Tema 4

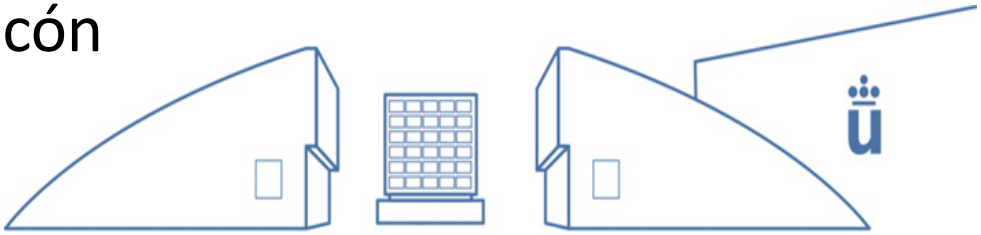
Gestión de procesos

Alberto Sánchez

Sofia Bayona

Luis Rincón

Universidad Rey Juan Carlos



Bibliografía

- **“Fundamentos de sistemas operativos”** *Silberschatz, Galvin, Gagne*. Editorial Mc Graw Hill.
  - Capítulo 3: 3.1, 3.2, 3.3, 3.4
  - Capítulo 4: 4.1, 4.2
  - Capítulo 5: 5.1, 5.2, 5.3
  - Capítulo 6: 6.1, 6.2, 6.3, 6.4, 6.5, 6.6
- **“Sistemas Operativos: una visión aplicada” 2ª ed.** *Carretero, García, De Miguel, Pérez.* Editorial Mc Graw Hill.
  - Capítulo 3: 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.9, 3.12, 3.13
  - Capítulo 4: 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8
  - Capítulo 6: 6.1, 6.2, 6.3, 6.6

Índice

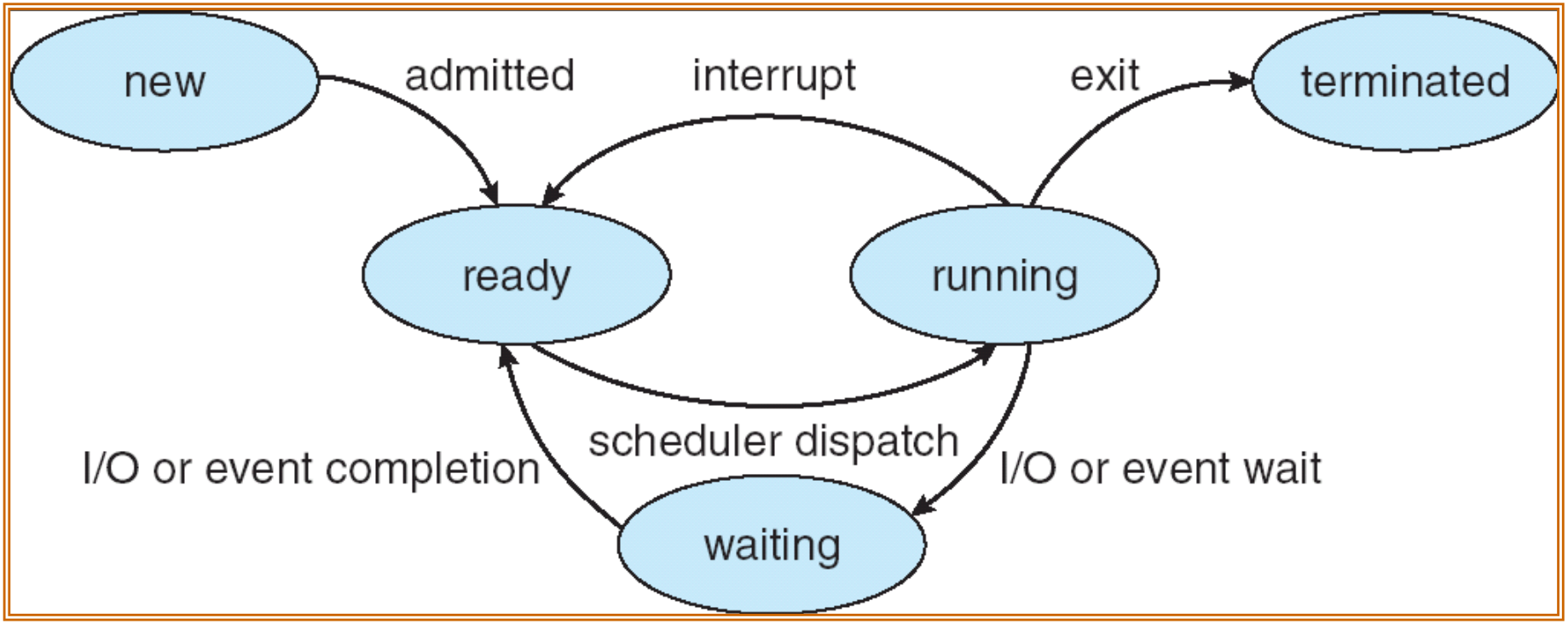
Concepto de proceso

- Un proceso es un programa en ejecución.
  - Puede haber varias réplicas de un mismo programa en ejecución.
  - Los procesos son independientes, aunque sean instancias de un mismo programa.
- Un proceso no es solamente el código del programa, también incluye la actividad actual como:
  - Contador de programa
  - Registros del procesador
  - Imagen de memoria

Estado del proceso

- Un proceso puede ir cambiando entre varios estados:
  - **Nuevo**: el proceso está siendo creado.
  - **Listo**: está a la espera de que le asignen un procesador.
  - **En ejecución**: se están ejecutando instrucciones del proceso.
  - **Bloqueado**: esperando que ocurra algún suceso (E/S, comunicación, sincronización . . .) que lo desbloquee.
  - **Terminado**.
- Otros estados:
  - **Preparado**: el proceso está a la espera para ser ejecutado cuando tenga suficientes recursos (ejemplo: ejecución por lotes).
  - **Suspendido**: no puede pasar a ejecución porque reside enteramente en la memoria de intercambio.
    - Se suspenden procesos para reducir el grado de multiprogramación.

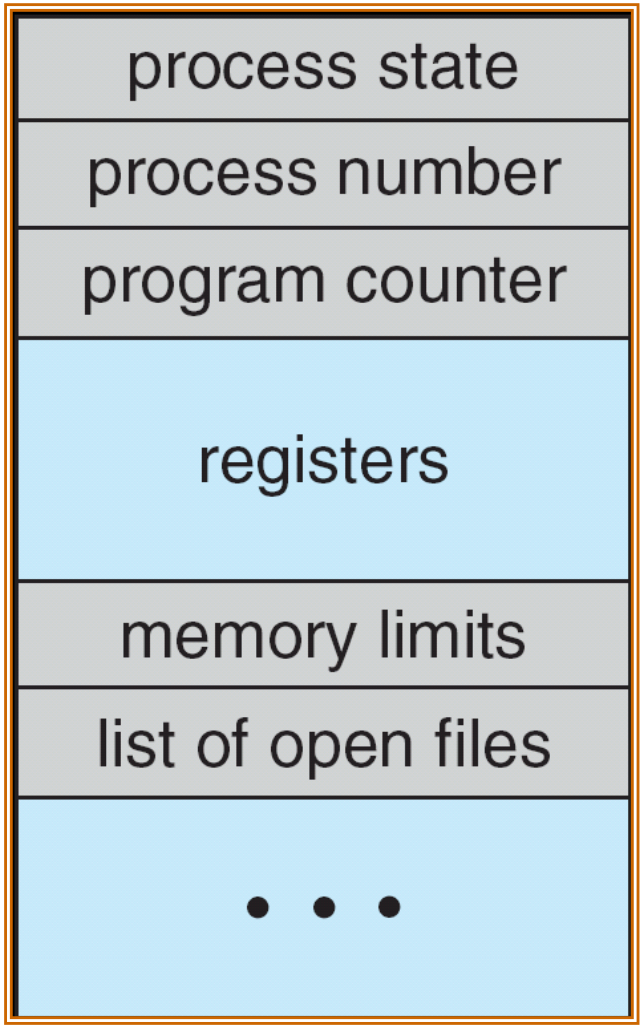
Estado del proceso



Bloque de control de proceso

- Cada proceso se representa dentro del sistema operativo con un ***bloque de control de proceso*** o ***PCB***.

- El PCB contiene información asociada al proceso, como:
  - ***Identificador del proceso (pid) y del padre (ppid)***
  - ***Identificador de usuario (uid) y de grupo (gid)***
  - ***Estado del proceso:*** listo, bloqueado,. . .
  - ***Estado del procesador:*** valores de los registros (incluyendo *contador de programa*). Esta información se guarda cuando se produce una interrrupción.
  - ***Información contable:*** tiempo empleado, límites, . . .
  - ***Información de planificación:*** prioridad, . . .
  - ***Información de gestión de memoria:*** descripción de los segmentos, tabla de páginas,...
  - ***Información del estado de E/S:*** disp. asignados, lista de descriptores de ficheros abiertos,…
  - ***Temporizadores, señales, semáforos, puertos,…***



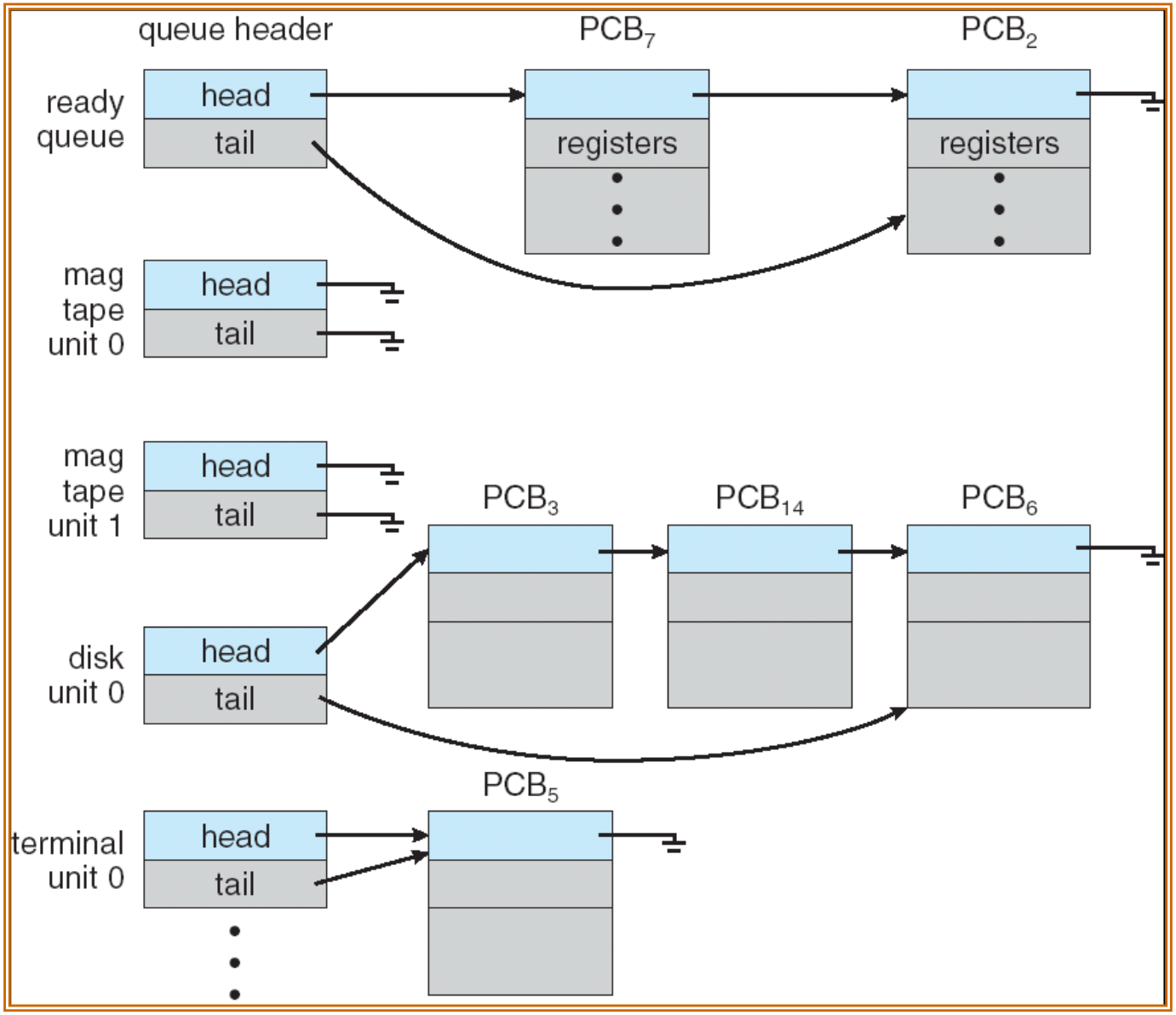
Tablas del Sistema Operativo

- Tabla de Procesos -> Tabla de PCBs
  - Cada elemento es el PCB de un proceso
- Tabla de ficheros abiertos del sistema
  - Cada proceso tiene su tabla de descriptores de ficheros, con apuntadores a entradas de esta tabla
- Tabla de Memoria
- Tabla de E/S
  - Almacena operaciones de E/S pendientes

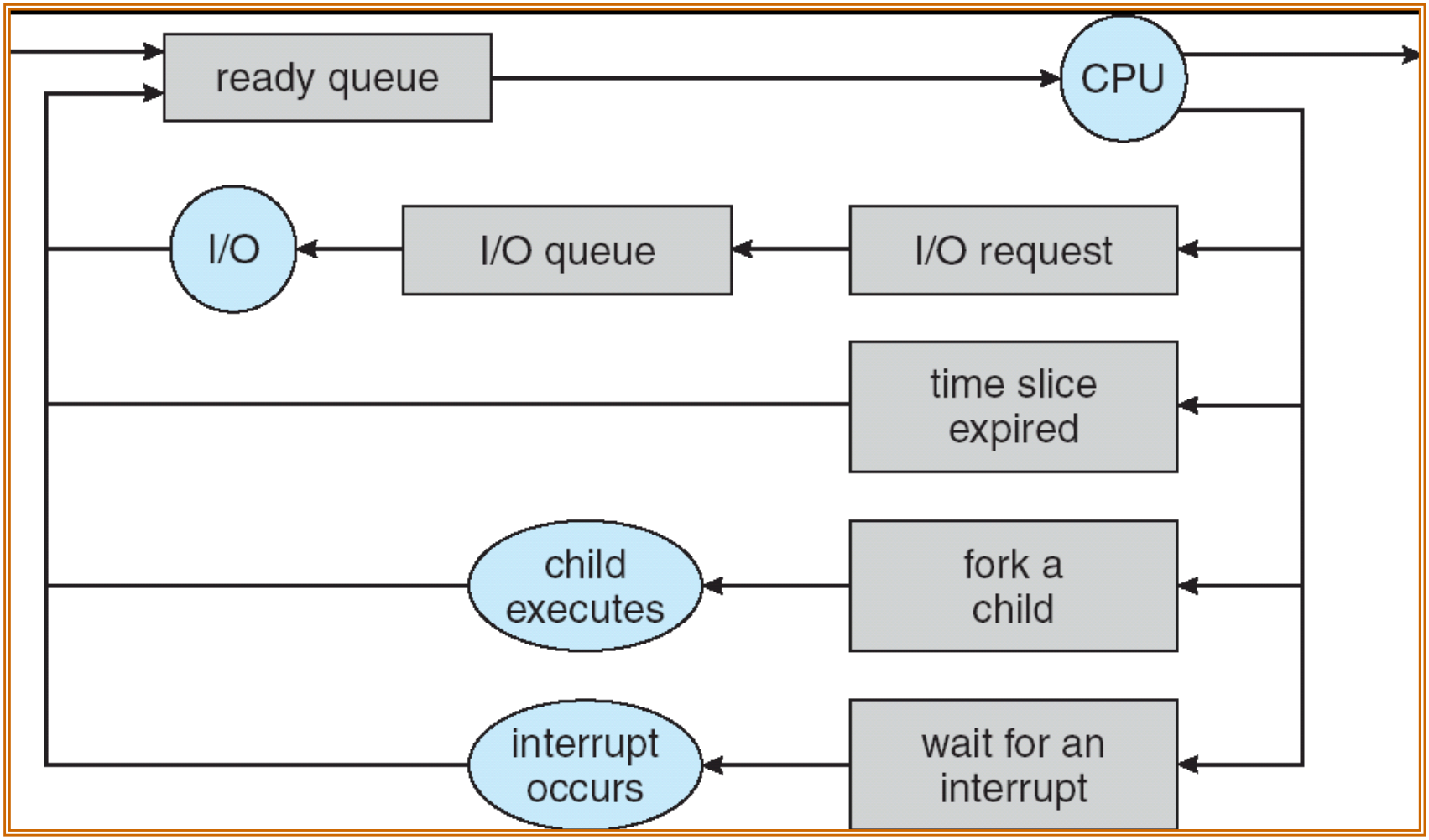
Planificación de procesos

- El objetivo es la multiprogramación: admitir varios procesos en ejecución al mismo tiempo (*concurrentemente*) para maximizar el uso de la CPU
- Los procesos se organizan en colas:
  - Una **cola de trabajos** que contiene todos los procesos del sistema
  - Una **cola de procesos preparados** que contiene todos los procesos listos esperando para ejecutar
  - Varias **colas de dispositivo** que contienen los procesos que están a la espera de alguna operación de E/S
- A lo largo de la vida de un proceso, los procesos migran de unas colas a otras

Planificación de procesos



Planificación de procesos



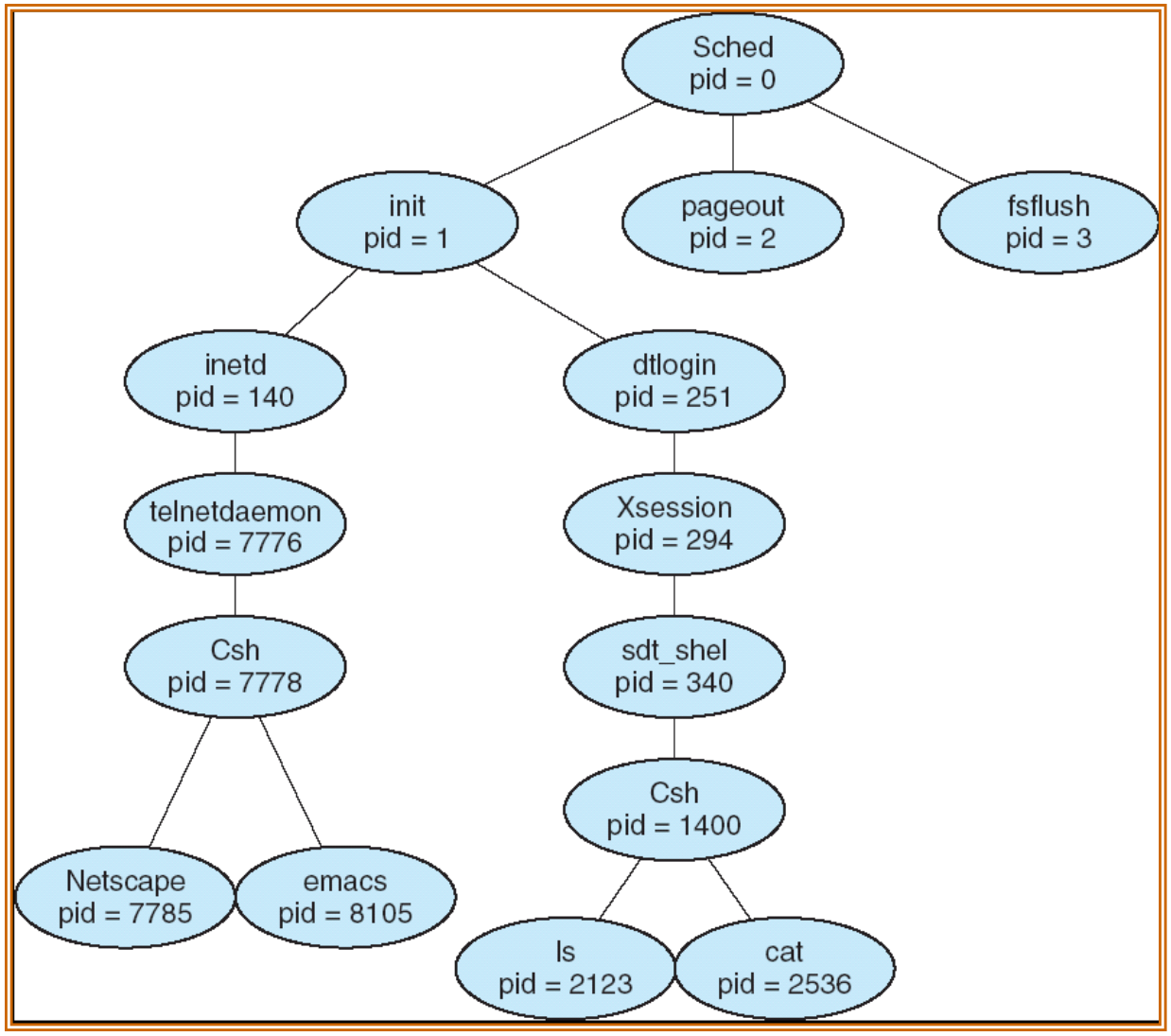
Índice

Creación de procesos

- Los procesos pueden crear nuevos procesos. El proceso creador se denomina padre y el proceso creado se denomina hijo.
- A su vez los hijos también pueden crear nuevos procesos, . . ., formándose un árbol de procesos.
- La mayoría de los sistemas operativos utilizan un **identificador de proceso (pid)** unívoco para cada proceso.

Creación de procesos

- Ejemplo: árbol de procesos



Creación de procesos

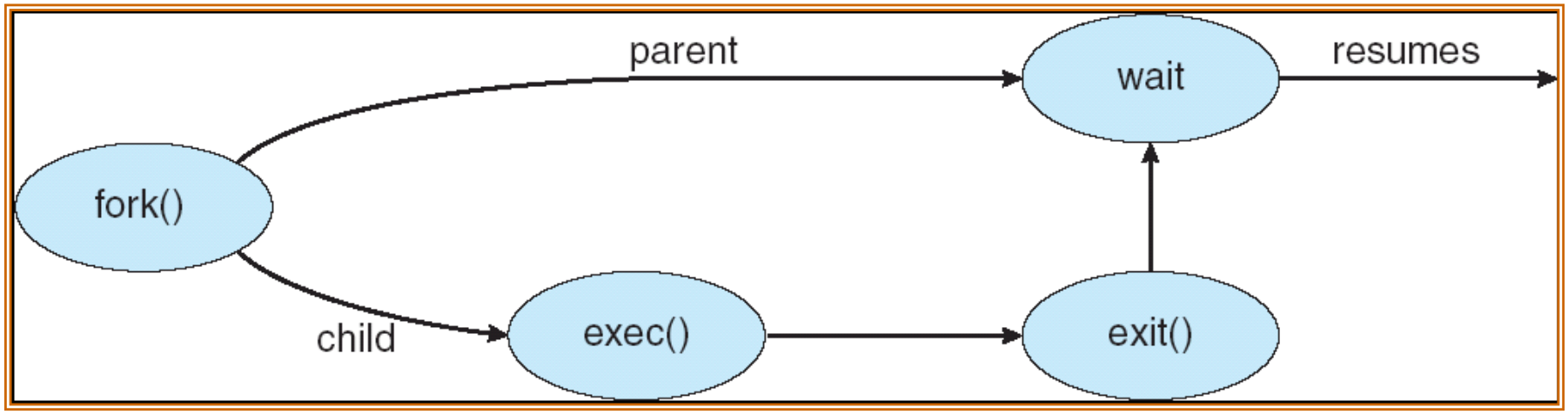
- Cuando un proceso crea otro proceso nuevo existen distintas posibilidades:
  - Ejecución:
    - Padre e hijo ejecutan concurrentemente.
    - El padre espera hasta que el hijo termine.
  - Recursos:
    - Padre e hijo comparten todos los recursos.
    - El hijo solo comparte una parte de los recursos del padre.
    - No se comparte ningún recurso.
  - Espacio de direcciones
    - El hijo tiene un duplicado del espacio de direcciones del padre.
    - El hijo carga un nuevo programa.

Terminación de procesos

- Un proceso termina cuando ejecuta su última instrucción y pide al SO que lo elimine
  - El proceso puede mandar información de finalización a su proceso padre
  - El SO libera los recursos asignados al proceso
- Un padre puede terminar la ejecución de alguno de sus hijos por diversos motivos:
  - El proceso hijo ha excedido el uso de algunos recursos
  - La tarea asignada al hijo ya no es necesaria
  - El padre abandona el sistema, y el SO no permite (no siempre) que un hijo continúe sin padre => terminación en cascada.

Creación y terminación de procesos

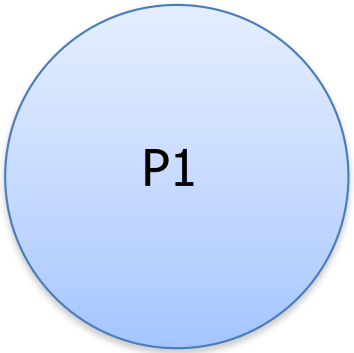
- Creación/terminación de procesos en UNIX:
  - **fork()** : llamada al sistema para crear un nuevo proceso.
  - **wait(…)**: llamada para esperar por un proceso. Se recibe la información de finalización del proceso por el que se espera
  - **exec (…)**: llamada al sistema para reemplazar el espacio de memoria actual de un proceso por un programa nuevo.
  - **exit(…)**: llamada para finalizar el proceso.

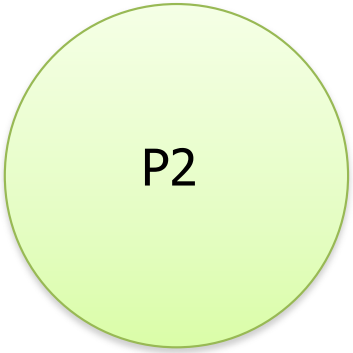


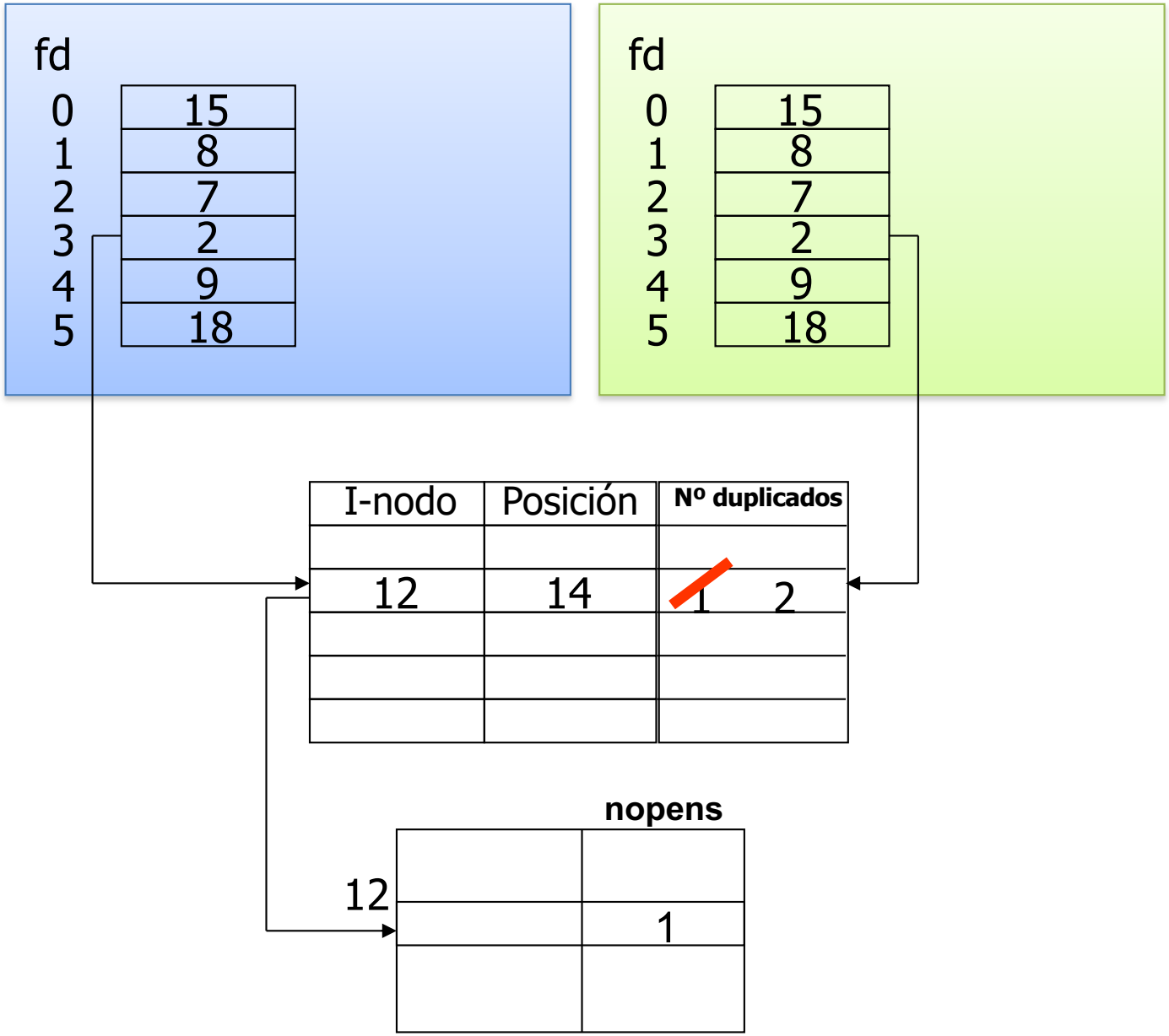
- **fork**:
  - Crea un proceso clonando al padre
  - Copia la imagen del padre en otra zona de memoria
  - Copia el PCB en la tabla de procesos
  - Cambia el pid y la descripción de memoria en el nuevo PCB
  - PC tiene el mismo valor que el del padre
  - Retorno:
    - El padre recibe el pid del proceso hijo
    - El hijo recibe 0
  - El hijo hereda los descriptores de ficheros del padre

Creación de procesos

```
....
pid = fork();
```







- **wait**:
  - Bloquea el padre hasta que finaliza un proceso hijo
  - Si el padre no realiza la operación wait, los hijos se vuelven zombies (se muestra como <defunct>)
  - Formas de llamada:
    - pid_t wait(int *status): Espera por cualquier hijo obteniendo su valor de retorno a partir de la variable status. Si no se quiere obtener se puede indicar NULL
      - Macros de apoyo:
        - WIFEXITED(status): distinto de cero si el hijo terminó.
        - WEXITSTATUS(status): Permite obtener el valor devuelto por el proceso hijo en la llamada exit. Solo se puede comprobar si WIFEXITED devuelve un valor distinto de cero.
    - pid_t waitpid(pid_t pid, int *status, int options): Espera por el hijo especificado en pid (-1, espera por cualquier proceso hijo).
      - options:
        - WNOHANG: Testea inmediatamente, sin bloquearse, si algún hijo ha terminado

Creación de procesos

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

int main() {
        pid_t  pid;
        pid = fork();
        if (pid < 0) { /* Error */
                fprintf(stderr, "Falló el fork()");
                exit(-1);
        }
        else if (pid == 0) { /* Proceso Hijo */
                printf("Hola, soy el hijo.\n");
                sleep(10);
                exit(0);
        }
        else { /* Proceso Padre */
                waitpid (pid, NULL, 0);
                printf ("El hijo terminó.\n");
                exit(0);
        }
}
```

- **exec**:
  - Ejecuta el programa indicado
  - Asigna un nuevo espacio de memoria al proceso
  - Se conserva el entorno del proceso y pid, ppid, uid, fd
  - Los descriptores se ficheros se mantienen
  - Carga datos iniciales en los segmentos
    - Rellena el PCB con valores iniciales de los registros y descripción de los segmentos
  - No regresa en caso de éxito
  - Formas de llamada:
    - int execv(const char *path, char *const argv[]);
    - int execvp(const char *file, char *const argv[]): hace funciones de busqueda en el path de la shell para encontrar el mandato
    - int execlp(const char *file, const char *arg, ...): se puede pasar como arg una lista de elementos "arg0", ..., "argn" acabados en NULL

Ejecución de procesos


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

int main() {
        pid_t  pid;
        pid = fork();
        int status;
        if (pid < 0) { /* Error */
                fprintf(stderr, "Falló el fork()");
                exit(-1);
        }
        else if (pid == 0) { /* Proceso Hijo */
                printf("Hola, soy el hijo.\n");
                execlp("ls", "ls", "-la", NULL);
                fprintf(stderr,"Se ha producido un error.\n");
                exit(1);
        }
        else { /* Proceso Padre */
                wait (&status);
                printf ("El hijo terminó.\n");
                exit(0);
        }
}
```

Índice

Cooperación entre procesos

- Un proceso es **independiente** si no puede afectar o verse afectado por otros procesos del sistema.
- Un proceso es **cooperativo** si puede afectar o verse afectado por otros procesos del sistema.
- Ventajas de la cooperación entre procesos:
  - Compartir información
  - Acelerar los cálculos
  - Modularidad
  - Conveniencia

- La cooperación entre procesos requiere mecanismos de **comunicación interprocesos** (**IPC**). Puede consistir en:
  - Avisar a un proceso de la ocurrencia de eventos:
    - **Señales**. Mecanismo estándar para informar a un proceso que ha ocurrido un evento
    - **Semáforos**. Mecanismo de bloqueo de un proceso hasta que ocurra un evento. (ver Sincronización de procesos)
  - Transferencia de datos entre procesos:
    - **Pipes**. Permite a un proceso hijo comunicarse con su padre a través de un canal de comunicación.
    - **Memoria compartida.** Se establece una región de memoria a la que pueden acceder los procesos cooperativos. Los procesos se comunican escribiendo y leyendo datos en la memoria compartida.
    - **Paso de mensajes**. Permite a los procesos comunicarse sin compartir el espacio de direcciones de memoria.

- Las señales son un mecanismo de sincronización entre procesos en S.O. compatibles con POSIX.
- Se utilizan para informar a un proceso de la ocurrencia de un evento o error.
- Las señales tienen frente al proceso el mismo comportamiento que las interrupciones frente al procesador:
  - Se interrumpe la ejecución del proceso.
  - Se salta a la rutina de tratamiento de la señal.
  - Una vez ejecutada la rutina de tratamiento, continua la ejecución del proceso.

Señales

- Cuando un proceso recibe una señal puede:
  - Ignorar la señal.
  - Invocar la rutina de tratamiento por defecto.
  - Invocar una rutina de tratamiento propia.
- Hay varias señales definidas en *signal.h*.
  - Algunas se pueden ignorar, otras no.
  - Algunas se pueden reprogramar, otras no.

- Algunas señales importantes son:

NOMBRE	DESCRIPCIÓN	REPROGRAMABLE	IGNORABLE
SIGKILL (9)	Destrucción inmediata del proceso	No	No
SIGSTOP (19)	Detiene el proceso	No	No
SIGTERM (15)	Terminación del proceso de forma normal	Sí	Sí
SIGINT (2)	Interrumpe el proceso (Por defecto termina el proceso) (Ctrl+C)	Sí	Sí
SIGUSR1 y SIGUSR2	Para uso del programador (Por defecto termina el proceso)	Sí	Sí
SIGCHLD	Un hijo ha terminado	Sí	Sí

Señales

- La llamada al sistema *signal(2)* permite establecer que acción tomar cuando se recibe una determinada señal:
  - `signal(int signum, sighandler_t handler)`: asigna a la señal de número *signum* el manejador *handler*. El manejador puede ser:
    - Una función a implementar `void handler(int sig)`
    - Una acción a realizar como:
      - `SIG_DFL`: Acción por defecto de la señal
      - `SIG_IGN`: Ignora la señal

Señales

- La llamada *kill(2)* permite enviar una señal a un proceso.
  - `kill(pid_t pid, int sig)`: envía la señal de número *sig* al proceso con identificador *pid.*
- También existe el mandato *kill(1)* desde consola*:*
  - `kill –signal pid` *:* envía la señal de número *signal* al proceso con identificador *pid*

- **Ejemplo con un solo proceso**

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

void manejador(int sig) {
	printf("Recibida señal %d!\n",sig);
}

int main() {
	signal(SIGINT, manejador);
	printf("Esperando señal.\n");
	pause();
	printf("Saliendo.\n");
	return 0;
}
```

- **Ejemplo con 2 procesos**

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

int pid;
int main() {
  signal(SIGUSR1, manejador);
  pid = fork();
  if( pid == 0 ) // Hijo
  {
    while(1);
  }
  else // Padre
  {
    kill(pid, SIGUSR1);
    wait(NULL);
    printf("El hijo terminó\n");
  }
  return 0;
}
```

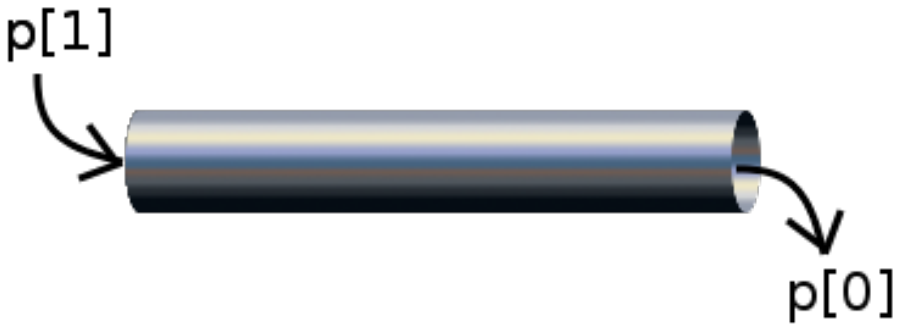
```
void manejador(int sig)
{
 if(pid == 0){
  printf("Hijo: Recibida señal %d\n", sig);
  kill(getppid(), SIGUSR1);
  exit(0);
 } else {
  printf("Padre: Recibida señal %d\n", sig);
 }
}
```

Pipes

- Las tuberías o *pipes* son un mecanismo de comunicación con dos extremos:
  - En un extremo se puede escribir y en el otro se puede leer.
  - **El extremo de lectura es bloqueante**. Si el proceso que lee no tiene nada que leer se bloquea hasta que pueda leer algo.
- Se manipulan a través de descriptores de ficheros.
  - Sobre un pipe se pueden utilizar las llamadas al sistema *read(2), write(2), close(2).*
  - Los pipes también pueden manipularse mediante el tipo de datos **FILE *** .
    - Primero se convierte el descriptor en FILE * mediante *fdopen(3)*
    - Después, sobre el FILE * es posible utilizar las funciones *fread(3), fwrite(3), fclose(3), fputs(3), fgets(3)*, etc.

- La llamada al sistema *pipe(2)* tiene la siguiente sintaxis:
  - `int pipe(int filedes[2]):` crea una tubería y asigna el descriptor de fichero de la entrada a *filedes[1]* y el descriptor de la salida a *filedes[0]*.
- Los pipes son **unidireccionales**:
  - Se debe cerrar el descriptor que no se usa

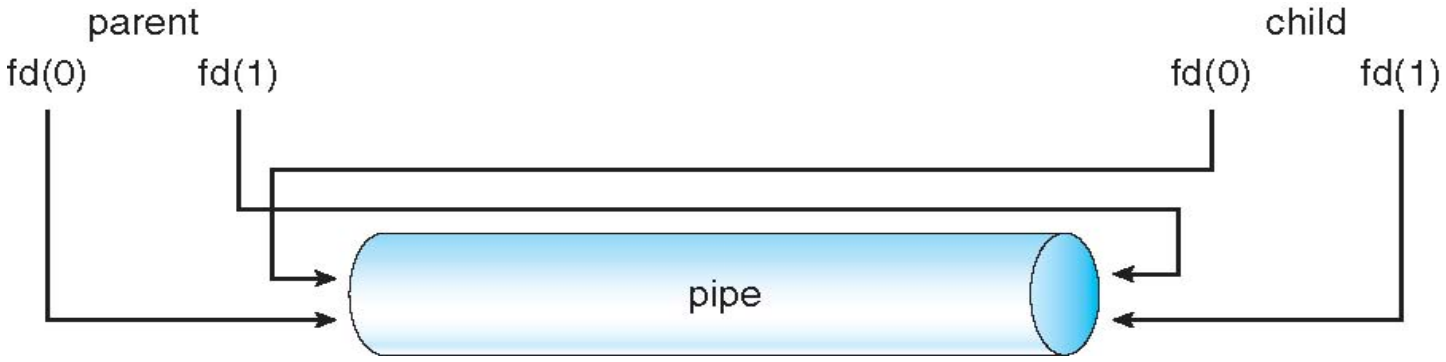
```
int p[2]
. . .
. . .
pipe(p);
```



Pipe + fork

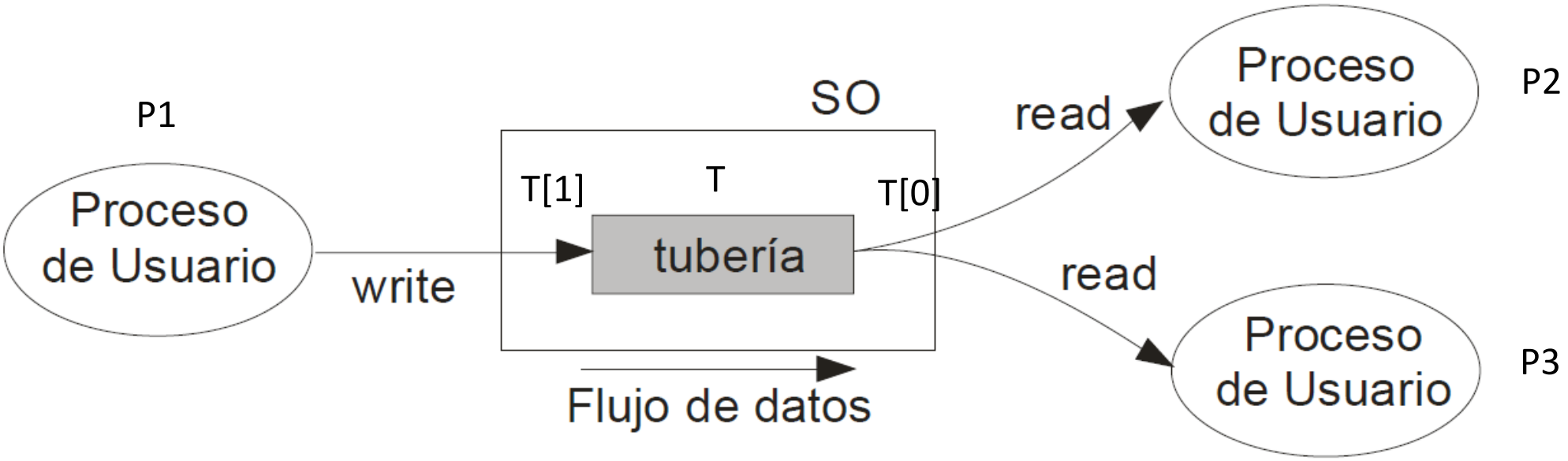
- El proceso hijo hereda los descriptores de los ficheros y pipes abiertos por el padre al hacer **fork()**

```
int fd[2], pid;
. . .
. . .
pipe(fd);
pid = fork();
. . .
```



- Antes de usar el pipe, si un proceso no va a usar alguno de los descriptores del mismo, debe cerrarlos.
- Después de usar el pipe, cada proceso deberá cerrar los descriptores que aún están abiertos.

Pipes: comunicación unidireccional



- Se pueden conectar varios procesos al mismo pipe pero siempre debe ir en la misma dirección:
  - Antes de empezar la comunicación:
    - P1 cierra T[0].
    - P2 y P3 cierran T[1].
  - Después de realizar la comunicación:
    - P1 cierra T[1].
    - P2 y P3 cierran T[0].

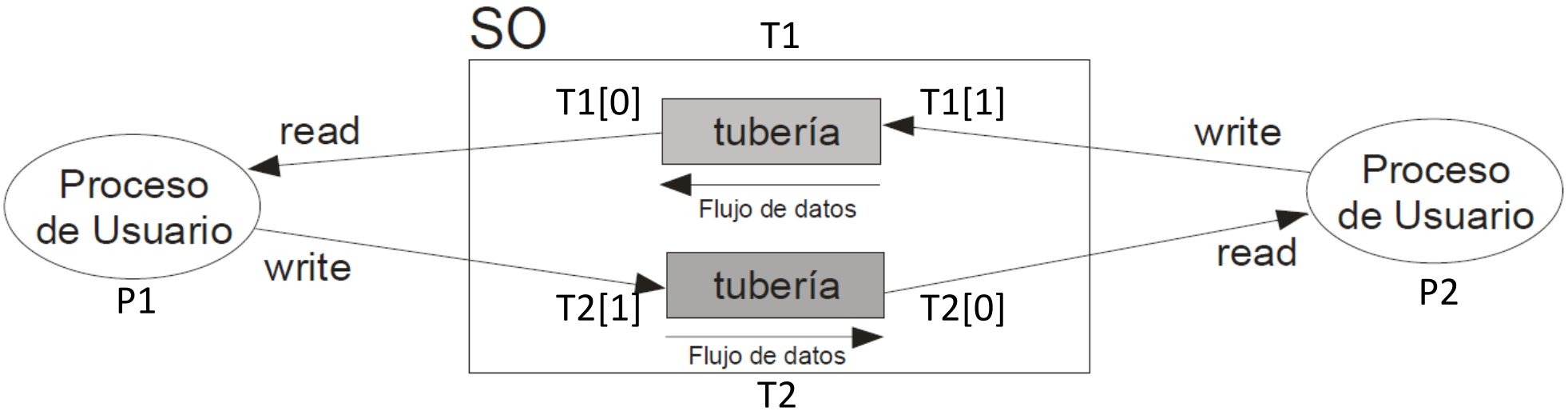
Ej: Pipes con descriptores de fichero


```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    int pipe_des[2], pid;
    char buf[1024];
    pipe(pipe_des);
    pid = fork();
    if(pid == 0) { // Hijo
        puts("Hijo: Hola, soy el hijo");
        close(pipe_des[1]); //El hijo solo recibe, cierro el pipe[1]
        read(pipe_des[0], buf, 1024);
        close(pipe_des[0]);
        printf("Hijo: Recibido el siguiente mensaje: \"%s\"\n", buf);
    }
    else { // Padre
        puts("Padre: Hola, soy el padre");
        close(pipe_des[0]); //El padre solo envía, cierro el pipe[0]
        write(pipe_des[1], "Hola hijo", 10);
        close(pipe_des[1]);
        wait(NULL);
        puts("Padre: El hijo terminó");
    }
    exit(0);
}
```

Ej: Pipes con FILE *


```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    int pipe_des[2], pid;
    char buf[1024];
    FILE *fd;
    pipe(pipe_des);
    pid = fork();
    if(pid == 0) { // Hijo
        close(pipe_des[1]); //El hijo solo recibe, cierro el pipe[1]
        fd = fdopen(pipe_des[0], "r"); // Tenemos el FILE * con el descriptor del pipe
        fgets(buf, 1024, fd);
        fclose(fd);
        printf("Hijo: Recibido el siguiente mensaje: \"%s\"\n ", buf);
    }
    else { // Padre
        close(pipe_des[0]); //El padre solo envía, cierro el pipe[0]
        fd = fdopen(pipe_des[1], "w"); // Tenemos el FILE * con el descriptor del pipe
        fprintf(fd,"Hola hijo");
        fflush(fd);
        fclose(fd);
        wait(NULL);
        puts("Padre: El hijo terminó");
    }
    exit(0);
}
```

Pipes: comunicación bidireccional



- Si se necesita comunicación bidireccional entre dos procesos A y B es necesario crear dos pipes: uno para enviar de A a B, y otra para enviar de B a A.
  - Antes de empezar la comunicación:
    - P1 cierra T1[1] y T2[0].
    - P2 cierra T1[0] y T2[1].
  - Después de realizar la comunicación:
    - P1 cierra T1[0] y T2[1].
    - P2 cierra T1[1] y T2[0].

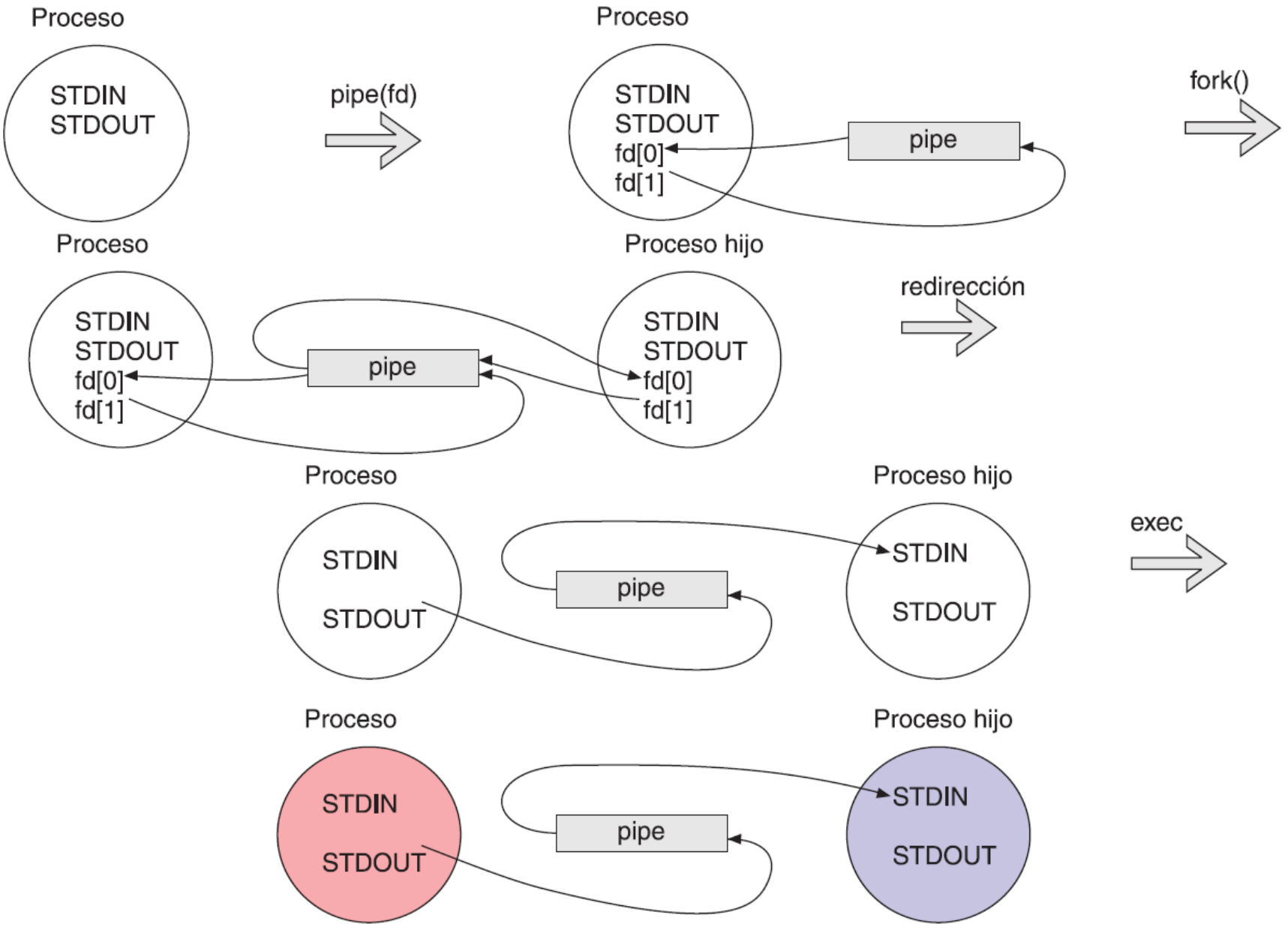
Ejemplo: redirecciones con pipes

- Se puede comunicar dos procesos redirigiendo la salida estándar del primero a la entrada estándar del segundo mediante pipes y dup.
- Ejemplo: comunicación padre-hijo donde ambos hacen **exec()**.

```
int fd[2], pid;
. . .
. . .
pipe(fd);
pid = fork();
// PROCESO PADRE
if (pid > 0) {
        close(fd[0]);
        close(STDOUT_FILENO);
        dup(fd[1]);
        close(fd[1]);
        exec...(...);
        ...
}
```

```
// PROCESO HIJO
else if (pid == 0) ) {
        close(fd[1]);
        close(STDIN_FILENO);
        dup(fd[0]);
        close(fd[0]);
        exec...(...);
        ...
}
```

Ejemplo: redirecciones con pipes (cont.)



Memoria compartida

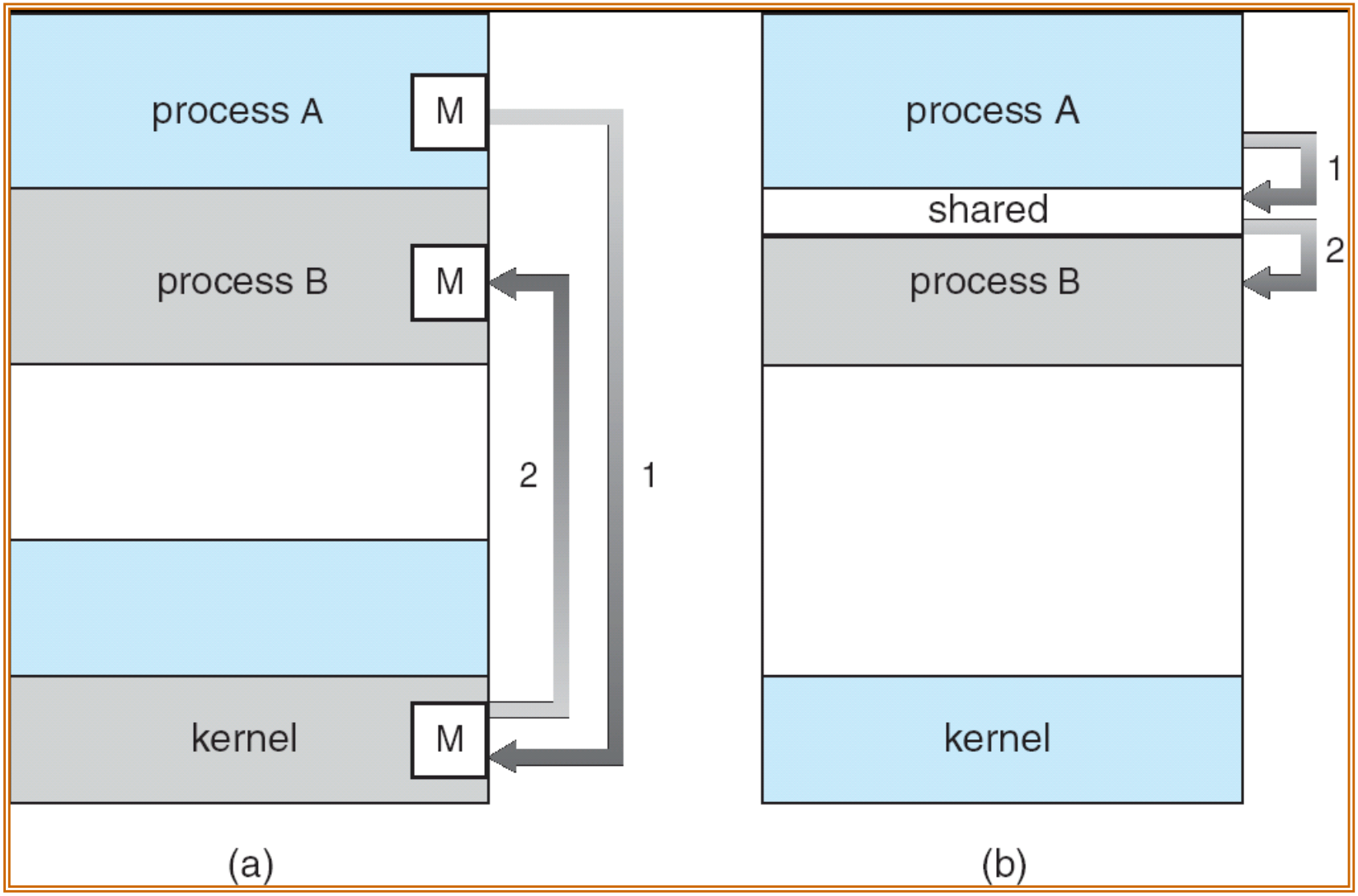
- El S.O proporciona algún tipo de mecanismo que permita a dos o más procesos compartir un segmento de memoria.
- Los procesos son los responsables del formato de los datos compartidos y de su ubicación. El S.O no interviene.
- Llamadas al sistema:
  - *shmget(…)/ shmctl()*: crear/eliminar segmento de memoria compartida
  - *shmat(…) /* shmdt(…)*:* asociar/desconectar un segmento de memoria compartida al espacio de direcciones del proceso

Paso de mensajes

- El paso de mensajes permite a los procesos comunicarse sin compartir el espacio de direcciones.
- Especialmente útil si los procesos que se comunican están en máquinas diferentes (memoria distribuida)
- Dos operaciones básicas: ***send*** y ***receive***.
- Si dos procesos quieren comunicarse:
  - Primero establecen enlace de comunicación.
  - Intercambiar mensajes mediante ***send/receive***
- Suele ocultar detalles de implementación:
  - Físicos: localización, red, . . .
  - Lógicos: comunicación directa/indirecta, síncrona/asíncrona, . . .

- **Sincronización:** el paso de mensajes puede ser bloqueante o no bloqueante.
  - ***Bloqueante o síncrono***:
    - Envío: el proceso que envía un mensaje se bloquea hasta que el receptor recibe el mensaje.
    - Recepción: el proceso que recibe se bloquea hasta que hay un mensaje disponible.
  - ***No bloqueante o asíncrono***:
    - Envío: el proceso transmisor envía el mensaje y continúa trabajando.
    - Recepción: el proceso que recibe extrae un mensaje válido o un mensaje nulo.

Paso de mensajes vs. Memoria compartida



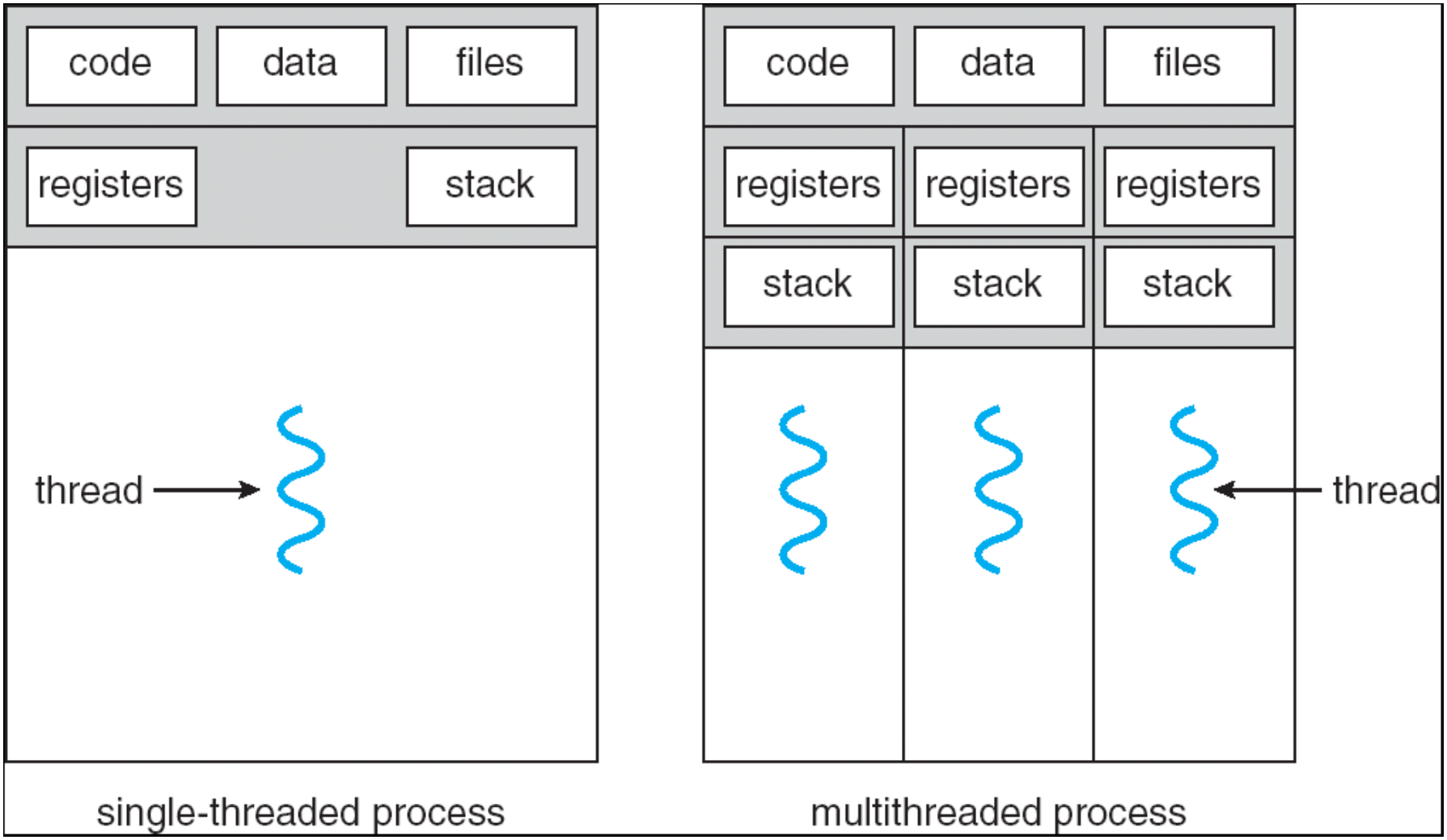
Índice

- Introducción a los procesos
- Planificación
- Creación y terminación de procesos
- Comunicación entre procesos
- **Threads**
- Sincronización de procesos
- Planificación de la CPU

Threads

- Las hebras, hilos o *threads* son una unidad básica de utilización de la CPU.
- Comparten con otros *threads* la sección de código, la sección de datos y otros recursos.
- Cada *thread* tiene su propio contador de programa, conjunto de registros de la CPU y pila, además de un identificador único.

Threads


code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

- Las ventajas de la programación multihilo son:
  - **Capacidad de respuesta:** permite a los programas continuar atendiendo al usuario aunque alguna de las tareas que esté realizando el programa sean muy largas.
  - **Compartición de recursos:** por defecto los threads comparten la memoria y recursos del proceso al que pertenecen.
  - **Economía:** es más "barato" en términos de uso de memoria y otros recursos crear nuevos threads que crear nuevos procesos.
  - **Utilización en arquitecturas multiprocesador:** permiten desarrollar aplicaciones paralelas de memoria compartida sin necesidad de realizar cambios de contexto.

- Dos tipos:
  - **Threads de usuario:** proporcionados por una librería de usuario:
    - Pthreads
    - Threads de Win32
    - Java Thread
  - **Threads del kernel:** proporcionados directamente por el sistema operativo.

Pthread

- Pthread es una API para la creación y sincronización de threads:
  - Para utilizarla se debe incluir la directiva `#include <pthread.h>`
  - Para compilar haciendo uso de la librería Pthread se debe incluir "*-lpthread*" en la compilación. Ej: `gcc file.c -lpthread -o programa`
  - Tipo de datos con identificador TID (`pthread_t`)
  - Cada thread tiene un conjunto de atributos (`pthread_attr_t`)
  - Se crea una nuevo thread con la función:
    `pthread_create(pthread_t *tid, pthread_attr_t *attr, void *funcion, void *param)`
  - En `funcion` se debe implementar la función que ejecuta la nueva hebra. Los threads siempre finalizan su ejecución con `pthread_exit(void *res)`
  - Para la sincronización de threads se utiliza `pthread_join(pthread_t tid,int **res)`. El thread que la llama espera a que finalice la hebra llamada antes de continuar. `res` devuelve el valor de retorno.

Pthread


```
#include <pthread.h>

int sum = 0;
void *suma(void *num) { // hebra
        int limite = *(int *) num;

        for(int i = 0; i <= limite; i++)
                sum += i;
        pthread_exit(NULL);
}


int main() {
        pthread_t tid;
        int N = 10;

        pthread_create(&tid, NULL, suma, (void *) &N);
        pthread_join(tid, NULL);
        printf("SUMA = %d\n", sum); //SUMA = suma de los N primeros numeros
        return 0;
}
```

Índice

Condiciones de carrera

- Varios procesos o threads en ejecución.
- Comparten datos.
- El acceso concurrente o en paralelo a esos datos puede llevar a resultados erróneos:
- Ejemplo: problema de productor/consumidor.
  - Se tiene una variable “cuenta” que contiene el número de elementos en el buffer.

Memoria compartida

- Problema del productor-consumidor.
  - Un buffer con capacidad para N elementos, compartido entre los dos procesos.
  - El productor inserta elementos en el buffer, y el consumidor los extrae.

```
//DATOS COMPARTIDOS
#define BUFFER_SIZE 10
typedef struct {
        . . .
} item;

item
buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int cuenta = 0;
```

```
while (true) {  //PRODUCTOR
   while (cuenta == BUFFER_SIZE); /*No
                                hacer nada */
   buffer[in] = produce_item();
   in = (in + 1) % BUFFER SIZE;
   cuenta++;
}
```

```
while (true) {  //CONSUMIDOR
  while (cuenta == 0);/*No hacer nada*/
  consume_item(buffer[out]);
  out = (out + 1) % BUFFER SIZE;
  cuenta--;
}
```

Condiciones de carrera

- cuenta++ en ensamblador:

  register1 = cuenta

  register1 = register1 + 1

  cuenta = register1

- cuenta-- en ensamblador:

  register2 = cuenta

  register2 = register2 – 1

  cuenta = register2

- Supongamos que *cuenta* vale 5, y los dos elementos están ejecutándose. Puede suceder que ambos lleguen a la instrucción que modifica *cuenta* y se ejecute lo siguiente:

```
T0: productor      register1 = cuenta {register1 = 5}
T1: productor      register1 = register1 + 1   {register1 = 6}
T2: consumidor     register2 = cuenta  {register2 = 5}
T3: consumidor     register2 = register2 - 1   {register2 = 4}
T4: productor      cuenta = register1   {cuenta = 6 }
T5: consumidor     cuenta = register2   {cuenta = 4}
```

- Se ha llegado al valor de *cuenta = 4* que es incorrecto.

Condiciones de carrera

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

//Variable compartida
volatile int j = 0;

void *th_func1(void *arg) {
        int i;
        for(i = 0; i < 10000; i++)
           j = j + 1;
        pthread_exit(NULL);
}

int main() {
        pthread_t th1, th2;

        pthread_create(&th1, NULL, th_func1, NULL);
        pthread_create(&th2, NULL, th_func1, NULL);

        pthread_join(th1, NULL);
        pthread_join(th2, NULL);

        printf("j = %d\n", j); // OJO: ¿j vale 20000?
}
```

Condiciones de carrera

- Esto sucede por permitir que ambos procesos manipulen variables compartidas de forma concurrente.
- Cuando varios procesos acceden a los mismos datos de forma concurrente y el resultado de la ejecución depende del orden concreto en que se realicen los accesos, se dice que hay una **condición de carrera**.
- Se necesitan mecanismos que impidan estas situaciones.

Sección crítica

- Sistema con *n* elementos de computo coordinándose.
- Cada elemento tiene una parte del código llamado **sección crítica** donde se accede a variables y recursos compartidos.
- Cuando un elemento está ejecutando su sección crítica, ningún otro puede ejecutar su correspondiente sección crítica.
- El *problema de la sección crítica* consiste en diseñar un protocolo que permita cooperar

```
do {
        Sección de
        entrada;

        SECCIÓN CRÍTICA

        Sección de
        salida;


        SECCIÓN RESTANTE

} while (TRUE);
```

Sección crítica

- Cualquier solución al problema de la sección crítica debe cumplir:
  - **Exclusión mutua**: si un elemento está ejecutando su sección crítica, ningún otro puede ejecutar su sección crítica.
  - **Progreso**: si ningún elemento está ejecutando su sección crítica y hay varios que quieren entrar en su sección crítica, solo los que no están en su sección restante pueden participar en la decisión de quien entra en la sección crítica, y esta decisión no puede posponerse indefinidamente.
  - **Espera limitada**: existe un límite de veces que se permite a otros entrar en la sección crítica después de que otro lo haya solicitado y antes de que se le conceda.
- Se utilizan herramientas que ofrecen abstracciones más simples, como los semáforos, mutex, monitores, etc

Semáforos

- Un semáforo es una variable entera a la que solamente se accede mediante dos operaciones atómicas: wait y signal.
- Las operaciones *wait* y *signal* deben ser **atómicas** (se ejecutan de principio a fin sin interrupciones), por lo que internamente utilizarán inhibición de interrupciones, TestAndSet, Swap, etc
- **wait**:

```
wait(S){
   S--;
   if (S < 0) {
      añadir el proceso a lista de bloqueados
      block(); }
}
```

- **signal**:

```
signal(S){
  S++;
  if(S <= 0) {
      sacar un proceso P de la lista
      wakeup(P);   }
}
```

Semáforos

- A menudo se diferencia entre semáforo binario y semáforo contador:
  - Binario: solo puede tomar valor 0 o 1. Se denomina **mutex** (*Mutual Exclusion*) y suele utilizarse para el acceso a una sección crítica
  - Contador: puede tomar valores dentro de un rango.
- Con un semáforo binario se puede resolver el problema de la sección crítica:

```
//Inicializado a 1 y compartido por varios procesos
Semáforo S;

wait(S);
    SECCIÓN CRÍTICA
signal(S);
```

- También se pueden utilizar para diversos problemas de sincronización:
  - Ejemplo: 2 procesos P1 y P2, con 2 instrucciones S1 y S2 que se quiere garantizar que S1 se ejecuta antes que S2. Se utiliza un semáforo S inicializado a 0:

```
//P1                    //P2
S1;                     wait(S);
signal(S);              S2;
```

- Los semáforos contadores se pueden utilizar para controlar el acceso a un determinado recurso formado por un número finito de instancias.
  - Ej: control de acceso de un parking en base a plazas disponibles

Interbloqueos

- Interbloqueo: dos o más elementos están esperando indefinidamente por un evento que solo puede generar uno de los bloqueados.
- Ejemplo: 2 semáforos S y Q inicializados a 1, y 2 elementos ejecutando P0 y P1.

```
   P0                          P1
wait (S);                wait (Q);
wait (Q);                wait (S);
   .                           .
   .                           .
   .                           .
signal  (S);             signal (Q);
signal (Q);              signal (S);
```

- Problema de productor-consumidor
  - Semáforos compartidos por los dos procesos: ***seccion*** inicializado a valor 1, ***full*** inicializado a valor 0, ***empty*** inicializado a valor N

```
//PRODUCTOR

while (true)  {

  //produce un item

  wait (empty);
  wait (seccion);
  . . .
  //inserta el item
  . . .
  signal (seccion);
  signal (full);
}
```

```
//CONSUMIDOR

while (true) {

  wait (full);
  wait (seccion);
  . . .
  //extrae un item
  . . .
  signal (seccion);
  signal (empty);

  //consume el item
}
```

Semáforos - Implementación

- Los semáforos nos son parte del estándar de *Pthread*, pero se pueden utilizar conjuntamente.
- Para usarlos se debe incluir la directiva: *#include <semaphore.h>*
- **Tipo de datos "semáforo":** *sem_t*
- **Inicialización de un semáforo:** *sem_init(*sem, int shared, int valor)*

  **sem*: puntero a una variable de tipo *sem_t*

  shared: si se puede utilizar solo entre threads creados dentro del que inicia el semáforo (0) o se puede heredar en fork (!=0)

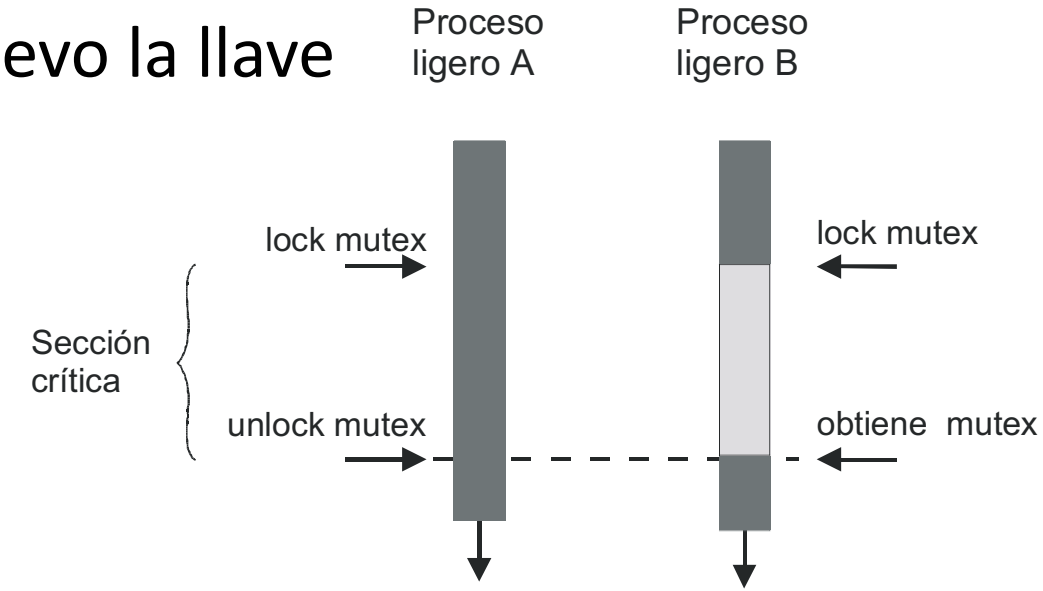
  valor: valor inicial
- **Destrucción de un semáforo:** *int sem_destroy (sem_t *sem);*
- Las operaciones **wait** y **signal** vistas se llaman:
  - *sem_wait(*sem)*: donde *sem* es un puntero a una variable *sem_t*
  - *sem_post(*sem)*: donde *sem* es un puntero a una variable *sem_t*

Semáforos - Implementación


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

volatile int j = 0;
sem_t sem;

void *th_func1(void *arg) {
        int i;
        for(i = 0; i < 10000; i++) {
                sem_wait(&sem); //Entrada en la sección crítica
                j = j + 1;
                sem_post(&sem); //Salida de la sección crítica
        }
        pthread_exit(NULL);
}

int main() {
        pthread_t th1, th2;
        sem_init(&sem, 0, 1);
        pthread_create(&th1, NULL, th_func1, NULL);
        pthread_create(&th2, NULL, th_func1, NULL);
        pthread_join(th1, NULL);
        pthread_join(th2, NULL);
        sem_destroy(&sem);
        printf("j = %d\n", j);      // j vale 20000
        return 0;
}
```

Mutex

- Mecanismo liviano idóneo para threads
- Los semáforos son más adecuados para procesos
- *Mutual Exclusion*
  - Semáforo binario sin memoria
    - Cerradura cerrada, llave puesta
    - Abro y cierro y me llevo la llave



- Dos acciones atómicas

```
mutex_lock(m);
<<sección crítica>>
mutex_unlock(m);
```

- No se puede hacer unlock sin haber hecho lock antes

Condiciones

- A veces el *thread* que está en la sección crítica no puede continuar, porque no se cumple cierta **condición** que sólo podría cambiar otro *thread* desde dentro de la sección crítica. Es preciso:
  - Liberar temporalmente el mutex que protege la sección crítica mientras se espera a que la condición se dé.
  - Sin abandonar la sección crítica.
  - De forma atómica.
- Acciones atómicas:

```
condition_wait(c,m){
        mutex_unlock(m);
        <<esperar aviso>>
        mutex_lock(m);
}
condition_signal(c) {
        if (<<alguien esperando<<)
                <<avisar que siga>>;
}
```

Mutex con condición

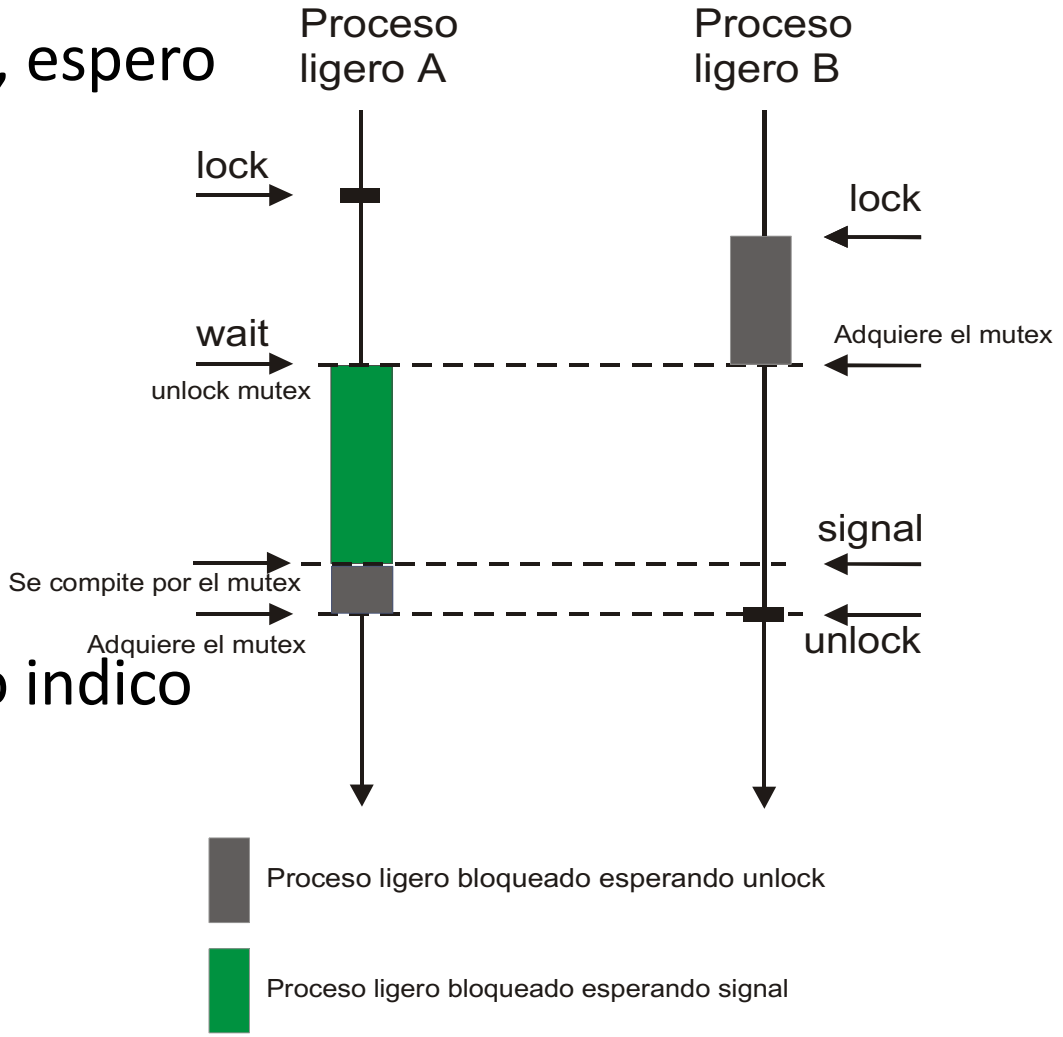
- **A**: Mientras no pueda seguir, espero

```
mutex_lock(mutex);
<<sección crítica...>>
while (<<no puedo continuar>>)
      condition_wait(cond,mutex);
<<...sección crítica>>
mutex_unlock(mutex);
```

  - Fundamental utilizar **while**

- **B**: Si alguien podría seguir, lo indico

```
mutex_lock(mutex);
<<sección crítica...>>
if (<<se podría continuar>>)
      condition_signal(cond);
<<...sección crítica>>
mutex_unlock(mutex);
```



Mutex - Implementación

- **Tipo de datos "mutex":** *pthread_mutex_t*
- **Inicialización de un mutex:** *int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *attr)*

  **mutex*: puntero a una variable de tipo mutex

  **attr:* atributos del mutex
- **Destrucción de un mutex:** *int pthread_mutex_destroy (pthread_mutex_t *mutex)*
- Operaciones básicas:
  - *int pthread_mutex_lock (pthread_mutex_t *mutex):* competir por coger el mutex
  - int pthread_mutex_unlock (pthread_mutex_t *mutex): devolver el mutex

Condiciones - Implementación

- **Tipo de datos "condición":** *pthread_cond_t*
- **Inicialización de una condición:** *int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *attr)*

  **cond*: puntero a una variable de tipo condición

  **attr:* atributos de la condición
- **Destrucción de una condición:** *int pthread_cond_destroy (pthread_cond_t *cond)*
- Operaciones básicas:
  - *int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex):* Sin salir de la sección crítica, libera temporalmente el mutex que la protege, para esperar a que se cumpla la condición
  - *int pthread_cond_signal (pthread_cond_t *cond)*: Señalar un cambio que permitiría continuar a uno de los hilos que esperan en la condición
  - *int pthread_cond_broadcast (pthread_cond_t *cond)*: Señalar un cambio que permitiría continuar a todos los hilos que esperan en la condición

Problema de productor-consumidor


```
#define BUFF_SIZE    1024
#define TOTAL_DATOS  100000
int n_datos;      /* Datos en el buffer */
int buffer[BUFF_SIZE];    /* buffer circular compartido */
pthread_mutex_t mutex;    /* Acceso a sección crítica */
pthread_cond_t no_lleno, no_vacio;  /* Condiciones de espera */

void main(void) {
        pthread_t th1, th2;

        pthread_mutex_init(&mutex, NULL);  /* Situación inicial */
        pthread_cond_init(&no_lleno, NULL);
        pthread_cond_init(&no_vacio, NULL);
        pthread_create(&th1, NULL, Productor, NULL); /* Arranque */
        pthread_create(&th2, NULL, Consumidor, NULL);
        pthread_join(th1, NULL);  /* Esperar terminación */
        pthread_join(th2, NULL);
        pthread_mutex_destroy(&mutex);     /* Destruir */
        pthread_cond_destroy(&no_lleno);
        pthread_cond_destroy(&no_vacio);
        exit(0);
}
```

Problema de productor-consumidor

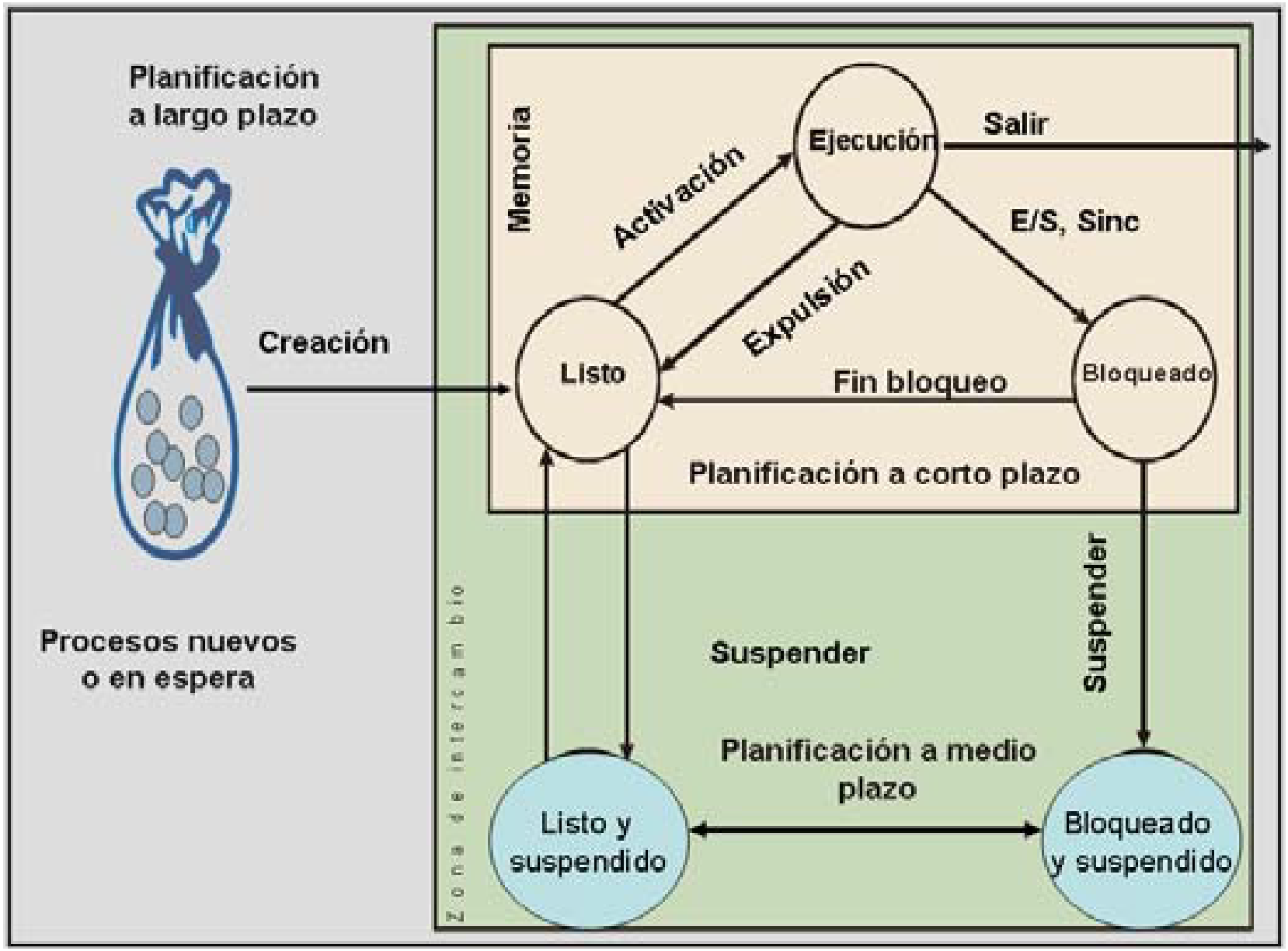

```
void Productor(void) {
          int i, dato;
          for(i=0; i < TOTAL_DATOS; i++) {
             <<Producir el dato>>
             pthread_mutex_lock(&mutex);
             while(n_datos == BUFF_SIZE)
                pthread_cond_wait(&no_lleno, &mutex);
             buffer[i % BUFF_SIZE] = dato;
             n_datos++;
             pthread_cond_signal(&no_vacio);
             pthread_mutex_unlock(&mutex);
          }
          pthread_exit(NULL);
}
void Consumidor(void) {
          int i, dato;
          for(i=0; i < TOTAL_DATOS; i++) {
             pthread_mutex_lock(&mutex);
             while(n_datos == 0)
                pthread_cond_wait(&no_vacio, &mutex);
             dato = buffer[i % BUFF_SIZE];
             n_datos--;
             pthread_cond_signal(&no_lleno);
             pthread_mutex_unlock(&mutex);
             <<Consumir el dato>>
          }
          pthread_exit(NULL);
}
```

Índice

Planificación de la CPU

- Cuando la CPU queda inactiva el SO debe seleccionar un proceso de la cola de procesos preparados y activarlo (cederle el uso de la CPU).
- La parte del SO que selecciona el proceso que pasa a ejecución es el planificador (scheduler).
  - El planificador se encarga de seleccionar los movimientos de procesos entre colas.
- Tipos de planificador:
  - A corto plazo: selecciona qué proceso de la cola de procesos preparados pasará a ejecución. Se invoca a menudo, y debe ser rápido.
  - A largo plazo: selecciona qué procesos nuevos deben pasar a la cola de procesos preparados. Se invoca con poca frecuencia: puede ser más lento.
    - Debe tratar de mantener una mezcla equilibrada de procesos limitados por E/S y procesos limitados por CPU.
    - Controla el grado de multiprogramación aceptando procesos nuevos o dejándolos en espera.
  - A medio plazo: trata la suspensión de procesos.
    - Controla el grado de multiprogramación suspendiendo procesos.
- Activador o despachador (dispatcher): parte del SO que realiza la activación de un proceso (cesión de la CPU al proceso).

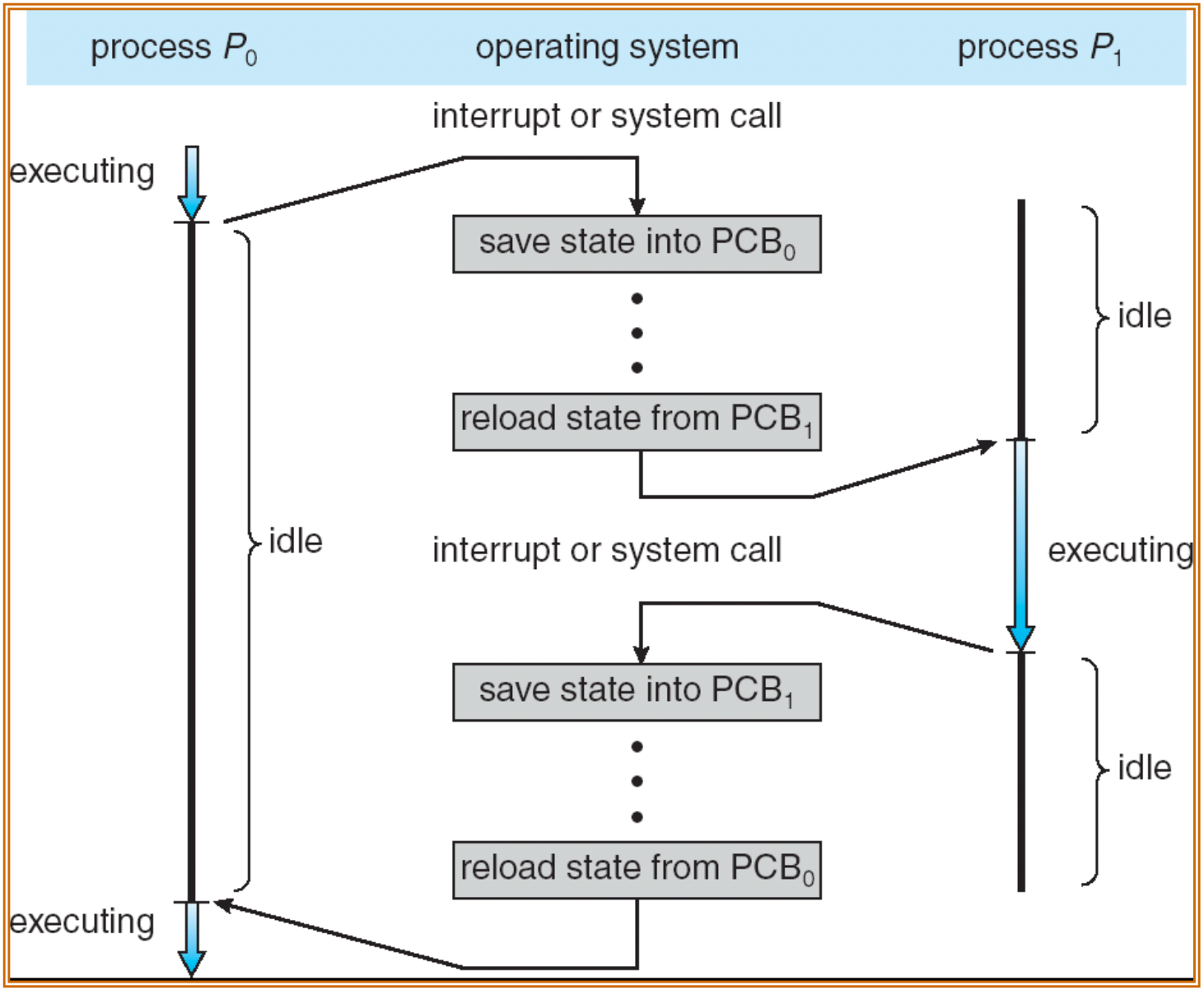
Planificación de la CPU



Cambio de contexto

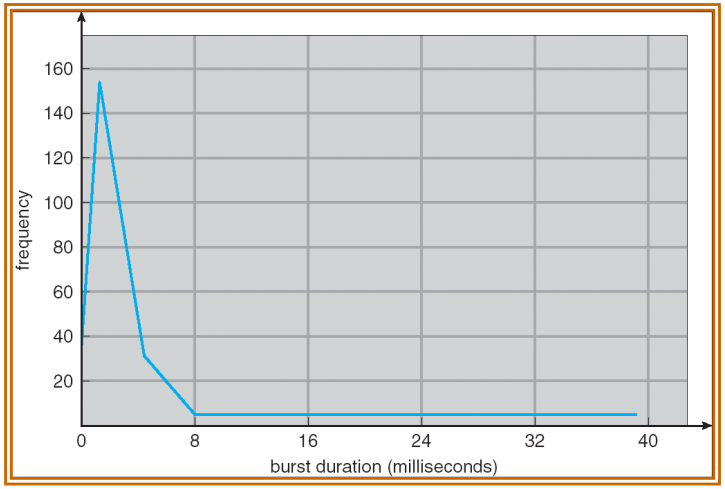
- El contexto de un proceso está formado por:
  - El estado del proceso.
  - El estado del procesador (valores de registros).
  - La información de gestión de memoria (tablas, punteros, etc).
- Cuando el despachador cede la CPU a otro proceso, el SO debe guardar el contexto del proceso actual (en su PCB) y restaurar el contexto del proceso que pasa a ejecución (desde su PCB).
- Al cambiar el proceso en ejecución en la CPU se produce un cambio de contexto.
- El cambio de contexto es tiempo perdido en gestión:
  - El procesador no hace trabajo útil durante ese tiempo.
  - El tiempo es dependiente de la arquitectura.

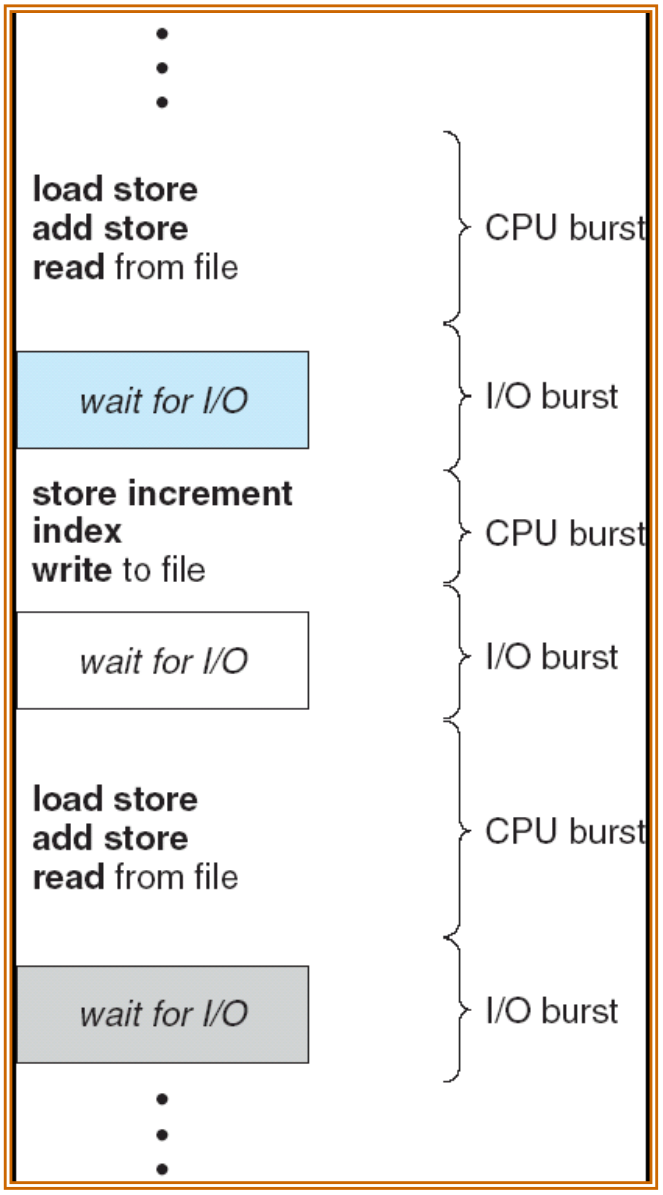
Cambio de contexto



Planificación de la CPU

- Los sistemas multiprogramados buscan maximizar el uso de la CPU.
- En un proceso se alternan ráfagas de uso de la CPU con ráfagas de Entrada/Salida.
- La duración y frecuencia de las ráfagas de CPU depende mucho de cada proceso, pero en general son muy cortas.





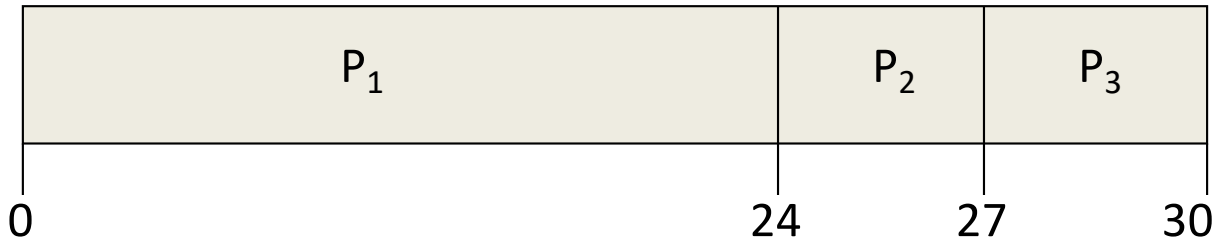
Planificación de la CPU

- En las siguientes situaciones puede ser necesario tomar una decisión sobre planificación:
  1. Cuando un proceso pasa del estado de ejecución al estado de espera.
  2. Cuando un proceso pasa del estado de ejecución al estado de preparado.
  3. Cuando un proceso pasa del estado de espera al estado de preparado.
  4. Cuando un proceso termina.
- Decisiones solamente en 1 y 4: planificación **sin desalojo**, **cooperativa** o **no expulsiva**.
- Decisiones en todos los casos: planificación **con desalojo**, **apropiativa (preemptive)** o **expulsiva**.

- Distintos criterios de planificación:
  - **Maximizar la tasa de procesamiento.** Número de procesos que se completan por unidad de tiempo.
  - **Minimizar el tiempo de ejecución.** Tiempo que pasa desde que se ordena ejecutar un proceso hasta que termina su ejecución.
  - **Minimizar el tiempo de espera.** Tiempo que pasa esperando en la cola de procesos preparados.
  - **Minimizar el tiempo de respuesta.** Tiempo que pasa desde que se envía una solicitud hasta que se empiezan a recibir resultados.

- **First-Come, First-Served (FCFS)**
  - Se asigna la CPU a los procesos en función de su orden de llegada.
  - Muy fácil de implementar con una cola FIFO.
  - Ejemplo:

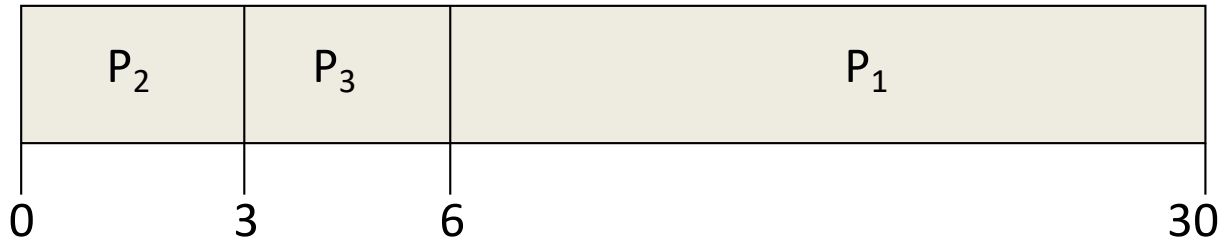
Proceso	Duración	t llegada
P1	24	0
P2	3	0
P3	3	0



Tiempo de espera: P1 = 0; P2 = 24; P3 = 27
Tiempo medio de espera: (0 + 24 + 27) / 3 = 17

- **First-Come, First-Served (FCFS)**
  - El tiempo de espera depende mucho del orden de llegada de los procesos y su duración (efecto convoy).
  - Ejemplo: mismos procesos en otro orden de llegada.

Proceso	Duración	t llegada
P2	3	0
P3	3	0
P1	24	0

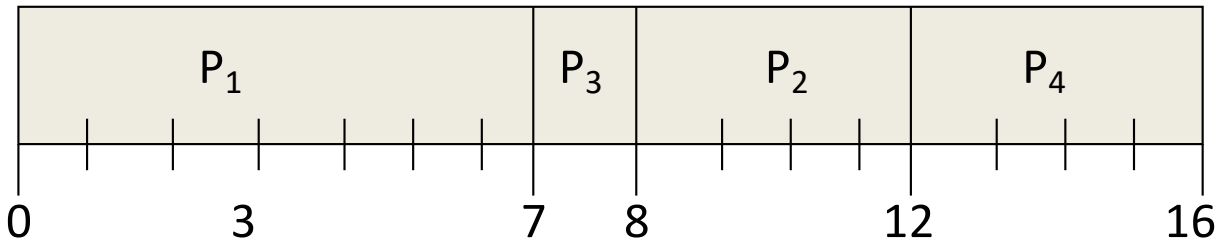


Tiempo de espera: P1 =6; P2 = 0; P3 = 3
Tiempo medio de espera: (6 + 0 + 3) / 3 = 3

- **Shortest Job First (SJF)**
  - Cuando la CPU está disponible se asigna al proceso que tiene la siguiente ráfaga de CPU más corta.
  - A igualdad de tiempo de ráfaga, se usa FCFS para decidir.
  - Este algoritmo se puede utilizar de forma cooperativa o de forma apropiativa.
  - Es el algoritmo óptimo en cuanto a tiempo medio de espera.

- **Shortest Job First (SJF)**
  - Ejemplo cooperativo.

Proceso	Duración	t llegada
P1	7	0
P2	4	2
P3	1	4
P4	4	5

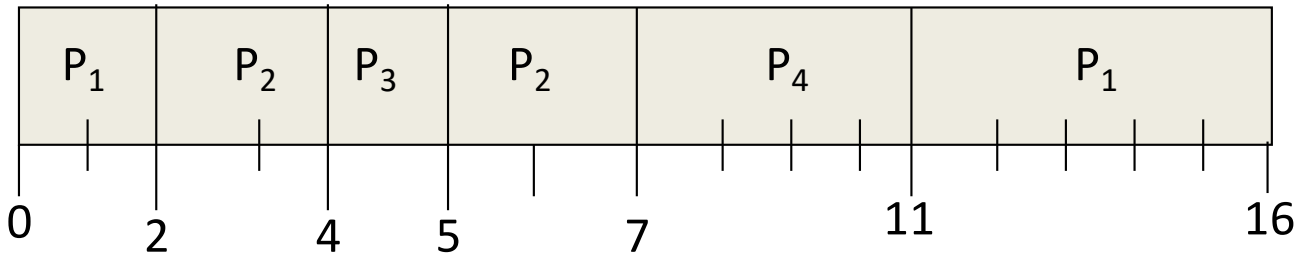


Tiempo de espera: P1 =0; P2 = 6; P3 = 3; P4 = 7

Tiempo medio de espera: (0 + 6 + 3 + 7) / 4 = 4

- **Shortest Job First (SJF)**
  - Ejemplo apropiativo.

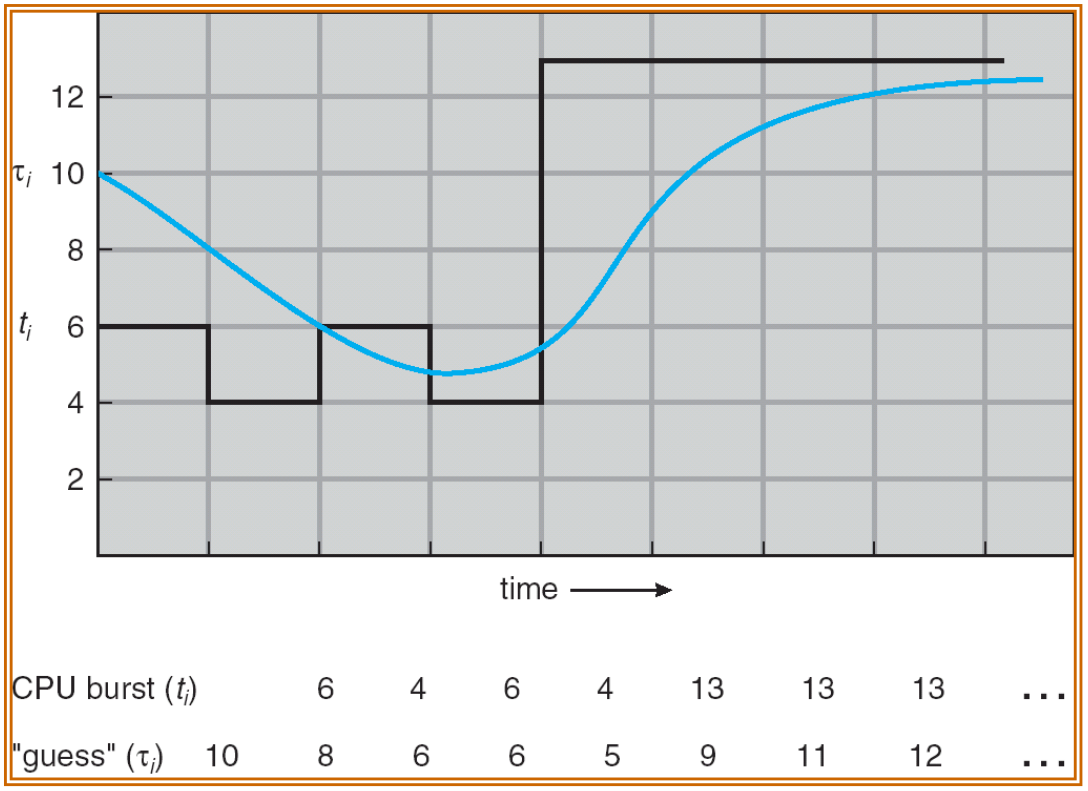
Proceso	Duración	t llegada
P1	7	0
P2	4	2
P3	1	4
P4	4	5



Tiempo de espera: P1 =9; P2 = 1; P3 = 0; P4 = 2
Tiempo medio de espera: (9 + 1 + 0 + 2) / 4 = 3

- **Shortest Job First (SJF)**
  - No se puede saber a priori cuánto durará la próxima ráfaga de CPU de un proceso, pero se puede estimar en función de la historia del proceso.

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

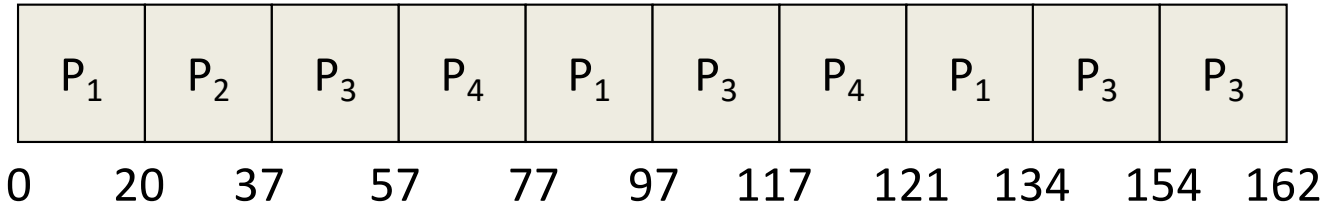


- **Planificación por prioridades**
  - A cada proceso se le asocia una prioridad. La CPU se asigna al proceso más prioritario.
    - Este algoritmo se puede utilizar de forma cooperativa o de forma apropiativa.
    - El algoritmo SJF es un caso particular del algoritmo por prioridades, donde la prioridad es la duración de la siguiente ráfaga de la CPU.
  - Problema: inanición. Es posible que algunos procesos nunca lleguen a ejecutarse.
    - Solución: envejecimiento. Se aumenta la prioridad de los procesos que llevan mucho tiempo sin ejecutarse.

- **Planificación por turnos (Round Robin)**
  - Típico de los sistemas de tiempo compartido.
  - Es similar a FCFS, pero cada cierto tiempo (*cuanto*) el S.O desaloja el proceso que se está ejecutando y lo pone al final de la cola de procesos preparados. El primer proceso de la cola de preparados toma la CPU.
  - El rendimiento del algoritmo depende mucho de la duración del *cuanto* de tiempo respecto a la duración media de las ráfagas de CPU.
    - Un *cuanto* muy grande en RR es casi lo mismo que FCFS
    - Un *cuanto* muy pequeño producirá muchos cambios de contexto (se pierde tiempo en cada cambio).

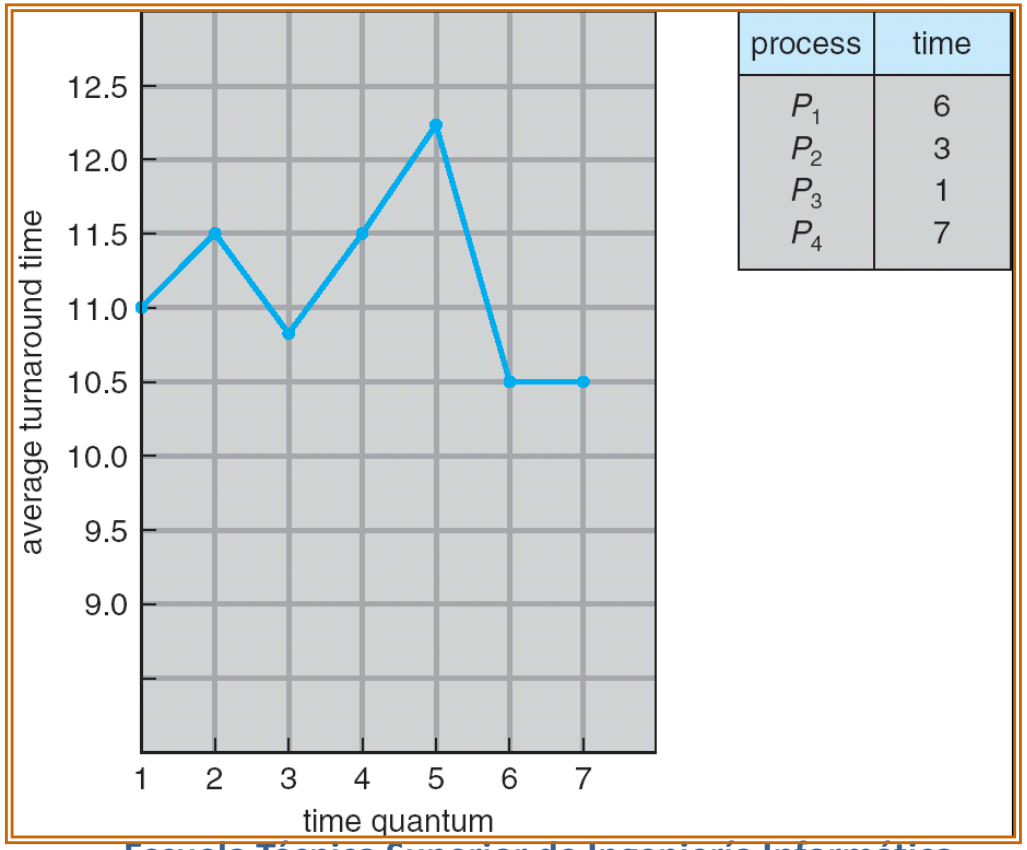
- **Planificación por turnos (Round Robin)**
  - Ejemplo con *cuanto* = 20:

Proceso	Duración	t llegada
P1	53	0
P2	17	0
P3	68	0
P4	24	0



Tiempo de espera: P1 =81; P2 = 20; P3 = 94; P4 = 97
Tiempo medio de espera: (81 + 20 + 94 + 97) / 4 = 73

- **Planificación por turnos (Round Robin)**
  - El tiempo medio de ejecución depende del *cuanto* de tiempo, y no tiene por que mejorar cuando se aumenta la duración del *cuanto*.



- **Planificación con colas multinivel**
  - La cola de procesos preparados se divide en varias colas. Cada cola tiene su propio algoritmo de planificación.
  - Ejemplo:
    - 2 colas: procesos interactivos y procesos por lotes.
    - Interactivos: planificación Round Robin
    - Por lotes: planificación FCFS.
    - Planificación entre las colas, varias opciones:
      - Prioridad fija: la cola de “interactivos” antes que “por lotes”
      - Repartir el tiempo: por ejemplo 80% “interactivos” y 20 % “por lotes”

- **Planificación con colas multinivel realimentadas**
  - Los procesos pueden moverse de una cola a otra.
  - Para definir un planificador mediante colas realimentadas se definen:
    - El número de colas.
    - El algoritmo de planificación de cada cola.
    - El método para determinar cuándo pasar un proceso a una cola de prioridad más baja.
    - El método para determinar cuándo pasar un proceso a una cola de prioridad más alta.
    - El método para determinar en qué cola se introduce un proceso nuevo.

- **Planificación con colas multinivel realimentadas**
  - Ejemplo: 3 colas. Cola 1 RR 8 ms, cola 2 RR 16 ms, cola 3 FCFS.