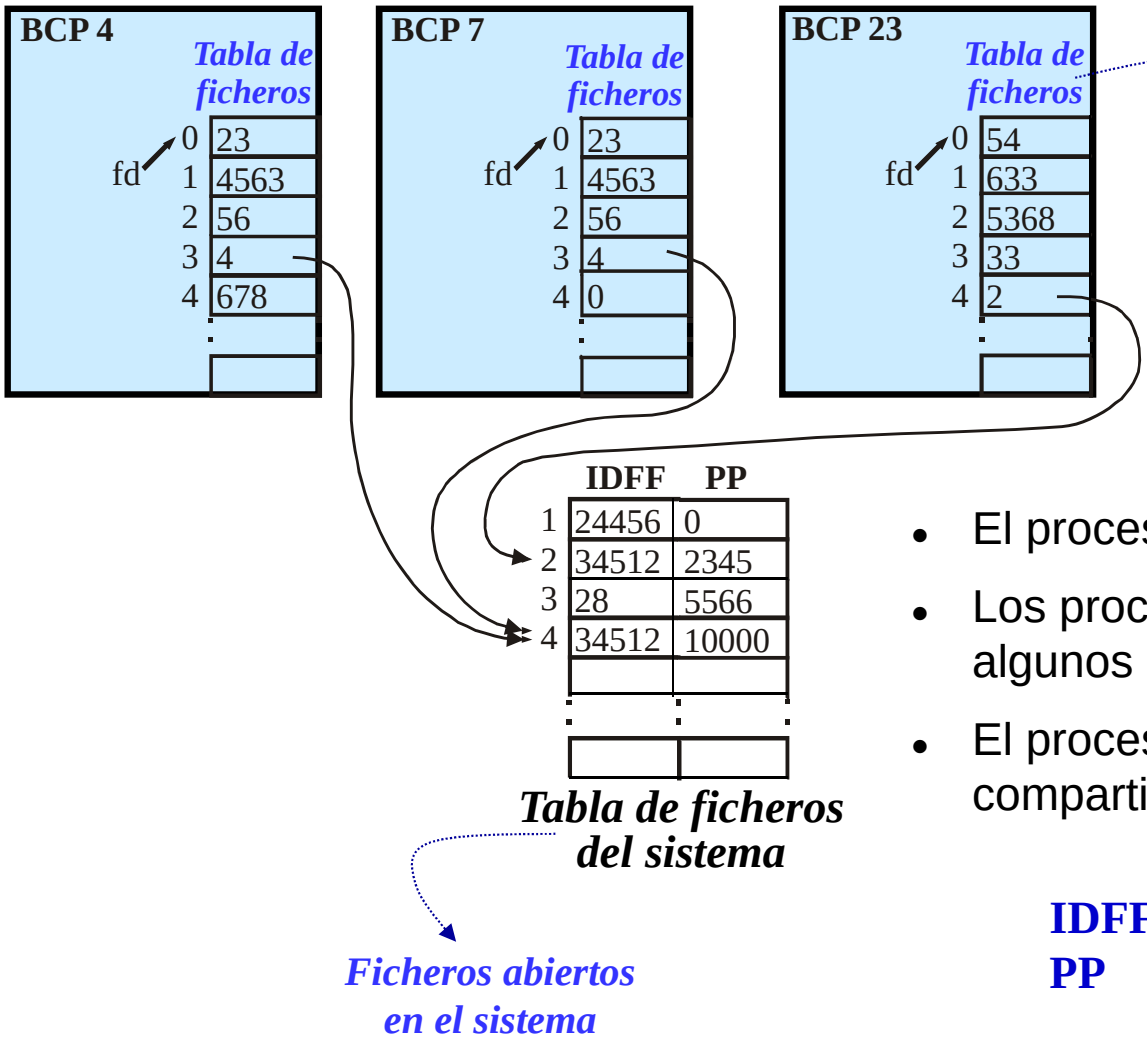
Información del BCP

- **¿Qué información del proceso se saca fuera del BCP?**
  - La que tiene tamaño variable
    - Razones de eficiencia
    - La tabla de procesos se construye como una estructura estática, formada por un número de BCP del mismo tamaño
    - Ejemplo: Tabla de páginas
  - La que se comparte:
    - El BCP es de acceso restringido al proceso que lo ocupa
    - Ejemplo: Punteros de posición de ficheros abiertos por el proceso

Información del BCP

- **Tabla de páginas:**
  - Describe la imagen de memoria del proceso
  - Tamaño variable
  - El BCP contiene el puntero a la tabla de páginas
  - La compartición de memoria requiere que sea externa al BCP
- **Punteros de posición de los ficheros:**
  - Si se asocian a la tabla de ficheros abiertos por los procesos (en el BCP) no se pueden compartir
  - Si se asocian al i-nodo se comparten siempre
  - Se ponen en una estructura común a los procesos y se asigna uno nuevo en cada servicio *open*

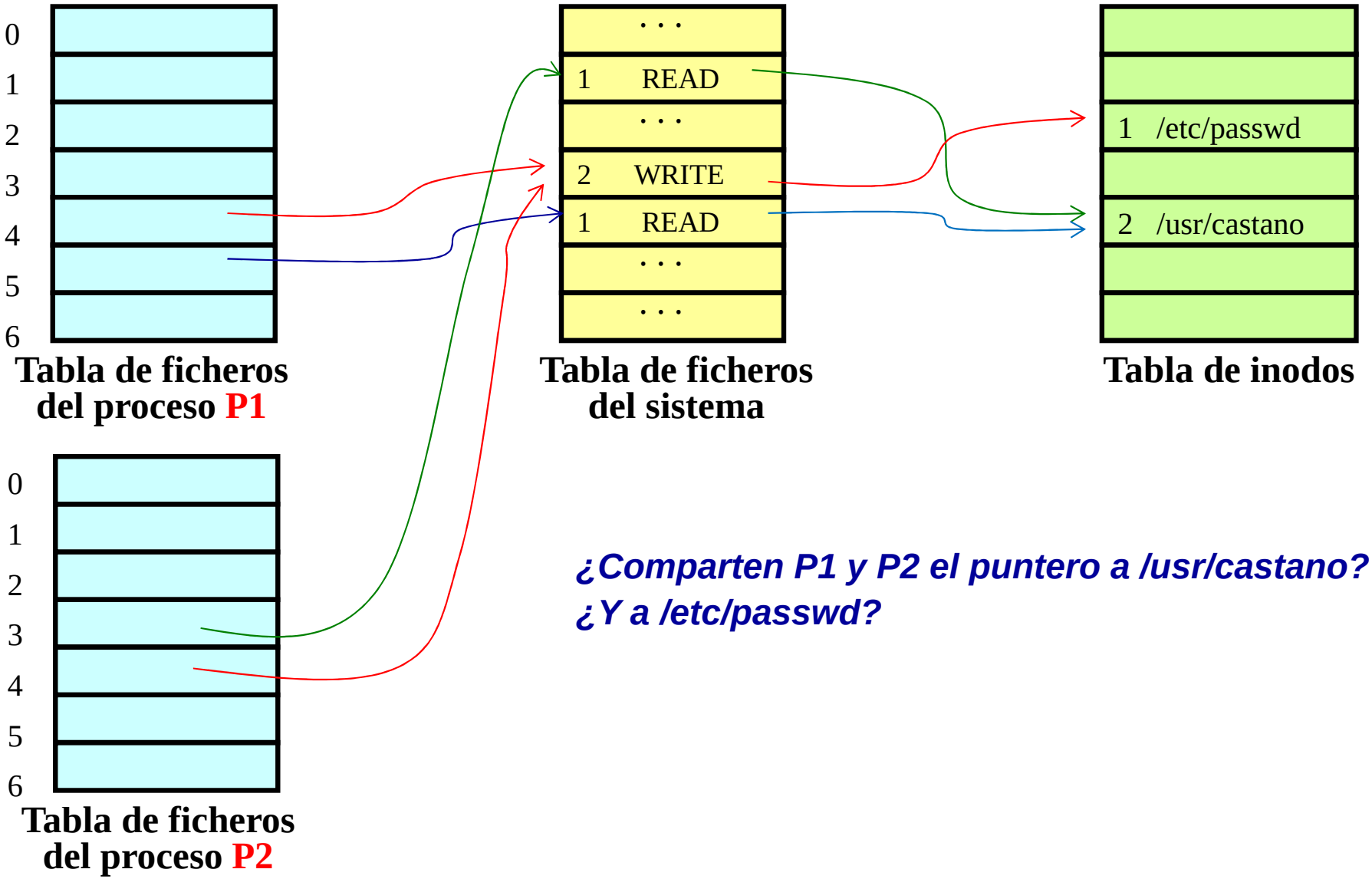
Compartir información



- El proceso con BCP7 es hijo del proceso con BCP4
- Los procesos con BCP7 y BCP4 comparten algunos punteros a los mismos ficheros
- El proceso con BCP23 abre uno de los ficheros compartido por los procesos con BCP7 y BCP4

IDFF = Identificativo de fichero
PP = Posición puntero al fichero

Compartir información



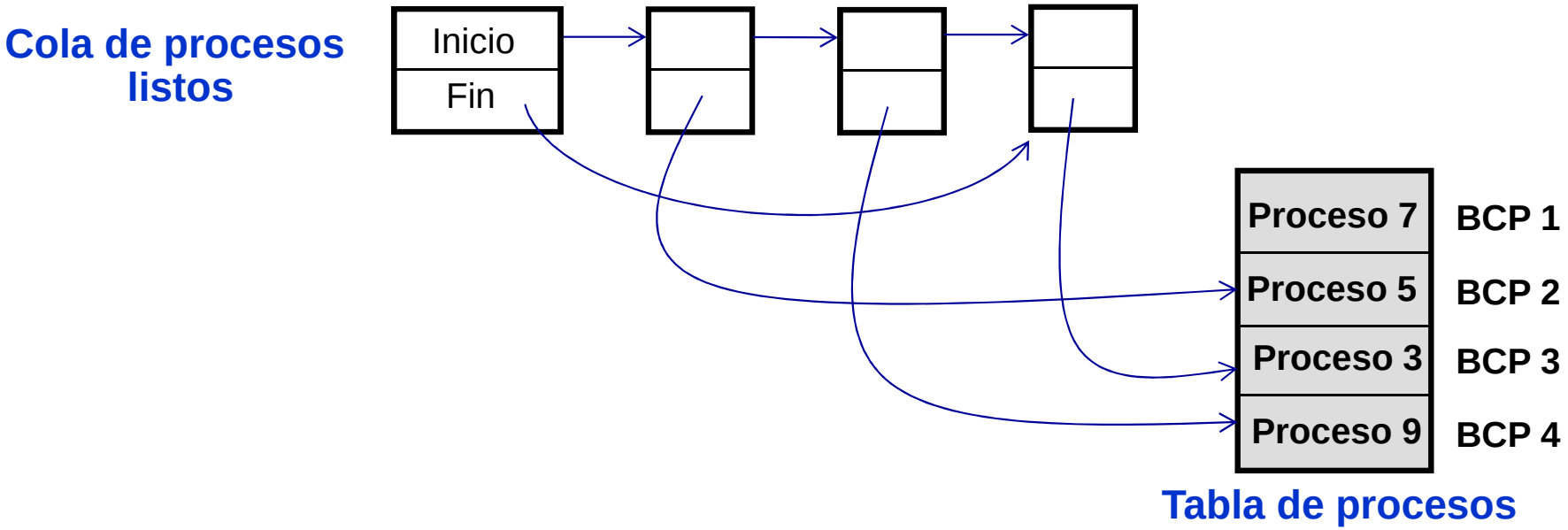
¿Comparten P1 y P2 el puntero a /usr/castano?
¿Y a /etc/passwd?

Tablas del SO

- **Tabla de procesos:** Tabla con los BCP de los procesos del sistema
- **Tabla de memoria:** Información sobre el uso de la memoria
- **Tabla de E/S:** Información asociada a los periféricos y a las operaciones de E/S
- **Tabla de ficheros:** Información sobre los ficheros abiertos en el sistema

Tablas del SO

- **Implementación de una cola de procesos listos:**



Implementación de un proceso

- **Descriptor de un proceso de Minix:**

Estado del procesador

Estado del proceso

```
struct proc{
   int p_reg[NR_REGS];          /* process' registers */
   int *p_sp;                   /* stack pointer */
   struct pc_psw p_pcpsw;       /* pc and psw as pushed by interrupt*/
   struct mem_map p_map[NR_SEGS];/* memory map */
   int *p_splimit;              /* lowest legal stack value */

   int p_pid;                   /* process id passed in from MM*/
   int p_flags;                 /* P_SLOT_FREE, SENDING, RECEIVING, */
                                /* or 0 if the process is runnable */

   real_time user_time;         /* user time in ticks */
   real_time sys_time;          /* sys time in ticks */
   real_time child_utime;       /* cumulative user time of children */
   real_time child_stime;       /* cumulative sys time of children */
   real_time p_alarm;           /* time of next alarm in ticks, or 0 */
```

Implementación de un proceso

- **Descriptor de un proceso de Minix (cont.):**

```
  struct proc *p_callerq;   /* head of list of procs wishing to send */
  struct proc *p_sendlink;  /* link to next proc wishing to send */
  message *p_messbuf;       /* pointer to message buffer */
  int p_getfrom;            /* from whom does process want to receive? */

  stuct proc *p_nextready;  /* pointer to next ready process */
  int p_pending;            /* bit map for pending signals */
} proc[NR_TASKS+NR_PROCS];
```

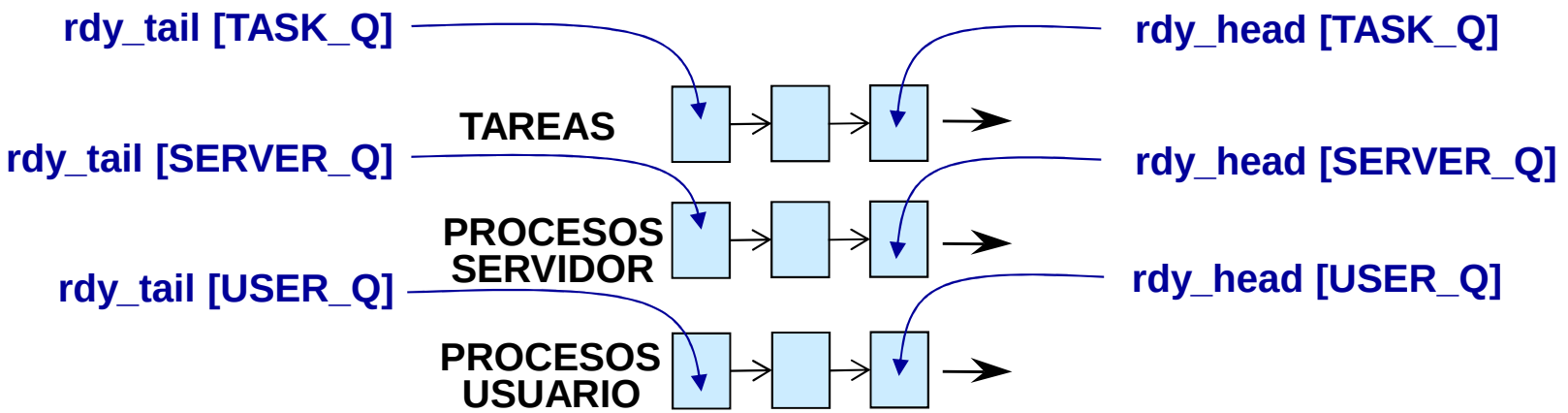
Puntero al siguiente proceso listo

Tabla de procesos del sistema

Implementación de una tabla de procesos

- **Tabla de procesos listos en Minix:**

```
/* The following items pertain to the 3 scheduling queues: */
#define NQ        3     /* # of scheduling queues */
#define TASK_Q    0     /* ready tasks   are scheduled via queue 0 */
#define SERVER_Q  1     /* ready servers are scheduled via queue 1 */
#define USER_Q    2     /* user tasks are scheduled via queue 2 */

struct proc *rdy_head[NQ];  /* pointers to ready list headers */
struct proc *rdy_tail[NQ];  /* pointers to ready list tails */
```



Tema 2. Procesos e hilos

UNIVERSITAT JAUME·I

Índice

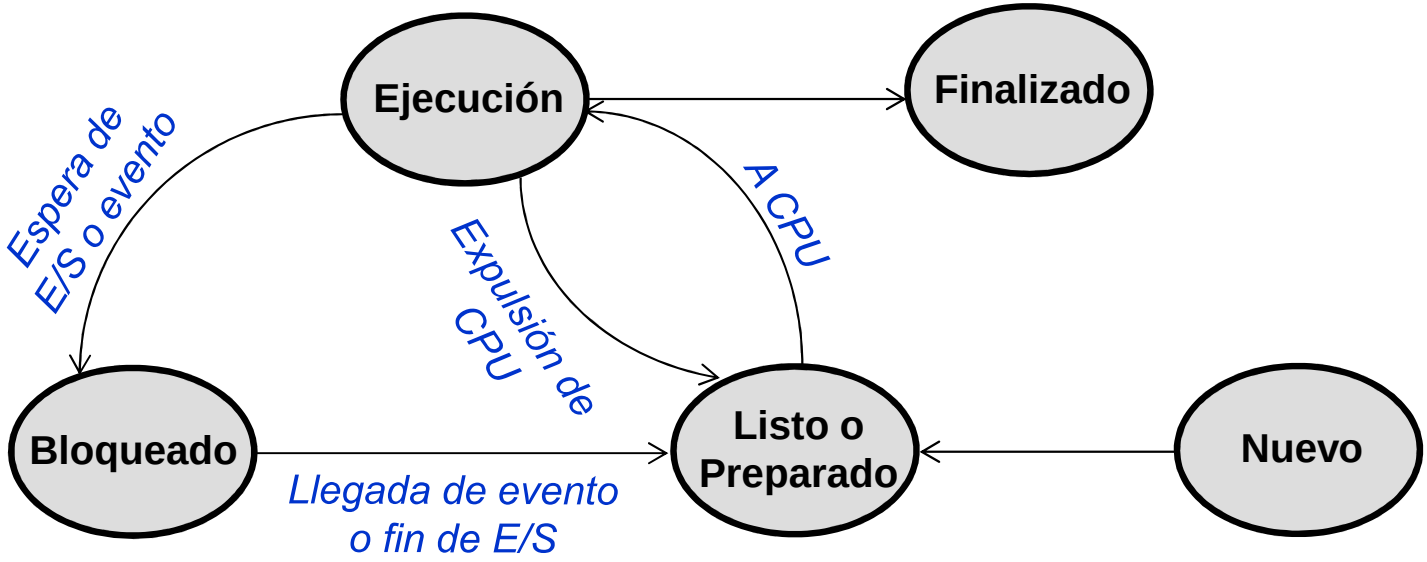
- Concepto de proceso
- Información del proceso
- Estados del proceso
- Formación de un proceso
- Cambio de contexto
- Servicios POSIX para gestión de procesos
- Concepto de hilo de ejecución
- Servicios POSIX para gestión de hilos
- Planificación de procesos e hilos

Estados de un proceso

- http://politube.upv.es/play.php?vid=1001 *Escucha solo hasta el minuto 3:43 del video*
- Cuando un proceso se ejecuta pasa por distintos *estados*
- **Posibles estados de un proceso:**
  - *Nuevo*: El proceso acaba de crearse
  - *En ejecución*: La CPU está ejecutando instrucciones del proceso
  - *Listo o preparado*: El proceso espera a ser asignado a la CPU
  - *Bloqueado*: El proceso espera a que ocurra un evento
  - *Finalizado*: El proceso ha finalizado su ejecución

Estados de un proceso

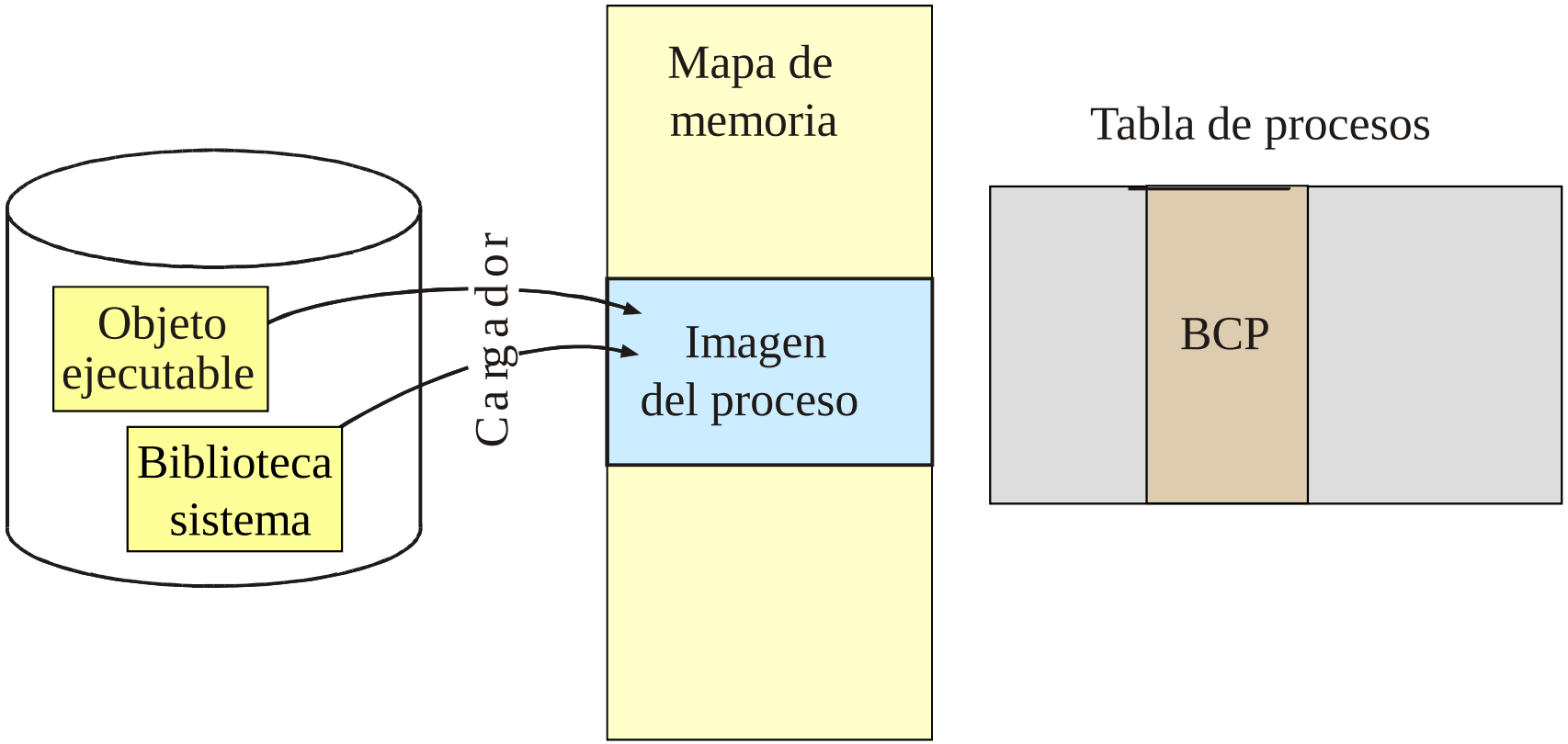
- **Diagrama de transición entre estados:**
  - Grafo dirigido:
    - Nodos: Posibles estados del proceso.
    - Arcos: Transiciones posibles entre estados.



UNIVERSITAT JAUME·I

Índice

Formación de un proceso



Formación de un proceso

- **Ciclo de vida de un proceso:**
  - Nace
  - Ejecuta algo
  - Muere o termina

Tema 2. Procesos e hilos

UNIVERSITAT JAUME·I

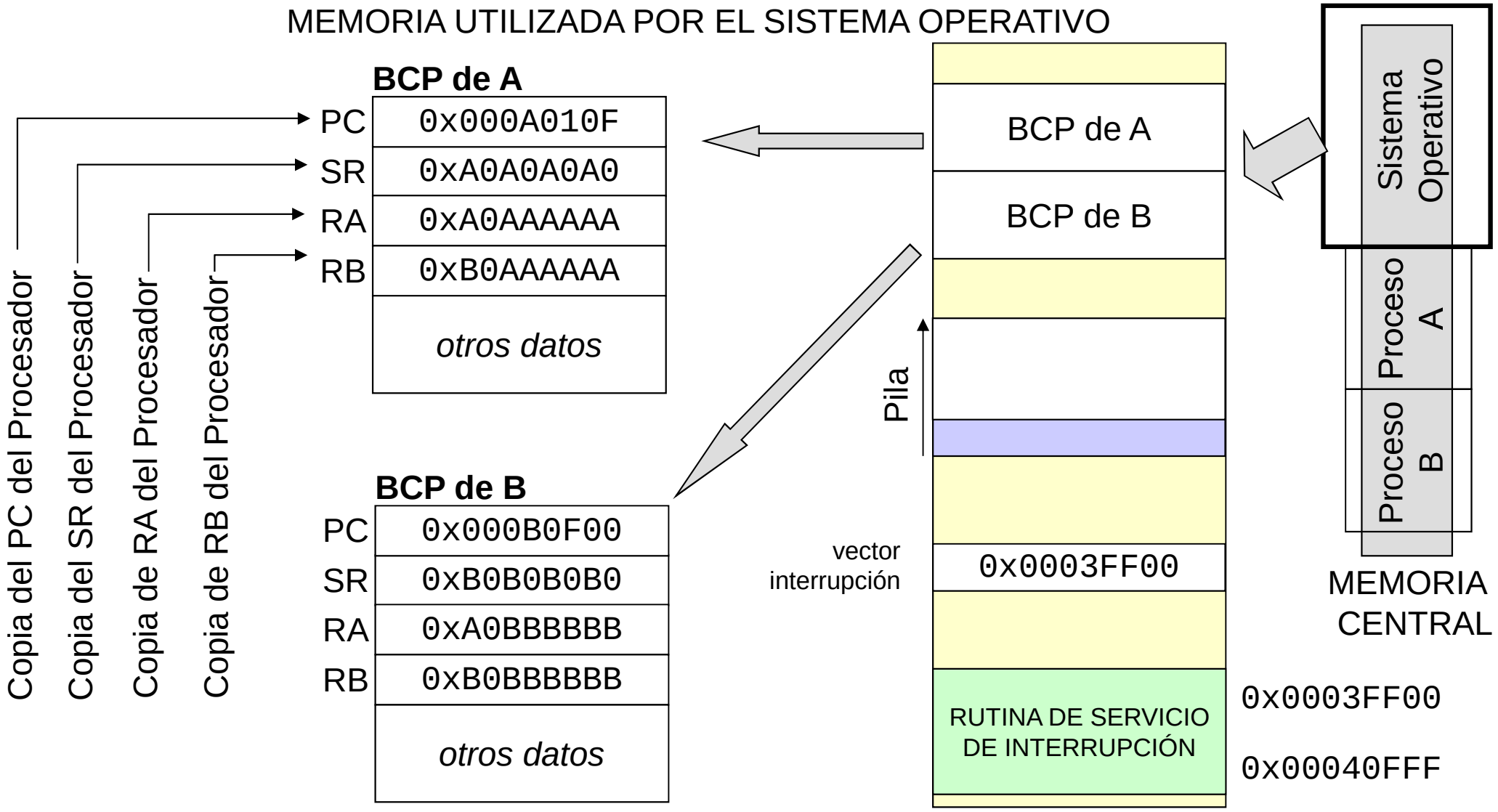
Índice

- Concepto de proceso
- Información del proceso
- Estados del proceso
- Formación de un proceso
- Cambio de contexto
- Servicios POSIX para gestión de procesos
- Concepto de hilo de ejecución
- Servicios POSIX para gestión de hilos
- Planificación de procesos e hilos

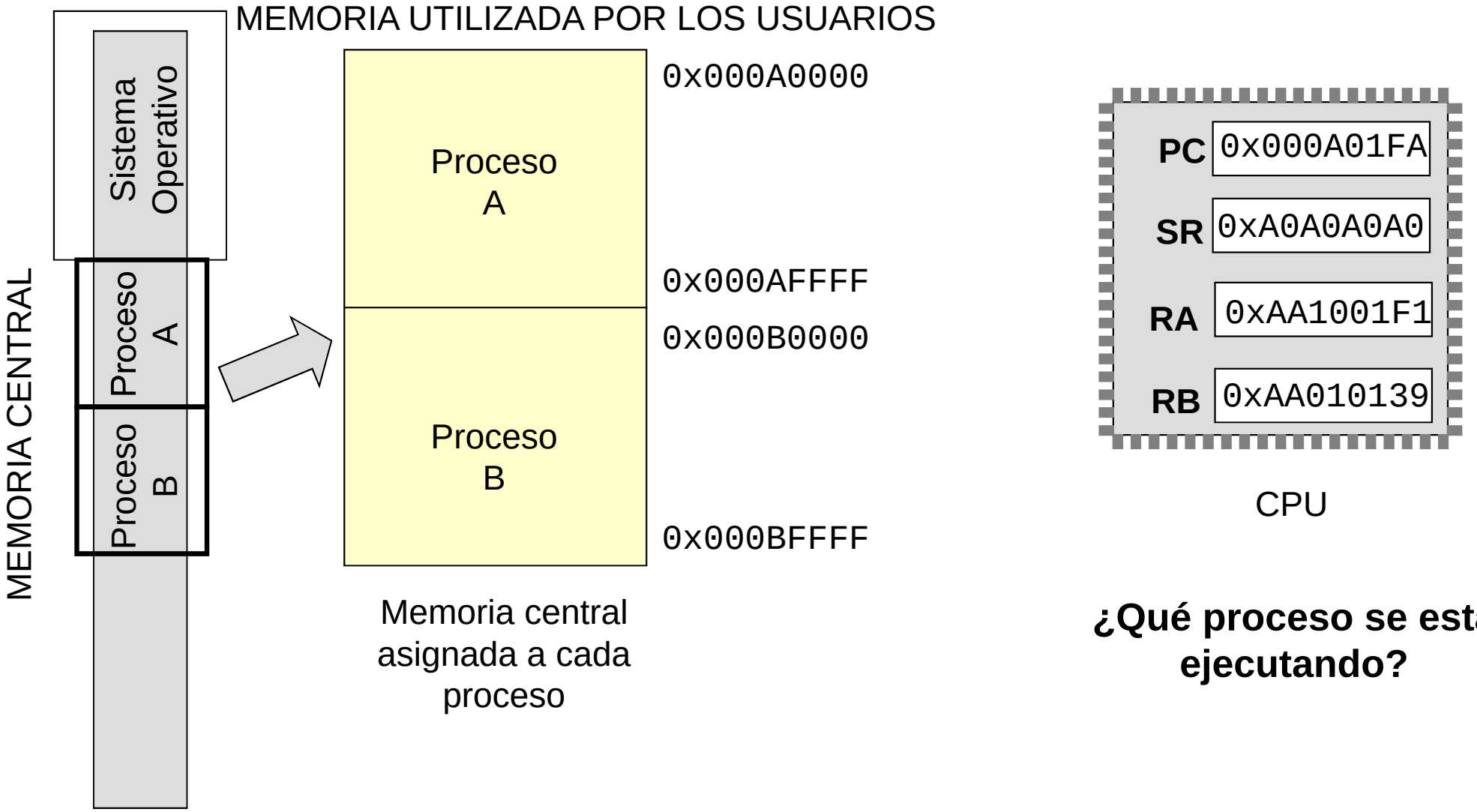
Cambio de contexto

- **¿Qué es un cambio de contexto?**
  - Mecanismo que “puede” poner en ejecución un proceso diferente al que se estaba ejecutando hasta el momento
  - Permite implementar multitarea sobre un solo procesador
  - Operación muy dependiente de la arquitectura del procesador sobre el que se esté ejecutando → Ha de realizarse a bajo nivel (en ensamblador)
  - Interesa que sea lo más rápido posible
  - Dura entre 1 y 1000 microsegundos
- **¿Cuándo se puede producir un cambio de contexto que implique cambio de proceso?**
  - En cualquier instante en el que el SO obtiene el control sobre el proceso actualmente en ejecución

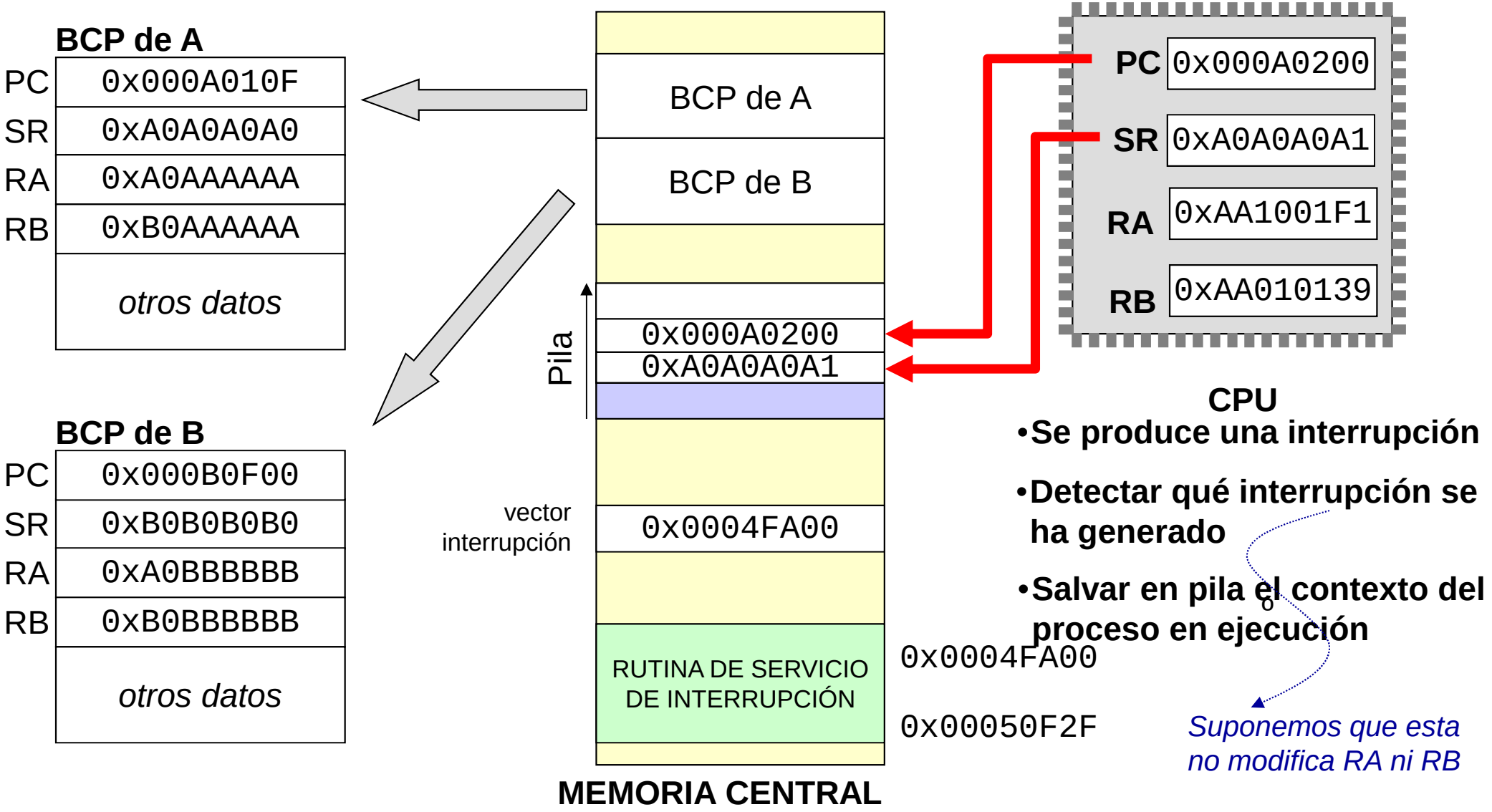
Cambio de contexto sin cambio de proceso



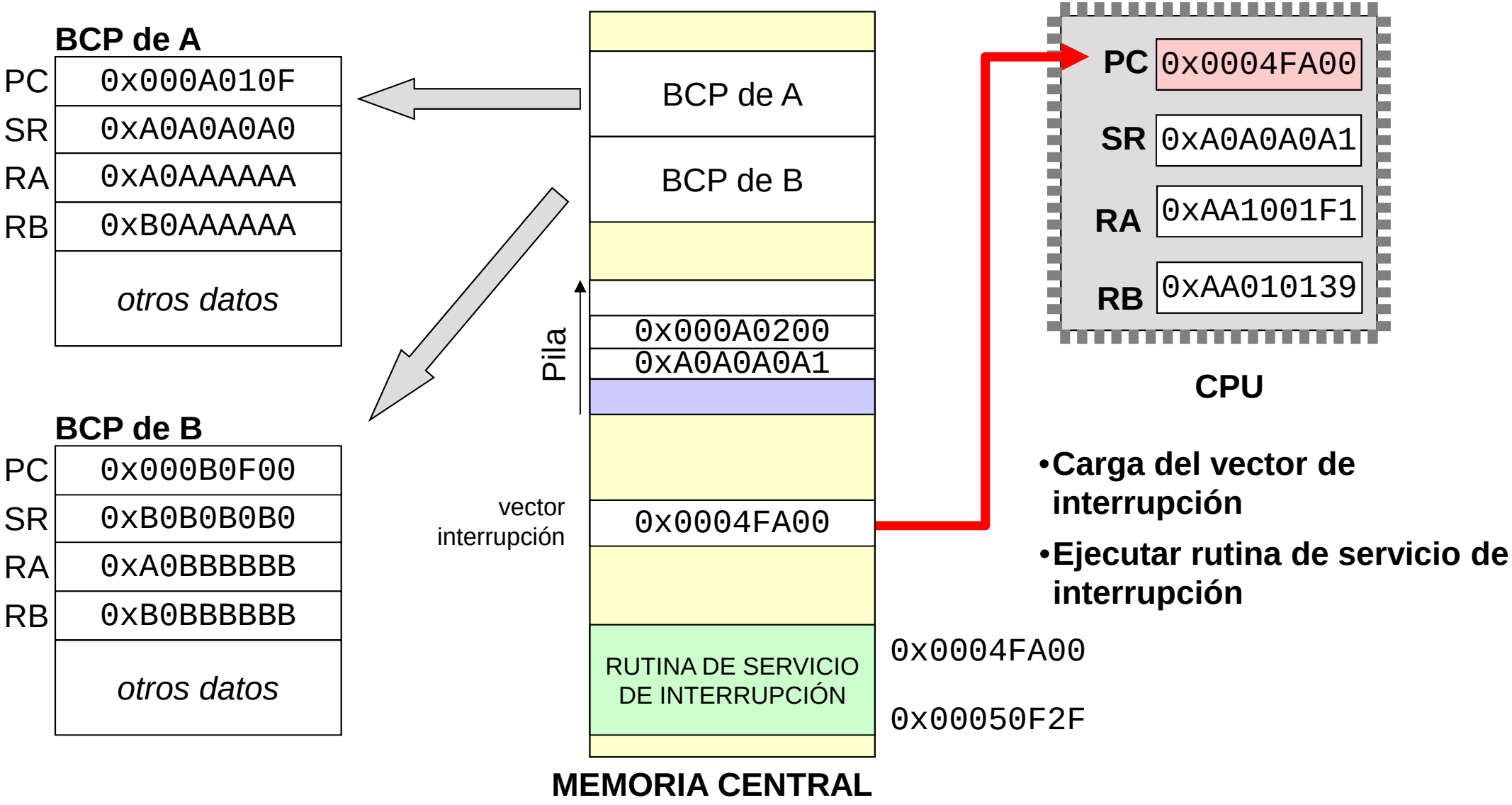
Cambio de contexto sin cambio de proceso



Cambio de contexto sin cambio de proceso

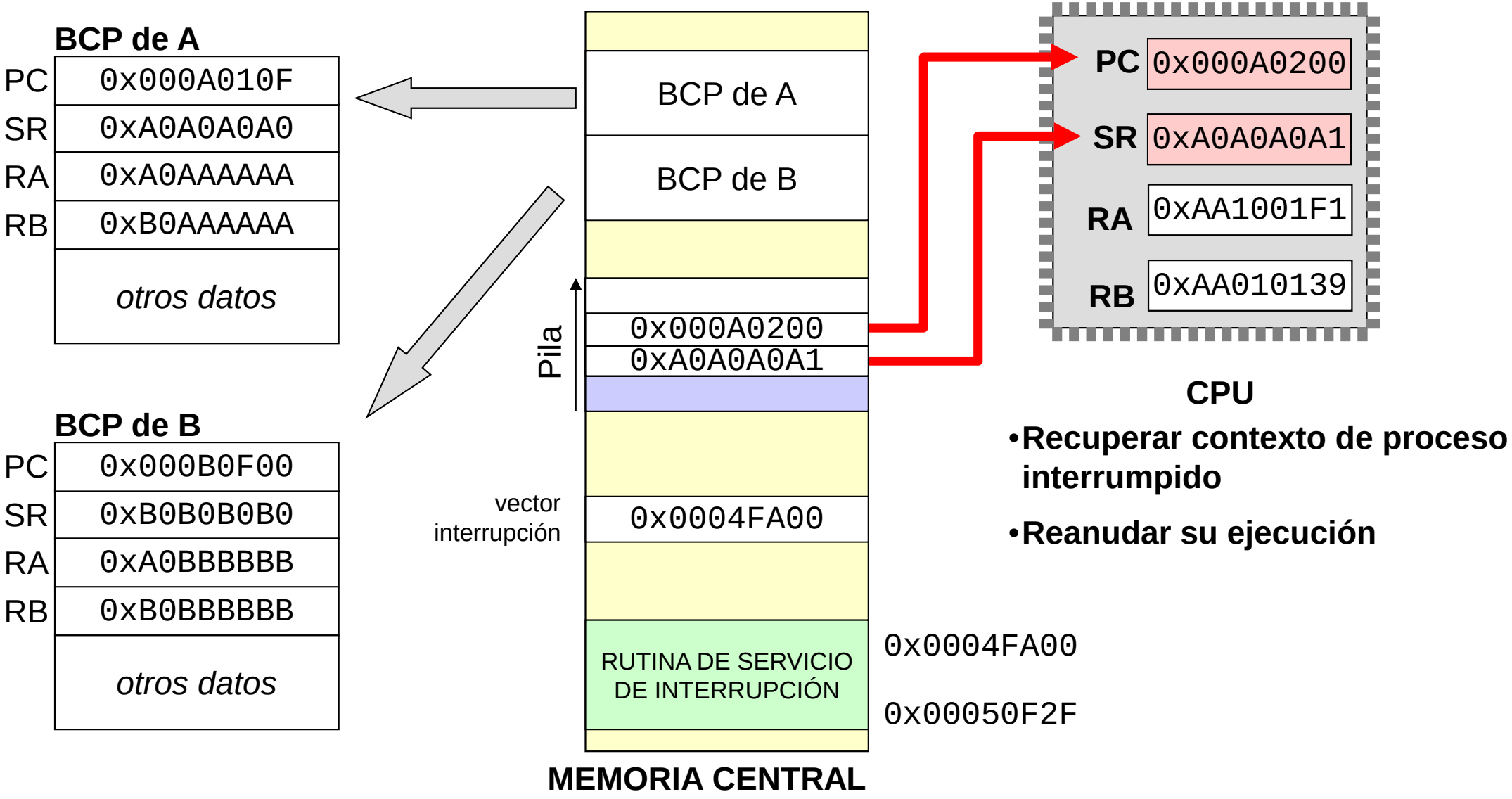


Cambio de contexto sin cambio de proceso

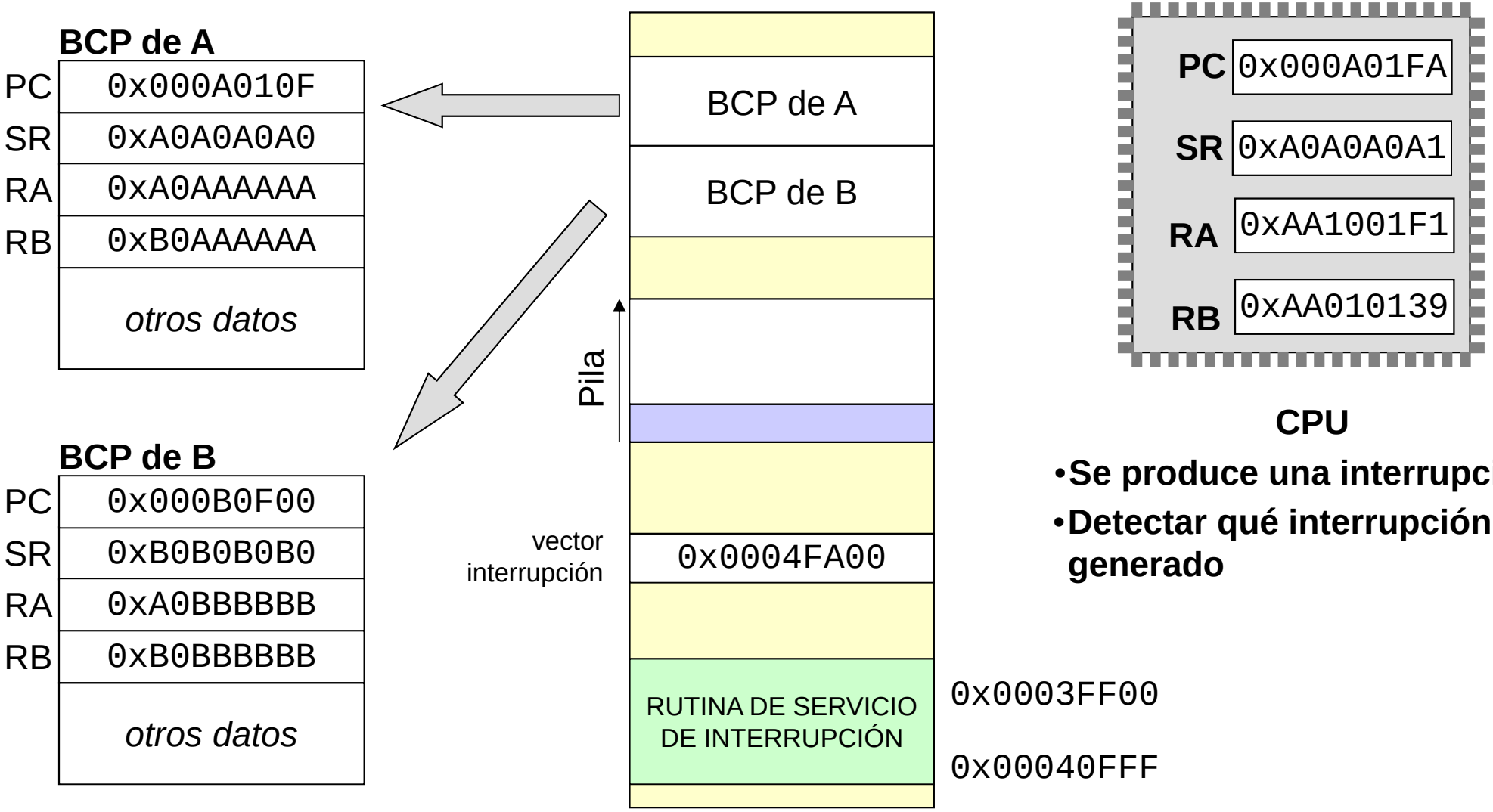


- **Carga del vector de interrupción**
- **Ejecutar rutina de servicio de interrupción**

Cambio de contexto sin cambio de proceso

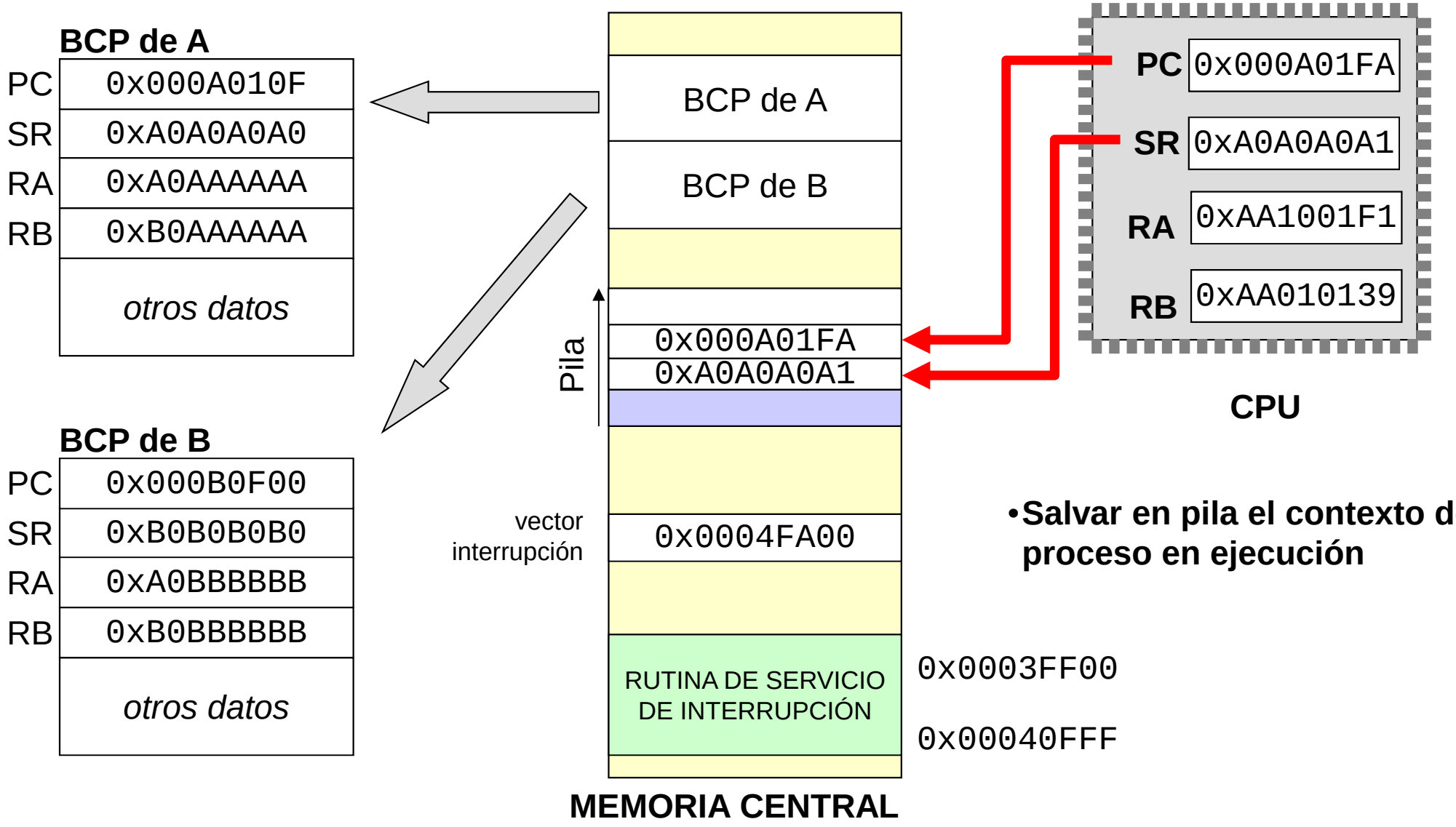


Cambio de contexto con cambio de proceso



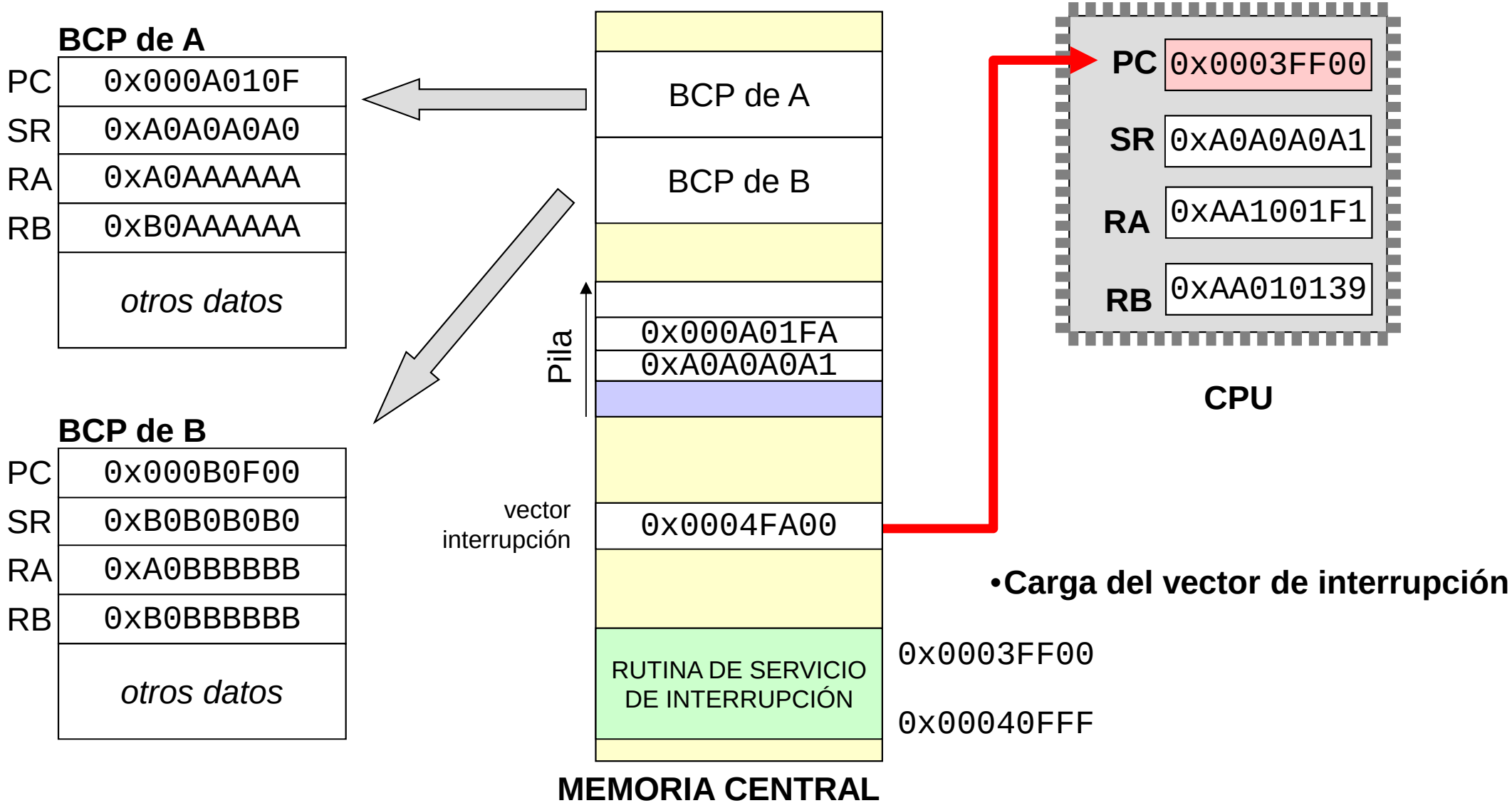
- **Se produce una interrupción**
- **Detectar qué interrupción se ha generado**

Cambio de contexto con cambio de proceso

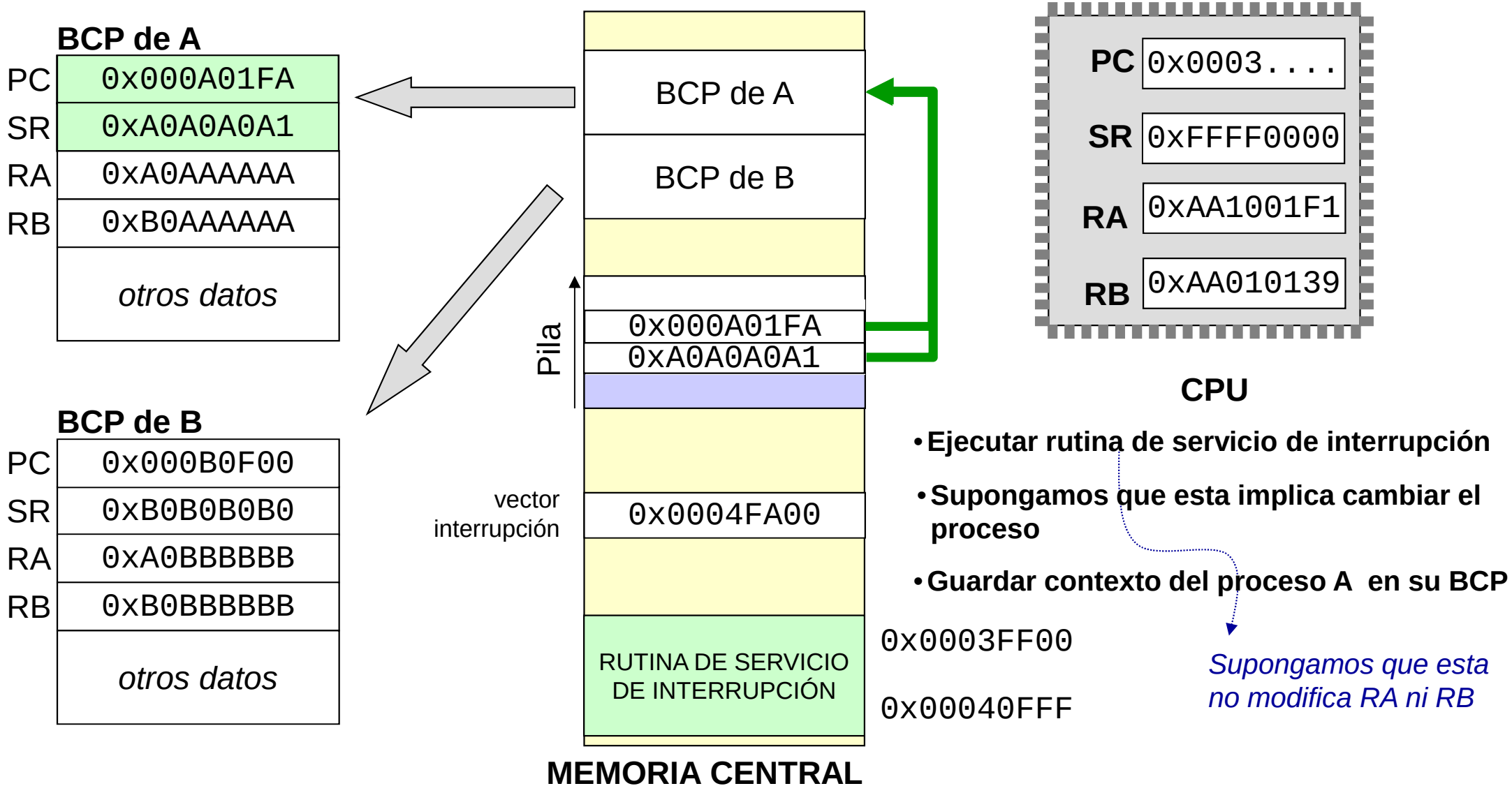


- **Salvar en pila el contexto del proceso en ejecución**

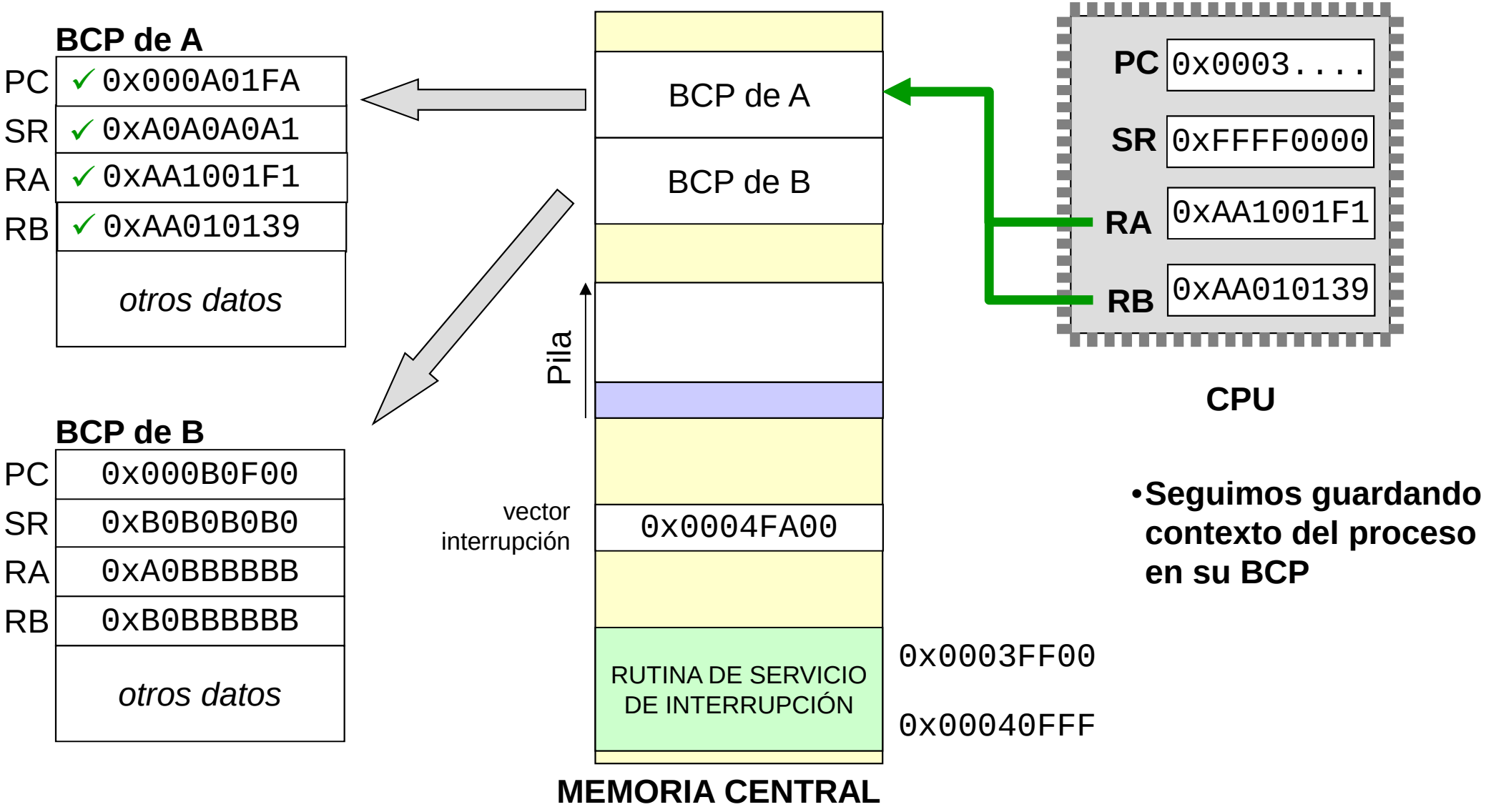
Cambio de contexto con cambio de proceso



Cambio de contexto con cambio de proceso

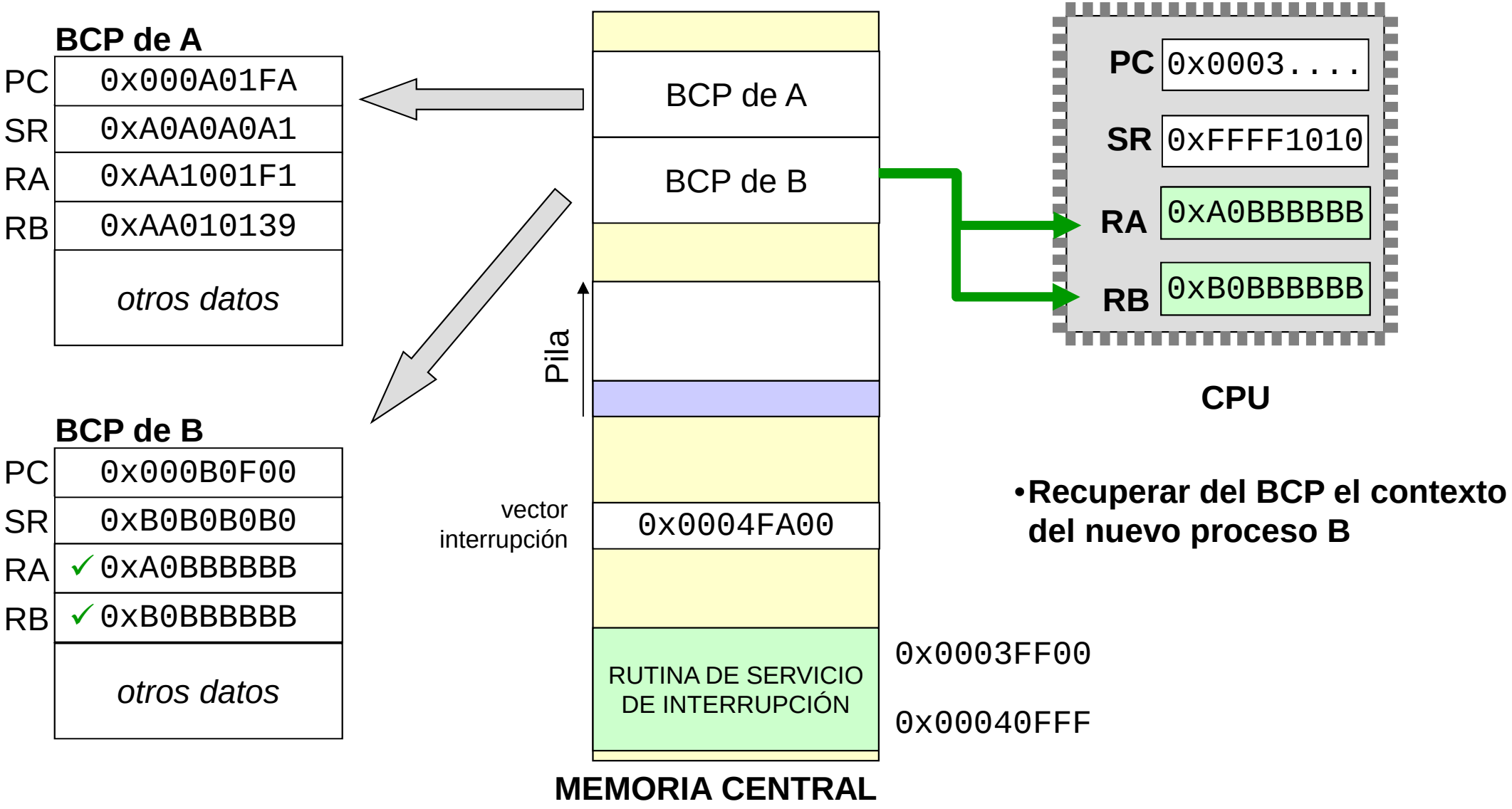


Cambio de contexto con cambio de proceso



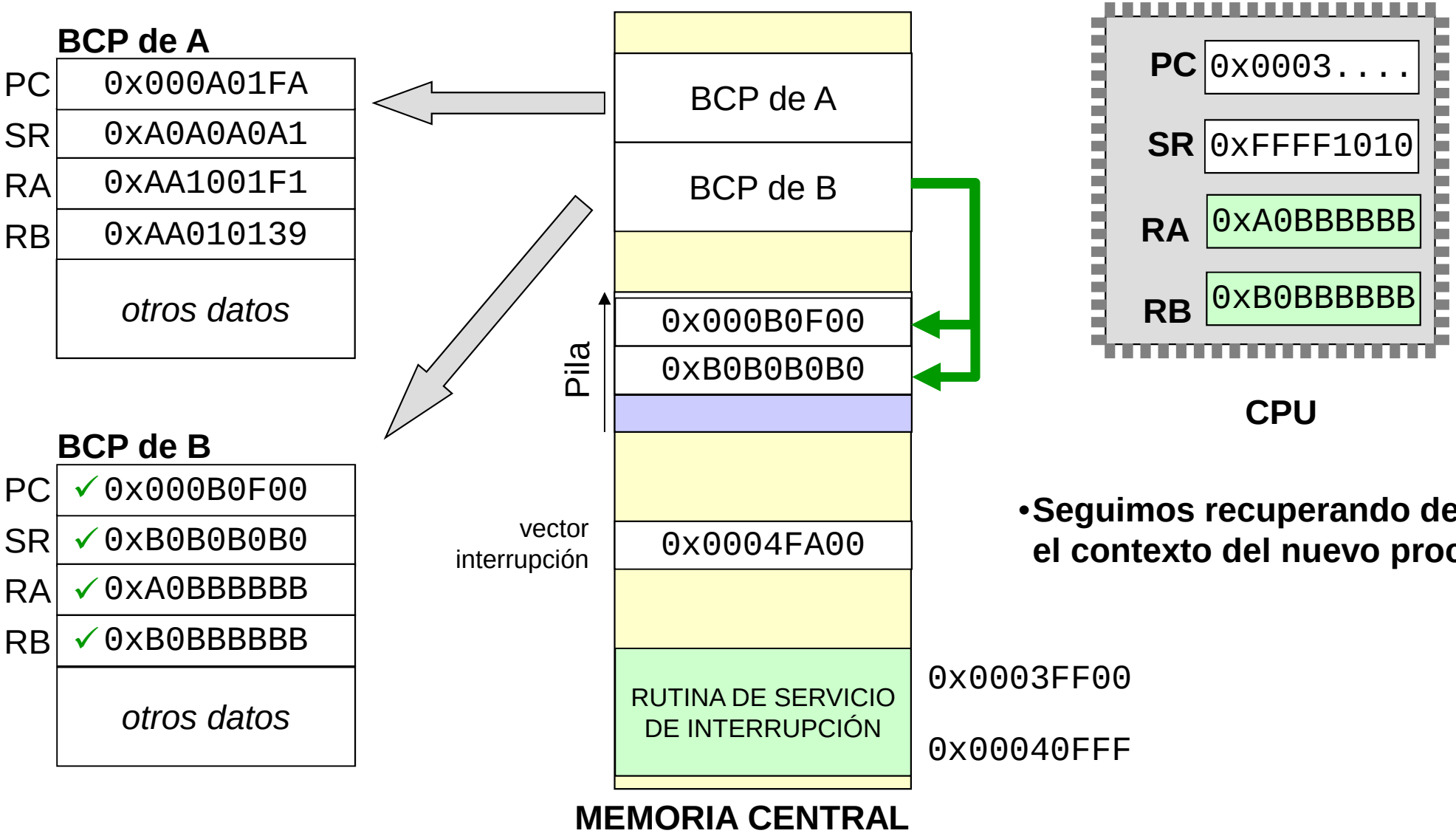
- **Seguimos guardando contexto del proceso en su BCP**

Cambio de contexto con cambio de proceso



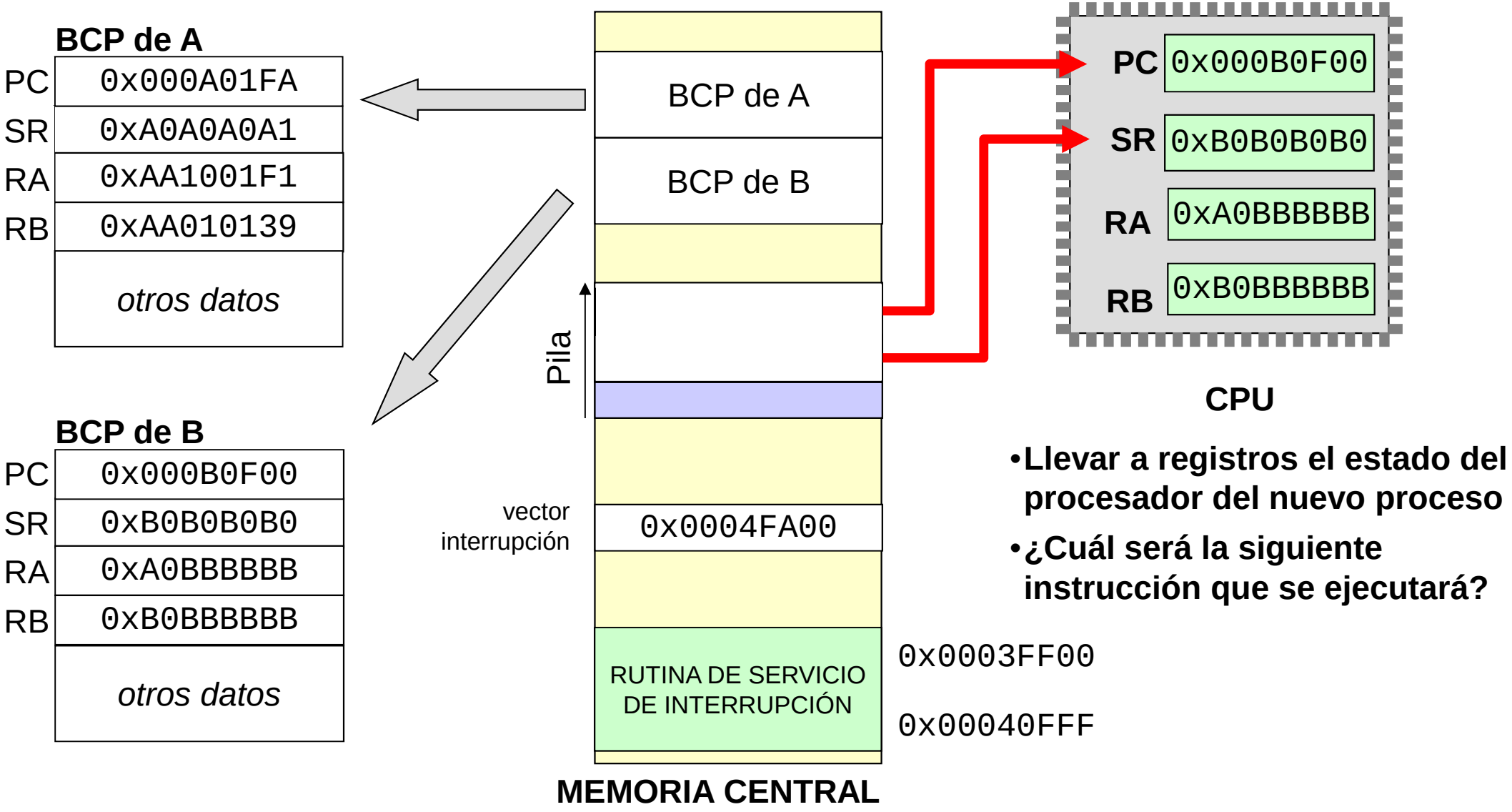
- **Recuperar del BCP el contexto del nuevo proceso B**

Cambio de contexto con cambio de proceso



- **Seguimos recuperando del BCP el contexto del nuevo proceso**

Cambio de contexto con cambio de proceso



- **Llevar a registros el estado del procesador del nuevo proceso**
- **¿Cuál será la siguiente instrucción que se ejecutará?**

Tema 2. Procesos e hilos

UNIVERSITAT JAUME·I

Índice

- Concepto de proceso
- Información del proceso
- Estados del proceso
- Formación de un proceso
- Cambio de contexto
- Servicios POSIX para gestión de procesos
- Concepto de hilo de ejecución
- Servicios POSIX para gestión de hilos
- Planificación de procesos e hilos

Servicios POSIX para gestión de procesos

- **Creación de procesos:**
  - `fork`
- **Terminación de procesos:**
  - `exit y wait`
- **Reinicialización del código de procesos:**
  - `execlp y execvp`
- **Obtención de identificadores:**
  - `getpid`, `getppid`, `getuid`, `getgrp` `y getpgrp`

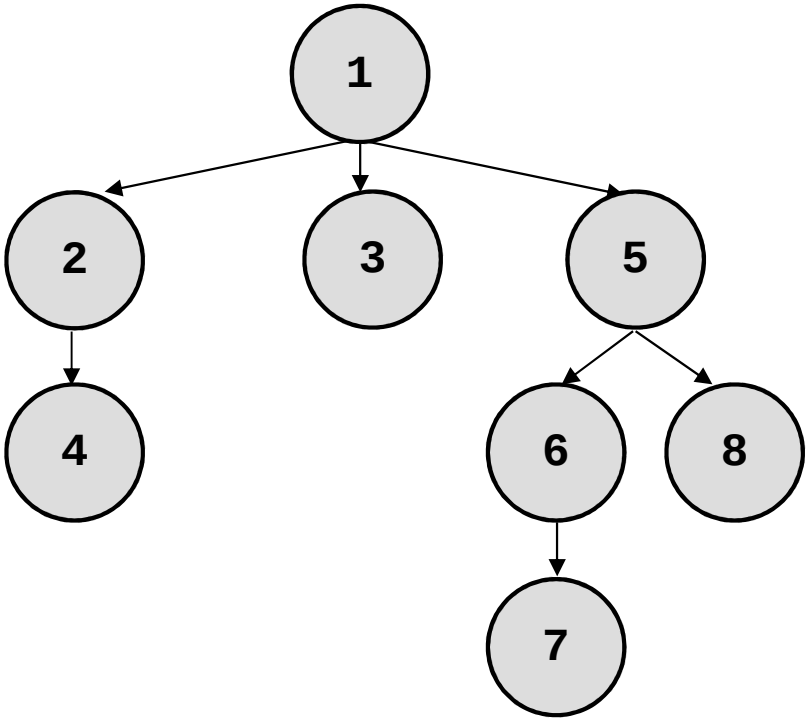
Conceptos previos

- **Identificación de procesos en POSIX:**
  - Todo proceso tiene asociados dos números desde su creación:
    - El *identificativo de proceso* (PID – “Process Identifier”):
      - ✓ Número entero positivo que lo identifica (de manera única)
      - ✓ No cambia nunca
    - El *identificativo del proceso padre* (PPID):
      - ✓ PID del proceso padre que lo creó

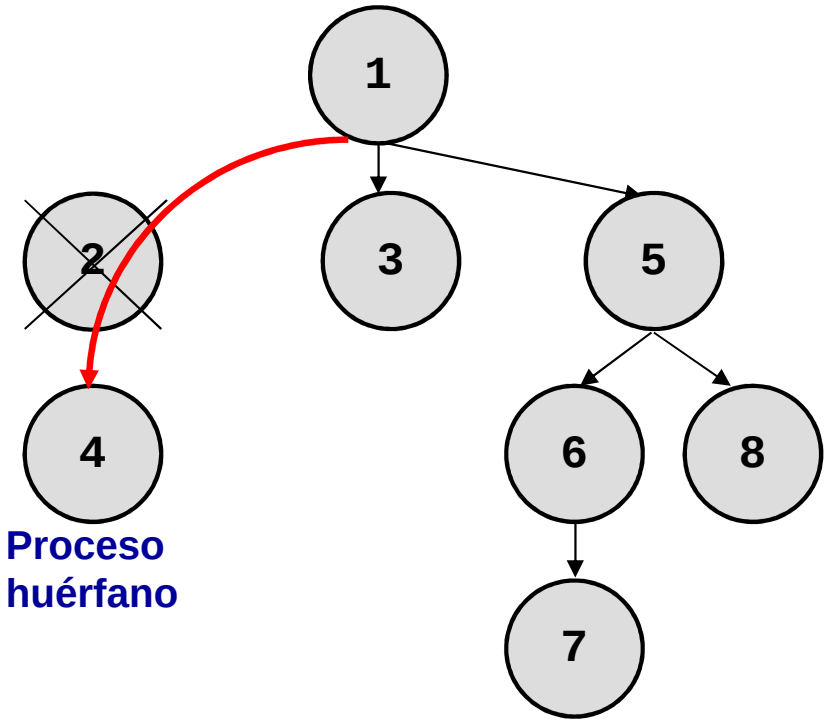
        ó
      - ✓ 1 (PID del proceso *init*) si el proceso está huérfano

Conceptos previos

- **Jerarquía (árbol) de procesos en POSIX:**
  - Proceso **init**:
    - PID = 1
    - Ancestro de todos los procesos



Finaliza el proceso con PID=2



Proceso huérfano

Conceptos previos

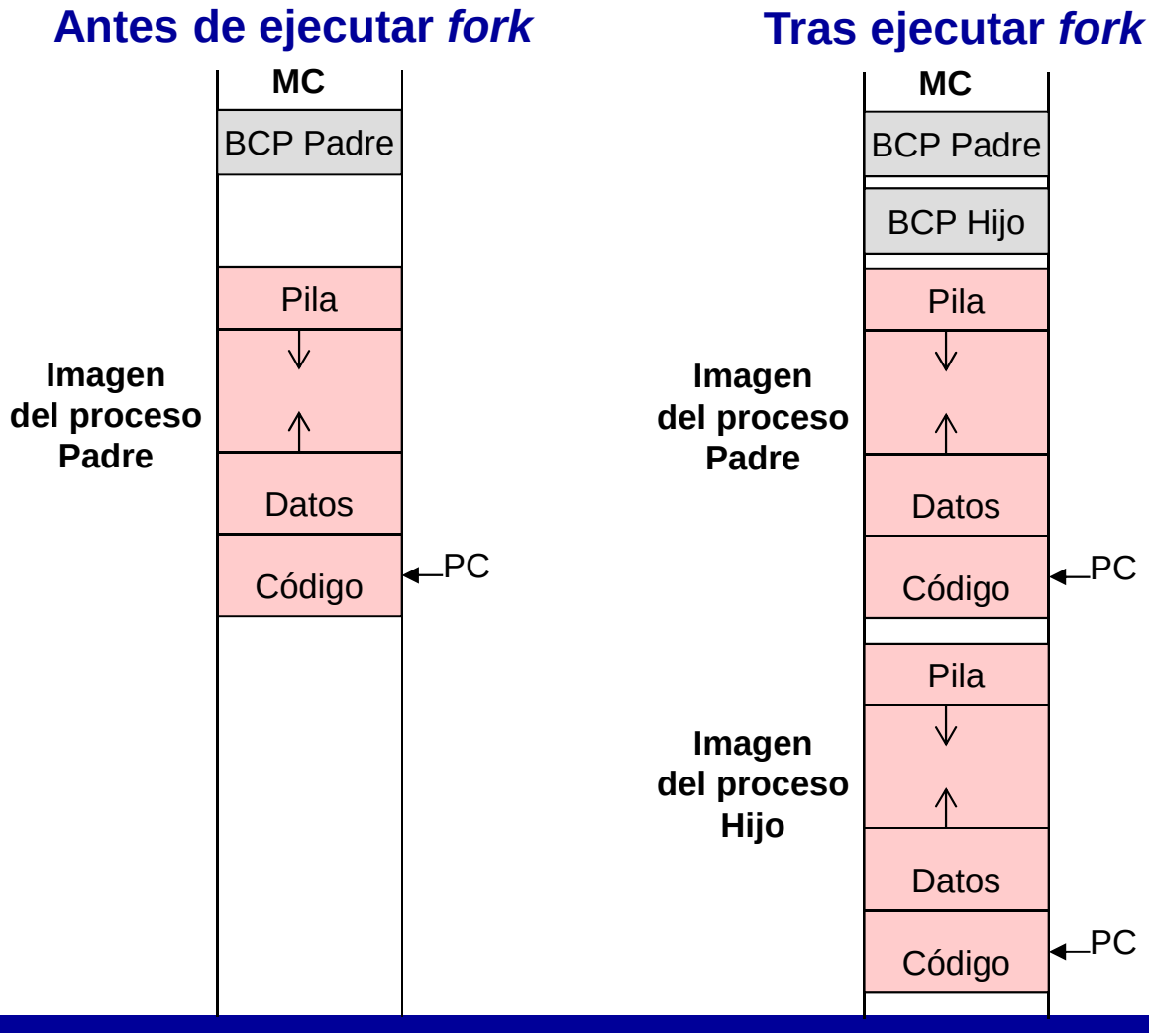
- **Identificación de usuarios en POSIX:**
  - Todo proceso tiene asociado un usuario (denominado *propietario*)
  - Cada usuario tiene un identificativo único (denominado *identificador de usuario*)

Función *fork*

- **Descripción:**
  - Creación de un proceso nuevo
    - Proceso padre: Proceso que invoca la función `fork`
    - Proceso hijo: Proceso nuevo creado con la función `fork`
  - Réplica exacta (**clonación**) del proceso que la invoca
  - *El nuevo proceso ocupa otra zona de memoria con el mismo contenido*
- **Tras la clonación:**
  - El PC de los dos procesos es idéntico
  - Ambos procesos comparten el puntero de posición de los ficheros abiertos en ese momento
  - Los cambios que se realicen posteriormente en uno de ellos no afectan al otro

Función *fork*

- **Creación de un proceso mediante *fork*:**



Función *fork*

- **Sintaxis:**

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

devuelve**:**

- Al proceso padre: El PID del proceso hijo
- Al hijo: Un cero
- Si error: -1

Función *fork*

http://politube.upv.es/play.php?vid=999

Función *fork*

- **Ejemplo 1:**

```
#include <unistd.h>
#include <stdio.h>
int main()
{
  if (fork()!=0) printf("Soy el padre\n");
    else printf("Soy el hijo\n");
}
```

Función *fork*

- **Ejemplo 2:**

```
#include <stdio.h>
int main()
{ int i,j;
  if (fork()!=0)      /* Proceso Padre */
  {for (i=0;i<100;i++)
    {
      for (j=0;j<100000;j++);
      printf("Proceso padre. Indice i=%d\n",i);
    }
  }
  else                /* Proceso Hijo */
  {for (i=0;i<100;i++)
    {
      for (j=0;j<100000;j++);
      printf("Proceso hijo. Indice i=%d\n",i);
    }
  }
}
```

Funciones *getpid* y *getppid*

- **Sintaxis:**

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

devuelven:

- `getpid`: El PID del proceso que realiza la llamada
- `getppid`: El PPID del proceso que realiza la llamada

Funciones *getpid* y *getppid*

- **Ejemplo:**

```
$cat MostrarIdsProceso.c
#include <unistd.h>
#include<stdio.h>
main()
{ printf("Inicio del proceso padre: PID =%d\n", getpid());
  if (fork()!=0)
     printf("Continuación del padre:   PID=%d\n", getpid());
  else
     printf("Inicio del proceso hijo:  PID=%d PPID=%d \n",getpid(),getppid());
  printf("Fin del proceso:          PID=%d\n", getpid());
}
```

Funciones *getpid* y *getppid*

- **Ejemplo (cont.):**

```
$make MostrarIdsProceso
$MostrarIdsProceso
Inicio del proceso padre: PID=587
Inicio del proceso hijo:  PID=588 PPID=587
Fin del proceso:          PID=588
Continuación del padre:   PID=587
Fin del proceso:          PID=587

$MostrarIdsProceso
Inicio del proceso padre: PID=587
Continuación del padre:   PID=587
Fin del proceso:          PID=587
Inicio del proceso hijo:  PID=588 PPID=587
Fin del proceso:          PID=588
```

Tras el *fork* continúa ejecutándose el hijo

Tras el *fork* continúa ejecutándose el padre

Función *getuid*

- **Sintaxis:**

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

devuelve:

  - El número de identificativo del usuario propietario del proceso que la invoca

Función *getpgrp*

- **Sintaxis:**

```
pid_t getpgrp(void);
```

devuelve:

  - El número de identificativo del grupo al que pertenece el proceso que la invoca

Función *setpgrp*

- **Sintaxis:**

```
pid_t setpgrp(void);
```

- **Descripción:**
  - El proceso que la invoca:
    - Crea un nuevo grupo de procesos
    - Se convierte en el líder del nuevo grupo
  - Devuelve el identificativo del nuevo grupo de procesos (el PID del proceso que la invoca)

Funciones *getpgrp* y *setpgrp*

- **Ejemplo:**

```
$cat MostrarGrupoProceso.c
#include<stdio.h>
main()
{ if (fork()!=0)
     printf("Padre:   PID=%d  PPID=%d  ID GRUPO PROC=%d\n",
             getpid(), getppid(), getpgrp());
  else
  {  printf("Hijo:    PID=%d  PPID=%d  ID GRUPO PROC=%d\n",
             getpid(), getppid(); getpgrp());
     setpgrp();
     printf("Hijo:    PID=%d  PPID=%d  ID GRUPO PROC=%d\n",
             getpid(), getppid(), getpgrp());
  }
}
```

Funciones *getpgrp* y *setpgrp*

- **Ejemplo (cont.):**

```
$make MostrarGrupoProceso
$MostrarGrupoProceso
Hijo:    PID=134  PPID=133  ID GRUPO PROC=133
Hijo:    PID=134  PPID=133  ID GRUPO PROC=134
Padre:   PID=133  PPID=130  ID GRUPO PROC=133

$MostrarGrupoProceso
Padre:   PID=133  PPID=130  ID GRUPO PROC=133
$Hijo:    PID=134  PPID=133  ID GRUPO PROC=133
Hijo:    PID=134  PPID=133  ID GRUPO PROC=134
```

Tras el *fork* continúa ejecutándose el hijo

Tras el *fork* continúa ejecutándose el padre

Funciones *getpgrp* y *setpgrp*

- **Ejemplo (cont.):**

```
$make MostrarGrupoProceso
$MostrarGrupoProceso
Hijo:    PID=134  PPID=133  ID GRUPO PROC=133
Hijo:    PID=134  PPID=133  ID GRUPO PROC=134
Padre:   PID=133  PPID=130  ID GRUPO PROC=133
$
$
$MostrarGrupoProceso
Padre:   PID=133  PPID=130  ID GRUPO PROC=133
$Hijo:    PID=134  PPID=133  ID GRUPO PROC=133
Hijo:    PID=134  PPID=133  ID GRUPO PROC=134
```

Al finalizar el padre se visualiza el prompt

¿Cómo conseguir una salida única?

Sincronización de procesos con *wait* y *exit*

Función *exit*

- **Descripción:**
  - Finaliza la ejecución del proceso que la invoca
  - No es imprescindible su uso pero sí recomendable
- **Sintaxis:**

```
void exit (int estado);
```

  - Deja un código de terminación en el parámetro `estado` que puede recoger el padre
  - No devuelve ningún valor

Función *exit*

- **Acciones que implica:**
  - Liberar la memoria que ocupa el proceso que la invoca y cerrar sus ficheros asociados (si no hay más procesos que los tengan abiertos)
  - Si el proceso padre está en espera (debido a un `wait`), desbloquearlo
  - Si el proceso padre no está ejecutando una función `wait`, el proceso que invoca la función `exit` queda en estado *zoombie* hasta que el padre ejecute una función `wait`.
    - Libera el espacio que ocupaba en memoria excepto su BCP
  - La terminación del proceso no finaliza de forma directa la ejecución de sus procesos hijos
    - Si tiene hijos éstos quedan *huérfanos* y son “adoptados” por el proceso *init* (sus PPID’s serán 1)

Función *wait*

- **Sintaxis:**

```
int wait(int *estado);
```

devuelve**:**

  - El PID del hijo que ha finalizado
  - SI error: -1
  - Parámetro `estado`:
    - Byte bajo: Número de señal que destruyó al hijo
    - Byte siguiente: Estado de terminación del hijo

- **Descripción:**
  - Suspende (bloquea) la ejecución del proceso que la invoca hasta que uno de sus procesos hijos (cualquiera) termina
  - Si no tiene hijos o los hijos han terminado, no tiene efecto (el proceso no se bloquea)

Función *wait*

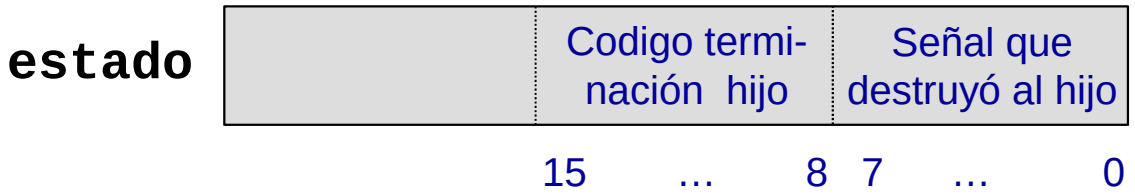
Hijo

```
exit(status);
```

Padre

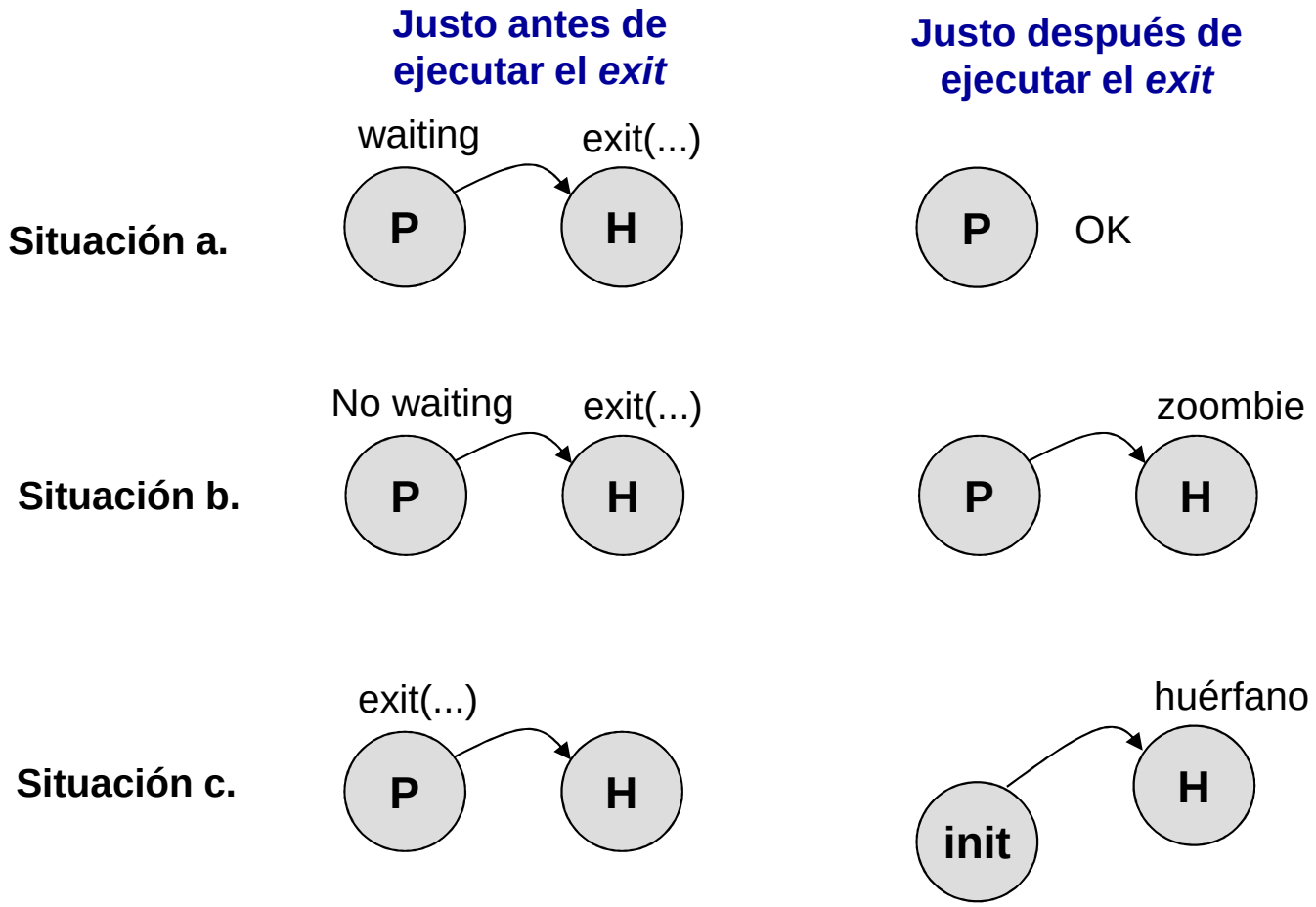
```
wait(&estado);
```

SO



Función *wait*

- **Sincronización de procesos mediante *wait* y *exit*:**



Función *wait*

- **Ejemplo 1:**

```
$cat espera.c
#include<stdio.h>
main()
{ int estado;
  if (fork()!=0)
  {   wait (&estado);
      printf("Padre\n");
  } else printf("Hijo\n");
  exit(0);
}
$make espera
$espera
Hijo
Padre
```

```
$cat no_espera.c
#include<stdio.h>
main()
{ if (fork()!=0) printf("Padre\n");
  else printf("Hijo\n");
  exit(0);
}
$make no_espera
$no_espera
Hijo
Padre
$no_espera
Padre
$Hijo
```

Orden de impresión diferente

Al finalizar el padre se visualiza el prompt

Función *wait*

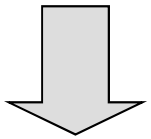
- **Ejemplo 2:**

```
$cat MostrarGrupoProceso.c
#include<stdio.h>
main()
{ int estado;

  if (fork()!=0)
  {  wait (&estado);
     printf("Padre:   PID=%d  PPID=%D  ID GRUPO PROC=%D\n",
             getpid(), getppid(), getpgrp());
  } else {
     printf("Hijo:    PID=%d  PPID=%D  ID GRUPO PROC=%D\n",
             getpid(), getppid(); getpgrp());
     setpgrp();
     printf("Hijo:    PID=%d  PPID=%D  ID GRUPO PROC=%D\n",
             getpid(), getppid(), getpgrp());
  }
  exit (0);
}
```

Funciones *getpgrp* y *setpgrp*

- **Ejemplo 2 (cont.):**

```
$make MostrarGrupoProceso
$MostrarGrupoProceso
Hijo:    PID=134  PPID=133  ID GRUPO PROC=133
Hijo:    PID=134  PPID=133  ID GRUPO PROC=134
Padre:   PID=133  PPID=130  ID GRUPO PROC=133
$
```



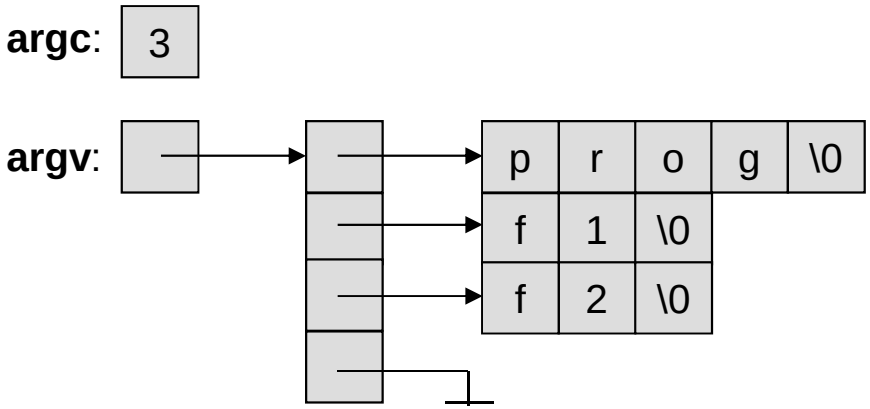
Orden de impresión único

Funciones *exec*

- **Paso de parámetros en la ejecución de un programa en C:**
  - Los argumentos pasados se almacenan en una estructura `argv`

```
int argc;
char *argv[];
```

  - Ejemplo: `$prog f1 f2`



Funciones *exec*

- **Descripción:**
  - Reinicialización (sobreescritura) del código de un proceso
- ***exec* vs. *fork*:**
  - `fork` crea un proceso que ejecuta el mismo programa que el padre

    2 procesos – 1 código
  - `exec` permite que un proceso ejecute un programa distinto

    1 proceso – 2 códigos

Funciones *exec*

- **Sintaxis:**

```
int execlp (fichero,arg0,arg1,...,argN,(char *)0);
const char *fichero;
const char *arg0;
const char *arg1;
...
const char *argN;

int execvp (fichero,argv);
const char *fichero;
char *argv[];
```

No devuelve el control al programa llamante excepto en caso de error

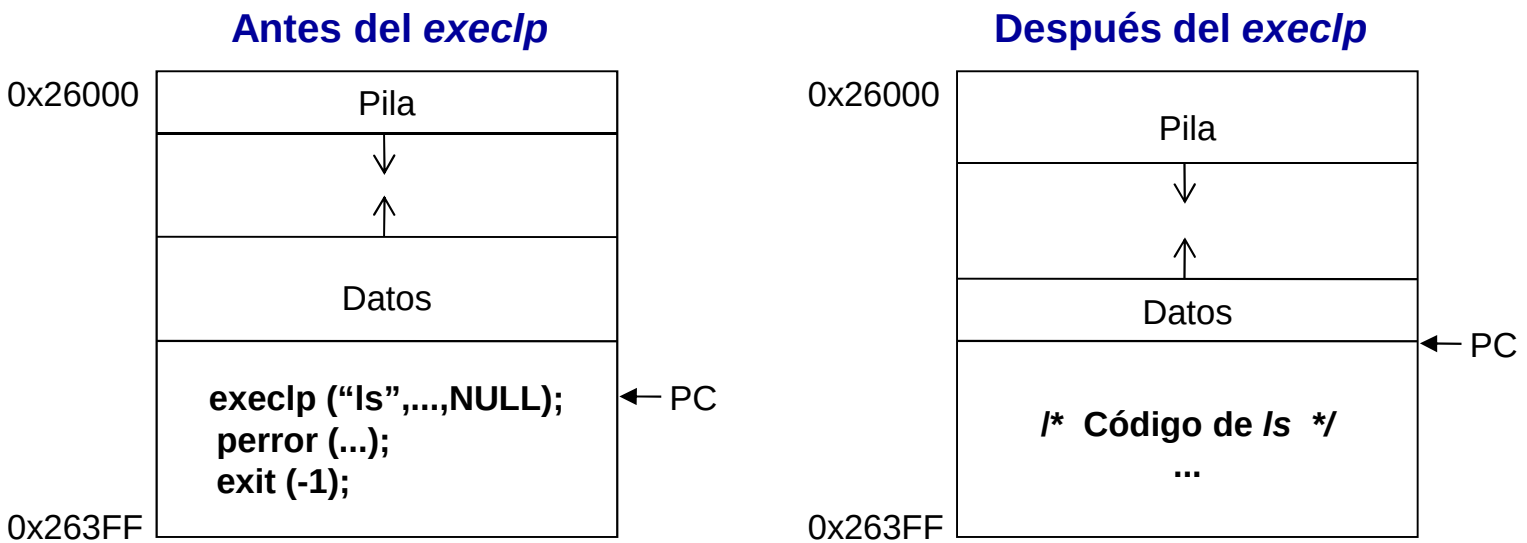
Función *execlp*

- **Ejemplo 1:**

```
$cat exe1.c
#include<stdio.h>
main()
{
   execlp("ls","ls","-l",NULL);
   perror("No se ha podido ejecutar ls -l");
   exit(-1);
}
$make exe1
$exe1
total 964
-rw-------    1 castano  icc       2292 Apr 16  1997 Aceptacion
-rw-r-----    1 castano  icc        731 Feb  4  1997 bibliografia
drwxr-x--x    2 castano  icc       4096 Oct  6  2000 figuras
-rw-r-----    1 castano  icc      35205 May 27  1997 texto.tex
```

Función *execlp*

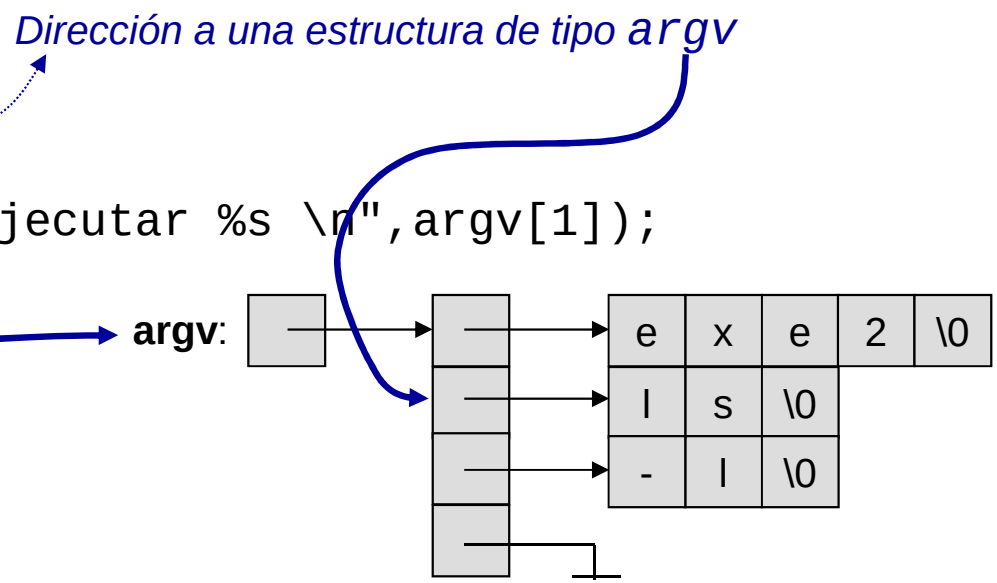
- **Ejemplo 1 (cont.):**



Función *execvp*

- **Ejemplo 2:**

```
$cat exe2.c
#include<stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
   execvp(argv[1],&argv[1]);
   printf ("No se ha podido ejecutar %s \n",argv[1]);
   exit(-1);
}
$
$
$make exe2
$exe2 ls -l
total 964
-rw-------    1 castano  icc       2292 Apr 16  1997 Aceptacion
-rw-r-----    1 castano  icc        731 Feb  4  1997 bibliografia
drwxr-x--x    2 castano  icc       4096 Oct  6  2000 figuras
-rw-r-----    1 castano  icc      35205 May 27  1997 texto.tex
```



Jerarquía de procesos

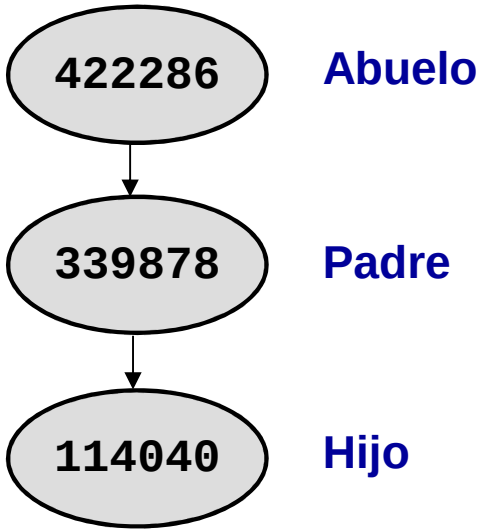
- **Ejemplo 1: Jerarquía Abuelo-Padre-Hijo**

```
#include<stdio.h>
main()
{int estado;
  if (fork()!=0)
  {    wait (&estado);
       printf("Proceso Abuelo   ");
       printf("PID=%d  PPID=%d\n",getpid(),getppid());
  } else {
    if (fork()!=0)
    {  wait (&estado);
       printf("Proceso Padre    ");
       printf("PID=%d  PPID=%d\n",getpid(),getppid());
    } else {
       printf("Proceso Hijo     ");
       printf("PID=%d  PPID=%d\n",getpid(),getppid());
    }
  }
  exit(0);
}
```

Jerarquía de procesos

- **Ejemplo 1: Jerarquía Abuelo-Padre-Hijo (cont.)**
  - Ejemplo de salida (única) de ejecución:

```
Proceso Hijo     PID=114040  PPID=339878
Proceso Padre    PID=339878  PPID=422286
Proceso Abuelo   PID=422286  PPID=211238
```



Jerarquía de procesos

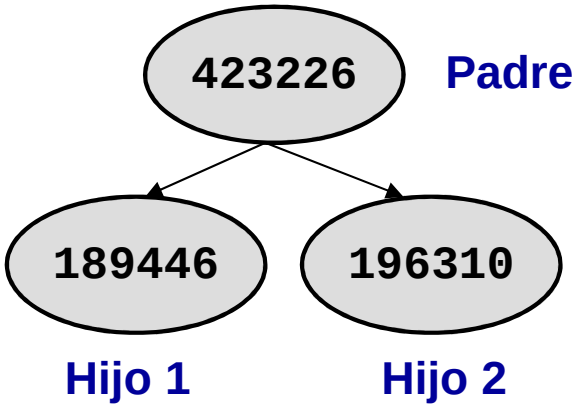
- **Ejemplo 2: Jerarquía Padre - Hijo 1- Hijo 2**

```
#include<stdio.h>
main()
{
  if (fork()!=0)
  {    printf("Proceso Padre    ");
       printf("PID=%d  PPID=%d\n",getpid(),getppid());
       if (fork()!=0)
       {  printf("Proceso Padre    ");
          printf("PID=%d  PPID=%d\n",getpid(),getppid());
       } else {
          printf("Proceso Hijo 2   ");
          printf("PID=%d  PPID=%d\n",getpid(),getppid());
       }
  } else {
       printf("Proceso Hijo 1   ");
       printf("PID=%d  PPID=%d\n",getpid(),getppid());
  }
  exit(0);
}
```

Jerarquía de procesos

- **Ejemplo 2: Jerarquía Padre - Hijo 1- Hijo 2 (cont.)**
  - Ejemplo de posible (no única) salida de ejecución:

```
Proceso Padre    PID=423226  PPID=211238
Proceso Hijo 1   PID=189446  PPID=423226
Proceso Padre    PID=423226  PPID=211238
Proceso Hijo 2   PID=196310  PPID=423226
```



Jerarquía de procesos

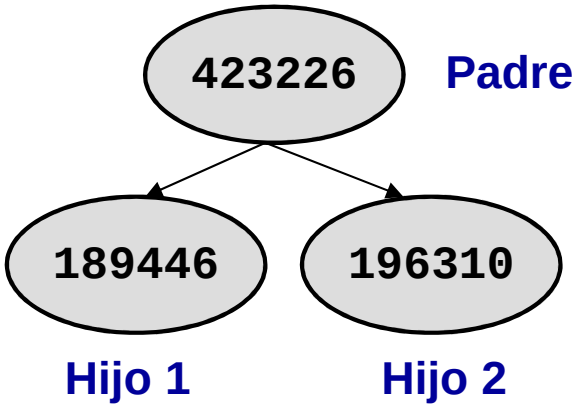
- **Ejemplo 2: Jerarquía Padre - Hijo 1- Hijo 2 con salida única**

```
#include<stdio.h>
main()
{                                  wait(NULL);
  if (fork()!=0)
  {    printf("Proceso Padre    ");
       printf("PID=%d  PPID=%d\n",getpid(),getppid());
       if (fork()!=0)
       {  printf("Proceso Padre    ");
          printf("PID=%d  PPID=%d\n",getpid(),getppid());
       } else {
          printf("Proceso Hijo 2   ");
          printf("PID=%d  PPID=%d\n",getpid(),getppid());
       }
  } else {
       printf("Proceso Hijo 1   ");
       printf("PID=%d  PPID=%d\n",getpid(),getppid());
  }
  exit(0);
}
```

Jerarquía de procesos

- **Ejemplo 2: Jerarquía Padre - Hijo 1- Hijo 2 con salida única (cont.)**
  - Ejemplo de salida (única) de ejecución:

```
Proceso Hijo 1   PID=189446  PPID=423226
Proceso Padre    PID=423226  PPID=211238
Proceso Hijo 2   PID=196310  PPID=423226
Proceso Padre    PID=423226  PPID=211238
```



Estado de finalización del proceso hijo

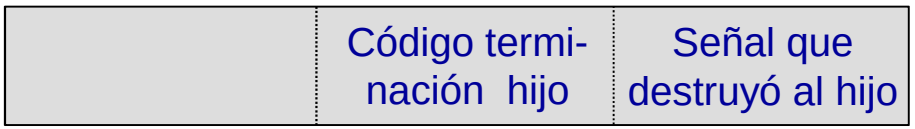
- **Ejemplo 3:**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>


int main(int argc, char **argv){
  int i,pid,estado;
  int j;

  j = 2;
  for (i=0;i<atoi(argv[1]);i++)
  {

     if (fork()==0) {
      execlp(argv[j],argv[j],argv[j+1],argv[j+2],NULL);
      printf("Llamada exec erronea\n");
      exit(-1);
     }
```

Estado de finalización del proceso hijo

- **Ejemplo 3 (cont.):**

```
else {
        j = j+3;
        pid=wait(&estado);
        estado = estado & 0x0000ff00;
        estado = estado >> 8;
        printf("Hijo %d finalizado con código= %d\n", pid, estado);
    }
  }
  exit(0);
}
```



Tema 2. Procesos e hilos

UNIVERSITAT JAUME·I

Índice

- Concepto de proceso
- Información del proceso
- Estados del proceso
- Formación de un proceso
- Cambio de contexto
- Servicios POSIX para gestión de procesos
- Concepto de hilo de ejecución
- Servicios POSIX para gestión de hilos
- Planificación de procesos e hilos

Concepto de hilo de ejecución

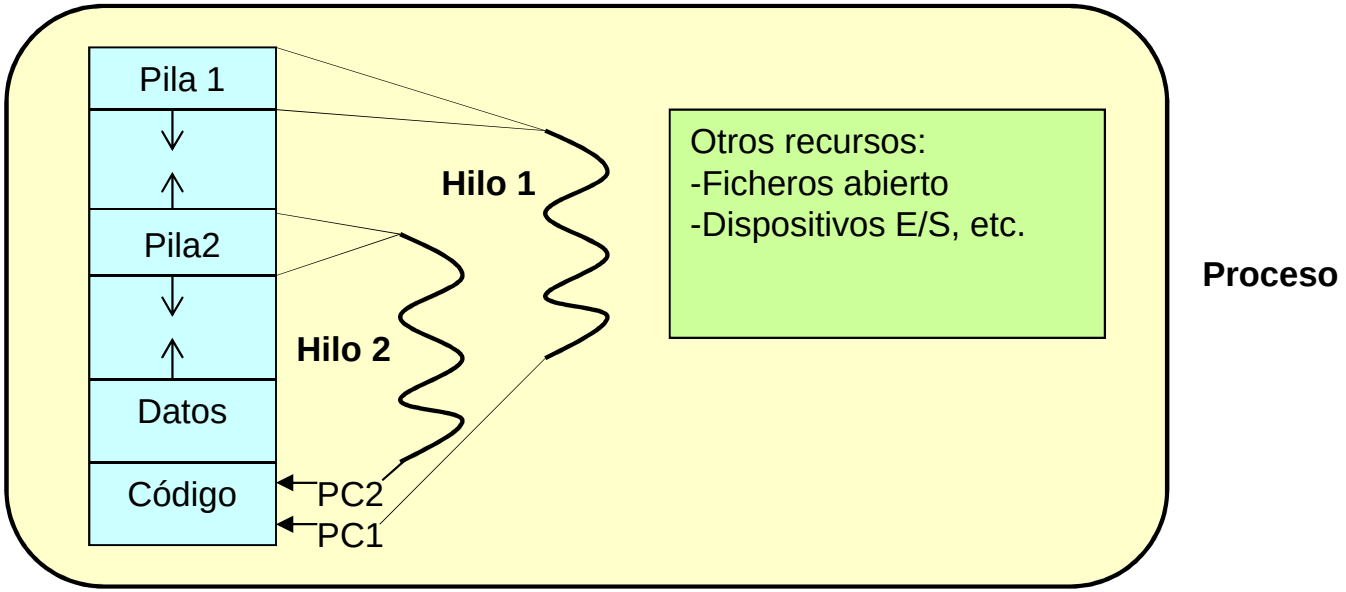
- **Características básicas del modelo tradicional de *proceso* (*pesado*):**
  - Ejecución secuencial.
  - Ejecución independiente.
- **Planteamiento:**
  - Varios procesos pueden cooperar para resolver una misma tarea del SO → Ejecución concurrente entre procesos → Comunicación entre procesos, por ejemplo, a través de memoria.
  - Un programa podría incluir varias actividades concurrentes → Ejecución concurrente dentro de un proceso.

Concepto de hilo de ejecución

- **¿Qué es un hilo de ejecución?**
  - También llamado *hebra*, *proceso ligero*, *flujo*, *subproceso* o *"thread"*.
  - Programa en ejecución que comparte la imagen de memoria y otros recursos del proceso con otros hilos.
  - Desde el punto de vista de programación: Función cuya ejecución se puede lanzar en paralelo con otras.
  - Un proceso puede contener uno o más hilos.

Concepto de hilo de ejecución

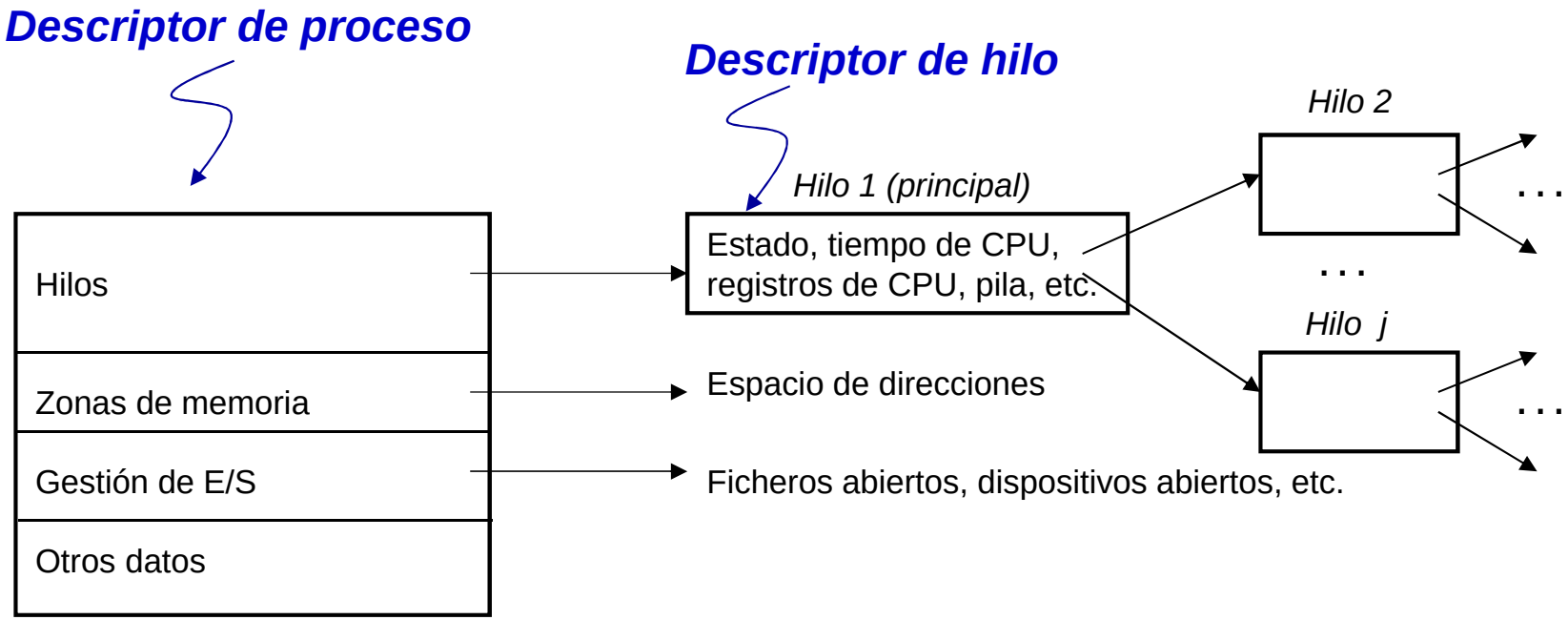
- **¿Qué es un hilo de ejecución (cont.)?**



Hilo: unidad de planificación Proceso: unidad de asignación de recursos

Concepto de hilo de ejecución

- **Descriptor de un proceso y de un hilo:**



Concepto de hilo de ejecución

- **Descriptor de un proceso y de un hilo:**
  - Todos los hilos de un proceso comparten el mismo entorno de ejecución (variables globales, espacio de direcciones, ficheros abiertos, etc.).
  - Cada hilo tiene su propio juego de registros de CPU, pila, variables locales, etc.
  - No existe protección entre hilos: un error en un hilo puede estropear la pila de otro.
  - Para ordenar la forma en la que los hilos acceden a datos comunes hay que emplear mecanismos de sincronización.

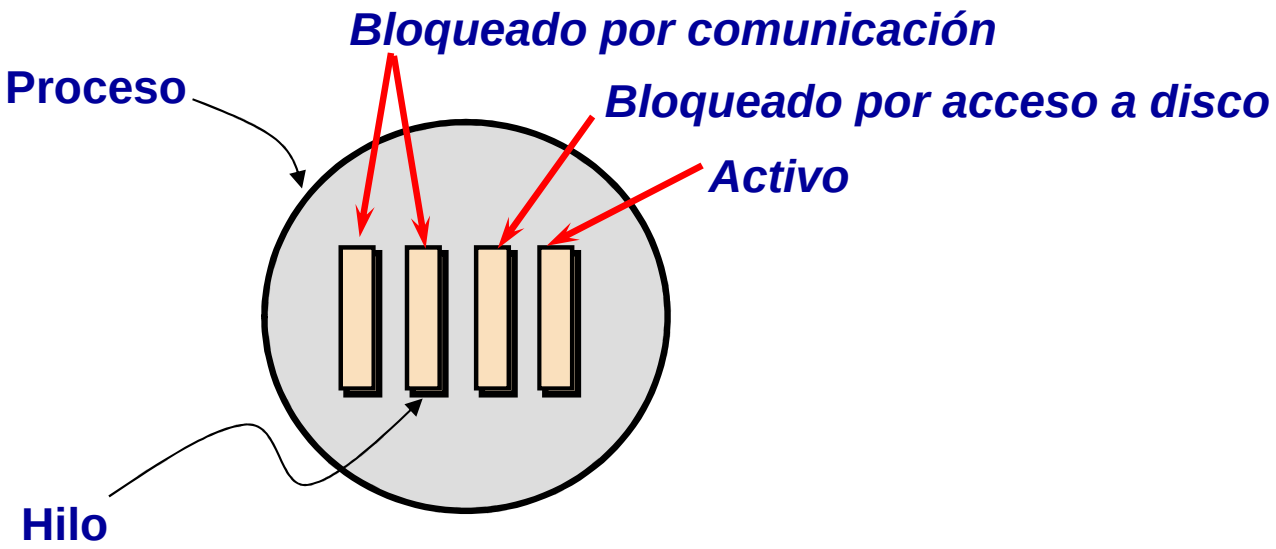
Entre procesos convencionales sí

MC

Variables globales a todos los hilos
Variables locales del hilo principal
Variables locales del hilo subordinado 1
Variables locales del hilo subordinado 2
...

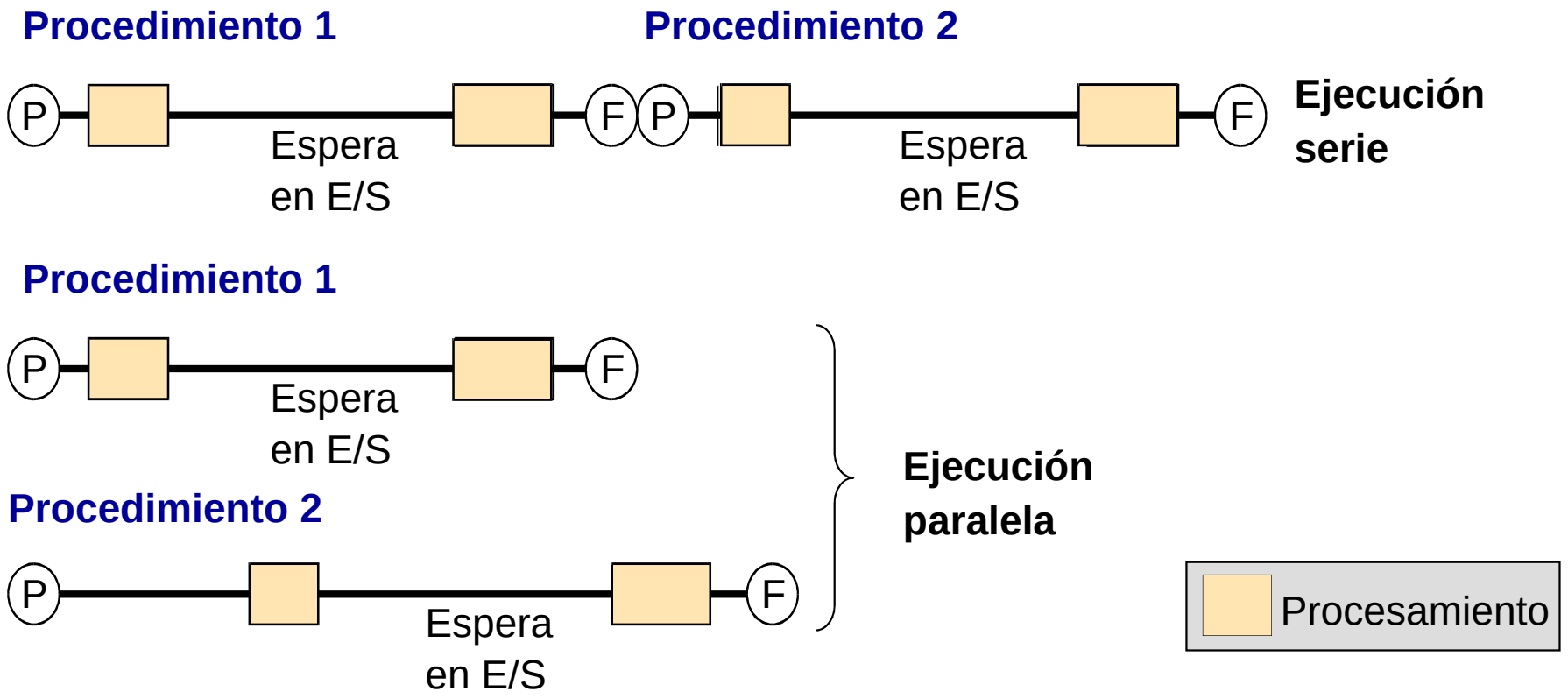
Estados de un hilo y de un proceso

- **Estado de un proceso con hilos:**
  - Combinación de los estados de sus hilos:
    - Si hay un hilo en ejecución → Proceso en ejecución
    - Si no hay hilos en ejecución pero sí preparados → Proceso preparado
    - Si todos sus hilos bloqueados → Proceso bloqueado

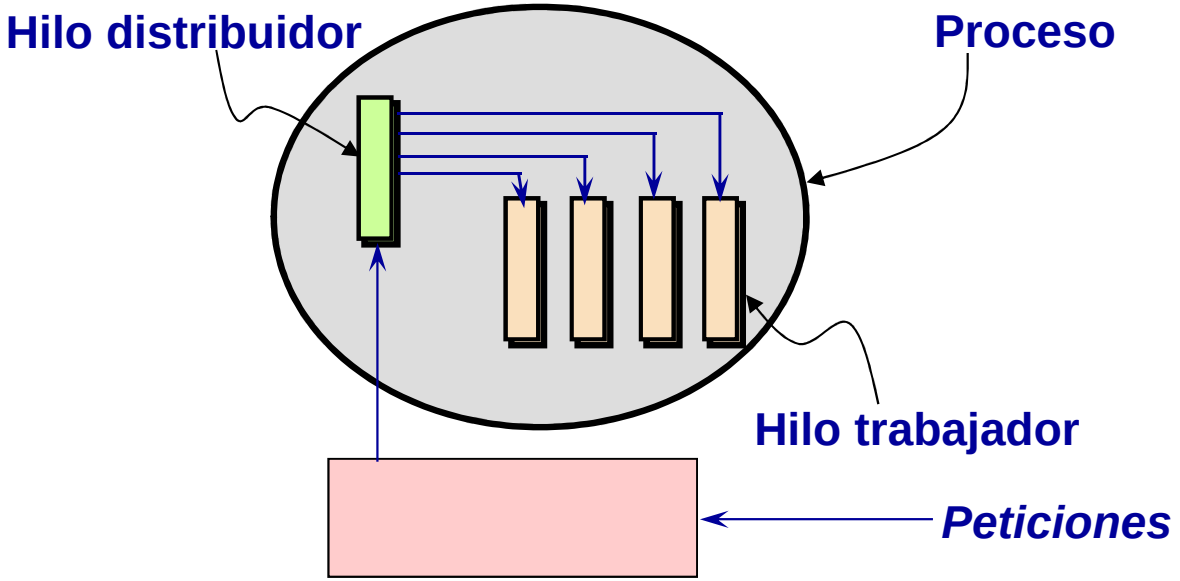


Paralelización usando hilos

- Los hilos permiten paralelizar la ejecución de una aplicación
- Mientras un hilo está bloqueado, otro podría ejecutarse
  - Uso de llamadas al sistema bloqueantes por hilo



Servidor con múltiples hilos



- **Hilos trabajadores:**
  - Pueden crearse a medida que se necesitan y destruirse al finalizar la tarea encomendada
  - Pueden existir siempre y quedar libres al finalizar la tarea encomendada
    - Más eficiente (evita el trabajo de crear y destruir hilos)

Servidor con múltiples hilos

- **Ejemplo: Servidor secuencial de ficheros**

Algoritmo

```
Mientras no haya que terminar {
    Esperar a que llegue una petición
    Comprobar que la petición es correcta
    Si (los datos no están en la caché) {
        Realizar operación de E/S bloqueante sobre disco
    }
    Enviar resultado
}
```

Caché para bloques accedidos recientemente

- Sencillo
- Prestaciones pobres: permanece bloqueado

Servidor con múltiples hilos

- **Ejemplo (cont.): Servidor de ficheros con múltiples hilos**

Hilo distribuidor

Algoritmo 1

Mientras no haya que terminar {
Esperar a que llegue una petición
Esperar trabajador libre
}

Hilo trabajador

Algoritmo 2

Mientras no haya que terminar {
Esperar trabajo
Comprobar que la petición es correcta
Si (los datos no están en la caché) {
Realizar operación de E/S bloqueante sobre disco
}
Enviar resultado
Avisar que está libre
}

- Mayor complejidad
- Buenas prestaciones

Ventajas de hilos de ejecución

- **Ventajas de utilizar múltiples hilos:**
  - La ejecución concurrente de hilos de un mismo proceso puede mejorar la eficiencia del sistema.
    - Paralelismo dentro del proceso (en multiprocesadores).
    - Las operaciones bloqueantes no paralizan al proceso (completo).
    - Comunicación entre hilos que comparten memoria más rápida que entre procesos.
  - Mayor eficiencia que con múltiples procesos en:
    - Creación/eliminación de unidades de planificación.
    - Cambio de unidad de planificación.
  - Facilidad de implementación.

Tema 2. Procesos e hilos

UNIVERSITAT JAUME·I

Índice

- Concepto de proceso
- Información del proceso
- Estados del proceso
- Formación de un proceso
- Cambio de contexto
- Servicios POSIX para gestión de procesos
- Concepto de hilo de ejecución
- Servicios POSIX para gestión de hilos
- Planificación de procesos e hilos

Servicios POSIX para gestión de hilos

UNIVERSITAT JAUME·I

Pthreads

```
int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy (pthread_attr_t *attr);

int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);

int pthread_create (pthread_t *thread,
    const pthread_attr_t  *attr, void *(*func)(void *),
    void *arg);

int pthread_exit(void *value);

int pthread_cancel (pthread_t *thread);

int pthread_join(pthread_t thid, void **value);

pthread_t pthread_self(void);
```

Servicios POSIX sobre atributos de hilos

- **Creación de atributos:**
  - Sintaxis:

```
int pthread_attr_init(pthread_attr_t *attr);
```

  - Descripción:
    - Inicia un objeto atributo de tipo `pthread_attr_t` con las propiedades que tendrán los hilos que se creen posteriormente
    - Los atributos permiten especificar: tamaño de pila, prioridad, política de planificación, etc.
    - Existen diversas llamadas para modificar los atributos

Servicios POSIX sobre atributos de hilos

- **Destrucción de atributos:**
  - Sintaxis:

```
int pthread_attr_destroy (pthread_attr_t *attr);
```

  - Descripción:
    - Destruye el objeto atributo de tipo `pthread_attr_t` pasado como argumento a la misma

Servicios POSIX sobre atributos de hilos

- **Establecimiento del estado de terminación:**
  - Sintaxis:

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
```

  - Descripción:
    - Establece el estado de terminación de un hilo:
      - Si "detachstate" = PTHREAD_CREATE_DETACHED
        - ✓ El hilo se considerará como “independiente”
        - ✓ El hilo liberará sus recursos cuando finalice su ejecución
      - Si "detachstate" = PTHREAD_CREATE_JOINABLE
        - ✓ El hilo se considerará como “no independiente”
        - ✓ El hilo no liberará todos los recursos (descriptor y pila) cuando finalice su ejecución, es necesario utilizar `pthread_join()`
        - ✓ Habitualmente, valor por defecto

Servicios POSIX sobre gestión de hilos

- **Creación de hilos:**
  - Sintaxis:

```
int pthread_create (pthread_t *thread,
    const pthread_attr_t  *attr, void *(*func)(void *),
    void *arg);
```

  - Descripción:
    - Crea un hilo con atributos `attr` que ejecuta `func` con argumentos `arg`

Servicios POSIX sobre gestión de hilos

- **Finalización de hilos:**
  - Sintaxis:

```
int pthread_exit(void *value);
```

  - Descripción:
    - Finaliza la ejecución del hilo que invoca la función, indicando su estado de terminación

- **Cancelación de hilos:**
  - Sintaxis:

```
int pthread_cancel(pthread_t *thread);
```

  - Descripción:
    - El hilo que invoca la función finaliza (cancela) la ejecución de otro hilo

Servicios POSIX sobre gestión de hilos

- **Suspensión de hilos:**
  - Sintaxis:

```
int pthread_join(pthread_t thid, void **value);
```

  - Descripción:
    - Suspende la ejecución de un hilo hasta que termina el hilo con identificador `thid` (no necesariamente un hilo hijo)
    - Deja el estado de terminación del hilo en la posición apuntada por `value`
    - Sólo se puede solicitar este servicio sobre hilos no independientes

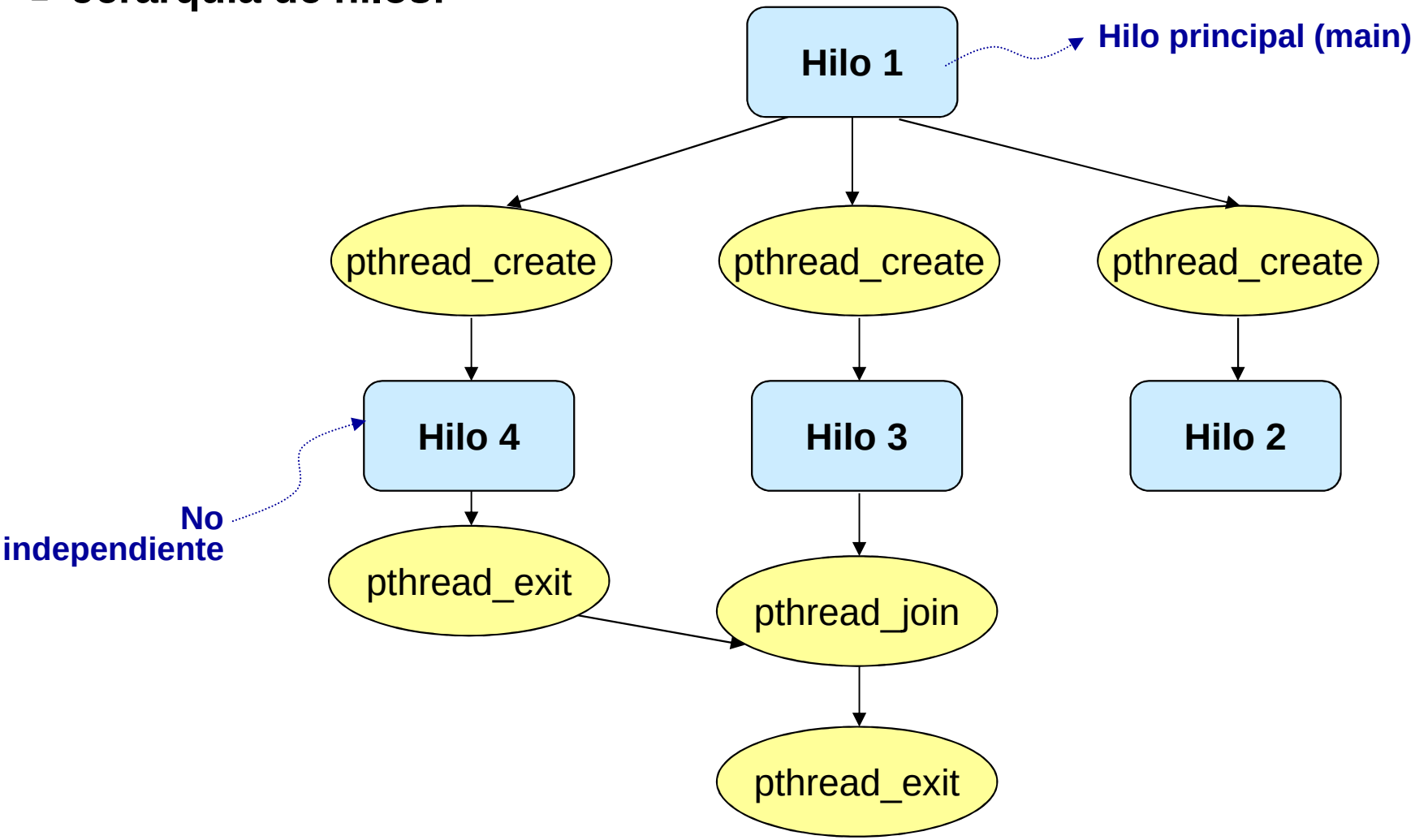
- **Identificación de hilos:**
  - Sintaxis:

```
pthread_t pthread_self(void);
```

  - Descripción:
    - Devuelve el identificador del hilo que ejecuta la llamada

Servicios POSIX sobre gestión de hilos

- **Jerarquía de hilos:**



Servicios POSIX sobre gestión de hilos

- **Compilación del código:**
  - Necesario enlazar con la librería de pthreads:

```
gcc  programa.c  -o  programa  –lpthread
```

¡IMPORTANTE!

Servicios POSIX para gestión de hilos

- **Ejemplo 1:**

```
#include <stdio.h>
#include <pthread.h>

#define MAX_THREADS 10

void *func() {
  printf("Thread %u \n", pthread_self());
  pthread_exit(0);
}

main() {
  int i;
  pthread_t hilo[MAX_THREADS];

  for(i = 0; i < MAX_THREADS; i ++)
    pthread_create(&hilo[i], NULL, &func, NULL);
  for(i = 0; i < MAX_THREADS; i ++)
    pthread_join(hilo[i], NULL);
  pthread_exit(0);
}
```

¡Ojo! No es %d

A partir de aquí, el hilo principal y los hilos creados se ejecutan concurrentemente. Cualquiera puede acabar 1º, 2º o último

Servicios POSIX para gestión de hilos

- **Ejemplo 2: Paso de argumento al hilo**

```
#include <pthread.h>
#include <stdio.h>

void *hilo (void *cadena)
{ int i;
  for (i=0; i<10; i++)
  {   sleep(2);
      printf("Hilo (%u): %u %s \n",pthread_self(),i, (char *) cadena);
  }
  pthread_exit(0);
}
int main()
{ char *cadena1="Hola";
  char *cadena2="Adios";

  pthread_t hilo1, hilo2;

  pthread_create(&hilo1, NULL,  &hilo, (void *) cadena1);
  pthread_create(&hilo2, NULL,  &hilo, (void *) cadena2);

  pthread_join(hilo1, NULL);
  pthread_join(hilo2, NULL);

  pthread_exit (0);
}
```

Servicios POSIX para gestión de hilos

- **Ejemplo 3: Suma de dos vectores repartida entre dos hilos**

```
#include <pthread.h>
#include <stdio.h>

#define N 10

int A[N], B[N], C[N];        /* Variables globales*/

void *hilo (void *arg)       /* Cada hilo suma una parte del vector */
{
  int inicio, fin, i;

  inicio = *((int *) arg);
  fin = inicio + N/2;
  printf("Hilo (%u): Suma desde componente %d hasta %d\n", pthread_self(),
          inicio, fin);
  for (i=inicio;i<fin;i++)
    C[i]=A[i]+B[i];
  pthread_exit (0);
}
```

Servicios POSIX para gestión de hilos

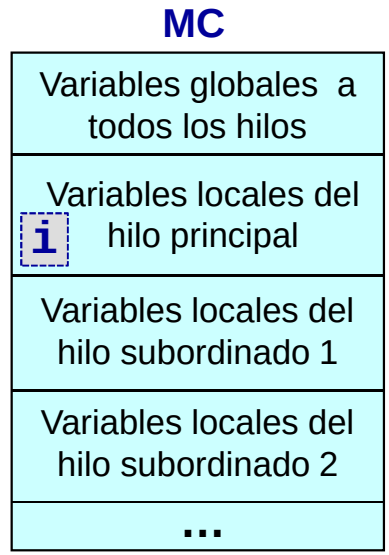
- **Ejemplo 3: Suma de dos vectores repartida entre dos hilos (cont.)**

```
main()
{ int j, dim1, dim2;
  pthread_t hilo1, hilo2;

  for (j=0;j<10;j++) {A[j] = j; B[j]=j*j; C[j]=0;}

  dim1 = 0;
  pthread_create(&hilo1, NULL, &hilo, (void *) &dim1);
  dim2 = N/2;
  pthread_create(&hilo2, NULL, &hilo, (void *) &dim2);

  pthread_join(hilo1,NULL);
  pthread_join(hilo2,NULL);

  for (j=0;j<10;j++)   printf("A[%d] = %d\n", j,A[j]);
  printf("\n");
  for (j=0;j<10;j++)   printf("B[%d] = %d\n", j,B[j]);
  printf("\n Suma:\n");
  for (j=0;j<10;j++)   printf("C[%d] = %d\n", j,C[j]);

  pthread_exit (0);
}
```

Servicios POSIX para gestión de hilos

- **Ejemplo 4: Paso de índice como argumento al hilo de forma INCORRECTA**

```
#include <stdio.h>
#include <pthread.h>

#define MAX_THREADS 10

void *func (void * arg) {
  int parametro;
  parametro = *(int *) arg;
  printf("Thread %u lee %d\n", pthread_self(),parametro);
  pthread_exit(0);
}

main() {
  int i;

  pthread_t hilo[MAX_THREADS];
  for(i = 0; i < MAX_THREADS; i ++)
    pthread_create(&hilo[i], NULL, &func, &i);
  for(i = 0; i < MAX_THREADS; i ++)
    pthread_join(hilo[i], NULL);
  pthread_exit(0);
}
```



Damos una dirección cuyo contenido cambia.

El valor que recibirá el hilo puede ser el de otra iteración

Servicios POSIX para gestión de hilos

- **Ejemplo 4: Paso de índice como argumento al hilo de forma INCORRECTA**

```
$ ejemplo4_no_ok
Thread 3046009712 lee 0
Thread 3035519856 lee 0
Thread 3025030000 lee 0
Thread 3056499568 lee 0
Thread 3014540144 lee 0
Thread 3077479280 lee 0
Thread 3004050288 lee 1
Thread 2993560432 lee 1
Thread 2983070576 lee 1
Thread 3066989424 lee 1
$
```

Cada hilo debería leer un número diferente 0, 1, 2, …

Servicios POSIX para gestión de hilos

- **Ejemplo 4: Paso de índice como argumento al hilo de forma CORRECTA**

```
#include <stdio.h>
#include <pthread.h>

#define MAX_THREADS 10

void *func (void * arg) {
  int parametro;
  parametro = (int) arg;

  printf("Thread %u lee %d\n", pthread_self(),parametro);
  pthread_exit(0);
}

main() {
  int i;
  pthread_t hilo[MAX_THREADS];

  for(i = 0; i < MAX_THREADS; i ++)
    pthread_create(&hilo[i], NULL, &func, (void *) i);

  for(i = 0; i < MAX_THREADS; i ++)
    pthread_join(hilo[i], NULL);
  pthread_exit(0);
}
```

Da un warning pero funciona

Servicios POSIX para gestión de hilos

- **Ejemplo 4: Paso de índice como argumento al hilo de forma CORRECTA**

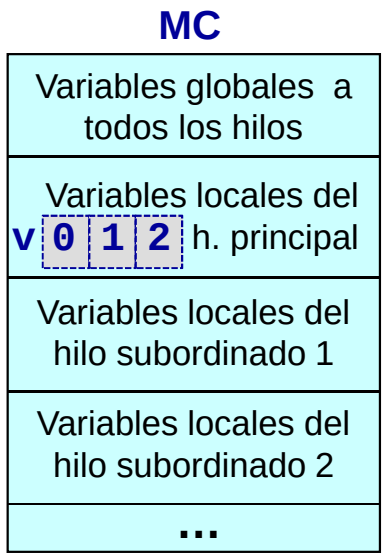
```
$ ejemplo4_ok
Thread 3046112112 lee 3
Thread 3035622256 lee 4
Thread 3056601968 lee 2
Thread 3067091824 lee 1
Thread 3025132400 lee 5
Thread 3014642544 lee 6
Thread 3004152688 lee 7
Thread 2993662832 lee 8
Thread 2983172976 lee 9
Thread 3077581680 lee 0
$
```

Cada hilo lee un número diferente 0, 1, 2, …

Servicios POSIX para gestión de hilos

- **Ejemplo 4: Paso de índice como argumento al hilo de OTRA forma CORRECTA**

```
#include <stdio.h>
#include <pthread.h>

#define MAX_THREADS 10

void *func (void * arg) {
  int parametro;
  parametro = * (int *) arg;
  printf("Thread %u lee %d\n", pthread_self(),parametro);
  pthread_exit(0);
}

main() {
  int       i, v[MAX_THREADS];
  pthread_t hilo[MAX_THREADS];

  for(i = 0; i < MAX_THREADS; i ++) v[i] = i;
  for(i = 0; i < MAX_THREADS; i ++)
    pthread_create(&hilo[i], NULL, &func, (void *) &v[i]);
  for(i = 0; i < MAX_THREADS; i ++)
    pthread_join(hilo[i], NULL);
  pthread_exit(0);
}
```



Servicios POSIX para gestión de hilos

- **Ejemplo 5: Paso de varios argumentos al hilo**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define MAX_THREADS 10

typedef struct {
  int i;
  int j;
} parametros;

void *func (void * arg) {
  parametros * argum;
  int argum1, argum2;

  argum = (parametros *) arg;
  argum1 = argum->i;
  argum2 = argum->j;

  printf("Thread %u lee %d y %d \n", pthread_self(), argum1, argum2);
  pthread_exit(0);
}
```

Servicios POSIX para gestión de hilos

- **Ejemplo 5: Paso de varios argumentos al hilo (cont.)**

```
main() {
  int i, j;
  pthread_t hilo[MAX_THREADS];
  parametros * argum;

  for(i = 0; i < MAX_THREADS; i ++)
  {
    j = i * i;
    argum = (parametros *) malloc(sizeof(argum));
    argum->i=i;
    argum->j=j;
    pthread_create(&hilo[i], NULL, &func, (void *) argum);
  }
  for(i = 0; i < MAX_THREADS; i ++)
    pthread_join(hilo[i], NULL);
  pthread_exit(0);
}
```

Se “empaquetan” los 2 argumentos

Servicios POSIX para gestión de hilos

- **Ejemplo 6: Devolución de un valor del hilo**

```
#include <stdio.h>
#include <pthread.h>

#define MAX_THREADS 10

void *func (void * arg) {
  int parametro, cuadrado;
  parametro = (int) arg;

  printf("Thread %u lee %d\n", pthread_self(),parametro);
  cuadrado = parametro * parametro;

  pthread_exit ((void *) cuadrado);
}
```

Servicios POSIX para gestión de hilos

- **Ejemplo 6: Devolución de un valor del hilo (cont.)**

```
main() {
  int i;
  pthread_t hilo[MAX_THREADS];
  int retorno;

  for(i = 0; i < MAX_THREADS; i ++)
    pthread_create(&hilo[i], NULL, &func, (void*) i);

  for(i = 0; i < MAX_THREADS; i ++)
  {
    pthread_join (hilo[i], (void *) &retorno);
    printf("Recojo  %d\n", retorno);
  }
  pthread_exit(0);
}
```

Servicios POSIX para gestión de hilos

- **Ejemplo 6: Devolución de un valor del hilo (cont.)**

```
$ ejemplo6
Thread 3045997424 lee 3
Thread 3035507568 lee 4
Thread 3025017712 lee 5
Thread 3056487280 lee 2
Thread 3014527856 lee 6
Thread 3004038000 lee 7
Thread 2993548144 lee 8
Thread 2983058288 lee 9
Thread 3066977136 lee 1
Thread 3077466992 lee 0
Recojo  0
Recojo  1
Recojo  4
Recojo  9
Recojo  16
Recojo  25
Recojo  36
Recojo  49
Recojo  64
Recojo  81
$
```

Servicios POSIX para gestión de hilos

- **Ejemplo 7:**

```
#include <pthread.h>
#include <stdio.h>

int cont=0;

void *hilo()
{ sleep(2);
  cont=cont+1;
  printf("Hilo           (%d): cont=%d \n",pthread_self(),cont);
  pthread_exit(0);
}

int main()
{ pthread_t hilo1, hilo2;
  printf("Hilo principal (%u): cont=%d \n", pthread_self(), cont);

  pthread_create(&hilo1, NULL,  &hilo, NULL);
  pthread_create(&hilo2, NULL,  &hilo, NULL);

  pthread_join(hilo1,NULL);
  pthread_join(hilo2,NULL);

  printf("Hilo principal (%u): cont=%d \n",pthread_self(),cont);
  exit(0);
}
```

¡Ojo! Varios hilos accediendo a la misma **variable global**.
Necesario sincronizar acceso
(lo veremos en Tema 3)

Servicios POSIX para gestión de procesos

- **Ejemplo 7:** Código pseudo-equivalente con gestión de procesos pesados:

¿Por qué?

```
#include <stdio.h>
int main()
{ int estado, cont=0;
  printf("Padre  (%d): cont=%d \n",getpid(),cont);
  if (fork() != 0)
  {
    if (fork() != 0)
    { wait(&estado);
      wait(&estado);
      printf("Padre  (%d): cont=%d \n",getpid(),cont);
    } else { /* HIJO 2 */
        cont=cont+1;
        printf("Hijo 2 (%d): cont=%d \n",getpid(),cont);
    }
  } else { /* HIJO 1 */
        cont=cont+1;
        printf("Hijo 1 (%d): cont=%d \n",getpid(),cont);
  }
 exit(0);
}
```

Servicios POSIX para gestión de hilos

- **Ejemplo 8:**

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

int x=0;

void *fhilo1()
{ int i, cont;
  for (i=0; i<3; i++) {
     cont=x+1;
     printf ("Suma 1\n");
     sleep (random()%3);
     x=cont;
  }
  pthread_exit (NULL);
}
```

```
void *fhilo2()
{ int i, cont;
  for (i=0; i<3; i++) {
     cont=x-1;
     printf ("Resta 1\n");
     sleep (random()%3);
     x=cont;
   }
   pthread_exit (NULL);
}
```

Servicios POSIX para gestión de hilos

- **Ejemplo (cont.):**

```
main()
{ pthread_t hilo1, hilo2;
  time_t t;
  srandom (time(&t));
  printf ("Valor inicial de x: %d \n",x);
  pthread_create(&hilo1, NULL,  &fhilo1, NULL);
  pthread_create(&hilo2, NULL,  &fhilo2, NULL);
  pthread_join(hilo1,NULL);
  pthread_join(hilo2,NULL);
  printf ("Valor final de x: %d \n",x);
  exit(0);
}
```

Funcionamiento incorrecto
Varios hilos accediendo a
la variable global x

Tema 2. Procesos e hilos

UNIVERSITAT JAUME·I

Índice

- Concepto de proceso
- Información del proceso
- Estados del proceso
- Formación de un proceso
- Cambio de contexto
- Servicios POSIX para gestión de procesos
- Concepto de hilo de ejecución
- Servicios POSIX para gestión de hilos
- Planificación de procesos e hilos

UNIVERSITAT JAUME·I

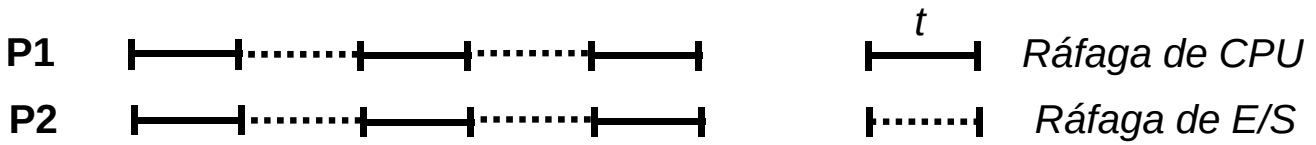
Planificación de procesos e hilos

- Conceptos básicos de planificación
- Parámetros de evaluación de algoritmos de planificación
- Algoritmos de planificación

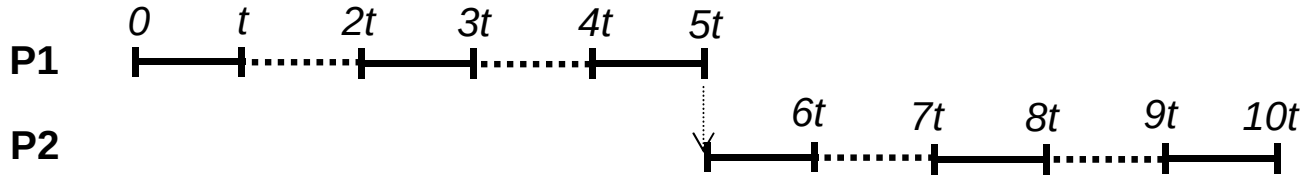
Conceptos básicos de planificación

- **Ciclo de ráfagas de CPU y de E/S:**
  - La ejecución de un proceso es una secuencia (ciclo) del tipo:

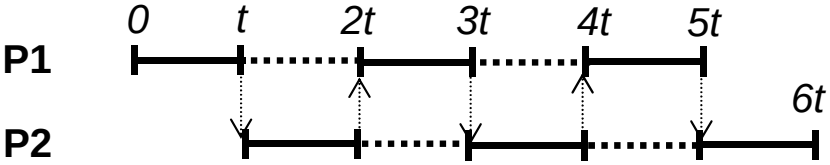
    CPU - E/S - CPU - E/S - CPU - E/S - ...
  - *Ejemplo*: Sean P1 y P2 dos procesos listos para ejecución.



Ejecución sin multiprogramación:



Ejecución con multiprogramación:



Conceptos básicos de planificación

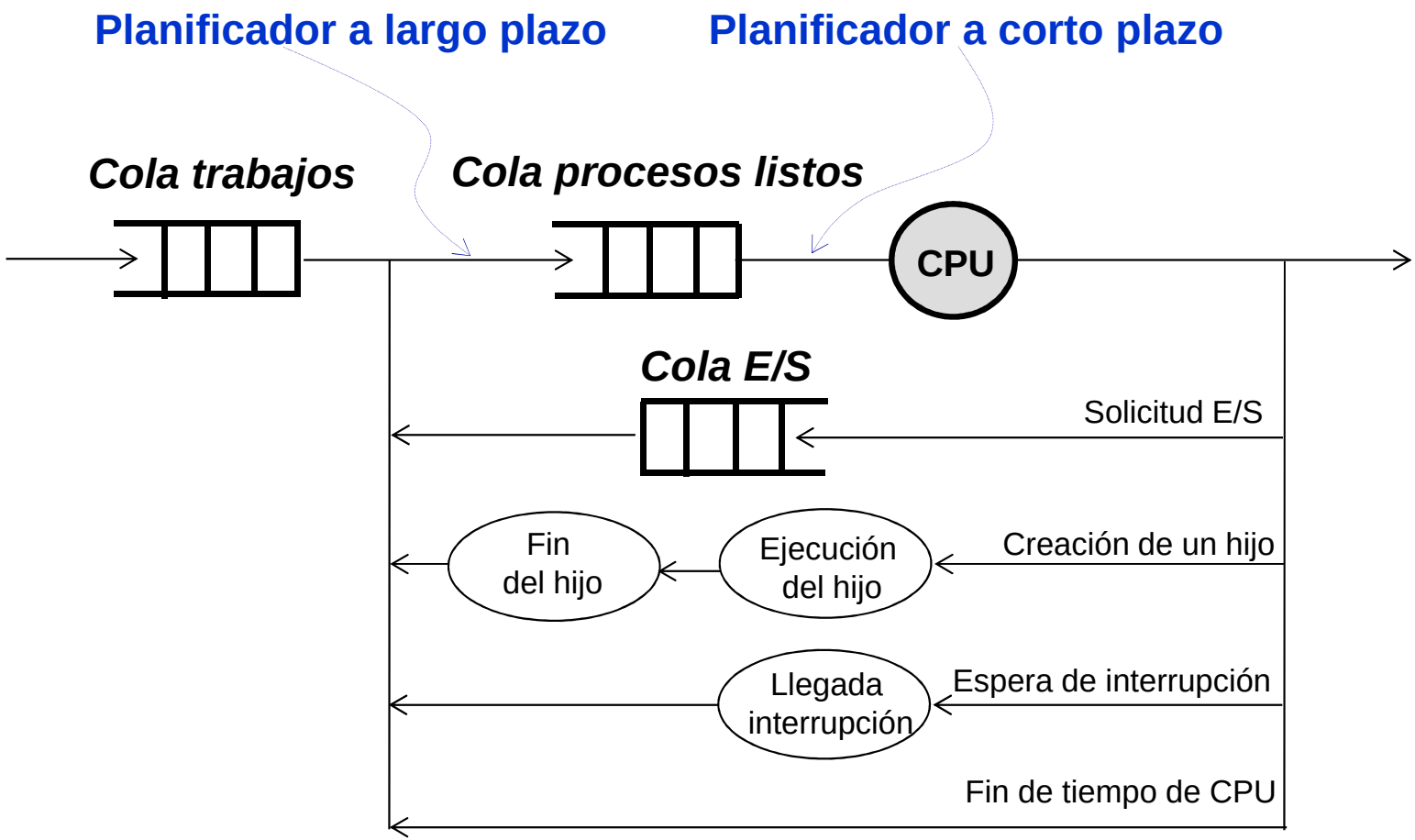
- Máxima utilización de la CPU con multiprogramación:
  - Cada vez que un proceso tiene que esperar, otro puede usar la CPU.
- **Clasificación de procesos en base a las ráfagas de CPU y E/S:**
  - *Procesos limitados por E/S*:
    - Emplean más tiempo en realizar E/S que en efectuar cálculos.
    - Muchas ráfagas de CPU pero cortas.
  - *Procesos limitados por la CPU:*
    - Emplean más tiempo en efectuar cálculos que en realizar E/S.
    - Pocas ráfagas de CPU pero largas.

Conceptos básicos de planificación

- **Colas de planificación:**
  - *Cola de trabajos*: Procesos del sistema que están esperando a ser asignados en memoria principal.
  - *Cola de procesos listos*: Procesos listos que están en memoria esperando ser ejecutados.
  - *Cola de dispositivos de E/S*: Colas de procesos esperando por un dispositivo de E/S.

Conceptos básicos de planificación

- **Migración de procesos entre las diferentes colas de planificación:**

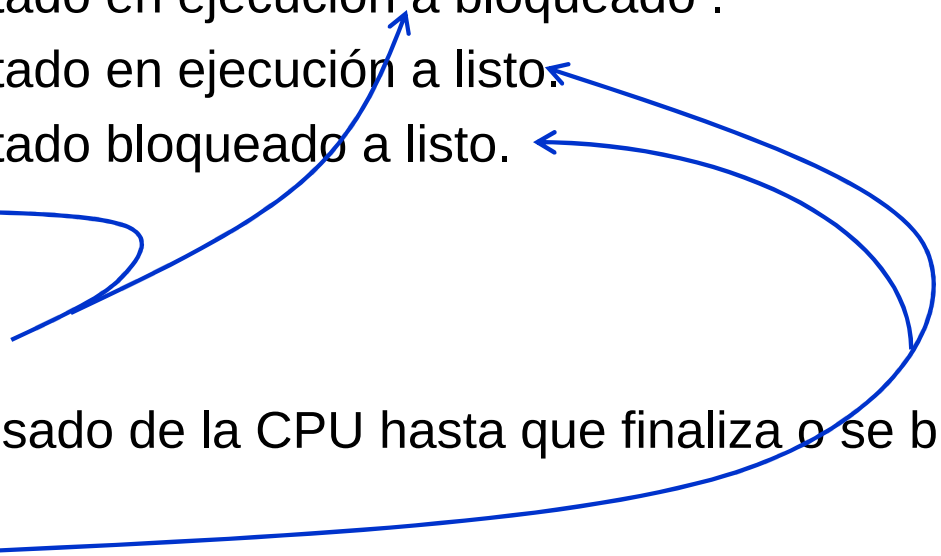


Conceptos básicos de planificación

- **Tipos de planificadores:**
  - *Planificador a largo plazo (planificador de trabajos):*
    - Selección de trabajos a cargar en memoria principal.
    - Invocado muy infrecuentemente (segundos o minutos).
    - Puede ser (más) lento.
    - Controla el grado de multiprogramación en el sistema.
  - *Planificador a corto plazo (planificador de la CPU):*
    - Selección del proceso listo que será ejecutado a continuación.
    - Invocado muy frecuentemente (milisegundos).
    - Debe ser rápido.
  - *Planificador a medio plazo*:
    - Traslado de un proceso en memoria principal a disco (intercambio o "swapping"). Posteriormente volverá a memoria principal.
    - Reduce la contienda por el uso de la CPU.
    - En ocasiones necesario ante los requisitos de memoria principal.

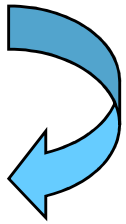
Planificador de la CPU

- **¿Cuándo puede actuar?**
  - Un proceso pasa de estado en ejecución a bloqueado .
  - Un proceso pasa de estado en ejecución a listo.
  - Un proceso pasa de estado bloqueado a listo.
  - Un proceso finaliza.
- **Panificación no expulsiva:**
  - Un proceso no es expulsado de la CPU hasta que finaliza o se bloquea.
- **Panificación expulsiva:**
  - Un proceso es expulsado de la CPU sin haber finalizado o antes de bloquearse.



Planificador de la CPU

- ***Recordando ...***
  - **Pasos para realizar un cambio de proceso:**
    - Guardar en la tabla de procesos el contexto del proceso en ejecución.
    - Seleccionar el siguiente proceso listo que pasa a ejecución.
    - Recuperar el contexto del nuevo proceso.



Planificador

  - El cambio de contexto debe ser lo más rápido posible.

Evaluación de algoritmos de planificación

- **Parámetros de evaluación de un algoritmo de planificación:**
  - *Utilización de la CPU*: Porcentaje de tiempo que el procesador está ocupado.
  - *Productividad de la CPU*: Número de trabajos por unidad de tiempo que finalizan.
  - *Tiempo de retorno*: Tiempo que tarda en ejecutarse un proceso.
  - *Tiempo de espera*: Tiempo de un proceso en lista de procesos listos.
  - *Tiempo de respuesta*: Tiempo de un proceso en dar la primera respuesta.
- **Criterios de optimización en planificación:**
  - Maximizar utilización y productividad de la CPU.
  - Minimizar tiempo de retorno, de espera y de respuesta.

Algoritmos de planificación

- **Algoritmos de planificación monoprocesador:**
  - Planificación por “turno de llegada” (FCFS)
  - Planificación por “primero el trabajo más corto” (SJF y SRTF)
  - Planificación basada en “prioridades”
  - Planificación por “turno rotatorio” (RR)
  - Planificación basada en “colas multinivel”
    - Colas no realimentadas
    - Colas realimentadas

Algoritmos de planificación

- **Algoritmo FCFS (“First-Come First-Served”):**
  - Los procesos pasan a CPU en orden de llegada a cola de procesos listos.

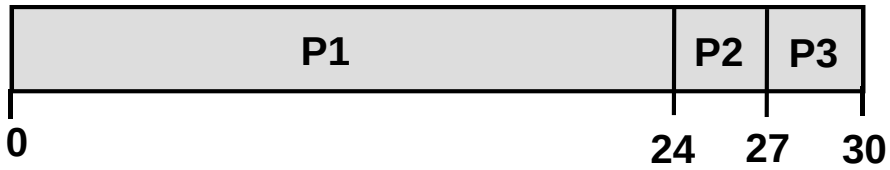
    Si el proceso en ejecución necesita E/S, se inserta al final de la cola de procesos listos al regresar a ésta.
  - Algoritmo no expulsivo.
  - Fácil implementación con cola FIFO.
  - Poco eficiente.

Algoritmos de planificación

- **Algoritmo FCFS (cont.):**
  - Ejemplo:

Proceso	***Ráfaga CPU***
P1	24 ut.
P2	3 ut.
P3	3 ut.

    - Orden de llegada (en instante 0) a cola de procesos listos: P1, P2, P3.
      - Diagrama de Gant para la planificación:



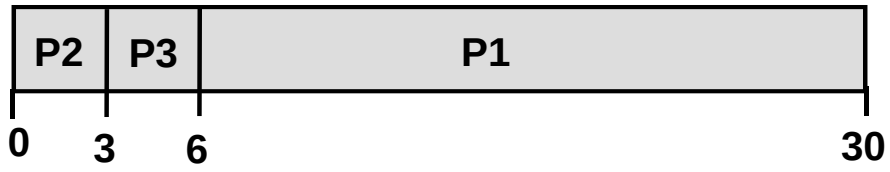
      - Tiempo de espera: TEP1=0 ut. TEP2=24 ut. TEP3=27 ut.
      - Tiempo medio de espera: (0+24+27)/3=17 ut.
      - *Efecto convoy*: Procesos en espera debido a procesos limitados por CPU.

Algoritmos de planificación

- **Algoritmo FCFS (cont.):**
  - Ejemplo:

Proceso	***Ráfaga CPU***
P1	24 ut.
P2	3 ut.
P3	3 ut.

    - Orden de llegada (en instante 0) a cola de procesos listos: P2, P3, P1.
      - Diagrama de Gant para la planificación:



      - Tiempo de espera: TEP1=6 ut. TEP2=0 ut. TEP3=3 ut.
      - Tiempo medio de espera: (6+0+3)/3=3 ut.
      - Mejores resultados que en el caso anterior.
      - Solución efecto convoy: Poner primero los trabajos más cortos.

Algoritmos de planificación

- **Algoritmo SJF (“Shortest Job First”):**
  - Asociar a cada proceso el tiempo de ráfaga de CPU.

    Seleccionar el proceso con menor ráfaga de CPU.

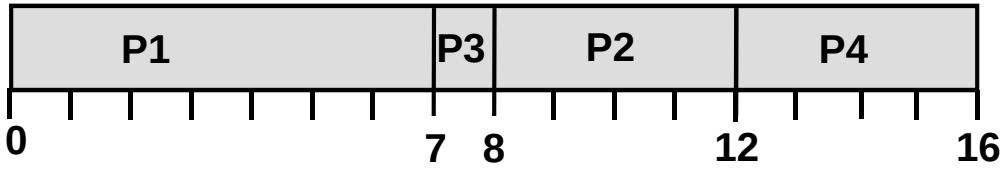
    En caso de empate, aplicar FCFS.
  - Algoritmo no expulsivo.

Algoritmos de planificación

- **Algoritmo SJF (cont.):**
  - Ejemplo:

Proceso	*Tiempo llegada*	*Ráfaga CPU*
P1	0	7
P2	2	4
P3	4	1
P4	5	4

  - ♣ Diagrama de Gant para la planificación:



  - ♣ Tiempo de espera: TEP1=0 ut. TEP2= 8-2 =6 ut.
    TEP3= 7-4 =3 ut. TEP4= 12-5 =7 ut.
  - ♣ Tiempo medio de espera: (0+6+3+7)/4=4 ut.

Algoritmos de planificación

- **Algoritmo SRTF (“Shortest Remaining Time First”):**
  - Asociar a cada proceso el tiempo de ráfaga de CPU restante.

    Seleccionar el proceso con menor ráfaga de CPU restante.

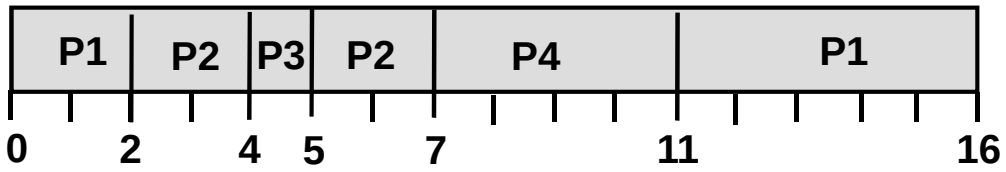
    En caso de empate, aplicar FCFS.
  - Algoritmo expulsivo: Realizar cambio de contexto si llega un proceso a la cola de procesos listos con ráfaga de CPU menor que el tiempo restante del proceso en ejecución.
  - Óptimo al minimizar el tiempo medio de espera.

Algoritmos de planificación

- **Algoritmo SRTF (cont.):**
  - Ejemplo:

Proceso	*Tiempo llegada*	*Ráfaga CPU*
P1	0	7
P2	2	4
P3	4	1
P4	5	4

  - ♣ Diagrama de Gant para la planificación:



  - ♣ Tiempo de espera: TEP1= 0+9 = 9 ut. TEP2= (2-2) + 1 =1 ut.
    TEP3= 4-4 =0 ut. TEP4= 7-5 =2 ut.
  - ♣ Tiempo medio de espera: (9+1+0+2)/4=3 ut.

Algoritmos de planificación

- **Cálculo de la siguiente ráfaga de CPU para algoritmos SJF y SRTF:**
  - No se puede conocer la longitud de la siguiente ráfaga de CPU.
  - Estimación en base al promedio exponencial de la longitudes de las ráfagas de CPU anteriores medidas:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\,\tau_n$$

Información reciente → αt_n

Datos históricos → $(1-\alpha)\,\tau_n$

siendo

- ♣ t_n : Longitud de la n-ésima ráfaga de CPU.
- ♣ τ_n: Longitud estimada para la n-ésima ráfaga de CPU.
- ♣ $0 \leq \alpha \leq 1$: Ponderación entre información reciente y datos históricos.

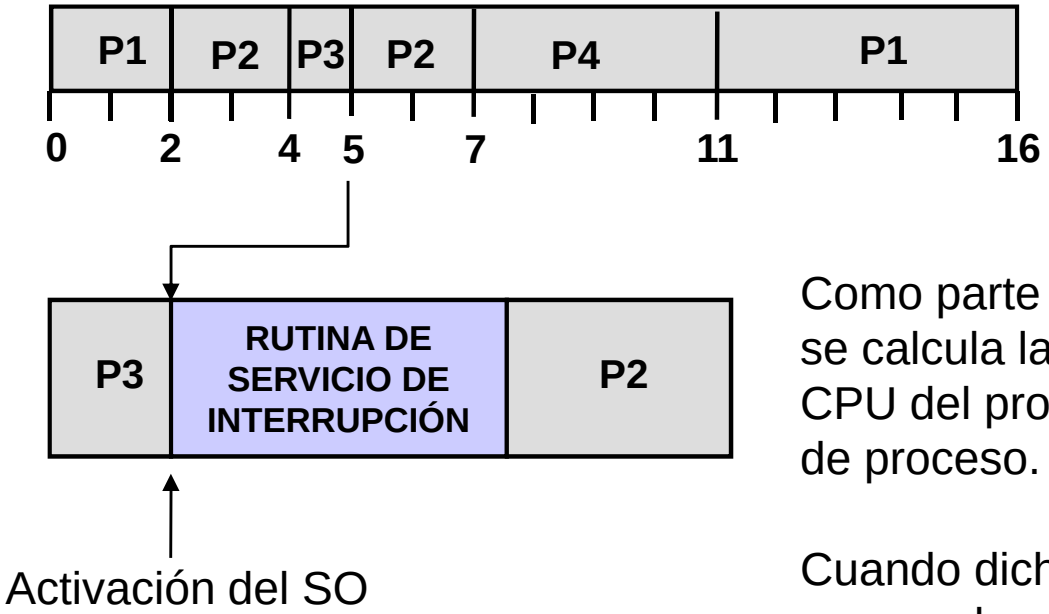
  - Expandiendo la fórmula:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\,\alpha\, t_{n-1} + \ldots + (1-\alpha)^j\,\alpha\, t_{n-j} + \ldots + (1-\alpha)^{n+1}\,\tau_0$$

Como $0 \leq \alpha \leq 1$, cada término tiene menos peso que su predecesor.

Algoritmos de planificación

- **¿Cuándo se aplica la fórmula de cálculo de la siguiente ráfaga de CPU?**
  - Al finalizar cada una de las ráfagas de CPU



Como parte de la rutina de servicio de interrupción se calcula la aproximación para el siguiente ciclo de CPU del proceso 3 y se almacena en su descriptor de proceso.

Cuando dicho proceso llegue de nuevo a estado preparado, ese dato se utilizará para que el planificador decida cuándo pasa a ejecución.

  - Información del descriptor de proceso sobre planificación SJF y SRTF: Longitud estimada para la siguiente ráfaga de CPU, τ_{n+1}.

Algoritmos de planificación

- **Algoritmo basado en prioridades:**
  - Asociar a cada proceso una prioridad (número entero).

    Asignar la CPU al proceso más prioritario.

    En caso de empate, aplicar FCFS.
  - Asumiremos mayor prioridad con menor número entero.
  - Algoritmo expulsivo o no expulsivo.

Algoritmos de planificación

- **Algoritmo basado en prioridades (cont.):**
  - ¿Por qué los algoritmos SJF, SRTF y FIFO son un caso especial de un algoritmo de planificación por prioridades?

Algoritmos de planificación

- **Algoritmo basado en prioridades (cont.):**
  - Problema: *Inanición*, los procesos con baja prioridad nunca acceden a la CPU.

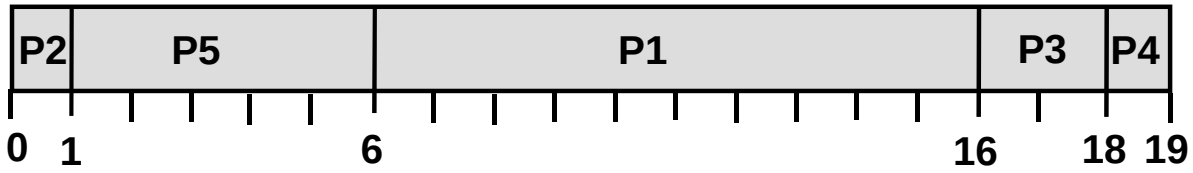
    Solución: *Envejecimiento*, la prioridad de un proceso aumenta gradualmente con el tiempo de espera ⌫ *Prioridades dinámicas*.
  - Información del descriptor de un proceso sobre planificación: Prioridad del proceso.

Algoritmos de planificación

- **Algoritmo basado en prioridades (cont.):**
  - Ejemplo:

Proceso	***Prioridad***	***Ráfaga CPU***
P1	3	10
P2	1	1
P3	3	2
P4	4	1
P5	2	5

    - Orden de llegada (en instante 0) a cola de procesos listos: P1, P2, P3, P4, P5.
    - Diagrama de Gant para la planificación:



    - Tiempo medio de espera: (6+0+16+18+1)/5=6,2 ut.

Algoritmos de planificación

- **Algoritmo RR ("Round Robin"):**
  - Especialmente diseñado para sistemas de tiempo compartido.
  - Cola circular de procesos listos.
  - Asociar a cada proceso un tiempo de posesión de CPU (*cuantum q*).

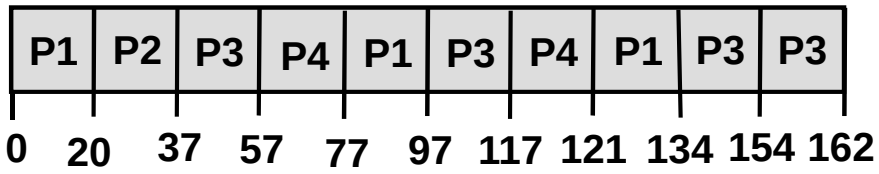
    Tras el cuantum *q* el proceso en ejecución se expulsa de la CPU y se pasa al final de la cola de procesos listos.
  - Algoritmo expulsivo cada cuantum *q* de tiempo.

Algoritmos de planificación

- **Algoritmo RR (cont.):**
  - Ejemplo:

Proceso	***Ráfaga CPU***
P1	53
P2	17
P3	68
P4	24

    - Con q=20
      - Diagrama de Gant para la planificación:



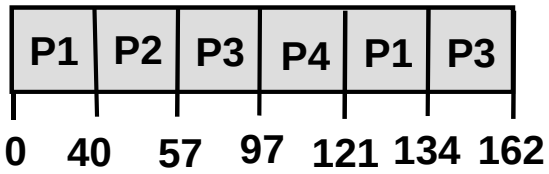
      - 9 cambios de contexto

Algoritmos de planificación

- **Algoritmo RR (cont.):**
  - Ejemplo:

Proceso	***Ráfaga CPU***
P1	53
P2	17
P3	68
P4	24

    - Con q=40
      - Diagrama de Gant para la planificación:



      - 5 cambios de contexto

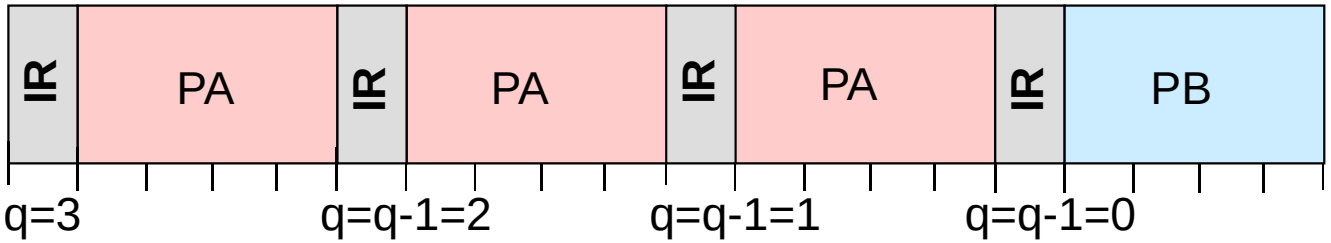
Algoritmos de planificación

- **Algoritmo RR (cont.):**
  - El rendimiento del algoritmo depende de *q*:
    - *q* grande → FIFO.
    - *q* pequeño → Muchos cambios de contexto y sobrecarga en la gestión de interrupciones de reloj.

    q debe ser grande respecto al tiempo de realizar un cambio de contexto.
  - Generalmente, tiempo de retorno mayor que con SRTF pero mejor tiempo de respuesta.

Algoritmos de planificación

- **Algoritmo RR (cont.):**
  - Una posible implementación es a través de la gestión de un contador dentro de la rutina de servicio de interrupción de reloj.



Proceso	***Ráfaga CPU***
PA	15
PB	6

- **IR:**
  - Rutina de servicio de interrupción de reloj
    q=q-1
    si (q==0) entonces
    q=q_0=3
    planificador
    finsi
  - Duración rutina de servicio: 1 ut.
- Frecuencia interrupción de reloj**:** Cada 5 ut.

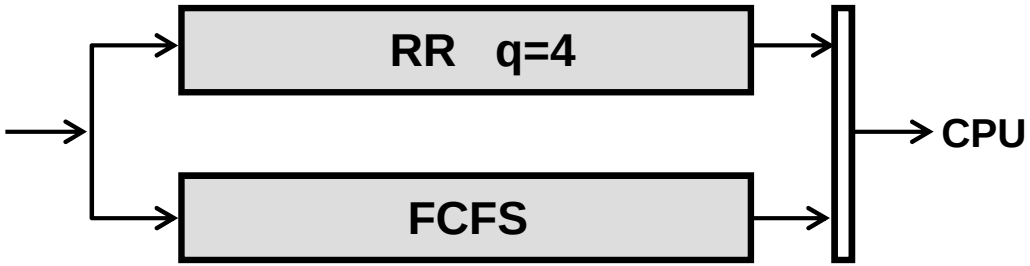
Como q=0:
- ✓ Proceso en ejecución a preparado.
- ✓ q=3.
- ✓ Elegir otro proceso para ejecución.

Algoritmos de planificación

- **Colas multinivel sin realimentación:**
  - Varias colas para los procesos listos.
  - Cada cola tiene su propio algoritmo de planificación.
  - Sistema de colas definido por:
    - Número de colas.
    - Algoritmo de planificación de cada cola.
    - Planificación entre colas.
      - Prioridad a cada cola.
      - Cuantum de CPU a cada cola, que se reparte entre los procesos de cada cola.

Algoritmos de planificación

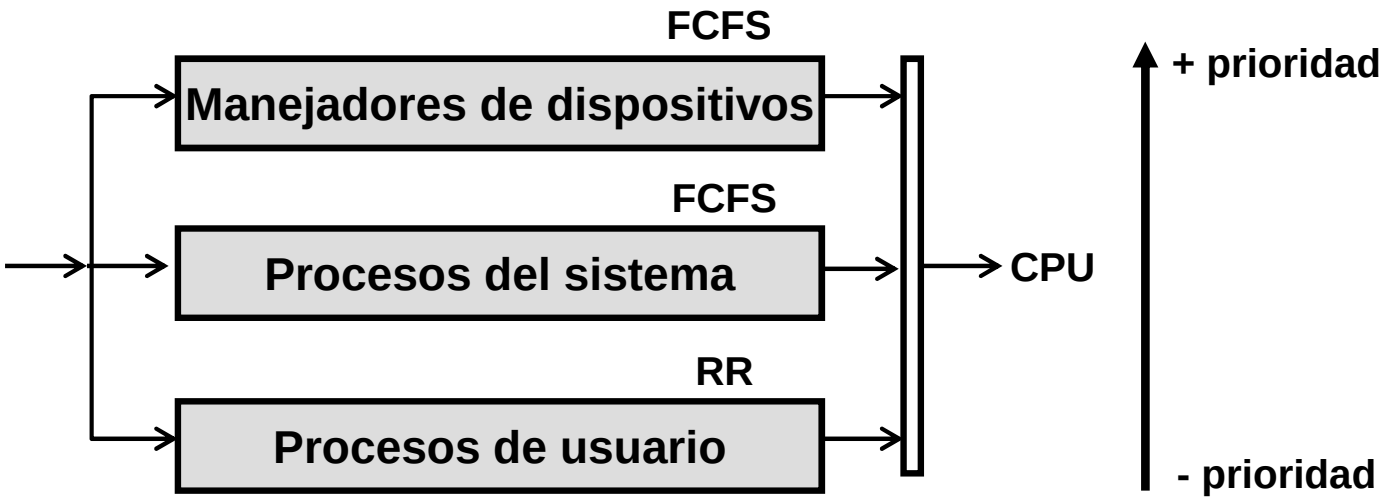
- **Colas multinivel sin realimentación (cont.):**
  - Ejemplo 1:



    - ♣ Cola de procesos en primer plano: RR con 80% de CPU.
    - ♣ Cola de procesos en segundo plano: FCFS con 20% de CPU.

Algoritmos de planificación

- **Colas multinivel sin realimentación (cont.):**
  - Ejemplo 2:
    - Planificador del SO minix.



Parcialmente expulsivo

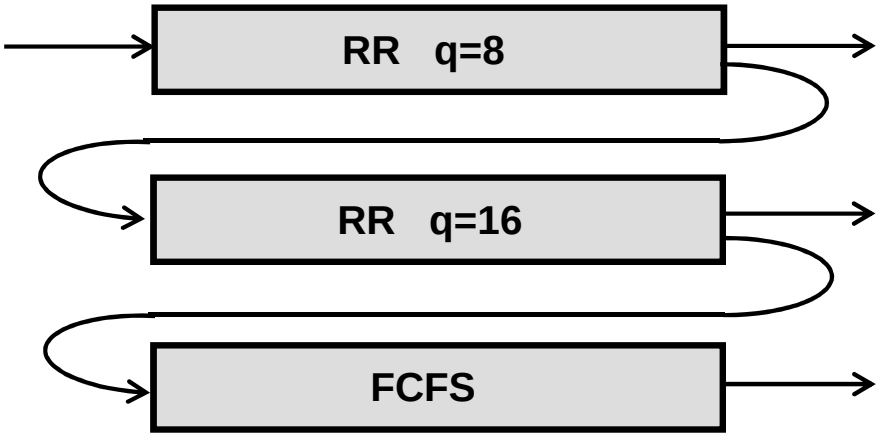
- Si llega un proceso del sistema o de dispositivo mientras se está ejecutando un proceso de usuario, este último es expulsado

Algoritmos de planificación

- **Colas multinivel con realimentación:**
  - Varias colas para los procesos listos.
  - Un proceso se mueve de una cola a otra.
  - Sistema de colas definido por:
    - Número de colas.
    - Algoritmo de planificación de cada cola.
    - Cola en la que entrará un proceso al llegar a preparado.
    - Planificación entre colas.
    - Método para determinar la realimentación:
      - Cuándo promover un proceso a una cola de mayor prioridad.
      - Cuándo degradar un proceso a una cola de menor prioirdad.

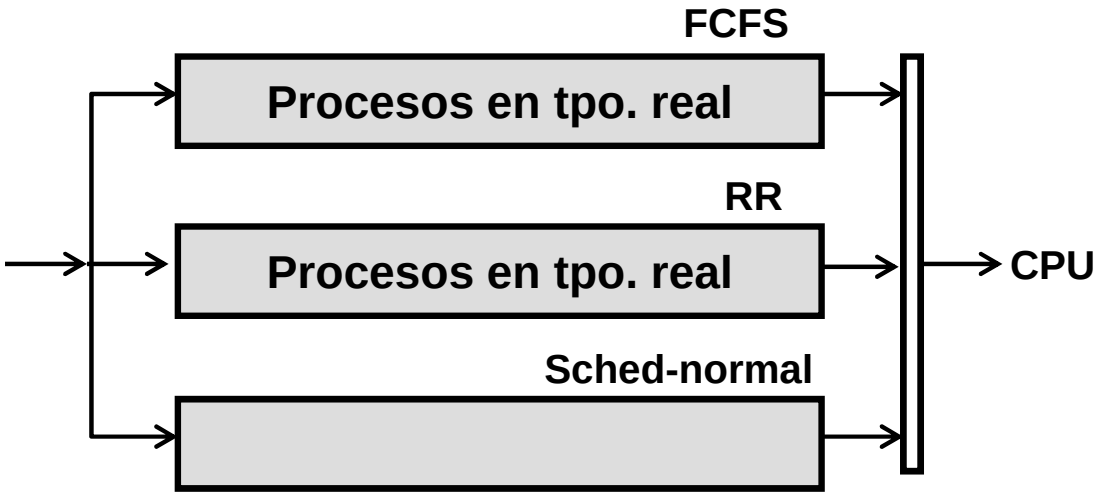
Algoritmos de planificación

- **Colas multinivel con realimentación (cont.):**
  - Ejemplo:



Algoritmos de planificación

- **Colas multinivel con y sin realimentación:**
  - Ejemplo:
    - Planificador simplificado del SO linux.



- Cada proceso tiene una prioridad
- Los procesos FCFS y RR tienen mayor prioridad
- Se selecciona el proceso con más prioridad de las tres colas
- No hay realimentación en los procesos FCFS y RR, en los otros sí

Algoritmos de planificación

- **Planificación de procesadores múltiples:**
  - Algoritmo de planificación complejo.
  - Sistema heterogéneo:
    - Cada procesador tiene su cola y algoritmo de planificación.
    - Los procesos han de ejecutarse en un procesador determinado.
  - Sistema homogéneo:
    - Cola común de procesos listos.

Algoritmos de planificación

- **Evaluación de algoritmos de planificación:**
  - *Modelado determinista*: Dada una carga, medir el rendimiento de cada algoritmo para dicha carga.
  - *Modelos de colas*: Estimar la distribución de probabilidad de ráfagas de CPU y de tiempos de llegada al sistema.
  - *Simulaciones*: Programar un modelo del sistema de computación.

Ejercicios

- **Ejercicio 1:**

  ¿Cuándo entra un proceso en estado zoombie?

  a) Cuando muere su padre y él no ha terminado todavía
  b) Cuando muere su padre sin haber hecho `wait` por él
  c) Cuando él muere y su padre no ha hecho `wait` por él
  d) Cuando él muere y su padre no ha terminado todavía

Ejercicios

- **Ejercicio 2:**

  ¿Cuál de las siguientes afirmaciones acerca de la llamada al sistema exec es falsa?

  a) Permite ejecutar mandatos con cualquier número de argumentos
  b) Si funciona bien, devuelve 0
  c) Puede cambiar el identificador efectivo de usuario
  d) Sólo retorna si va mal

Ejercicios

- **Ejercicio 3:**

  Si un proceso ejecuta el siguiente código:

```
if (fork() != 0) wait (&estado);
else execvp (...);
```

  ¿qué información comparten el proceso original (padre) y el proceso creado (hijo)?

Ejercicios

- **Ejercicio 4:**

  Indicar qué saldrá por pantalla tras ejecutar desde el shell el programa:

```
#include<unistd.h>
main()
{ printf("%d\n", getppid());
}
```

  a) El identificativo de proceso del shell desde el que ejecuto el programa

  b) El identificativo de proceso del programa que estoy ejecutando

  c) El identificativo del proceso init del sistema

  d) Un cero si la llamada getppid tiene éxito o –1 si no lo tiene

Ejercicios

- **Ejercicio 5:**

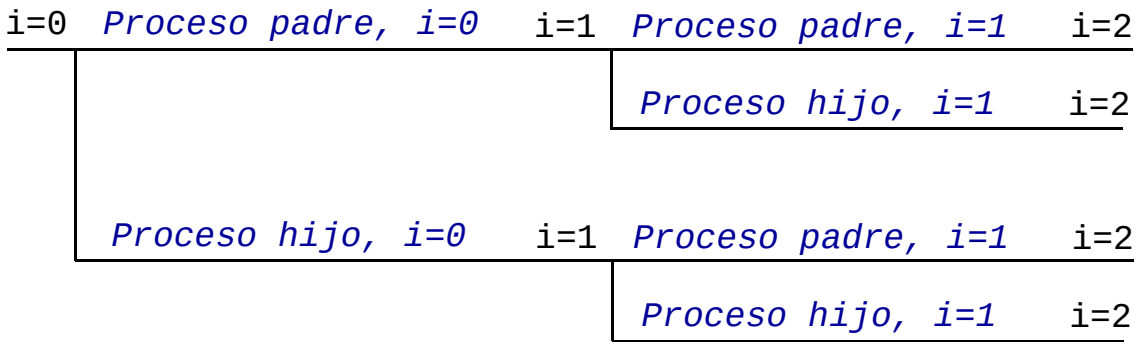
  ¿Cuántas veces saldrán por pantalla los mensajes `Proceso padre` y `Proceso hijo` al ejecutar el siguiente programa?

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

main()
{
  int i;

  for (i=0; i<2; i++)
     if (fork()!=0) printf("Proceso padre, i=%d\n",i);
     else           printf("Proceso hijo,  i=%d\n",i);
  exit(0);
}
```

Ejemplos

- **Ejercicio 5 (sol.):**

  3 veces `Proceso padre` y otras 3 `Proceso hijo`.



  Una posible salida podría ser:

```
Proceso hijo , i=0
Proceso padre, i=0
Proceso hijo , i=1
Proceso padre, i=1
Proceso hijo , i=1
Proceso padre, i=1
```

  ¿Por qué? ¿Salida única?

Ejemplos

- **Ejercicio 5 (cont.):**

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

main()
{
  int i;

  for (i=0; i<2; i++)
     if (fork()!=0)
     {              wait(NULL);
                    printf("Proceso padre, i=%d\n",i);
     }
     else           printf("Proceso hijo,  i=%d\n",i);
  exit(0);
}
```

Salida única

```
Proceso hijo , i=0
Proceso hijo , i=1
Proceso padre, i=1
Proceso padre, i=0
Proceso hijo , i=1
Proceso padre, i=1
```

Ejercicios

- **Ejercicio 6:**

  ¿Cuál es la salida del siguiente programa?

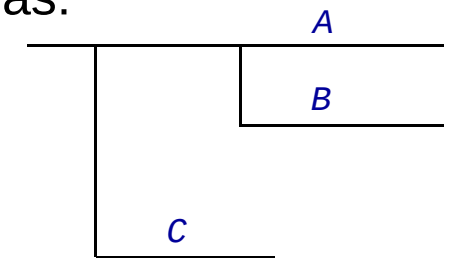
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

main()
{ if (fork()!=0){
     if (fork()!=0){
        wait (NULL);
        wait (NULL);
        printf("A");
     } else
        printf("B");
   } else
        printf("C");
  exit(0);
}
```

Ejercicios

- **Ejercicio 6 (sol.):**

  Dos posibles salidas:

  CBA
  BCA



  Para que la salida fuese única habría que poner uno de los `wait` en el código del proceso padre antes de que este crease el segundo hijo:

```
main()
{ if (fork()!=0){
      wait (NULL);
      if (fork()!=0){
         wait (NULL);
         printf("A");
      } else
         printf("B");
    } else
         printf("C");
  exit(0);
}
```

Ejercicios

- **Ejercicio 7:**

  Analiza este programa, muestra la jerarquía de procesos que se genera y lo que se imprime por pantalla tras su ejecución. Asigna un identificador a cada proceso.

```
// OJO! Faltan los includes
int main(){
int i, contador=1;

for (i=0;i<4;i++){
  if (fork() == 0){
    contador = contador * i;
    printf("Proceso=%d, iteracion=%d, contador=%d\n", getpid(), i, contador);
    exit(0);
  } else {
    contador = contador + i;
    wait(NULL);
  }
}
printf("Proceso=%d, iteracion=%d, contador=%d\n", getpid(), i,contador);
exit(0);
}
```

Ejercicios

- **Ejercicio 7 (sol.):**

  El proceso que ejecuta el programa crea cuatro procesos hijos en el bucle `for`. Pero no crea otro hijo hasta que el anterior haya finalizado. Cada hijo actualiza su variable `contador` multiplicando el valor que tenían esta y la variable `i` cuando el padre lo creó. Después, el hijo muestra por pantalla su identificativo de usuario y el valor de sus variables `i` y `contador` y finaliza. El padre, una vez que crea un hijo, incrementa su variable `contador` en tantas unidades como valga su variable `i`. Luego, espera a que el último hijo que ha creado finalice y después pasa a la siguiente iteración del bucle `for`. Cuando han finalizado los cuatro hijos, el padre sale del bucle `for`, muestra por pantalla su identificativo de usuario y el valor de sus variables `i` y `contador` y finaliza.

  Cabe recordar que, cuando se crea un hijo, las variables que utiliza este tienen en ese instante el mismo valor que tenían en el padre. Pero padre e hijo no comparten esas variables, cada uno tiene las suyas en un espacio de direcciones diferente. Y, por lo tanto, a partir de ese instante, **los cambios que realice uno en sus variables no afectan a las del otro**.

Ejercicios

- **Ejercicio 7 (sol.):**

  También matizamos que, a diferencia de lo que ocurre en el lenguaje Java, cuando finaliza un bucle `for` en C, el índice utilizado por este en las diferentes iteraciones no pierde su valor. Por tanto, en nuestro programa la variable `i` vale 4 cuando el proceso padre ejecuta la función `printf` que precede a la función `exit`.

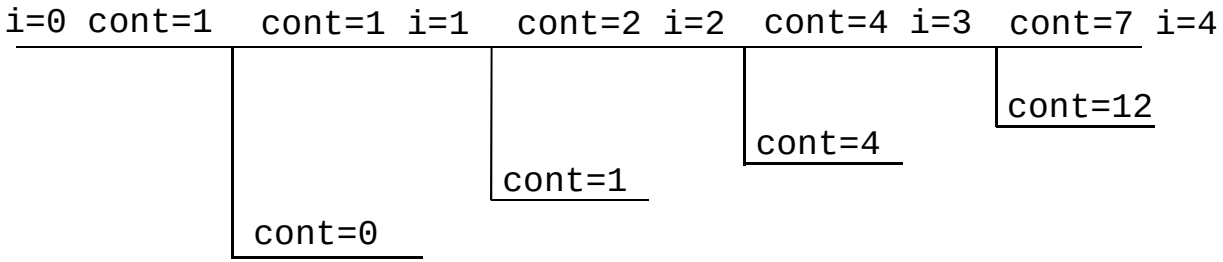
  Por tanto, si asumimos que el identificador del proceso padre es 5311 y el de los cuatro hijos que crea 5312, 5313, 5314 y 5315, la única posible salida de la ejecución del programa sería:

```
proceso=5312, iteracion=0, contador=0
proceso=5313, iteracion=1, contador=1
proceso=5314, iteracion=2, contador=4
proceso=5315, iteracion=3, contador=12
proceso=5311, iteracion=4, contador=7
```

Ejercicios

- **Ejercicio 7 (sol.):**

```
proceso=5312, iteracion=0, contador=0
proceso=5313, iteracion=1, contador=1
proceso=5314, iteracion=2, contador=4
proceso=5315, iteracion=3, contador=12
proceso=5311, iteracion=4, contador=7
```



Ejercicios

- **Ejercicio 8:**

  ¿Cuál es la salida del siguiente programa?

```
#include <unistd.h>
main()
{
  if (fork()!=0){
    wait (NULL);
    execlp(“ls”,”ls”,NULL);
    perror("ls");
    exit (-1);
  } else printf(“Escribe el proceso hijo\n”);
  printf(“Escribe alguien\n”);
  exit(0);
}
```

Ejercicios

- **Ejercicio 8 (sol.):**

  Salida única.

  Una posible salida:

```
Escribe el proceso hijo
Escribe alguien
ejercicio5.c  ejercicio6.c  ejercicio7.c
```

Ejercicios

- **Ejercicio 9:**

  Mostrar un gráfico que ilustre la jerarquía de procesos creada con el siguiente código:

```
#include<stdio.h>
main()
{
  int i;

  for (i=0; i<4; i++)
     if (fork()!=0) break;
  printf("PID=%d  PPID=%d\n",getpid(),getppid());
  exit(0);
}
```

  ¿En qué orden se escribirán los mensajes y acabarán las ejecuciones de los distintos procesos?

Ejercicios

- **Ejercicio 9 (cont.):**

  Explicar qué ha ocurrido en la ejecución del programa anterior si la salida de la ejecución es la siguiente:

```
PID=29591  PPID=29590
PID=29590  PPID=11653
$ PID=29592  PPID=1
PID=29593  PPID=29592
PID=29594  PPID=29593
```

Ejercicios

- **Ejercicio 9 (cont.):**

  ¿Qué diferencia hay entre este programa y el anterior?

```
#include<stdio.h>
main()
{
int i;

for (i=0; i<4; i++)
  if (fork()!=0) {wait(NULL); break;}
printf("PID=%d  PPID=%d\n",getpid(),getppid());
exit(0);
}
```

Ejercicios

- **Ejercicio 9 (cont.):**

  Una posible salida de la ejecución del programa anterior:

```
PID=481  PPID=480
PID=480  PPID=479
PID=479  PPID=478
PID=478  PPID=477
PID=477  PPID=11653
```

Ejercicios

- **Ejercicio 10:**

  Realizar un programa que cree tres procesos con una jerarquía del tipo abuelo-padre-hijo. El hijo ejecutará el comando `ls -l`. El padre, el comando `echo HOLA`. Y el abuelo, el comando `grep A file`, siendo `file` un fichero que se pasa al programa como argumento.

  En primer lugar se mostrará la salida del comando que ejecuta el hijo, después la del padre y, por último, la del proceso abuelo.

Ejercicios

- **Ejercicio 10 (sol.):**

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
if (argc != 2) {printf("Número de parámetos incorrecto\n"); exit(-1);}

if (fork()!=0)
{ /* P1 */
    wait(NULL);
    printf("*** EJECUCION DE P1 ***\n");
    execlp("grep","grep","A",argv[1],NULL);
    perror("Error en exec de P1");
    exit (-1);
} else
```

Ejercicios

- **Ejercicio 10 (sol.):**

```
{
    if (fork()!=0)
    { /* P2 */
        wait(NULL);
        printf("*** EJECUCION DE P2 ***\n");
        execlp("echo","echo","HOLA",NULL);
        perror("Error en exec de P2");
        exit (-1);
    } else {
          /* P3 */
        printf("*** EJECUCION DE P3 ***\n");
        execlp("ls","ls","-l",NULL);
        perror("Error en exec de P3");
        exit (-1);
    }
}
}
```

Ejercicios

- **Ejercicio 11:**

  Indicar cuál de los siguientes atributos es compartido entre todos los hilos de un mismo proceso:

  a) La pila de ejecución

  b) El estado de ejecución

  c) El contexto de ejecución donde salvar el contexto del hilo cuando éste no esté en ejecución

  d) El código ejecutable

Ejercicios

- **Ejercicio 12:**

  Cuando se crea un nuevo hilo, éste tiene acceso a:

  a) Una copia del segmento de datos del proceso
  b) No tiene acceso al segmento de datos del proceso
  c) El segmento de datos del proceso
  d) El segmento de datos del proceso pero marcándolo previamente como “copy-on-write”

Ejercicios

- **Ejercicio 13:**

  Implementa un programa que calcule la suma de los elementos de un vector de MAX números enteros aleatorios.

  El programa debe crear tantos hilos como se le indique en el primer argumento. Y cada hilo ha de realizar la suma de un fragmento consecutivo del vector de un tamaño aproximadamente igual para todos los hilos. Así, si denotamos por N al número total de hilos a crear, el primero de ellos sumará las posiciones 0,1,...,(MAX div N)-1, el segundo (MAX div N), (MAX div N)+1,...,2*(MAX div N)-1 y así sucesivamente.

  El hilo principal esperará a que acaben todos los hilos, calculará la suma final a partir de las sumas parciales ya obtenidas y mostrará el resultado por pantalla.

  Utiliza un vector accesible por todos los hilos que almacene las sumas parciales calculadas por estos. Así, el hilo 0 guardará su suma en la componente 0 del vector, el hilo 1 en la componente 1 y así sucesivamente.

Ejercicios

- **Ejercicio 14:**

  Modifica el ejercicio anterior para que los hilos devuelvan al hilo principal como código de terminación (esto es, en `pthread_exit`) la suma parcial que han calculado.

Planificación de procesos: Ejercicios

- **Ejercicio 15:**

  Si para el proceso que se modeliza según el siguiente esquema

CPU (2 udt)	E/S (3 udt)	CPU (3 udt)	E/S (2 udt)	CPU (2 udt)

se sabe que su tiempo de espera ha sido de 8 udt. y que el tiempo total necesario para su ejecución ha sido de 25 udt., ¿cuánto tiempo ha estado dicho proceso bloqueado esperando para realizar alguna de las operaciones de E/S?

Planificación de procesos: Ejercicios

- **Ejercicio 16:**

  Un sistema se compone de los siguiente recursos:

  - una CPU
  - tres unidades de almacenamiento masivo (UAM1, UAM2 y UAM3).

  Existen dos tipos de procesos según las necesidades de utilización de los diferentes recursos:

Procesos Tipo_1	**Procesos Tipo_2**
1 ut para CPU	6 ut para CPU
3 ut para UAM1	1 ut para UAM1
2 ut para CPU	3 ut para CPU
6 ut para UAM3	2 ut para UAM2
1 ut para CPU	1 ut para CPU
	1 ut para UAM3
	2 ut para CPU

Planificación de procesos: Ejercicios

- **Ejercicio 16 (cont. enunciado):**

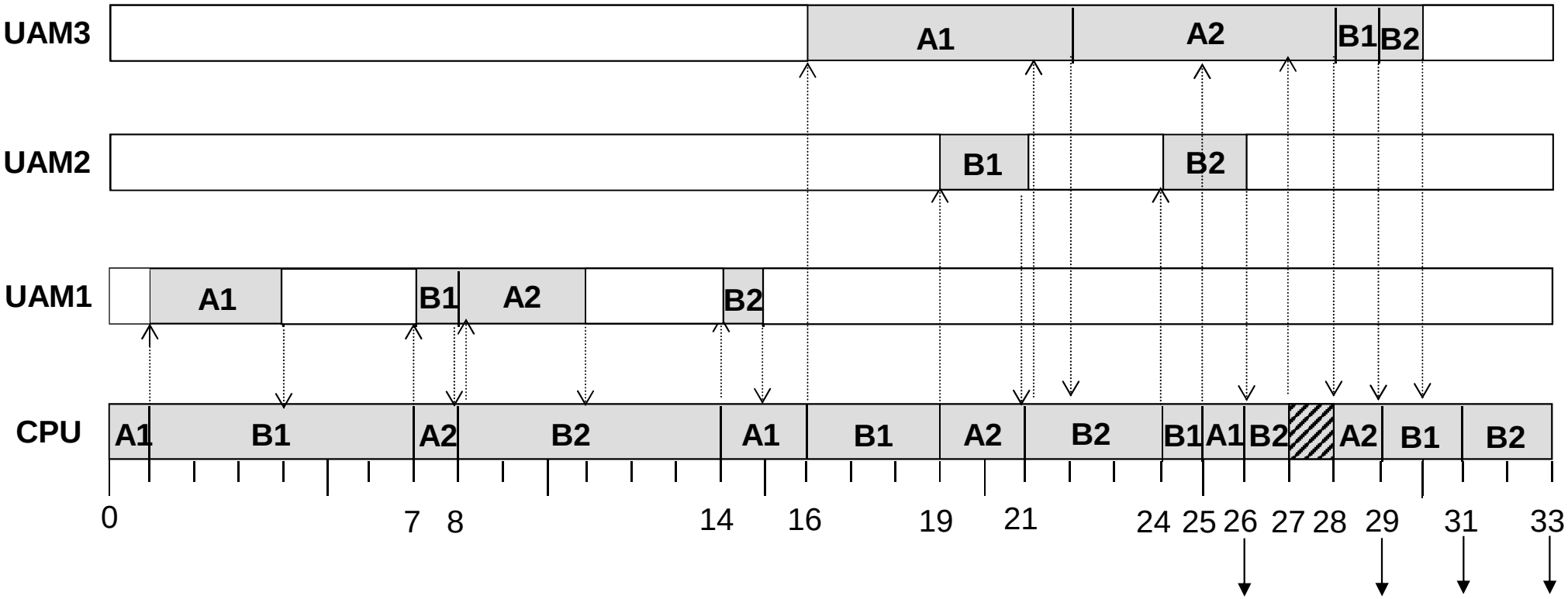
  Suponiendo que en el sistema existen dos trabajos del Tipo_1 (A1 y A2) y dos trabajos del Tipo_2 (B1 y B2), y que han llegado a preparados en el orden A1, B1, A2, B2, se debe calcular el tiempo de permanencia en el sistema, tiempo de espera medio y la utilización de la CPU, según el algoritmo de planificación de la CPU sea:

  - FCFS.
  - Prioridad expulsiva (con mayor prioridad los procesos de tipo 1).
  - SJF.
  - SRJF.
  - Round-Robin con quantum q=2 ut.

  Se supone para todos los casos que en el resto de colas del sistema se rige mediante la estrategia de planificación FCFS.

Planificación de procesos: Ejercicios

- **Ejercicio 16 (solución):**
  - Planificación FCFS:

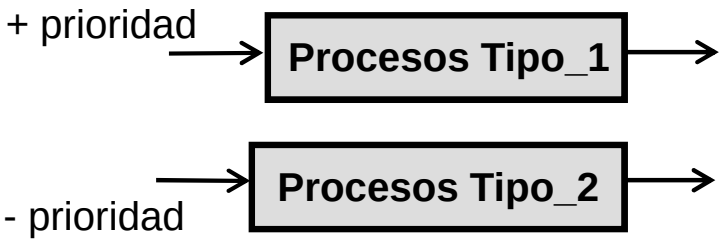


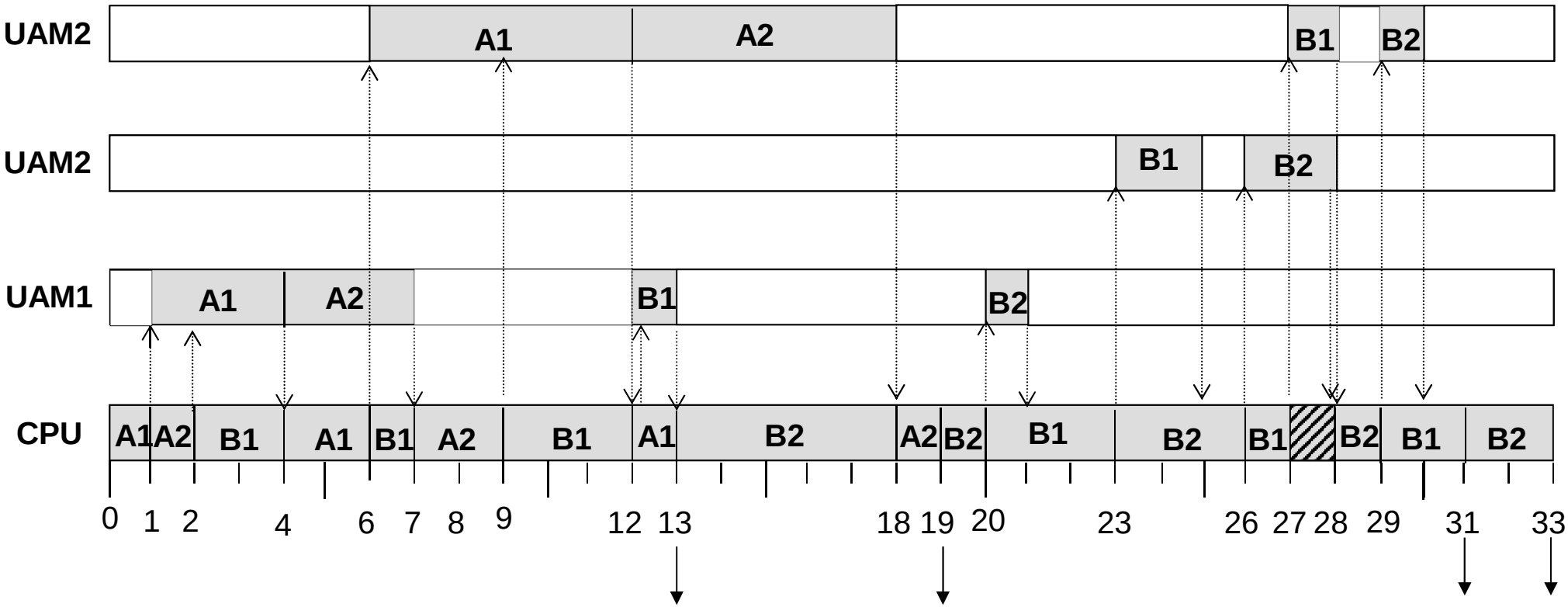
Planificación de procesos: Ejercicios

- **Ejercicio 16 (solución):**
  - Planificación FCFS (cont.):
    - Tiempos de permanencia (TP):
      - TPA1=26 ut. TPA2=29 ut. TPB1=31 ut. TPB2=33 ut.
    - Tiempos de espera (TE):
      - TEA1=0+(14-4)+(25-22)=13 ut.
      - TEA2=7+(19-11)+0=15 ut.
      - TEB1=1+(16-8)+(24-21)+0=12 ut.
      - TEB2=8+(21-15)+0+(31-30)=15 ut.
    - Tiempo medio de espera: (13+15+12+15)/4= 13,75 ut.
    - Utilización de CPU: (33-1)/33*100= 96,96%.

Planificación de procesos: Ejercicios

- **Ejercicio 16 (solución):**
  - Planificación con prioridad expulsiva:





Planificación de procesos: Ejercicios

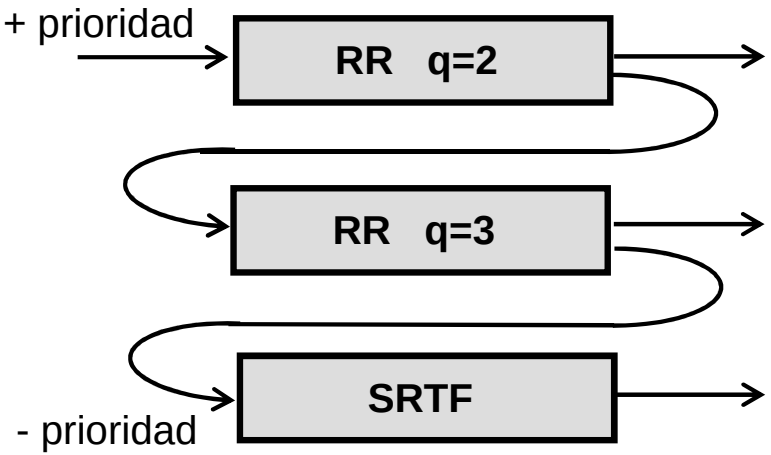
- **Ejercicio 16 (solución):**
  - Planificación con prioridad expulsiva (cont.):
    - Tiempos de permanencia (TP):
      - TPA1=13 ut. TPA2=19 ut. TPB1=31 ut. TPB2=33 ut.
    - Tiempos de espera (TE):
      - TEA1=0 ut.
      - TEA2=1 ut.
      - TEB1=2+2+2+7+1+1=15 ut.
      - TEB2=13+1+2+1=17 ut.
    - Tiempo medio de espera: (0+1+15+17)/4= 8,25 ut.
    - Utilización de CPU: (33-1)/33*100= 96,96%.

Planificación de procesos: Ejercicios

- **Ejercicio 17:**

  Se tienen los siguientes datos de un conjunto de procesos:

Proceso	T. ciclo de CPU	T. llegada a preparado
A	7	0
B	8	3
C	8	4
D	5	11
E	6	12
F	2	20



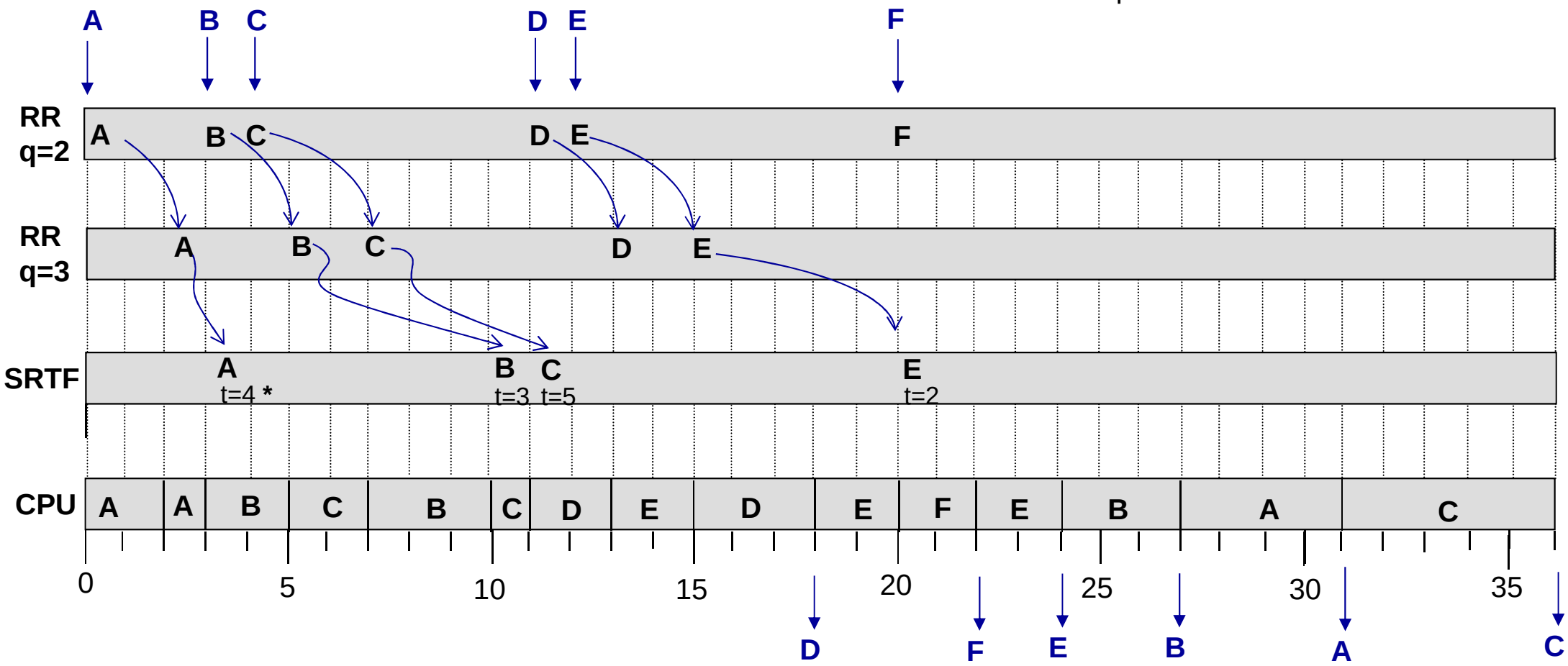
El sistema operativo utiliza un algoritmo de planificación con 3 colas multinivel con prioridad expulsiva con las características indicadas en el esquema de arriba.

Obténgase el diagrama de ocupación de la CPU.

Planificación de procesos: Ejercicios

- **Ejercicio 17 (solución):**

* tiempo restante de ciclo de CPU



Planificación de procesos: Ejercicios

- **Ejercicio 17 (solución):**

  Evolución de los tiempos restantes de la ráfaga de CPU de los procesos listos:

Proceso	T. restante de ciclo de CPU
A	~~7~~ ~~5~~ ~~4~~ 0
B	~~8~~ ~~6~~ ~~3~~ 0
C	~~8~~ ~~6~~ ~~5~~ 0
D	~~5~~ ~~3~~ 0
E	~~6~~ ~~4~~ ~~2~~ 0
F	~~2~~ 0

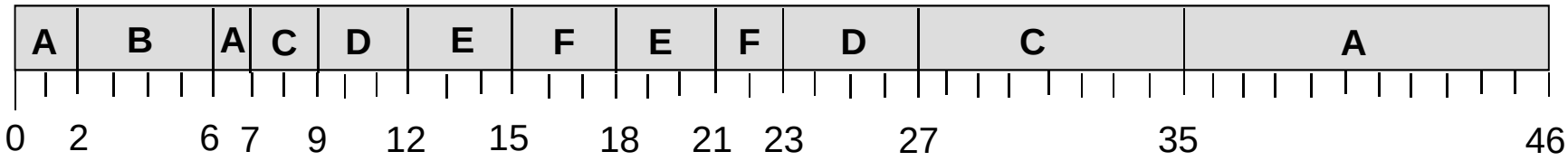
Planificación de procesos: Ejercicios

- **Ejercicio 18:**

Se tienen los siguientes datos de un conjunto de procesos:

Proceso	T. ciclo de CPU	T. llegada a preparado
A	14	0
B	4	2
C	10	7
D	7	9
E	6	12
F	5	15

Al aplicarse un determinado algoritmo de planificación se obtiene la siguiente ocupación de CPU:

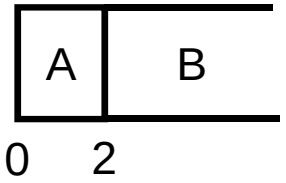


Indíquese un posible algoritmo de planificación que lleve a la ocupación mostrada.

Planificación de procesos: Ejercicios

- **Ejercicio 18 (solución):**

Instante 2:



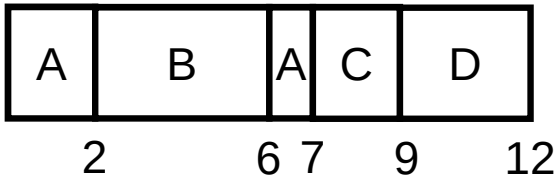
Se produce una expulsión de la CPU del proceso A.

Esto se puede deber a:

- Algoritmo de planificación RR con cuantum = 2 udt.
- Algoritmo SRTF, ya que el ciclo de CPU del proceso B tiene una estimación menor (4ut) que lo que le resta al proceso A (14-2=12ut).
- Algoritmo de prioridades expulsivas generales, siendo el proceso B más prioritatio que el A.

Planificación de procesos: Ejercicios

- **Ejercicio 18 (solución):**



Instante 6:

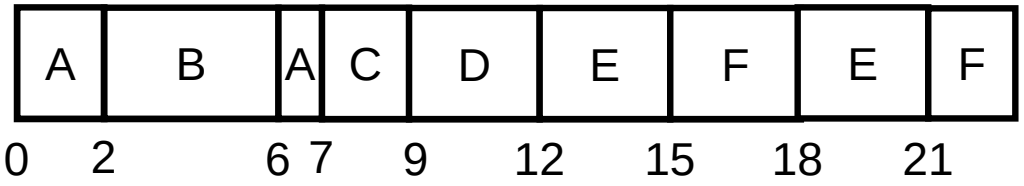
Finaliza el ciclo de CPU del proceso B; esto elimina la planificación RR de las posibilidades anteriores. Continúa su ejecución el proceso A.

Instantes 7 y 9:

Situación análoga a las anteriores.

Planificación de procesos: Ejercicios

- **Ejercicio 18 (solución):**

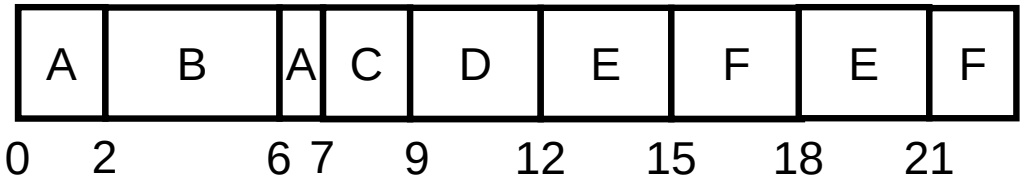


Instante 12:

Este hecho indica que entre el proceso D y E no se está siguiendo una planificación SRTF ya, que a pesar de que el proceso E tiene estimado un ciclo de CPU mayor se expulsa al proceso D. Las únicas alternativas que quedan entre estos dos proceso es un algoritmo de prioridades expulsivas, donde el proceso E sea más prioritario que el D o bien un algoritmo RR con cuantum 3 ut.

Planificación de procesos: Ejercicios

- **Ejercicio 18 (solución):**



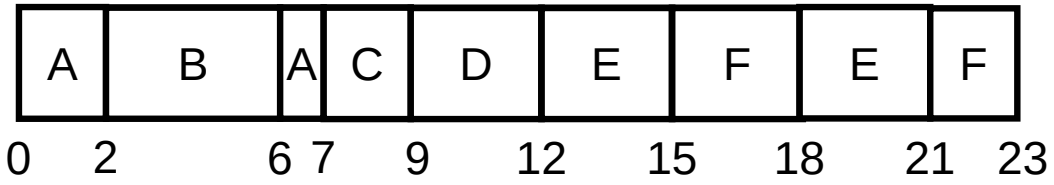
Instante 15:

Se tiene un hecho similar al que se producía entre el proceso E y el D en el instante 12. En este caso todavía no se puede asegurar que sea un algoritmo de prioridades generales.

En este instante se descarta el RR entre los procesos D, E y F, ya que si fuese así, el proceso D debería pasar en este momento a ejecución.

Planificación de procesos: Ejercicios

- **Ejercicio 18 (solución):**



Instante 18:

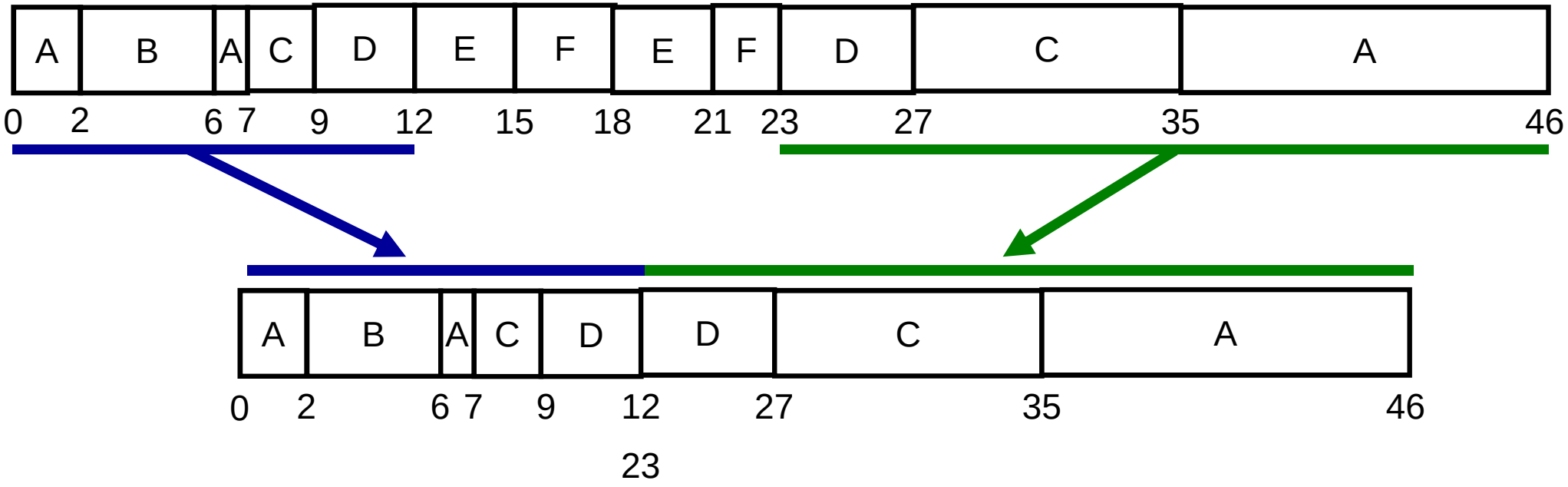
En este instante se ve claramente que entre los procesos E y F se sigue una estrategia de planificación RR con cuantum igual a 3 unidades de tiempo. En este instante no se produce ningún hecho remarcable, pero sin embargo el proceso E es expulsado de la CPU, signando ésta al proceso F. Esta situación sólo se da en el algoritmo RR.

Hasta el instante 23 se muestra claramente un algoritmo RR entre E y F.

Planificación de procesos: Ejercicios

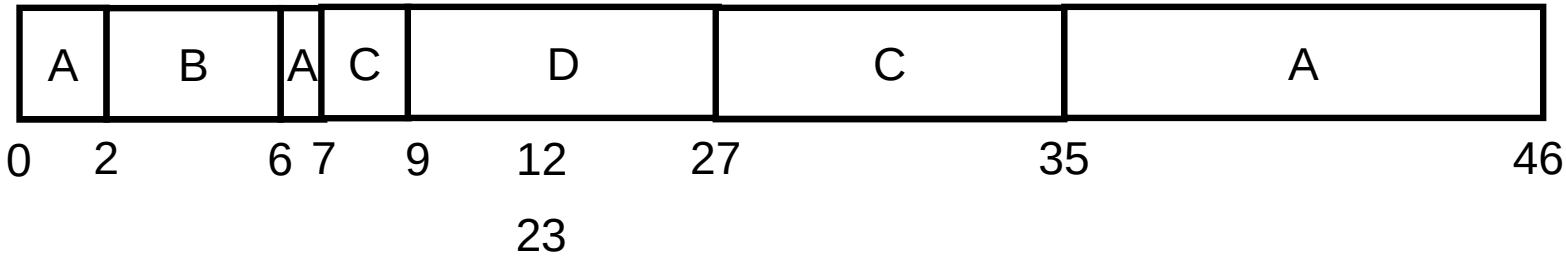
- **Ejercicio 18 (solución):**

  Ya se ha indicado que entre el instante 12 y el 23 se muestra un algoritmo RR con cuantum 3 ut entre los proceso E y F. Si se elimina este intervalo de tiempo de la representación se tiene:



Planificación de procesos: Ejercicios

- **Ejercicio 18 (solución):**



Esta figura representa claramente un estrategia SRTF entre los procesos A, B, C y D

Planificación de procesos: Ejercicios

- **Ejercicio 18 (solución):**

  Por tanto, la estrategia de planificación que se ha seguido es:

  Colas multinivel:

  - Número de colas 2 (C1, C2).
  - Planificación en las colas:
    - cola C1 RR con cuantum 3 ut.
    - cola C2 SRTF.
  - Planificación entre colas: Prioridades expulsivas, siendo C1 la más prioritaria.
  - Realimentación: NO
  - Cola de entrada:
    - C1 para los procesos E y F
    - C2 para los procesos A, B, C y D.

Planificación de procesos: Ejercicios

- **Ejercicio 19:**

  Suponiendo que en el instante 0 existen dos procesos preparados cuya modelización consta de un único ciclo de CPU de 10 udt., ¿cuál es el tiempo de espera medio si el algoritmo de planificación es RR con quantum 6 udt.?

Planificación de procesos: Ejercicios

- **Ejercicio 20:**

  La planificación de procesador de un SO sigue una política RR en la que el quantum asignado a un proceso finaliza cuando se produce la 3ª interrupción de reloj desde que un proceso ha pasado a estado de ejecución.

  La rutina de servicio de interrupción de reloj consume 1 udt. de procesador.

  Cada vez que se inicia un quantum se programa el reloj del sistema para que genere una interrupción cada 4 udt. (en ese tiempo no se contabiliza el que se utiliza para ejecutar rutinas de servicio de interrupción).

  Hay un único proceso en el sistema que pasa a ejecución en el instante 0 (justo al finalizar una rutina de servicio de interrupción de reloj) y cuyo ciclo (ráfaga) de CPU tiene una duración de 8 udt.,

Planificación de procesos: Ejercicios

- **Ejercicio 20 (cont. enunciado):**

  ¿Cuál de las siguientes afirmaciones es cierta?

  (a) El ciclo de CPU del proceso ocupa 2 quantums enteros.

  (b) Durante la ejecución de esta ráfaga se producen 3 interrupciones de reloj.

  (c) En el instante 10 el proceso realiza una llamada al SO.

  (d) El ciclo de CPU del proceso finaliza en el instante 9.

Planificación de procesos: Ejercicios

- **Ejercicio 21:**

Sea el siguiente fragmento de programa:

```
if (fork() != 0) wait(&estado);
        else write (fd,&dato,sizeof(dato));
exit (0);
```

i) ¿Cuántas llamadas al SO debidas a la ejecución del proceso padre y cuántas debidas a la ejecución del proceso hijo se producirán hasta la finalización de los mismos? Asumir que durante su ejecución no se producirá ningún error.

Planificación de procesos: Ejercicios

- **Ejercicio 21 (cont. enunciado):**

  ii) En dicho código las operaciones de E/S se gestionan con un único acceso al dispositivo de E/S de coste 5 udt. Las funciones de llamada al SO se modelizan con un ciclo de CPU de 4 udt., en el que en el instante 3 se realiza la llamada al SO. Las rutinas de servicio de interrupción SW tienen un coste de 2 udt. y las de servicio de interrupción HW (más prioritarias que las anteriores) un coste de 1 udt. Suponiendo una política de planificación SRTF, ¿cuál de las siguientes afirmaciones es cierta?

  (a) En el instante 11 el hijo invoca la llamada al sistema `write`.

  (b) En el instante 5 el padre invoca la llamada al sistema `wait`.

  (c) En el instante 18 los dos procesos están bloqueados.

  (d) En el instante 21 está bloqueado el padre y en ejecución el hijo.

Planificación de procesos: Ejercicios

- **Ejercicio 22:**

  Colas multinivel:

  - Número de colas 2 (C1, C2).
  - Planificación en las colas:
    - Cola C1: RR; el cuantum finaliza en la 2ª interrupción de reloj (IR) desde que el proceso está en ejecución.
    - Cola C2: SJF.
  - Planificación entre colas: Prioridades expulsivas, siendo C1 la más prioritaria.
  - Realimentación:
    - 1ª transición Ejecución → Preparado: Al final de C1.
    - 2ª, 3ª,... transición Ejecución → Preparado: A C2
  - Cola de entrada: C1.

Planificación de procesos: Ejercicios

- **Ejercicio 22:**

  Colas multinivel:

  - Número de colas 2 (C1, C2).
  - Planificación en las colas:
    - Cola C1: RR; el cuantum finaliza en la 2ª interrupción de reloj (IR) desde que el proceso está en ejecución.
    - Cola C2: SJF.
  - Planificación entre colas: Prioridades expulsivas, siendo C1 la más prioritaria.
  - Realimentación:
    - 1ª transición Ejecución → Preparado: Al final de C1.
    - 2ª, 3ª,... transición Ejecución → Preparado: A C2
  - Cola de entrada: C1.

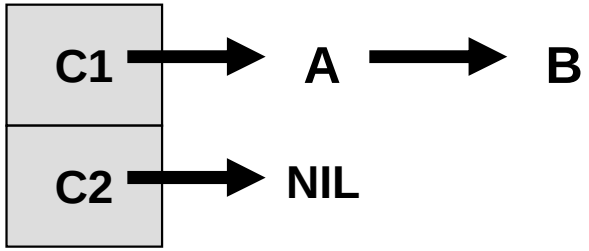
Planificación de procesos: Ejercicios

- **Ejercicio 22 (cont.):**

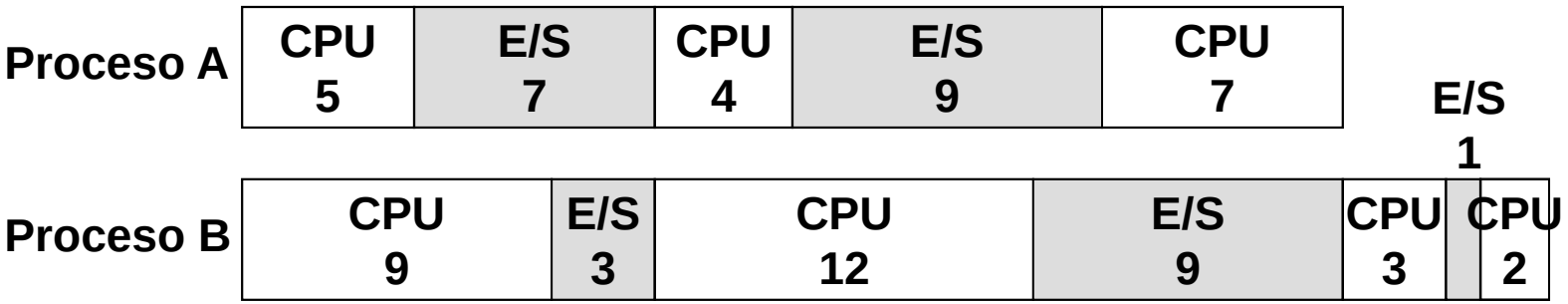
Funcionamiento del sistema:

- Cada 4ut se produce una IR
- Rutinas de servicio de interrupción SW 2 ut.
- Rutinas de servicio de interrupción HW 1 ut.
- Niveles de prioridad en las interrupciones.
  - Nivel 0 (más prioritarias) IH (no reloj)
  - Nivel 1 IR
  - Nivel 2 (menos prioritarias) IS

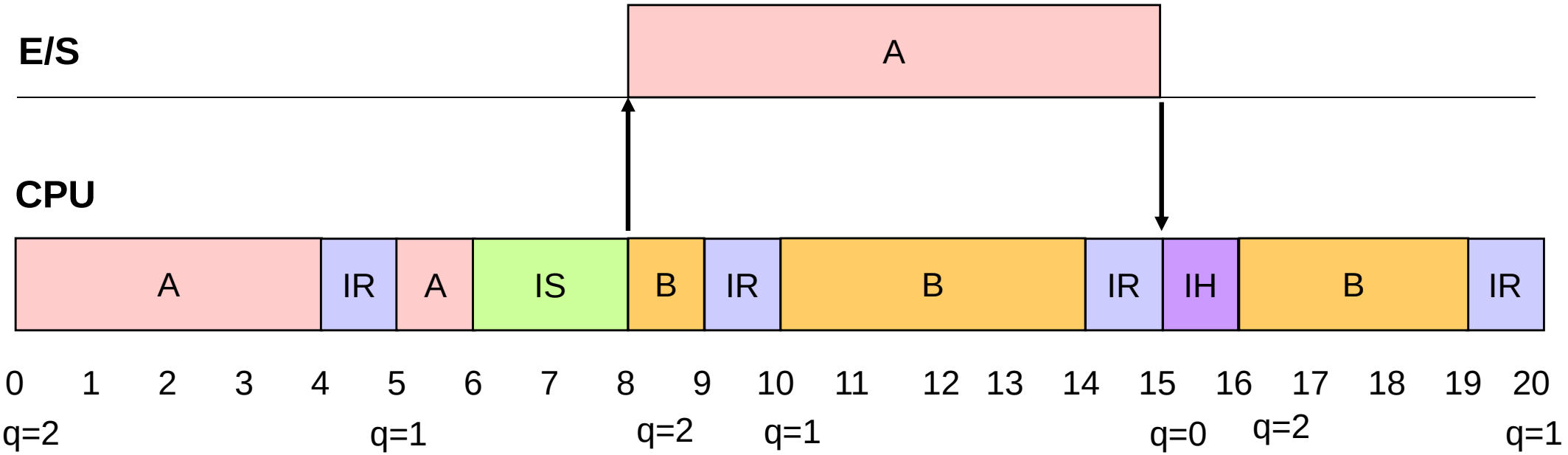
Situación inicial



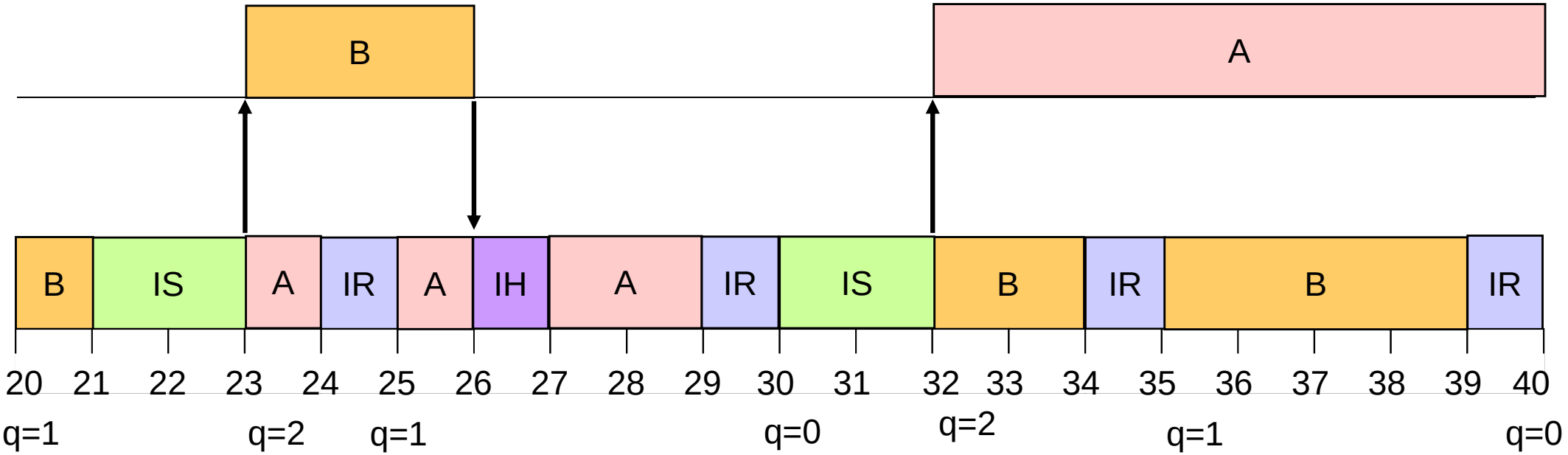
Modelos de los procesos:



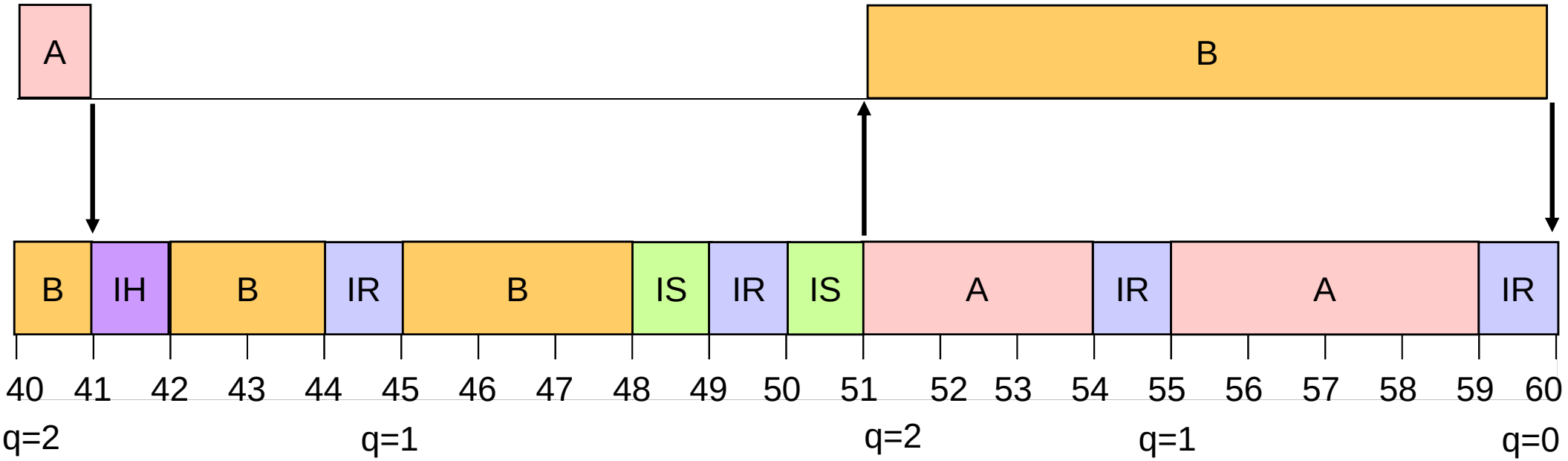
Planificación de procesos: Ejercicios



Planificación de procesos: Ejercicios



Planificación de procesos: Ejercicios



Planificación de procesos: Ejercicios