Tema 3. Concurrencia entre procesos

UNIVERSITAT JAUME·I

Índice

- Procesamiento concurrente
- El problema de la sección crítica
- Problemas clásicos de comunicación y sincronización
- Mutex y variables de condición
- Tuberías

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
  - Describir y reconocer el problema de la sección crítica en la ejecución concurrente de procesos y de hilos
  - Describir y reconocer el problema de la sincronización entre procesos y entre hilos
  - Describir el funcionamiento del mutex como una posible solución al problema de la sección crítica
  - Describir el funcionamiento del mutex y las variables de condición como una posible solución a problemas de sincronización

UNIVERSITAT JAUME·I

Resultados de aprendizaje (cont.)

- Relacionar el concepto de proceso e hilo con el modelo de ejecución de la arquitectura y los problemas inherentes de planificación, comunicación y sincronización
  - Analizar y desarrollar programas en los que se resuleve el problema de la sección crítica mediante mutex
  - Analizar y desarrollar programas en los que se resuelve la necesidad de sincronización utilizando mutex y variables de condición
  - Describir el funcionamiento de una tubería como una posible solución al problema de sincronización y comunicación entre procesos
  - Desarrollar programas en los que se plantea la necesidad de sincronización y comunicación de procesos y proponer una solución utilizando tuberías

Bibliografía

- J. Carretero et al. *Sistemas Operativos: Una Visión Aplicada.* McGraw-Hill. 2007. Capítulo 6
- POSIX threads programming. https://computing.llnl.gov/tutorials/pthreads/#Joining
- 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

UNIVERSITAT JAUME·I

Bibliografía por secciones

- Procesamiento concurrente
- El problema de la sección crítica
- Problemas clásicos de comunicación y sincronización
- Mutex y variables de condición

Carretero

- Tuberías

Badía et al. y ***Carretero***

Tema 3. Concurrencia entre procesos

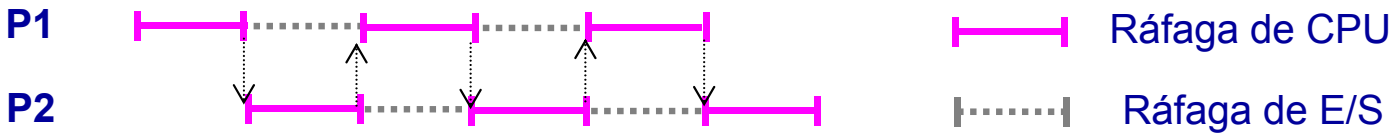
UNIVERSITAT JAUME·I

Índice

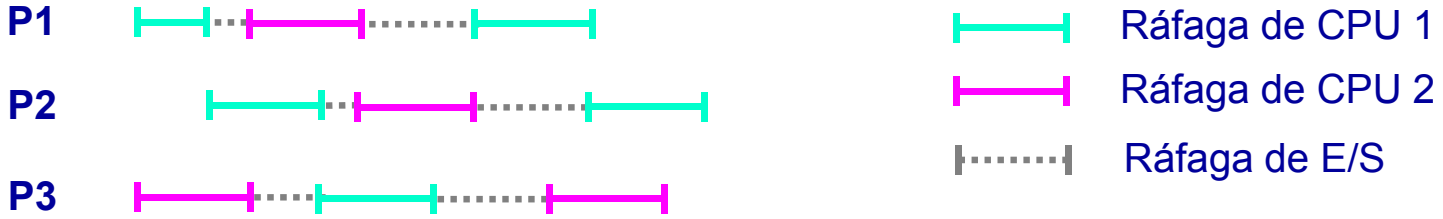
- Procesamiento concurrente
- El problema de la sección crítica
- Problemas clásicos de comunicación y sincronización
- Mutex y variables de condición
- Tuberías

Procesamiento concurrente

- **Modelos de computadora en los que se puede dar:**
  - *Multiprogramación en un único procesador*
    - Procesamiento concurrente: base de los SOs multiprogramados



  - *Multiprocesador*
    - Los procesos concurrentes no sólo pueden intercalar su ejecución sino también superponerla
    - Existe verdadera ejecución simultánea de procesos



  - *Multicomputador* (proceso distribuido)

Procesamiento concurrente

- **Razones de la ejecución concurrente:**
  - Compartir recursos físicos
  - Compartir recursos lógicos

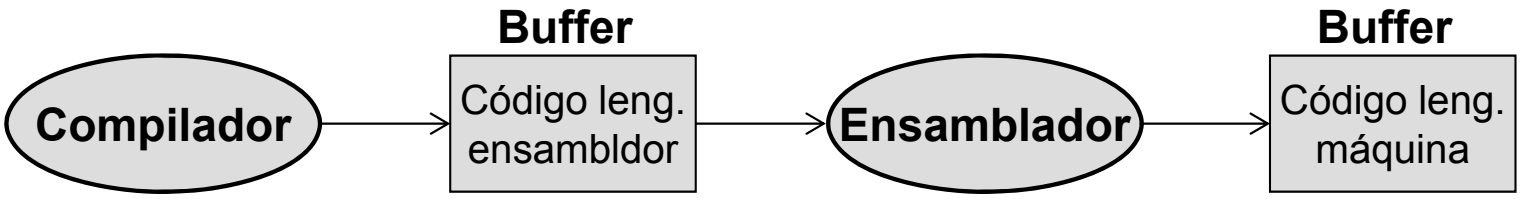
  → Mejor aprovechamiento de recursos
  - Acelerar los cálculos
  - Modularidad
  - Comodidad

  → Facilitar programación

Procesamiento concurrente

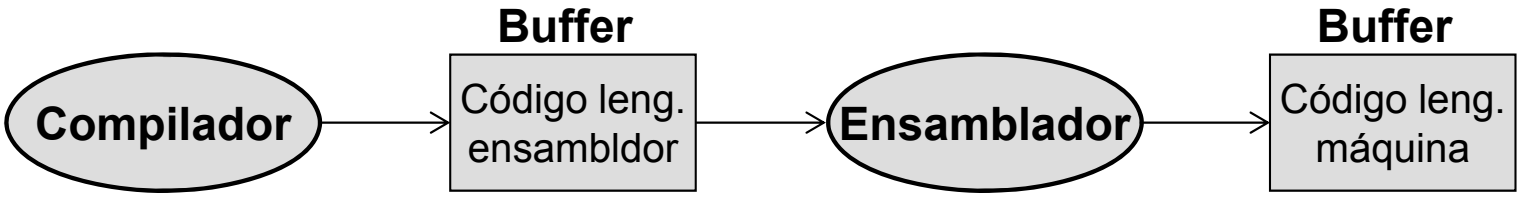
- **Tipos de procesos concurrentes:**
  - **Procesos independientes** no pueden afectar o ser afectados por la ejecución de otro proceso

    Procesos cooperantes que comparten datos pueden generar inconsistencia en esos datos



Procesamiento concurrente

- Interacción entre procesos:
  - Comparten y/o compiten por recursos
  - Ejecución sincronizada
- Se necesita:
  - Mecanismos de **sincronización y comunicación entre procesos**
  - Ejecución ordenada para conseguir datos consistentes



Condiciones de carrera

- **Problemas de la concurrencia: Ejemplo**

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

int x=0;

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

¿Funcionamiento correcto? ➔ Condiciones de carrera

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

Condiciones de carrera

- **Problemas de la concurrencia: Ejemplo**

```
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);
}
```

Condiciones de carrera

- **Problemas de la concurrencia: Ejemplo**
  - Orden de ejecución **A**:

H1.I1, H1.I2, H1.I1, H1.I2, H1.I1, H1.I2 , H2.E1, H2.E2 , H2.E1, H2.E2 , H2.E1, H2.E2 ⟶ **Valor final de x: 0**

```
int x=0;
void *fhilo1(void *arg)
{ int i, cont;
  for (i=0; i<3; i++) {
H1.I1 ⟶  cont=x+1;
          …
H1.I2 ⟶  x=cont;
}
void *fhilo2(void *arg)
{ int i, cont;
 for (i=0; i<3; i++) {
H2.E1 ⟶ cont=x-1;
          …
H2.E2 ⟶ x=cont;
}
```

Condiciones de carrera

- **Problemas de la concurrencia: Ejemplo**
  - Orden de ejecución **B**:

H1.I1, H2.E1, H2.E2, H1.I2, H1.I1, H1.I2, H1.I1, H1.I2 , H2.E1, H2.E2 , H2.E1, H2.E2

```
int x=0;
void *fhilo1(void *arg)
{ int i, cont;
  for (i=0; i<3; i++) {
H1.I1 ──→  cont=x+1;
           …
H1.I2 ──→  x=cont;
}
void *fhilo2(void *arg)
{ int i, cont;
 for (i=0; i<3; i++) {
H2.E1 ──→ cont=x-1;
          …
H2.E2 ──→ x=cont;
}
```

Valor final de x: 1

	x	H1		H2	
		i	cont	i	cont
H1.I1	0	0	1		
H2.E1				0	-1
H2.E2	-1				
H1.I2	1				
H1.I1		1	2		
H1.I2	2				
H1.I1		2	3		
H1.I2	3				
H2.E1				1	2
H2.E2	2				
H2.E1				2	1
H2.E2	1				

Condiciones de carrera

- **Problemas de la concurrencia: Ejemplo**
  - No funciona la ejecución concurrente debido a **condiciones de carrera**: el resultado final depende del orden de ejecución
  - El problema se agrava porque **H1.I1**, **H1.I2**, **H2.E1**, **H2.E2** no son atómicas (son divisibles y pueden ser interrumpidas)
    - Ejecución de **H1.I1** (`cont:=x+1`) en lenguaje máquina:

```
R1          ←  x   /* R1 es un registro de la CPU
*/
R1          ← R1 + 1
cont        ← R1
```

  - Solución a condiciones de carrera: Ejecución del código en que un hilo o proceso accede a datos compartidos en **exclusión mutua**

Tema 3. Concurrencia entre procesos

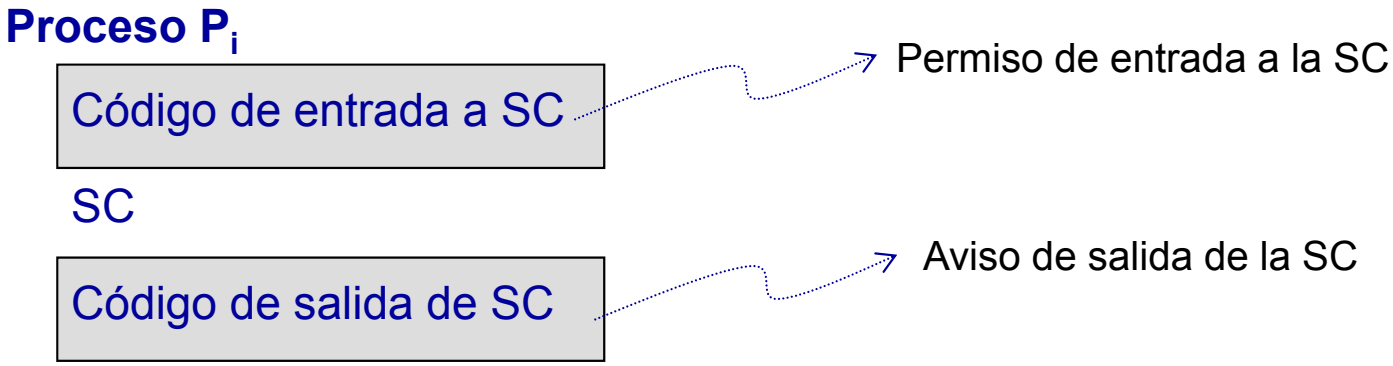
UNIVERSITAT JAUME·I

Índice

- Procesamiento concurrente
- El problema de la sección crítica
- Problemas clásicos de comunicación y sincronización
- Mutex y variables de condición
- Tuberías

El problema de la sección crítica

- **Planteamiento:**
  - **n** procesos **P_i i=1,..,n** compitiendo por usar ciertos datos compartidos
  - Cada proceso tiene un fragmento de código, llamado **sección crítica** (SC), en el que el proceso accede a los datos compartidos
- **Problema:**
  - Asegurar que cuando un proceso está ejecutando su sección crítica ningún otro proceso puede estar ejecutando su sección crítica
- **Solución:**
  - Añadir código adicional a los programas para acceder a y salir de la SC



El problema de la sección crítica

UNIVERSITAT JAUME·I

- **Requisitos que ha de cumplir una solución al problema de la SC:**
  - **Exclusión mutua**:
    - Sólo debe haber un proceso ejecutando la SC
  - **Progreso**:
    - Un proceso fuera de la SC no debe bloquear a otro que quiere entrar
  - **Espera limitada**:
    - Un proceso que quiere entrar en la SC no espera indefinidamente

El problema de la sección crítica

- **Herramientas de comunicación proporcionadas por el SO:**
  - Archivos
  - Tuberías
  - Variables en memoria compartida
  - Paso de mensajes
- **Herramientas de sincronización proporcionadas por el SO (o por el entorno de desarrollo):**
  - Tuberías
  - Semáforos
  - Mutex y variables condicionales
  - Paso de mensajes
  - Señales
  - Monitores

UNIVERSITAT JAUME·I

Índice

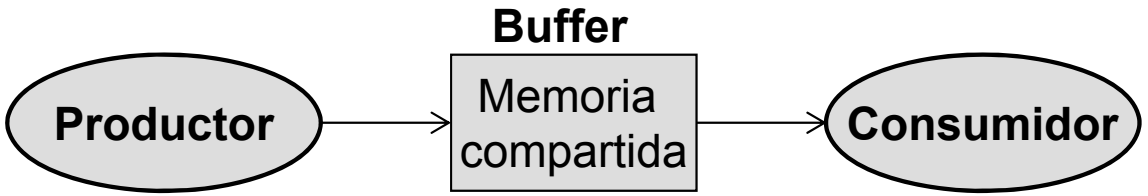
- Procesamiento concurrente
- El problema de la sección crítica
- Problemas clásicos de comunicación y sincronización
- Mutex y variables de condición
- Tuberías

Problemas clásicos de sincronización

- Problemas tipo de sincronización habituales en SOs:
  - El problema del productor-consumidor.
  - El problema de los lectores-escritores.
  - El problema de los filósofos comensales.

Problemas clásicos de sincronización

- **El problema del productor-consumidor:**
  - Planteamiento:
    - El proceso **productor** produce información y la almacena en un buffer.
    - El proceso **consumidor** accede al buffer y consume la información.
    - El productor y el consumidor comparten variables.
    - Tamaño del buffer limitado o ilimitado.



Problemas clásicos de sincronización

- **El problema del productor-consumidor con buffer ilimitado:**
  - El consumidor no puede acceder al buffer si está vacío.
- **El problema del productor-consumidor con buffer limitado:**
  - El productor no puede acceder al buffer si está lleno.
  - El consumidor no puede acceder al buffer si está vacío.

Sincronización necesaria

- **Ejemplos de procesos productor-consumidor habituales en SOs:**
  - Programa que produce caracteres para ser impresos y que son consumidos por el manejador de impresora.
  - Compilador que produce código ensamblador y que es consumido por el programa ensamblador.

Problemas clásicos de sincronización

- **Especificación del problema del productor-consumidor con buffer limitado:**

Datos compartidos

Tipos

elemento=…

Variables

buffer: array [0..n-1] de elemento	{ Array circular con **capacidad n** }
n_elementos = 0	{ Número de elementos que hay en el buffer }
entrada = 0	{ Primer entrada libre del buffer }
salida = 0	{ Primer entrada ocupada del buffer }

Problemas clásicos de sincronización

- **Especificación del problema del productor-consumidor con buffer limitado:**

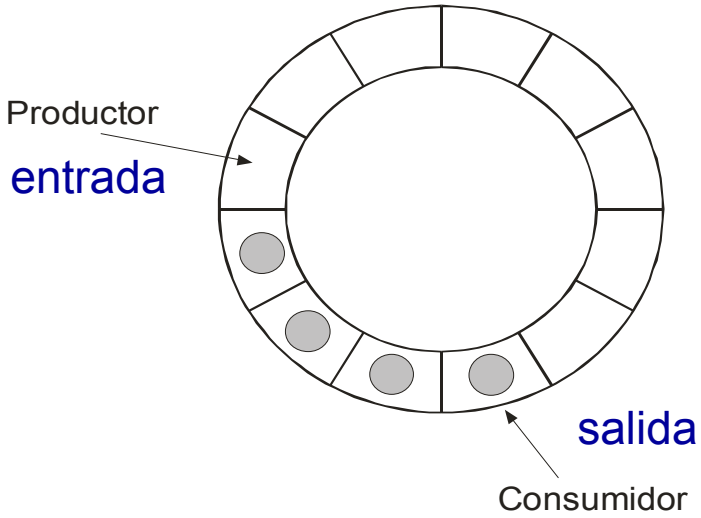
Proceso Productor

```
mientras (cierto)
    Producir elemento
    mientras (n_elementos==n) hacer nada
    buffer[entrada ] ← elemento
    entrada = entrada + 1 mod n
     n_elementos = n_elementos + 1
```

Proceso Consumidor

```
mientras (cierto)
    mientras (n_elementos==0) hacer nada
    elemento ← buffer[salida]
    salida = salida + 1 mod n
    n_elementos = n_elementos - 1
    Consumir elemento
```

Han de ser operaciones atómicas



Tema 3. Concurrencia entre procesos

UNIVERSITAT JAUME·I

Índice

- Procesamiento concurrente
- El problema de la sección crítica
- Problemas clásicos de comunicación y sincronización
- Mutex y variables de condición
- Tuberías

EI/MT1014 Sistemas Operativos

Mutex

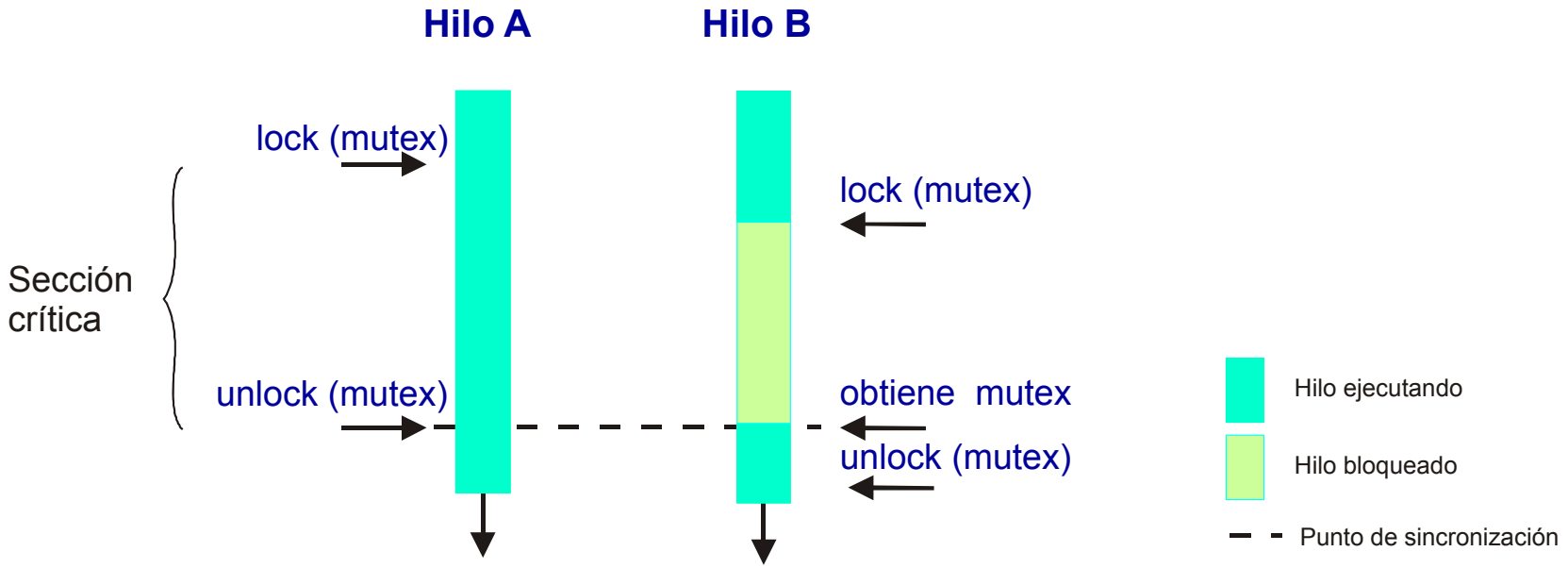
- **Definición de mutex:**
  - Mecanismo de sincronización (sencillo y eficiente) indicado para hilos
  - Se emplea para obtener acceso exclusivo a recursos compartidos y para "serializar" el acceso a la SC en exclusión mutua

    Sólo un hilo puede tener acceso simultáneamente al mutex
  - Dos operaciones atómicas:
    - `lock(m)`:
      - Intenta bloquear el mutex `m`
      - Si el mutex ya está bloqueado el hilo se suspende
    - `unlock(m)`:
      - Desbloquea el mutex `m`
      - Si existen hilos bloqueados en el mutex se desbloquea a uno

Secciones críticas con mutex

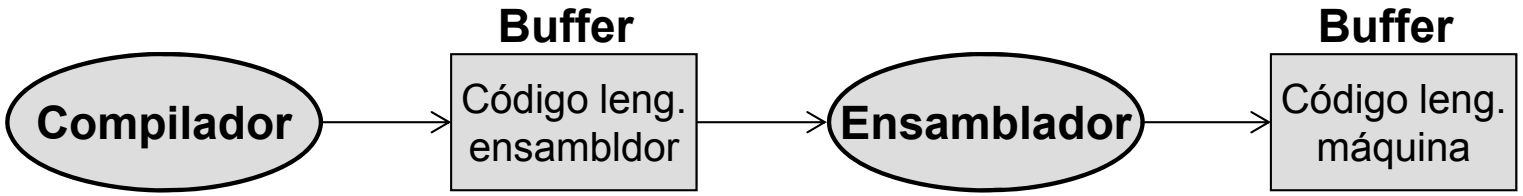
- **Utilización del mutex:**

```
lock(m);          /* Entrada en la SC */
< seccion critica >
unlock(m);        /* Salida de la SC */
```



Variables condicionales

- **Definición de variable condicional:**
  - Variable de sincronización asociada a un mutex
  - Se usa entre `lock` y `unlock`
  - Dos operaciones atómicas asociadas:
    - **`wait (condition, mutex)`:**
      - Bloquea al hilo que la ejecuta y le expulsa del mutex
    - **`signal (condition, mutex)`:**
      - Desbloquea a uno o varios procesos suspendidos en la variable condicional `condition`
      - El proceso que se despierta compite de nuevo por el mutex



Uso de mutex y variables condicionales

Hilo A e Hilo C

```
lock(mutex);
...
while (condicion == TRUE)
        wait(condition, mutex);
condicion = TRUE;
...
unlock(mutex);
```

Hilo B

```
lock(mutex);
...
condicion = FALSE;
signal(condition, mutex);
unlock(mutex);
```

Uso de mutex y variables condicionales

- **Un ejemplo…**

  SC: WC unitario con lavabo

  Var. condición: `haya_jabon`

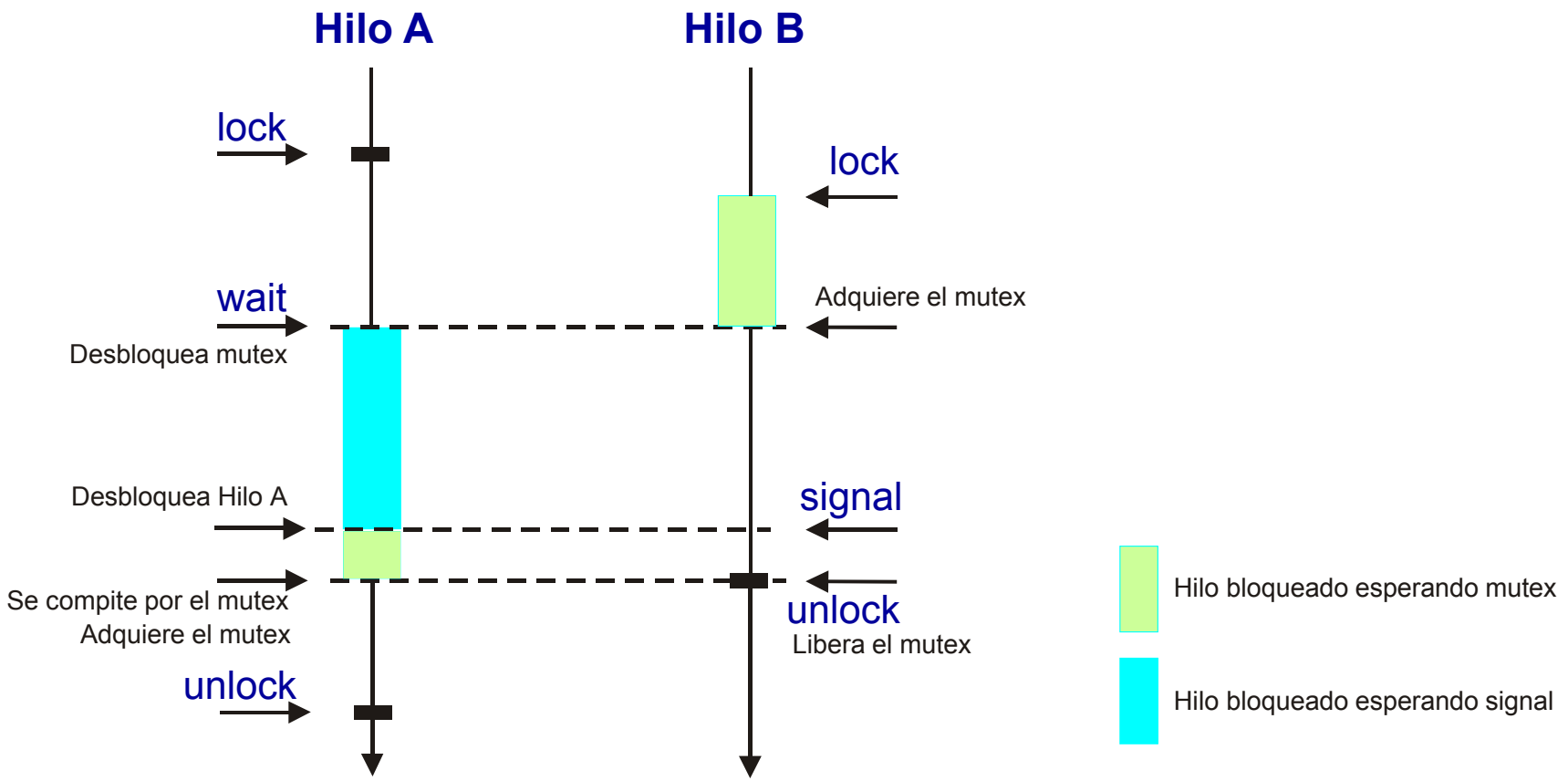
Hilo A e Hilo C

```
lock(mutex);
...
while (gotas_de_jabon_que_quedan==0)
    wait (haya_jabon, mutex);
gotas_de_jabon_que_quedan--;
...
unlock(mutex);
```

Hilo B

```
lock(mutex);
...
gotas_de_jabon_que_quedan= MAX_GOTAS;
signal (haya_jabon, mutex);
unlock(mutex);
```

Uso de mutex y variables condicionales



Uso de mutex y variables condicionales

- **La importancia del WHILE:**
  - Si varios procesos puden hacer que la condición sea cierta, el WHILE es necesario

Hilo A e Hilo C

```
lock(mutex);
...
while (condicion == TRUE)
        wait(condition, mutex);
condicion = TRUE;
...
unlock(mutex);
```

Hilo B

```
lock(mutex);
...
condicion = FALSE;
signal(condition, mutex);
unlock(mutex);
```

Uso de mutex y variables condicionales

UNIVERSITAT JAUME·I

- **La importancia del WHILE:**

```
condicion = TRUE;
```

Hilo A:

```
lock(mutex);
while ... wait
```
→ Hilo A bloqueado

Hilo B:

```
lock(mutex);
condicion = FALSE;
signal
```
→ Hilo A desbloqueado

Hilo C:

```
lock(mutex);
```

Hilo B:

```
unlock(mutex);
```

Hilo C:

```
while ...
```
→ No entra dentro del while

```
condicion = TRUE;
unlock(mutex);
```

Hilo A:

```
while ... wait
```
→ Hilo A bloqueado

Hilo A e Hilo C

```
lock(mutex);
...
while (condicion == TRUE)
        wait(condition, mutex);
condicion = TRUE;
...
unlock(mutex);
```

Hilo B

```
lock(mutex);
...
condicion = FALSE;
signal(condition, mutex);
unlock(mutex);
```

Si hubiese habido un IF → Error

Servicios POSIX sobre mutex

- **Identificación de un mutex en POSIX:**
  - Variable del tipo `pthread_mutex_t`
- **Funciones sobre mutex en POSIX:**

```
int pthread_mutex_init   (pthread_mutex_t * mutex,
                          pthread_mutexattr_t * attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock   (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

devuelven:

- Si todo ha ido bien: 0
- Si error: -1

Inicialización y destrucción de mutex

- **Inicialización de un mutex:**
  - Sintaxis:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t * attr);
```

  - Descripción:
    - Inicializa un mutex identificado a través de `mutex` con los atributos especificados a través de `attr` (atributos por defecto si NULL)
- **Destrucción de un mutex:**
  - Sintaxis:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

  - Descripción:
    - Destruye un mutex identificado a través de `mutex`

Operaciones *lock* y *unlock*

- **Operación `lock` sobre un mutex POSIX:**
  - Sintaxis:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- **Operación `unlock` sobre un mutex POSIX :**
  - Sintaxis:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Servicios POSIX sobre mutex

- **Ejemplo:**

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

int x=0;

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

¿Cuál es la sección crítica en cada hilo?

```
void *fhilo2(void *arg)
{ 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 sobre mutex

- **Ejemplo (solución):**

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

pthread_mutex_t mutex;      /* Mutex que controla acceso a la SC */
int x=0;

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

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

¿Qué pasa si lo ponemos fuera del for?

Servicios POSIX sobre mutex

- **Ejemplo (cont.):**

```
main()
{ pthread_t hilo1, hilo2;
  time_t t;

  pthread_mutex_init(&mutex, NULL);
  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);
  pthread_mutex_destroy(&mutex);
  exit(0);
}
```

Servicios POSIX sobre variables de condición

- **Identificación de una variable de condición en POSIX:**
  - Variable del tipo `pthread_cond_t`
- **Funciones sobre variables de condición en POSIX :**

```
int pthread_cond_init     (pthread_cond_t * cond,
                           pthread_condattr_t * attr);
int pthread_cond_destroy  (pthread_cond_t *cond);
int pthread_cond_wait     (pthread_cond_t * cond,
                           pthread_mutex_t * mutex);
int pthread_cond_signal   (pthread_cond_t * cond);
int pthread_cond_broadcast(pthread_cond_t * cond);
```

devuelven:

- Si todo ha ido bien: 0
- Si error: -1

Inicialización y destrucción de vars. de condición

- **Inicialización de una variable de condición:**
  - Sintaxis:

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *attr);
```

  - Descripción:
    - Inicializa una variable de condición identificada a través de `cond` con los atributos especificados a través de `attr` (atributos por defecto si NULL)
- **Destrucción de una variable de condición:**
  - Sintaxis:

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

  - Descripción:
    - Destruye una variable de condición identificada a través de `cond`

Operación *wait* sobre variables de condición

UNIVERSITAT JAUME·I

- **Operación `wait` sobre una variable de condición:**
  - Sintaxis:

```
int pthread_cond_wait(pthread_cond_t*cond,
                      pthread_mutex_t*mutex);
```

  - Descripción:
    - Suspende al hilo hasta que otro hilo señaliza la variable condicional `cond`
    - Se libera el `mutex` atómicamente
    - Cuando se despierta el hilo vuelve a competir por el `mutex`

Operación *signal* sobre variables de condición

- **Operación `signal` sobre una variable de condición:**
  - Sintaxis:

```
int pthread_cond_signal(pthread_cond_t * cond);
```

  - Descripción:
    - Se reactiva uno de los hilos que están suspendidos en la variable condicional `cond`
    - No tiene efecto si no hay ningún hilo esperando
- **Operación `broadcast` sobre una variable de condición:**
  - Sintaxis:

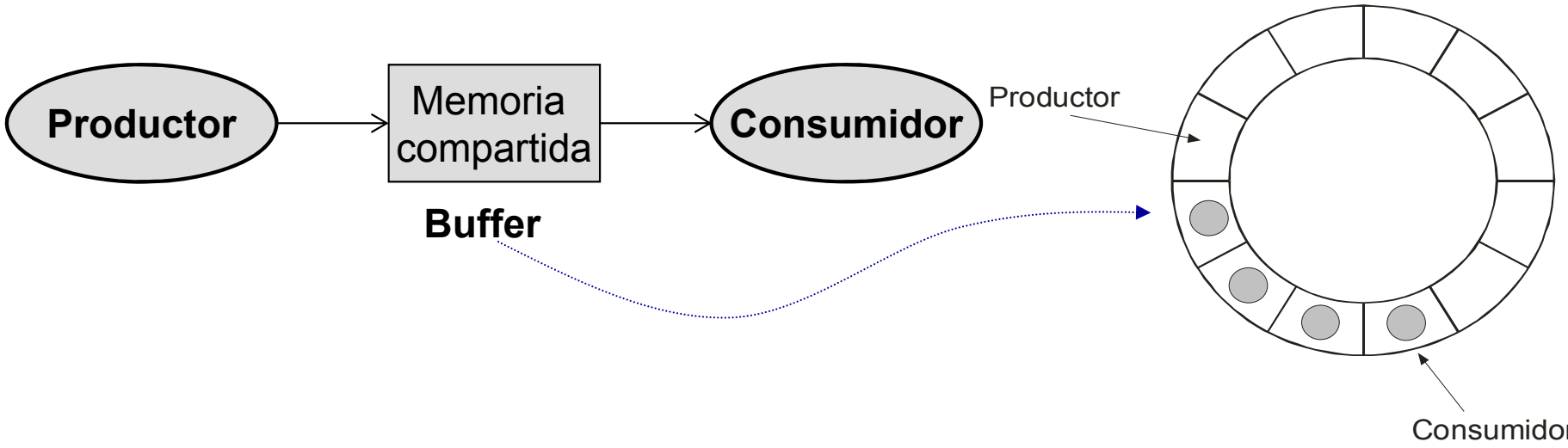
```
int pthread_cond_broadcast(pthread_cond_t * cond);
```

  - Descripción:
    - Todos los hilos suspendidos en la variable condicional `cond` se reactivan
    - No tiene efecto si no hay ningún hilo esperando

Servicios POSIX sobre mutex y vars de condición

- **Ejemplo: El problema del productor-consumidor con buffer limitado (circular)**
  - Planteamiento:
    - El proceso productor produce información y la almacena en un buffer
    - El proceso consumidor accede al buffer y consume la información
    - El productor y el consumidor comparten variables → **Acceso a SC**
    - El productor no puede acceder al buffer si está lleno
    - El consumidor no puede acceder al buffer si está vacío

    → **Sincronización**



Servicios POSIX sobre mutex y vars de condición

UNIVERSITAT JAUME·I

- **Ejemplo (cont.):**

```
#include <pthread.h>
#define MAX_BUFFER          1024      /* Tamaño del buffer */
#define DATOS_A_PRODUCIR    100000    /* Datos a producir */
pthread_mutex_t mutex;                /* Mutex que controla acceso al buffer */
pthread_cond_t no_lleno;              /* Controla el llenado del buffer */
pthread_cond_t no_vacio;              /* Controla el vaciado del buffer */
int n_elementos=0;                    /* Número de elementos en el buffer */
int buffer[MAX_BUFFER];               /* Buffer común */
```

Servicios POSIX sobre mutex y vars de condición

- **Ejemplo (cont.):**

```
void *Productor(void *arg)  {                          /* Código del Productor */
   int dato, i ,pos = 0;
   for(i=0; i < DATOS_A_PRODUCIR; i++ )  {
        dato = i;                                      /* Producir dato */
        pthread_mutex_lock(&mutex);                    /* Acceder al buffer */
        while (n_elementos == MAX_BUFFER)              /* Si buffer lleno */
            pthread_cond_wait(&no_lleno, &mutex);      /* se bloquea */
        buffer[pos] = i;
        pos = (pos + 1) % MAX_BUFFER;
        n_elementos ++;
        pthread_cond_signal(&no_vacio);                /* Buffer no vacío */
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}
```

¿Se puede sustituir por "if"?

¿Hace falta añadir if (n_elementos == 1)?

Servicios POSIX sobre mutex y vars de condición

- **Ejemplo (cont.):**

```
void *Consumidor(void *arg)   {                          /* Código del Consumidor */
int dato, i ,pos = 0;
    for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
        pthread_mutex_lock(&mutex);                      /* Acceder al buffer */
        while (n_elementos == 0)                         /* Si buffer vacío */
            pthread_cond_wait(&no_vacio, &mutex);        /* se bloquea */
        dato = buffer[pos];
        pos = (pos + 1) % MAX_BUFFER;
        n_elementos --;
        pthread_cond_signal(&no_lleno);                  /* Buffer no lleno */
        pthread_mutex_unlock(&mutex);
        printf("Consume %d \n", dato);                   /* Consume dato */
    }
    pthread_exit(0);
}
```

¿Hace falta añadir
if (n_elementos == MAX_BUFFER – 1)?

Servicios POSIX sobre mutex y vars de condición

UNIVERSITAT JAUME·I

- **Ejemplo (cont.):**

```
main(){
    pthread_t th1, th2;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&no_lleno, NULL);
    pthread_cond_init(&no_vacio, NULL);
    pthread_create(&th1, NULL, &Productor, NULL);
    pthread_create(&th2, NULL, &Consumidor, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&no_lleno);
    pthread_cond_destroy(&no_vacio);
    exit(0);
}
```

Servicios POSIX sobre mutex y vars de condición

UNIVERSITAT JAUME·I

- **Ejemplo: El problema del productor-consumidor con buffer limitado (circular)**
  - ¿Cómo modificarías el código anterior para que hubiese:
    - Dos hilos productores y
    - Un hilo consumidor
  - ¿Se podría sustituir el ***while*** del código del productor por un ***if***?
  - ¿Se podría sustituir el ***while*** del código del consumidor por un ***if***?

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 1:**

  Sea el siguiente código que bloquea a un hilo hasta que se cumpla una determinada condición:

```
1: pthread_mutex_lock(&mutex);
2: while (ocupado == true)
3:    pthread_cond_wait(&cond, &mutex);
4: ocupado = true;
5: pthread_mutex_unlock(&mutex);
```

  Y sea el siguiente código que permite desbloquear al hilo que ejecute el código anterior:

```
6: pthread_mutex_lock(&mutex);
7: ocupado = false;
8: pthread_cond_signal(&cond);
9: pthread_mutex_unlock(&mutex);
```

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

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

  Supongamos que el valor de la variable `ocupado` es `true` y que existen dos hilos en el sistema (denotados por A y B).

  (a) Asumiendo que el hilo A ejecuta el primer fragmento de código y el hilo B el segundo, mostrar algunas secuencias de ejecución posibles.

  (b) ¿Es posible la secuencia de ejecución A1 A2 B6 B7 B8 A2 A3 B9, donde Xi denota que el hilo X ejecuta la instrucción i?

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 1 (sol.):**

(a) A1 A2 A3 B6 B7 B8 B9 A2 A4 A5

A1 B6 A2 A3 B7 B8 B9 A2 A4 A5

B6 B7 B8 B9 A1 A2 A4 A5

B6 B7 A1 B8 B9 A2 A4 A5

(b) La secuencia A1 A2 B6 B7 B8 A2 A3 B9 no es posible.

Cuando el hilo B ejecuta la instrucción 6, el mutex está bloqueado.

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 1 (sol.):**

  Sea el siguiente código que bloquea a un hilo hasta que se cumpla una determinada condición:

```
1: pthread_mutex_lock(&mutex);
2: while (ocupado == true)
3:    pthread_cond_wait(&cond, &mutex);
4: ocupado = true;
5: pthread_mutex_unlock(&mutex);
```

A1 A2 A3 B6 B7 B8 B9 A2 A4 A5

A1 B6 A2 A3 B7 B8 B9 A2 A4 A5

B6 B7 B8 B9 A1 A2 A4 A5

B6 B7 A1 B8 B9 A2 A4 A5

  Y sea el siguiente código que permite desbloquear al hilo que ejecute el código anterior:

```
6: pthread_mutex_lock(&mutex);
7: ocupado = false;
8: pthread_cond_signal(&cond);
9: pthread_mutex_unlock(&mutex);
```

A1 A2 B6 B7 B8 A2 A3 B9

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 2:**

  Realizar un programa que cree dos hilos: El primero de ellos generará los 100 primeros números pares y el segundo los 100 primeros números impares, mostrándose todos estos por pantalla con la restricción de que no pueden aparecen dos números pares seguidos o dos números impares seguidos.

  Utilizar mutex y variables de condición como herramienta de sincronización.

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 2 (sol.):**

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

#define MAX      200

pthread_mutex_t  mutex;
pthread_cond_t   ImprPar, ImprImpar;
int              continuar_Par = 1;

void *GeneraImpares(void *arg)
{ int i;

  for (i=1; i<MAX; i=i+2)
  { pthread_mutex_lock(&mutex);
    while (continuar_Par)
      pthread_cond_wait(&ImprImpar,&mutex);
    printf("(Hilo %d): %d\n",
           pthread_self(),i);
    continuar_Par=1;
    pthread_cond_signal(&ImprPar);
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}
```

¿Qué pasa si lo ponemos fuera del for?

¿Qué pasa si lo ponemos fuera del mutex?

¿Se puede sustituir por broadcast?

```
void *GeneraPares(void *arg)
{ int i;

  for (i=0; i<MAX; i=i+2)
  { pthread_mutex_lock(&mutex);
    while (!continuar_Par)
      pthread_cond_wait(&ImprPar,&mutex);
    printf("(Hilo %d): %d\n",
           pthread_self(),i);
    continuar_Par=0;
    pthread_cond_signal(&ImprImpar);
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}
```

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 2 (sol.):**

```
main()
{
  pthread_t Pares, Impares;

  pthread_mutex_init(&mutex,NULL);
  pthread_cond_init(&ImprPar,NULL);
  pthread_cond_init(&ImprImpar,NULL);

  pthread_create(&Pares, NULL, &GeneraPares, NULL);
  pthread_create(&Impares, NULL, &GeneraImpares, NULL);

  pthread_join(Pares,NULL);
  pthread_join(Impares,NULL);

  pthread_cond_destroy(&ImprPar);
  pthread_cond_destroy(&ImprImpar);
  pthread_mutex_destroy(&mutex);
  exit(0);

}
```

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

¿Es correcta la siguiente solución?

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

#define MAX      200

pthread_mutex_t  mutex;
int              continuar_Par = 1;

void *GeneraImpares(void *arg)
{ int i;

  for (i=1; i<MAX; i=i+2)
  {
    pthread_mutex_lock(&mutex);
    while (continuar_Par) ;
    printf("(Hilo %d): %d\n",
           pthread_self(),i);
    continuar_Par=1;
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}
```

```
void *GeneraPares(void *arg)
{ int i;

  for (i=0; i<MAX; i=i+2)
  {
    pthread_mutex_lock(&mutex);
    while (!continuar_Par) ;
    printf("(Hilo %d):
           %d\n",pthread_self(),i);
    continuar_Par=0;
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}
```

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 2 (sol.):**

```
main()
{
  pthread_t Pares, Impares;

  pthread_mutex_init(&mutex,NULL);

  pthread_create(&Pares, NULL, &GeneraPares, NULL);
  pthread_create(&Impares, NULL, &GeneraImpares, NULL);

  pthread_join(Pares,NULL);
  pthread_join(Impares,NULL);

  pthread_mutex_destroy(&mutex);
  exit(0);

}
```

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 3:**

  Implementar un programa en lenguaje de programación C que cree *MaxA* hilos de tipo *A* y *MaxB* hilos de tipo *B*. Todos los hilos pueden acceder a una serie de datos comunes con la restricción de que, en un instante determinado, únicamente pueden acceder a los datos hilos del mismo tipo. Utilizar mutex y variables de condición como herramienta de sincronización.

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 3 (cont.):** Ejemplo de una posible salida del programa:

```
(A: 1 0 16386) Accedo a datos
(A: 1 0 16386) Dejo acceso a datos

(B: 0 1 131081) Accedo a datos
(B: 0 2 163851) Accedo a datos
(B: 0 3 114696) Accedo a datos
(B: 0 3 131081) Dejo acceso a datos
(B: 0 3 98311) Accedo a datos
(B: 0 3 163851) Dejo acceso a datos
(B: 0 2 114696) Dejo acceso a datos
(B: 0 2 147466) Accedo a datos
(B: 0 2 147466) Dejo acceso a datos
(B: 0 1 98311) Dejo acceso a datos

(A: 1 0 32771) Accedo a datos
(A: 1 0 32771) Dejo acceso a datos
(A: 1 0 49156) Accedo a datos
(A: 2 0 65541) Accedo a datos
(A: 3 0 81926) Accedo a datos
(A: 3 0 81926) Dejo acceso a datos
(A: 2 0 49156) Dejo acceso a datos
(A: 1 0 65541) Dejo acceso a datos
```

PID del hilo

Nº procesos tipo B accediendo a datos

Nº procesos tipo A accediendo a datos

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 3 (sol.):**

```
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#define MaxA    10
#define MaxB    10

int nA=0, nB=0;
pthread_mutex_t m;
pthread_cond_t  noprocsA;
pthread_cond_t  noprocsB;
void *f_hiloA(void *arg);
void *f_hiloB(void *arg);
```

```
main()
{
 pthread_t hiloA[MaxA], hiloB[MaxB];
 int i;
 time_t t;

 srandom(time(&t));
 pthread_mutex_init(&m, NULL);
 pthread_cond_init(&noprocsA,NULL);
 pthread_cond_init(&noprocsB,NULL);

 for (i=0; i<MaxA; i++)
    pthread_create(&hiloA[i], NULL, &f_hiloA, NULL);
 for (i=0; i<MaxB; i++)
    pthread_create(&hiloB[i], NULL, &f_hiloB, NULL);

 for (i=0; i<MaxA; i++) pthread_join(hiloA[i], NULL);
 for (i=0; i<MaxB; i++) pthread_join(hiloB[i], NULL);

 pthread_cond_destroy(&noprocsA);
 pthread_cond_destroy(&noprocsB);
 pthread_mutex_destroy(&m);
 exit(0);
}
```

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 3 (sol.):**

```
void *f_hiloA (void *arg)
{   sleep(random()%5);
    pthread_mutex_lock(&m);
    while (nB > 0) pthread_cond_wait(&noprocsB,&m);
    nA++;
    pthread_mutex_unlock(&m);
    /*Acceso a datos */
    printf("(A: %d %d %d) Accedo a datos\n",nA,nB,pthread_self());
    sleep(random()%3);
    printf("(A: %d %d %d) Dejo acceso datos\n",nA,nB,pthread_self());
    /* Fin acceso datos */
    pthread_mutex_lock(&m);
    nA--;
    if (nA == 0)
      {pthread_cond_broadcast(&noprocsA);
       printf("\n");}
    pthread_mutex_unlock(&m);
    pthread_exit(0);
}
```

¿Se puede sustituir por signal?

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 3 (sol.):**

```
void *f_hiloB (void *arg)
{  sleep(random()%5);
   pthread_mutex_lock(&m);
   while (nA > 0) pthread_cond_wait(&noprocsA,&m);
   nB++;
   pthread_mutex_unlock(&m);
   /*Acceso a datos */
   printf("(B: %d %d %d) Accedo a datos\n",nA,nB,pthread_self());
   sleep(random()%3);
   printf("(B: %d %d %d) Dejo acceso a datos\n",nA,nB,pthread_self());
   /* Fin acceso datos */
   pthread_mutex_lock(&m);
   nB--;
   if (nB == 0)
     {pthread_cond_broadcast(&noprocsB);
      printf("\n");}
   pthread_mutex_unlock(&m);
   pthread_exit(0);
}
```

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 4:**

  Dado el código que aparece a continuación, indicar razonadamente la veracidad de las siguientes afirmaciones:

  a) hilo 1 e hilo2 nunca ejecutarán `pthread_cond_signal(&cond)`.

  b) Antes de la instrucción `pthread_mutex_destroy(&mutex)` sería necesaria la instrucción `pthread_cancel(hilo3)`.

  c) Un hilo que ejecute la instrucción `pthread_cond_wait(&cond2,&mutex)` nunca saldrá de la situación de bloqueo y será eliminado por el hilo principal con la correpondiente instrucción `pthread_cancel`.

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 4 (cont.):**

  d) El funcionamiento del programa sería el mismo si eliminamos de la función `f_hilo1y2` la línea:

```
if (cont == MAXCONT) pthread_cond_wait(&cond2,&mutex);
```

  e) El programa no finaliza nunca.

  f) La variable `cont` nunca tendrá un valor superior a 20.

  g) El funcionamiento del programa sería diferente si en la función `f_hilo1y2` se pusiese la instrucción `pthread_mutex_lock` justo antes del `for` y la instrucción `pthread_mutex_unlock` justo después del `for`.

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 4 (cont.):**

```
#include <pthread.h>
#define MAXCONT 20
#define MAXITER 25
int cont=0;
pthread_mutex_t mutex; pthread_cond_t  cond, cond2;

void *f_hilo1y2(void *arg)
{ int i;
  for (i=0; i<MAXITER; i++)
  { pthread_mutex_lock(&mutex);                    /* Apartado G */
    if (cont == MAXCONT)
       pthread_cond_wait(&cond2,&mutex);           /* Apartados C y D */
    cont=cont+1;
    if (cont == MAXCONT) {
       pthread_cond_signal(&cond);                 /* Apartado A */
       printf("Hilo %d \n",pthread_self());
    }
    pthread_mutex_unlock(&mutex);                  /* Apartado G */
  }
  pthread_exit(0);
}
```

Mutex y variables de condición: Ejercicios

UNIVERSITAT JAUME·I

- **Ejercicio 4 (cont.):**

```
void *f_hilo3(void *arg)
{ pthread_mutex_lock(&mutex);
  while (cont < MAXCONT) pthread_cond_wait(&cond, &mutex);
  pthread_mutex_unlock(&mutex);
  pthread_exit(0);
}
main()
{ pthread_t hilo1, hilo2, hilo3;
  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&cond,NULL); pthread_cond_init(&cond2,NULL);

  pthread_create(&hilo1, NULL, f_hilo1y2, NULL);
  pthread_create(&hilo2, NULL, f_hilo1y2, NULL);
  pthread_create(&hilo3, NULL, f_hilo3,   NULL);

  pthread_join(hilo3,NULL);

  pthread_cancel(hilo1);
  pthread_cancel(hilo2);

  pthread_mutex_destroy(&mutex);                    /* Apartado B */
  pthread_cond_destroy(&cond);
  pthread_cond_destroy(&cond2);
  exit(0);
}
```

UNIVERSITAT JAUME·I

Índice

- Procesamiento concurrente
- El problema de la sección crítica
- Problemas clásicos de comunicación y sincronización
- Mutex y variables de condición
- Tuberías

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
  - …
  - Describir el funcionamiento de una tubería como una posible solución al problema de sincronización y comunicación entre procesos
  - Desarrollar programas en los que se plantea la necesidad de sincronización y comunicación de procesos y proponer una solución utilizando tuberías

UNIVERSITAT JAUME·I

Bibliografía por secciones

- Procesamiento concurrente
- El problema de la sección crítica
- Problemas clásicos de comunicación y sincronización
- Mutex y variables de condición

} ***Carretero***

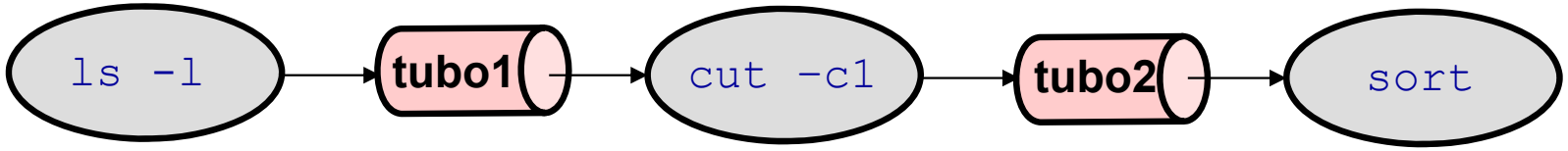
- Tuberías

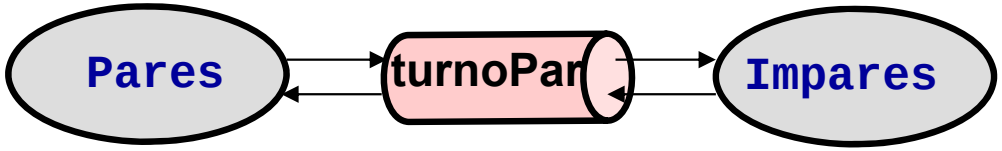
} ***Badía et al.*** y ***Carretero***

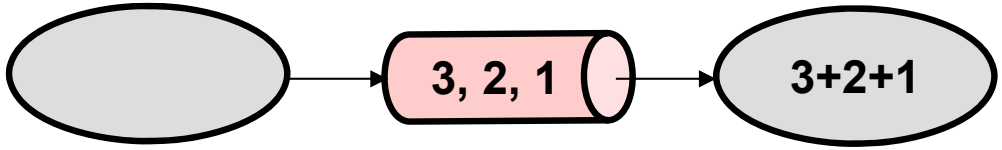
Tuberías

- Mecanismo de comunicación y sincronización

```
ls -l | cut -c1 | sort
```





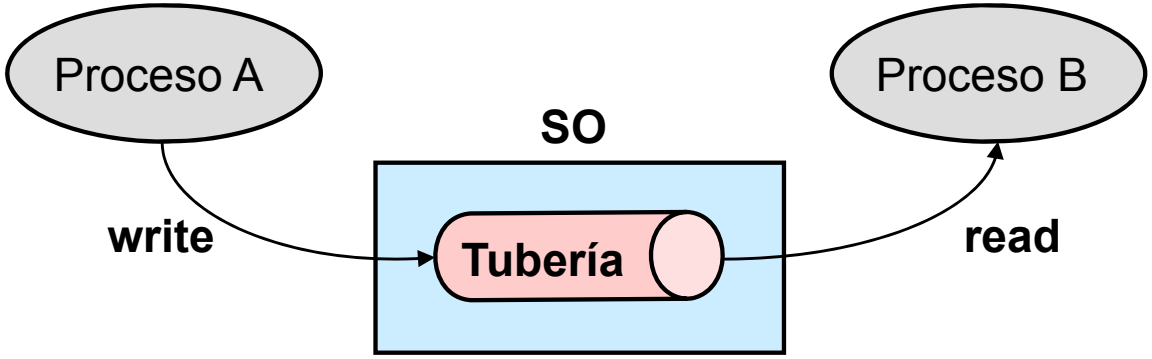


Tuberías

- “Pseudo-archivo” manejado por el SO.
- Cada proceso ve a la tubería como un conducto con dos extremos:
  - Uno para escribir o insertar datos.
  - Otro para leer o extraer datos.
- La lectura/escritura de datos se realiza mediante los servicios de lectura/escritura de archivos.

Tuberías

- Flujo de datos unidireccional y FIFO.



- Para un flujo de datos bidireccional hay que crear dos tuberías.

Tuberías

- **Tipos de tuberías:**
  - Sin nombre: Sólo pueden usarla procesos que desciendan del proceso que la creó.
  - Con nombre: Pueden usarla procesos independientes.
- **Servicios POSIX:**
  - Creación de tuberías sin nombre:

```
int pipe (int tub[2]);
```

  - Creación de tuberías con nombre:

```
int mkfifo (char *fifo, mode_t mode);
```

Tuberías

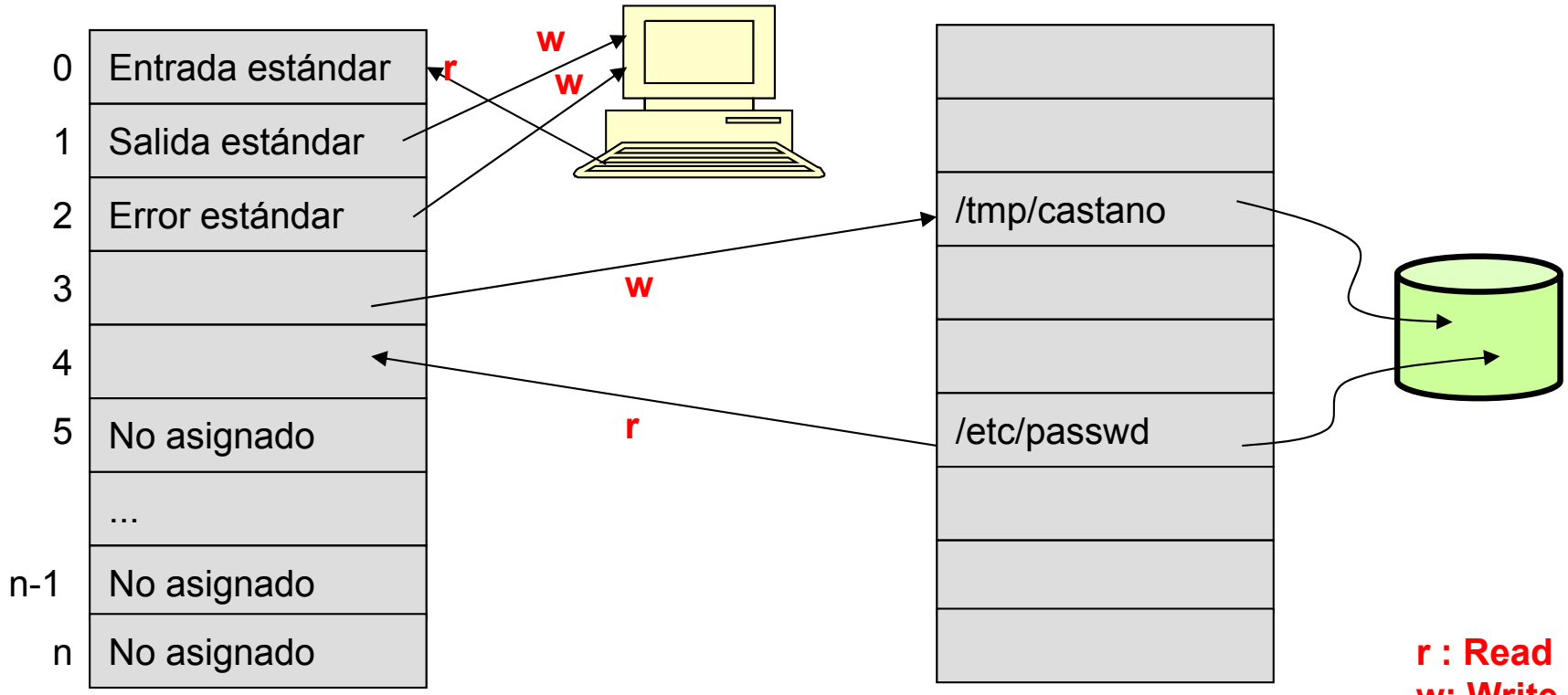
- **Escritura en una tubería:**
  - Introduce datos en orden FIFO.
  - Operación atómica.
  - Si la tubería está llena o se llena durante la escritura, la operación bloquea al proceso escritor hasta que se pueda completar.
  - Si no hay ningún proceso con la tubería abierta para lectura, la operación devuelve error.

Tuberías

- **Lectura de una tubería:**
  - Obtiene datos (y los elimina de la tubería).
  - Operación atómica.
  - Si la tubería está vacía, la llamada bloquea al proceso en la operación de lectura hasta que algún proceso escriba datos en la misma.
  - Si la tubería almacena *M* bytes y se quieren leer *n* bytes:
    - Si $M \geq n$, se devuelven y eliminan *n* bytes.
    - Si $M < n$, se devuelven y eliminan *M* bytes.
  - Si no hay escritores y la tubería está vacía, la operación devuelve fin de archivo.

Conceptos previos

- **Tabla de descriptores de ficheros (tabla de ficheros abiertos por el proceso):**
  - Vector que apunta a los ficheros utilizados por un proceso
  - Cada proceso tiene su propia tabla de descriptores de ficheros



T. ficheros abiertos por proceso

T. ficheros abiertos en el sistema

Conceptos previos

- **Apertura de un fichero:**
  - Devuelve un **descriptor de fichero** (índice entero a la tabla de descriptores de ficheros) que identifica al fichero
  - Al crear un proceso hay 3 descriptores de fichero abiertos por defecto:
    - Entrada estándar: Descriptor 0
    - Salida estándar: Descriptor 1
    - Error estándar: Descriptor 2

Conceptos previos

- **Puntero de L/E de un fichero:**
  - Característica de un fichero, no de un proceso
  - Se guarda en el tabla de ficheros abiertos en el sistema

Servicios POSIX para gestión de ficheros

- **Abrir un fichero:** `open`
- **Cerrar un fichero:** `close`
- **Crear un fichero:** `creat`
- **Leer datos de un fichero:** `read`
- **Escribir datos en un fichero:** `write`
- **Duplicar un descriptor de fichero:** `dup`

Función *open*

- **Sintaxis:**

```
int open(const char *nombre, int modo, mode_t permisos);
```

devuelve:

- Un descriptor de fichero
- Si error: -1

Parámetro `modo`:

- O_RDONLY (0)
- O_WRONLY (1)
- O_RDWR (2)
- O_CREAT
- O_TRUNC
- O_APPEND
- O_EXCL

- **Ejemplos:**
  - `fd = open ("prueba",1);`
  - `fd = open ("prueba",O_WRONLY|O_CREAT|O_TRUNC,0666);`
  - `fd = open ("prueba",O_WRONLY|O_CREAT|O_APPEND,0666);`
  - `fd = open ("prueba",O_WRONLY|O_CREAT|O_EXCL,0666);`

Función *close*

- **Sintaxis:**

```
int close(int desc_fich);
```

  devuelve:

  - Si todo ha ido bien: 0
  - Si error: -1

- **Ejemplo:**

```
close (fd);
```

Función *creat*

- **Sintaxis:**

```
int creat(const char *nombre, mode_t permisos);
```

devuelve:

  - Un descriptor de fichero
  - Si error: -1

- **Ejemplos:**
  - `fd = creat ("prueba",0600);`
  - `fd = creat ("/usr/castano/prueba",0666);`

Función *read*

- **Sintaxis:**

```
int read(int desc_fich, void *dato, size_t n_bytes);
```

  devuelve:

  - El número de bytes que ha podido leer
  - Si error: -1

- **Descripción:**
  - Lee los `n_bytes` bytes siguientes a la posición actual del puntero del fichero con descriptor `desc_fich` y los almacena a partir de la posición de memoria `dato`
  - Modifica el valor del puntero del fichero

- **Ejemplo:**
  - `n = read (fd,&i,sizeof(i));`

Función *write*

- **Sintaxis:**

```
int write(int desc_fich, const void *dato, size_t n_bytes);
```

devuelve:

  - El número de bytes escritos
  - Si error: -1

- **Descripción:**
  - En el fichero con descriptor `desc_fich` se escriben los `n_bytes` bytes que hay en memoria central a partir de la posición `dato`
  - Modifica el valor del puntero del fichero

Siempre que no se produzca error

- **Ejemplo:**
  - `n = write (fd,"Esto es el dato del fichero\0",28);`

Ejemplo

- **Ejemplo 1:**

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

int main()
{
  int i, fd, dato, vector[10];

  fd=creat("prueba",0600);                          /* (a) */
  for (i=0;i<10;i++) vector[i]=i;                   /* (b) */
  write(fd,vector,sizeof(vector));                  ≡ (a)
  close(fd);                                        /* (c) */

  fd=open("prueba",O_RDONLY);                       /* (d) */
  while (read(fd,&dato,sizeof(int))>0)
  {   printf("Leido el numero %d\n",dato); }
  close(fd);                                        /* (c) */
  exit(0);
}
```

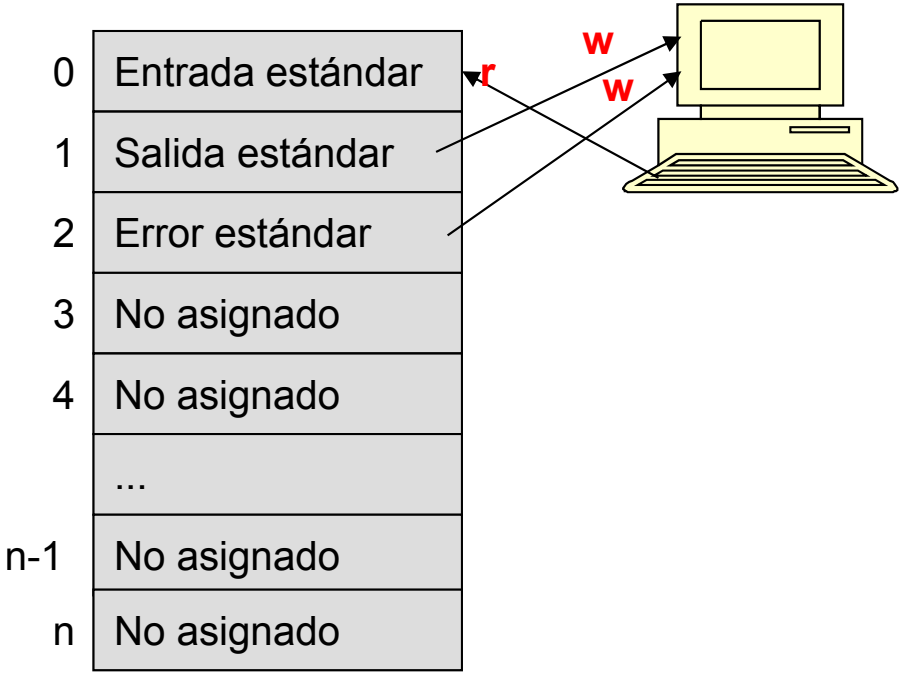
O también:

```
for (i=0;i<10;i++)
  write(fd,&i,sizeof(i));
```

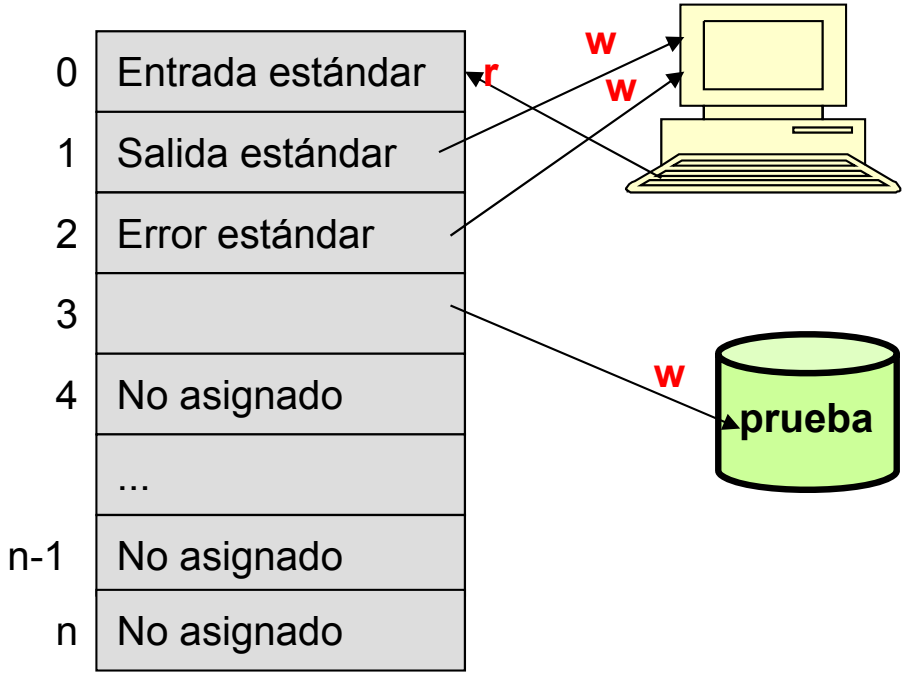
Ejemplo

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

Evolución de la tabla de descriptores de ficheros



(a)

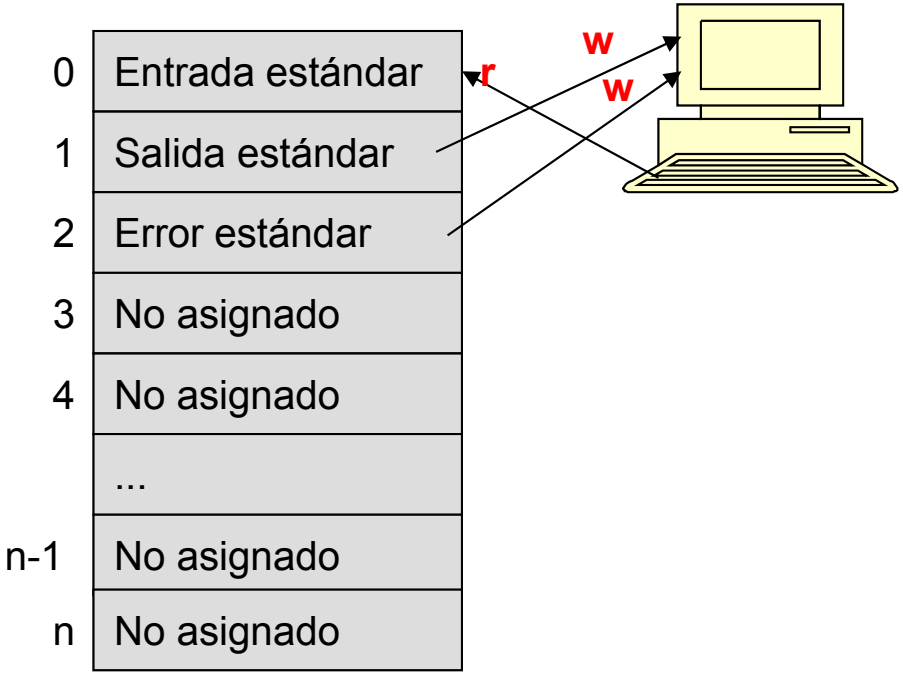


(b)

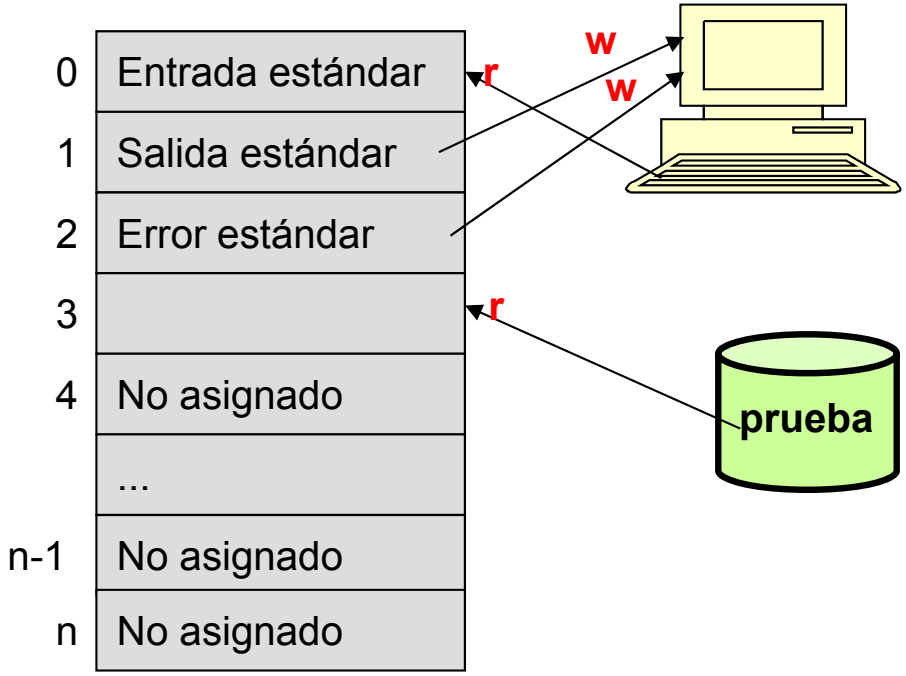
Ejemplo

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

Evolución de la tabla de descriptores de ficheros (cont.)



(c)



(d)

Función *dup*

- **Sintaxis:**

  ```
  int dup(int desc_fich);
  ```

  devuelve:

  - El nuevo descriptor de fichero para el fichero con descriptor `desc_fich`
  - Si error: -1

- **Descripción:**
  - Duplicación de un descriptor de fichero
  - Dos puntos de acceso al mismo fichero
    - Puntero de fichero único (compartido por ambos descriptores)

- **Ejemplo:**
  - `fd2 = dup (fd1);`

Ejemplo

- **Ejemplo 2:**

```
$ cat dup1.c
#include<unistd.h>
#include<fcntl.h>
#include<stdio.h>

int main()
{
   int fichero, duplicado, i, dato, salir;

   fichero = creat("prueba",0644);
   for (i=0;i<10;i++) write(fichero,&i,sizeof(int));
   close(fichero);

   fichero = open("prueba",O_RDONLY);    /* (a) */
   printf("Abierto el fichero con descriptor %d\n",fichero);
   duplicado=dup(fichero);               /* (b) */
   printf("Duplicado sobre descriptor %d\n",duplicado);
```

Ejemplo

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

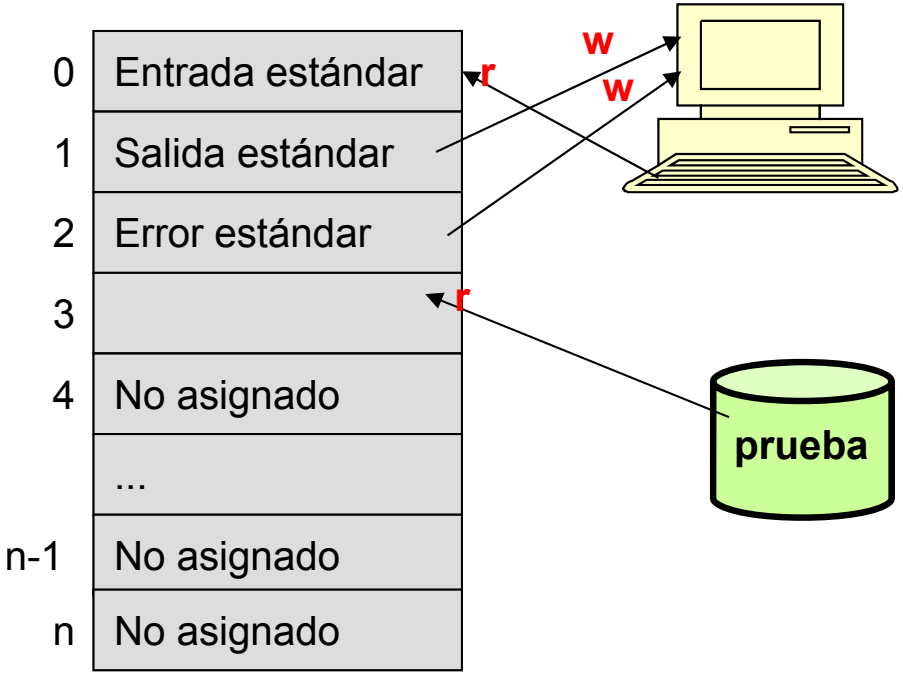
```
if (read(…)==0) salir=1;
else            salir=0;
```

```
salir=0;
while (!salir)
{
  salir = (read(fichero,&dato,sizeof(int))==0);
  if (!salir)
  {
    printf("Dato leido mediante fichero = %d\n",dato);
    salir = (read(duplicado,&dato,sizeof(int))==0);
    if (!salir)
    {   printf("Dato leido mediante duplicado = %d\n",dato);
    } else {
        printf("Final de fichero encontrado en duplicado\n");
    }
  } else {printf("Final de fichero encontrado en fichero\n");}
}
close(fichero);
close(duplicado);
exit(0);
}
```

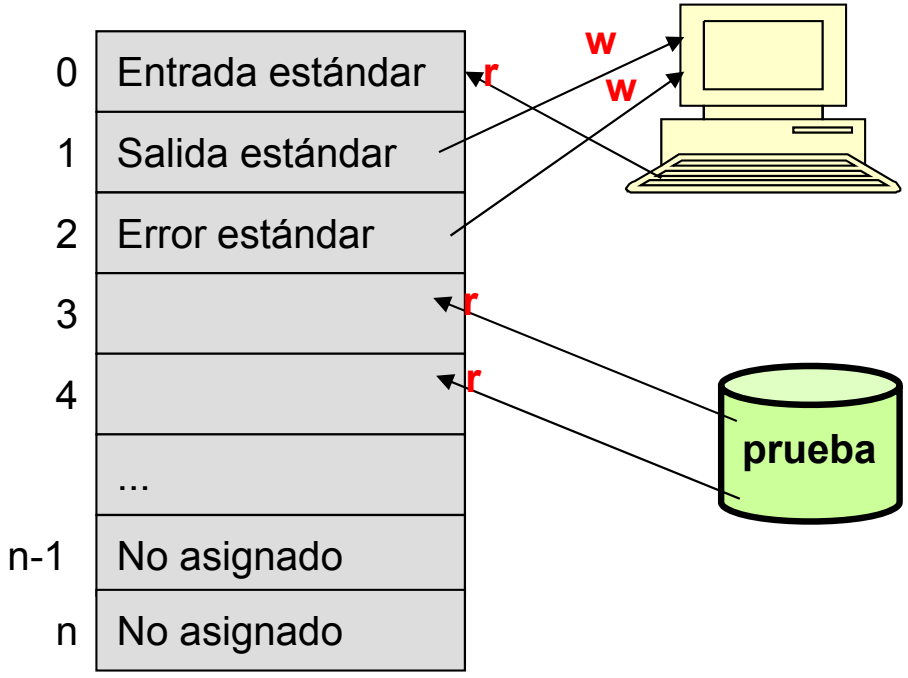
Ejemplo

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

Evolución de la tabla de descriptores de ficheros



(a) fichero=3



(b) fichero=3
duplicado=4

Ejemplo

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

```
$ make dup1
$ ./dup1
Ya se ha creado el fichero
Abierto el fichero con descriptor 3
Duplicado sobre descriptor 4
Dato leido mediante fichero = 0
Dato leido mediante duplicado = 1
Dato leido mediante fichero = 2
Dato leido mediante duplicado = 3
Dato leido mediante fichero = 4
Dato leido mediante duplicado = 5
Dato leido mediante fichero = 6
Dato leido mediante duplicado = 7
Dato leido mediante fichero = 8
Dato leido mediante duplicado = 9
Final de fichero encontrado en fichero
$
```

Función *pipe*

- **Sintaxis:**

  ```
  int pipe(int descriptor[2]);
  ```

  devuelve:

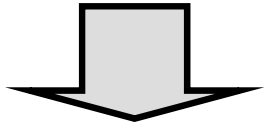
  - Si todo ha ido bien: 0
  - Si error: -1

- **Descripción:**
  - Crea una tubería sin nombre
    - Lectura de datos de la tubería vía `descriptor[0]`
    - Escritura de datos en la tubería vía `descriptor[1]`

- **Ejemplo:**
  - ```
    int tuberia[2];
    pipe (tuberia);
    ```
    ```

Función *pipe*

- **Lectura de una tubería:**
  - `read` bloquea al proceso que la realiza hasta que:
    - Existe algún dato por leer ó
    - Se ha alcanzado el final de fichero
- **Fin de fichero de una tubería:**
  - Cuando se cierran todos los descriptores de entrada (escritura) a esa tubería



Deben cerrarse los descriptores de tubería que no vayan a ser usados

- **Comunicación de padre e hijo mediante una tubería:**
  - `pipe` antes de `fork`

Ejemplos

- **Ejemplo 3:**

¿Cuál es la salida por pantalla si argv[1] es 3?

```
#include <stdio.h>

int main(int argc, char *argv[])
{ int tubo[2], status, i, dato, suma;

  pipe(tubo);                    /* (a) */
  if (fork()!=0)                 /* (b) */

  { close(tubo[0]);              /* (c) */
    dato=atoi(argv[1]);
    for (i=1;i<=dato;i++) write(tubo[1],&i,sizeof(i));
    close(tubo[1]);
    wait(&status);
    exit(0);

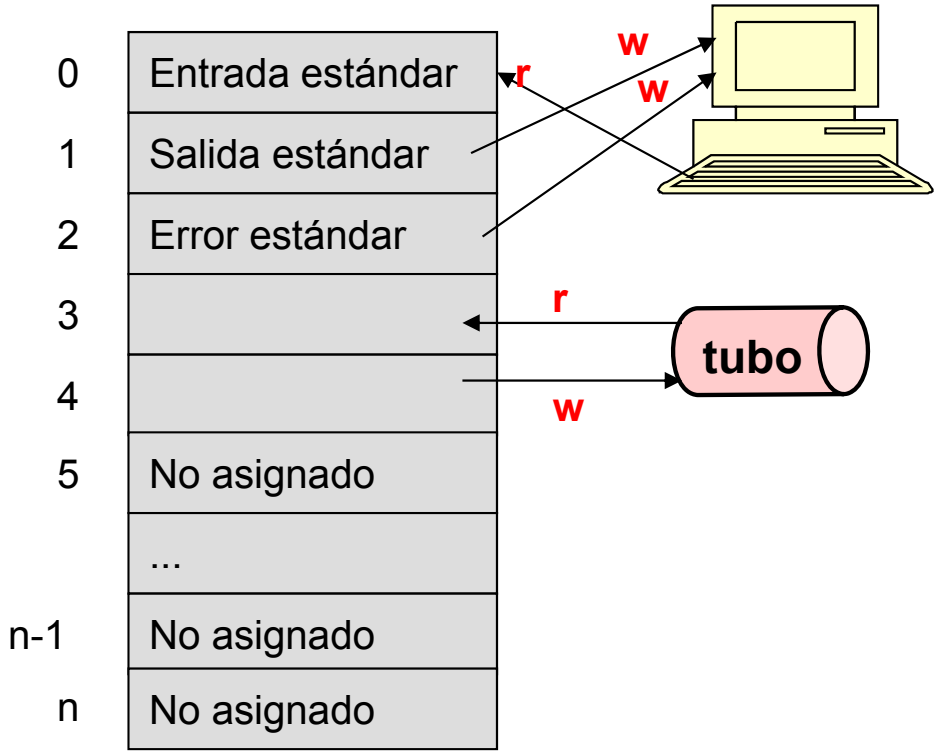
  } else {

    close(tubo[1]);              /* (c) */
    suma = 0;
    while (read(tubo[0],&dato,sizeof(int))>0) suma = suma + dato;
    close(tubo[0]);
    printf("La suma es: %d \n",suma);
    exit(0);
  }
}
```

Ejemplos

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

Evolución de la tabla de descriptores de ficheros

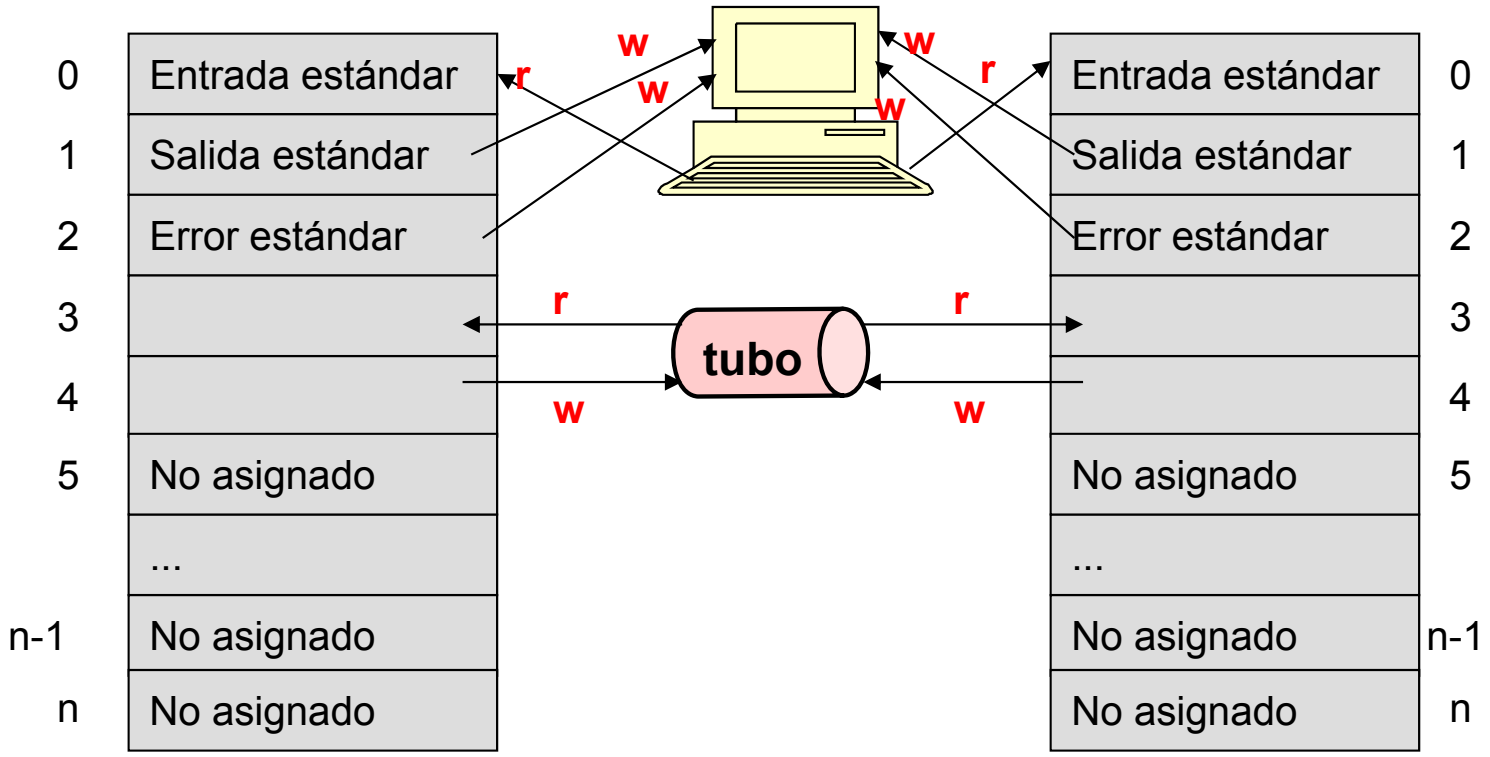


(a)

Ejemplos

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

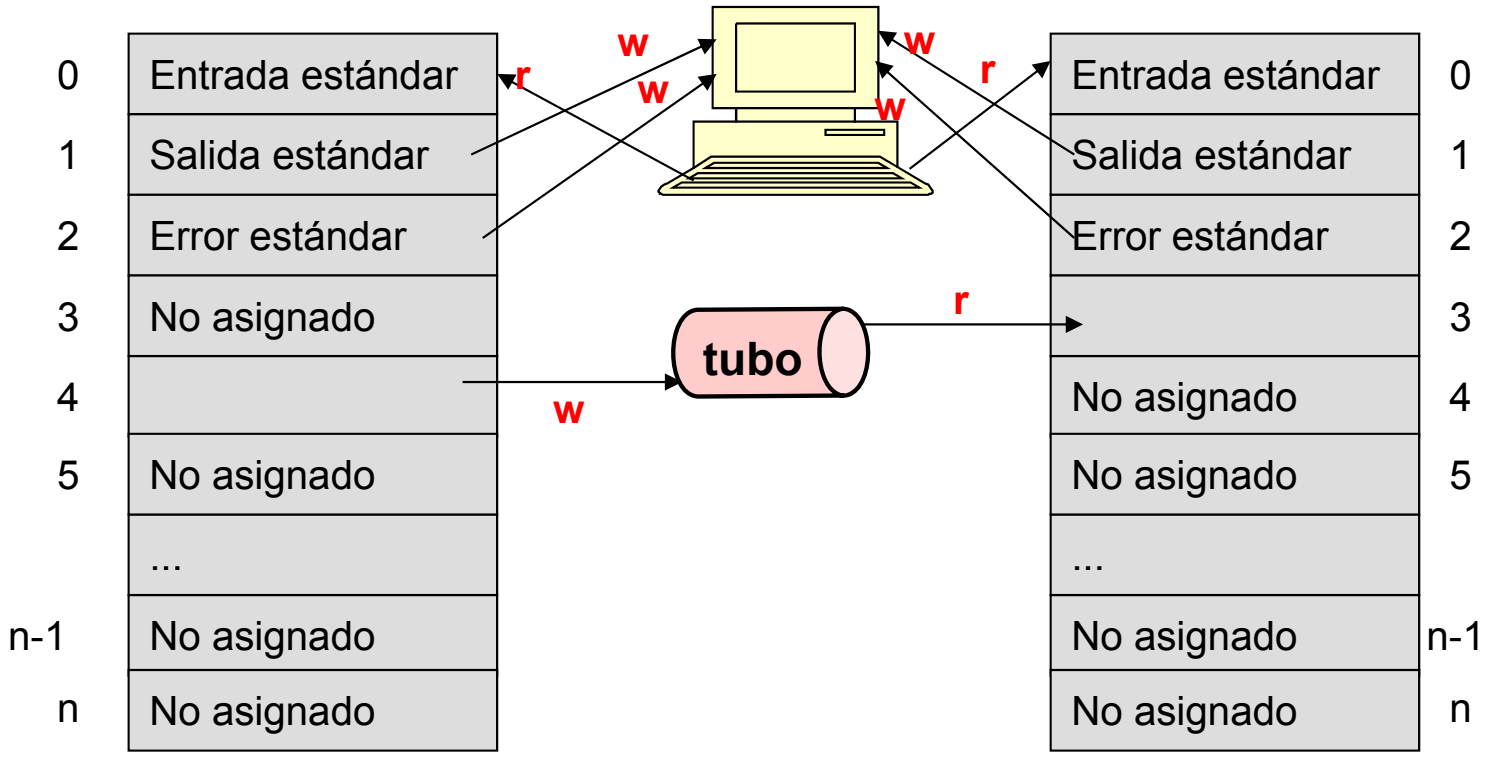
Evolución de la tabla de descriptores de ficheros (cont.)



Ejemplos

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

Evolución de la tabla de descriptores de ficheros (cont.)



(c)

Ejemplos

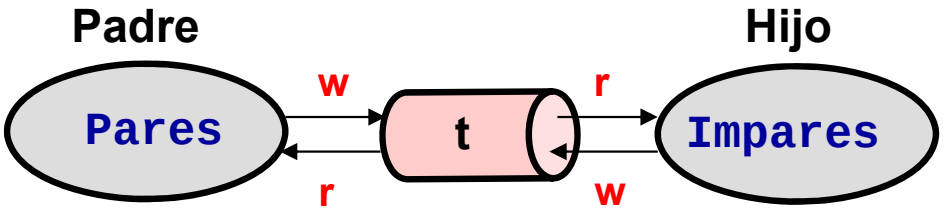
- **Ejemplo 4:**

  Realizar un programa que cree un proceso hijo. A continuación, el proceso padre generará los 100 primeros números pares y el hijo los 100 primeros números impares, mostrándose todos estos por pantalla con la restricción de que no pueden aparecen dos números pares seguidos o dos números impares seguidos.

  Utilizar tuberías como herramienta de sincronización.

Ejemplos

- **Ejemplo 4 (cont.): Usando una única tubería**




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

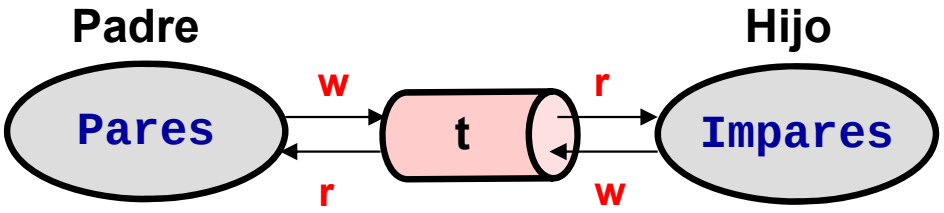
#define MAX  200

main()
{  int  i, turno_par, t[2];

pipe(t);
```

Ejemplos

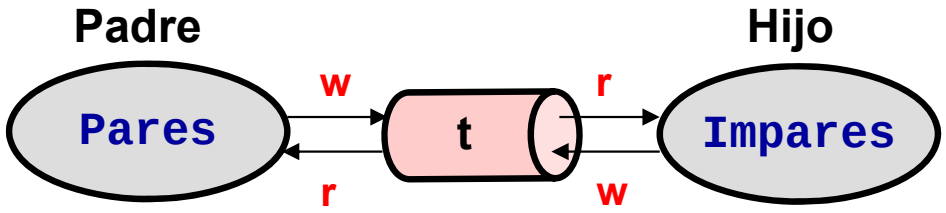
- **Ejemplo 4 (cont.): Usando una única tubería**




```
turno_par = 1;
write (t[1],&turno_par, sizeof(int));
if (fork() != 0)
{ i=0;          /* El padre imprime los pares */
  while (i <= MAX)
  { read (t[0],&turno_par, sizeof(int));
    if (turno_par)
        {   printf("Proceso %d: %d\n", getpid(), i);
            i = i + 2;
            turno_par=0;
         }
     write (t[1],&turno_par, sizeof(int));
  }
  close(t[0]);
  close(t[1]);
  wait(NULL);
} else {
```

Ejemplos

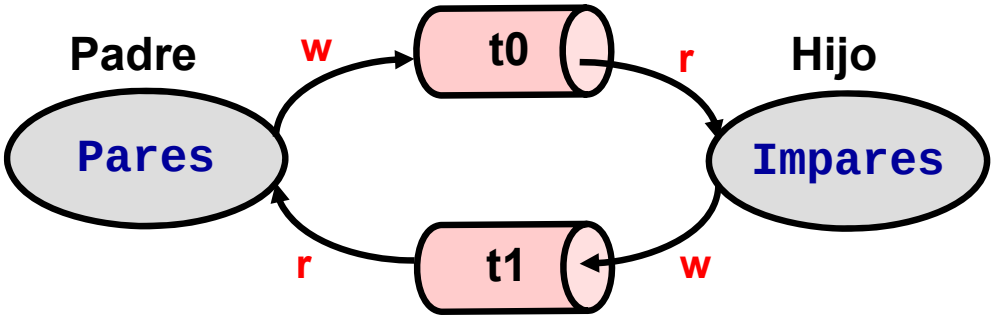
- **Ejemplo 4 (cont.): Usando una única tubería**




```
/* El hijo imprime los impares */
 i=1;
 while (i <= MAX)
 {  read (t[0],&turno_par, sizeof(int));
    if (!turno_par)
    {    printf("Proceso %d: %d\n", getpid(), i);
         i = i + 2;
         turno_par=1;
         write (t[1],&turno_par, sizeof(int));
    } else
         write (t[1],&turno_par, sizeof(int));
 }
 close(t[0]);
 close(t[1]);
}
exit(0);
}
```

Ejemplos

- **Ejemplo 4 (cont.): Usando dos tuberías**




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

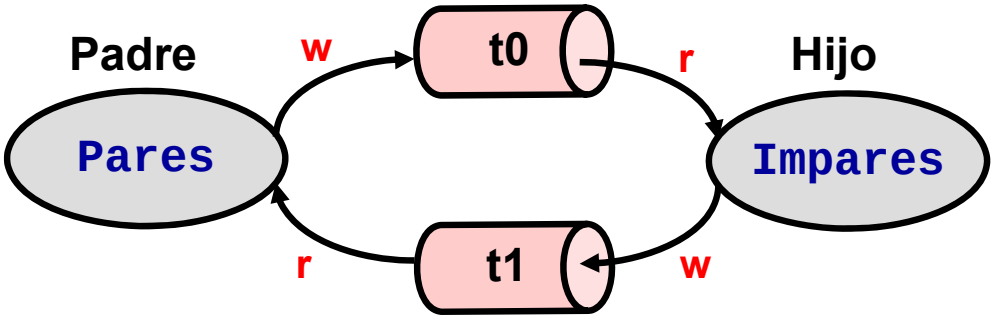
#define MAX  200

main()
{int  i, testigo= 1, t0[2], t1[2];

pipe(t0);
pipe(t1);
```

Ejemplos

- **Ejemplo 4 (cont.): Usando dos tuberías**

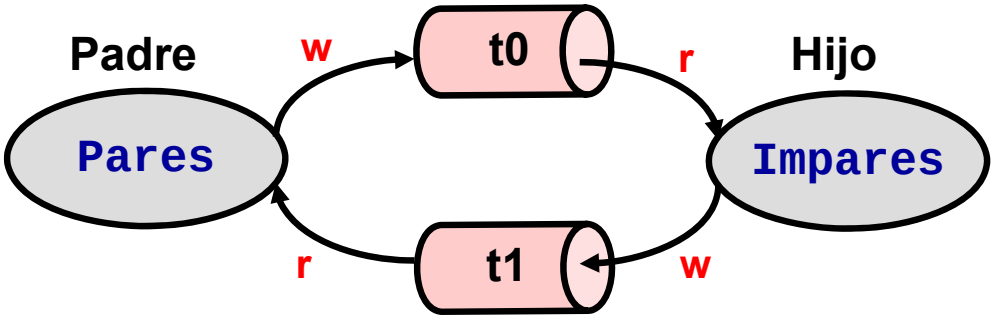



```
write (t1[1],&testigo, sizeof(int));
if (fork() != 0)
{ /* El padre imprime los pares */
  close(t0[0]);
  close(t1[1]);
  for (i=0; i<MAX; i=i+2)
  {  read (t1[0],&testigo, sizeof(int));
     printf("Proceso %d: %d\n", getpid(), i);
     write (t0[1],&testigo, sizeof(int));
  }
  close(t0[1]);
  close(t1[0]);
  wait(NULL);

} else {
```

Ejemplos

- **Ejemplo 4 (cont.): Usando dos tuberías**



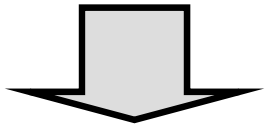

```
    /* El hijo imprime los impares */
    close(t0[1]);
    close(t1[0]);
    for (i=1; i<MAX; i=i+2)
    {  read (t0[0],&testigo, sizeof(int));
       printf("Proceso %d: %d\n", getpid(), i);
       write (t1[1],&testigo, sizeof(int));
    }
    close(t1[1]);
    close(t0[0]);
}
exit(0);
}
```

Resolución de línea de comandos del shell

- **Ejemplo 5:**

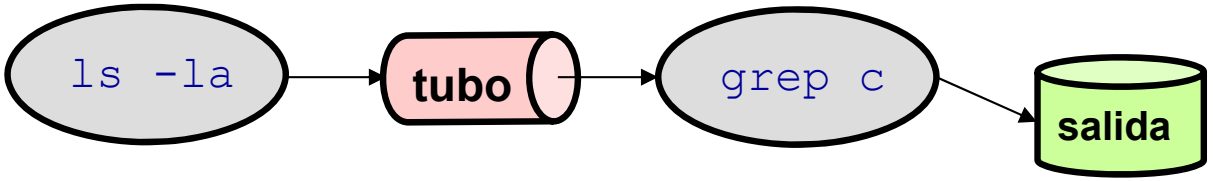
```
ls -la | grep c > salida
```

¿Cómo se resuelve?



Pasos para la resolución:

1. Determinar flujo de información



2. Establecer jerarquía de procesos
   - Padre: `grep c > salida`
   - Hijo: `ls -la`
3. Redireccionar entrada y salida estándar de procesos

Resolución de línea de comandos del shell

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

```
#include<stdio.h>
#include<string.h>
int main()
{int tubo[2], fsalida;
 pipe(tubo);                                    /* (a) */
 if (fork()!=0)
  { /* PROCESO PADRE */
    close(0);
    dup(tubo[0]);
    close(tubo[0]);
    close(tubo[1]);
    fsalida=creat("salida",0644);
    close(1);
    dup(fsalida);
    close(fsalida);
    execlp("grep","grep","c",NULL);             /* (b) */
    perror("Error en ejecucion de grep");
  }
```

Resolución de línea de comandos del shell

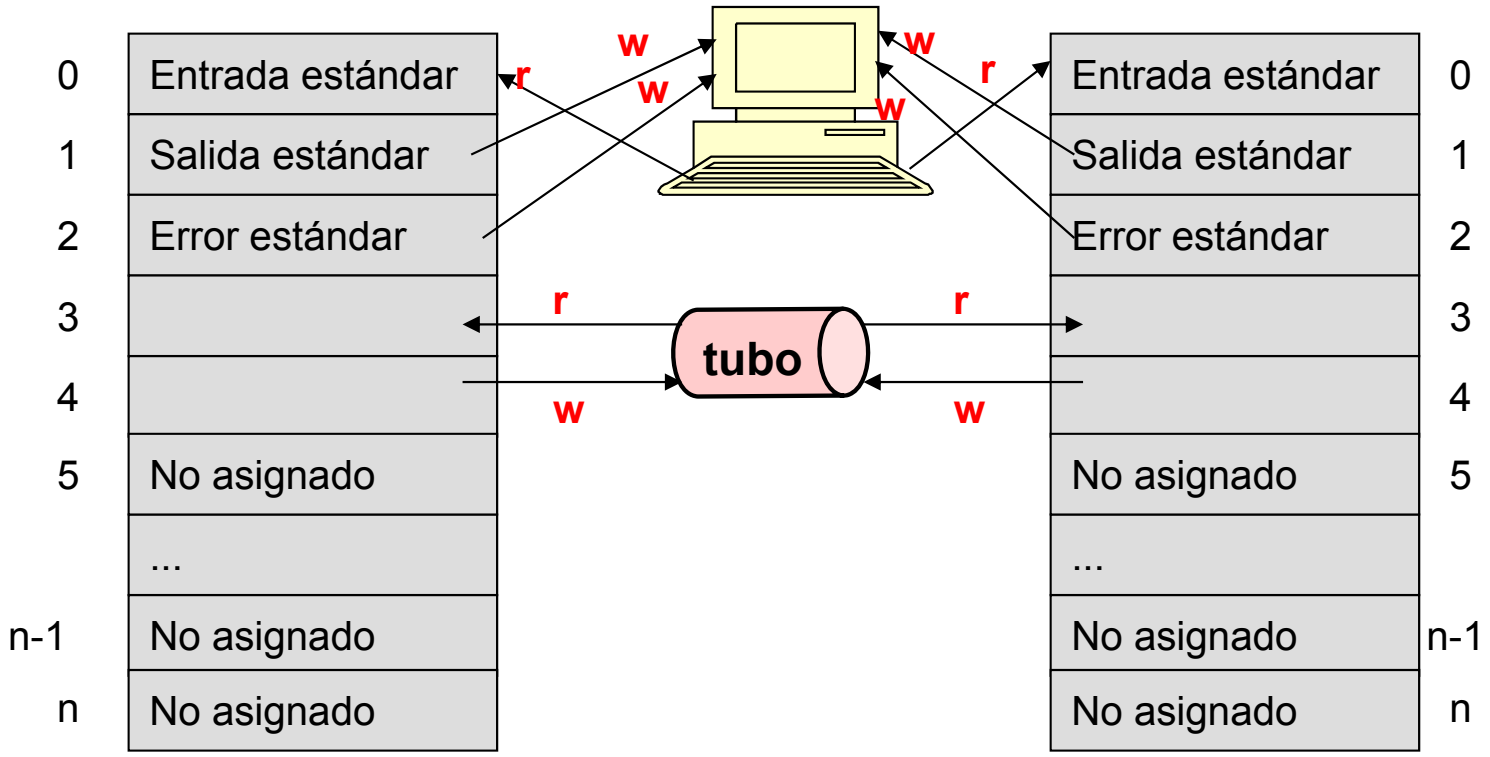
- **Ejemplo 5 (cont.):**

```
else
{ /* PROCESO HIJO */
  close(tubo[0]);
  close(1);
  dup(tubo[1]);
  close(tubo[1]);
  execlp("ls","ls","-la",NULL);          /* (b) */
  perror("Error en ejecucion de grep");
}
}
```

Ejemplos

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

Evolución de la tabla de descriptores de ficheros (cont.)

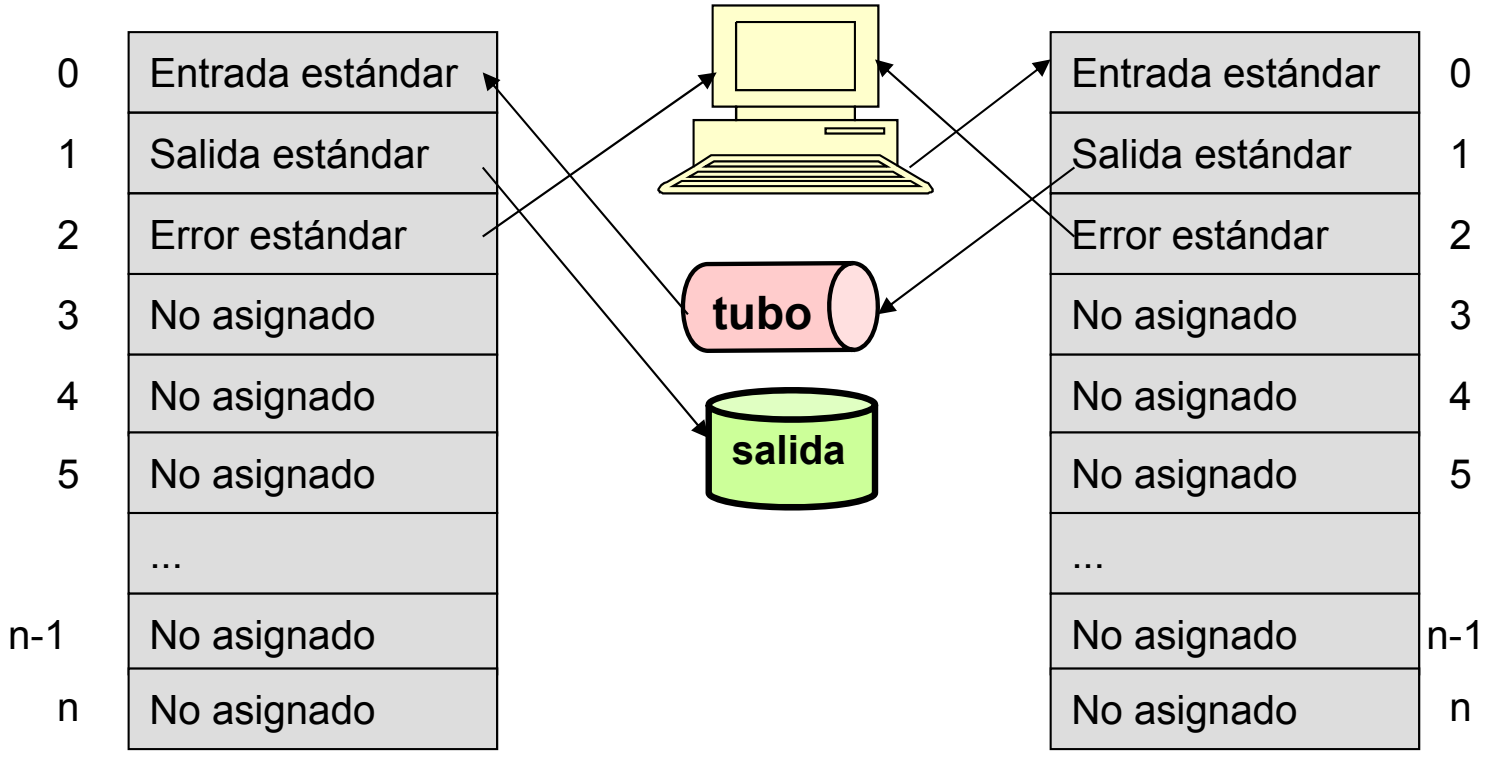


(a)

Ejemplos

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

Evolución de la tabla de descriptores de ficheros (cont.)



Proceso padre

Entrada estándar: tubo[0]
Salida estándar: salida

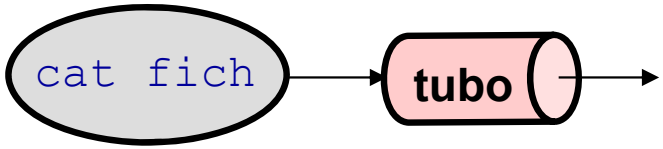
(b)

Proceso hijo

Entrada estándar: teclado
Salida estándar: tubo[1]

Redireccionamientos

- **Situación 1:**



- Solución 1:

```
while ((read(fd_fich,&dato,sizeof(dato))>0)
      write(tubo[1],&dato,sizeof(dato));
```

char

1 proceso

- Solución 2:

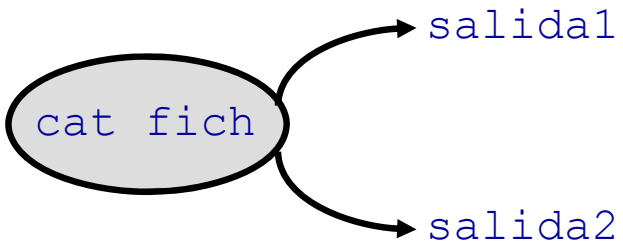
  Redireccionar la salida estándar a `tubo[1]`

```
execlp ("cat","cat","fich",NULL);
```

1 proceso

Redireccionamientos

- **Situación 2:**



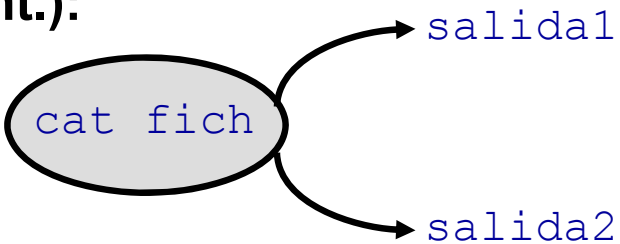
  - Solución 1:

```
while ((read(fd_fich,&dato,sizeof(dato))>0)
      { write(fd_salida1,&dato,sizeof(dato);
        write(fd_salida2,&dato,sizeof(dato);}
```

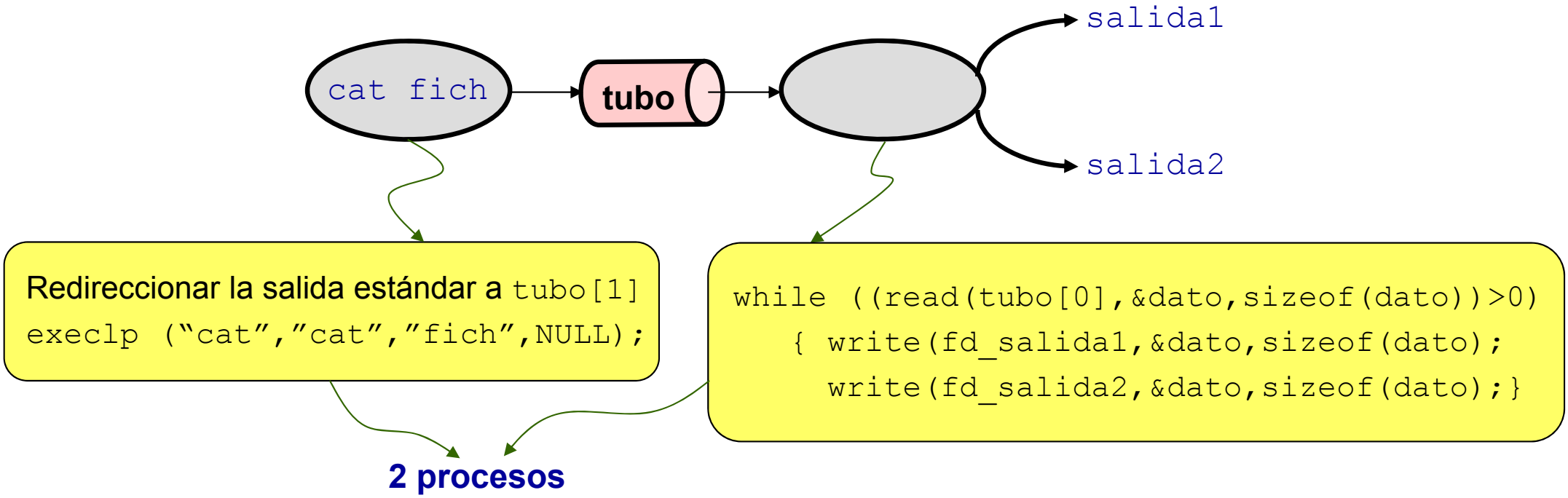
1 proceso

Redireccionamientos

- **Situación 2 (cont.):**

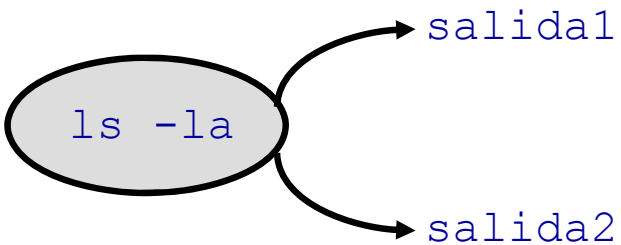


  - Solución 2:

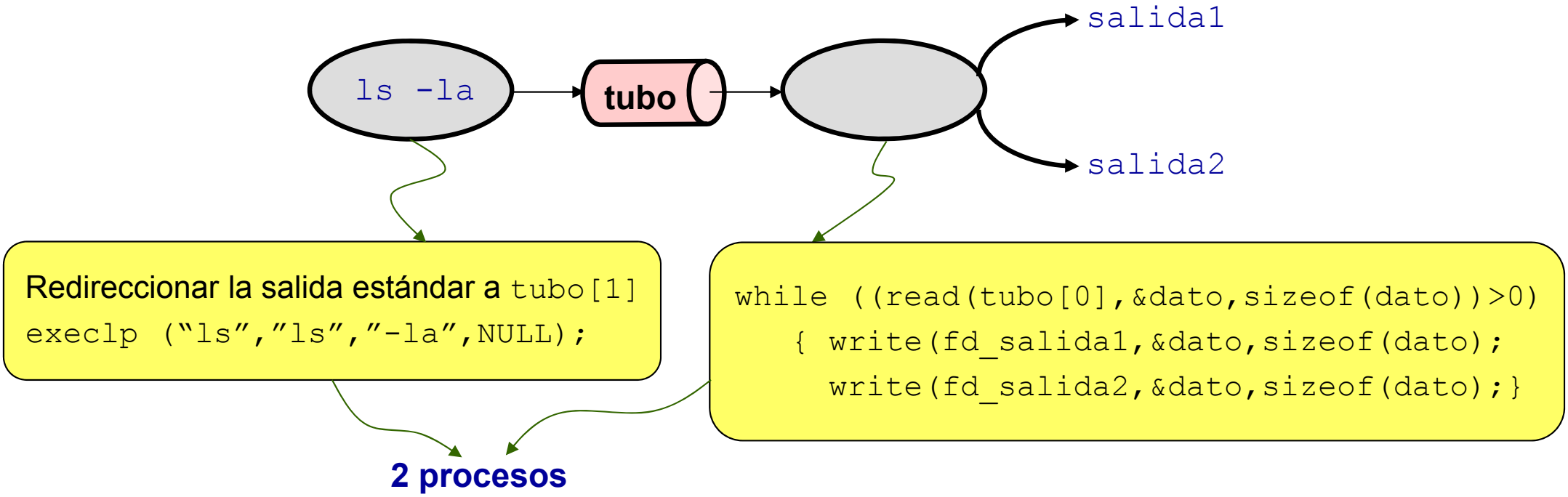


Redireccionamientos

- **Situación 3:**



  - Solución:



Tuberías: Ejercicios

- **Ejercicio 5:**

  Sea el programa que se especifica a continuación (y en el que se han numerado las líneas).

  a. Si el valor del argumento `argv[1]` es 5, ¿cuál es la salida por pantalla de la ejecución del programa?
  b. ¿Qué varía con respecto al apartado anterior si se elimina la línea 11?
  c. ¿Qué ocurre si en el programa original se elimina la línea 10?
  d. ¿Qué ocurre si en el programa original se eliminan las líneas 10 y 11?
  e. ¿Qué ocurre si en el programa original se elimina la línea 18?
  f. ¿Qué ocurre con el proceso hijo si se intercambia el orden de las líneas 4 y 5? ¿Cuál sería entonces la salida por pantalla del proceso hijo si el valor del argumento `argv[1]` fuese 5?

Tuberías: Ejercicios

- **Ejercicio 5:**

```
1.  #include <stdio.h>
2.  int main(int argc, char *argv[])
3.  { int tubo[2], status, i, dato, res, pid;
4.    pipe(tubo);
5.    pid=fork();
6.    if (pid != 0)
7.   {  close(tubo[0]);
8.      dato=atoi(argv[1]);
9.      for (i=1;i<=dato;i++) write(tubo[1],&i,sizeof(i));
10.     close(tubo[1]);
11.     wait(&status);
12.     printf ("Proceso B finaliza\n");
13.     exit(0);
14.  } else {
15.     close(tubo[1]);
16.     res = 1;
17.     while (read(tubo[0],&dato,sizeof(int))>0) res = res * dato;
18.     close(tubo[0]);
19.     printf("Proceso A finaliza. Resultado: %d \n", res);
20.     exit(0);
21.  }
22. }
```

Tuberías: Ejercicios

- **Ejercicio 6:**

  Dado el siguiente código:

```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#include <stdlib.h>
main()
{ int fd, leidos; char buffer[256];
  fd=open("fichero",O_RDONLY);
  while ( leidos = read(fd,buffer,sizeof(buffer)))
    write(1,buffer,sizeof(buffer));
  close(fd);
  exit(0);
}
```

  a. ¿Cuántas llamadas al SO debidas a la ejecución de este proceso se producirán (asumiendo que no se produce ningún error) si `fichero` tiene un tamaño de 1254 bytes?

  b. ¿Cuál será el valor final de la variable `leidos`?

Tuberías: Ejercicios

- **Ejercicio 7:**

  ¿Cuál es el resultado de ejecutar el siguiente fragmento de código?

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

main()

{ int tubo[2]; char dato='H';

  pipe (tubo);
  close (0);
  dup (tubo[0]);
  close (tubo[0]);
  write(tubo[1],&dato,1);
  while (read(0,&dato,1)>0)
  {  write(tubo[1],&dato,1); printf ("Leído %s \n",&dato);}
  close (tubo[1]);
  exit(0);
}
```

Tuberías: Ejercicios

- **Ejercicio 8:**

  Implementar un programa que, mediante llamadas al sistema, muestre por pantalla la salida de ejecutar la línea de comandos

  ```
  grep argv[1] argv[2] | cut –d: -f1
  ```

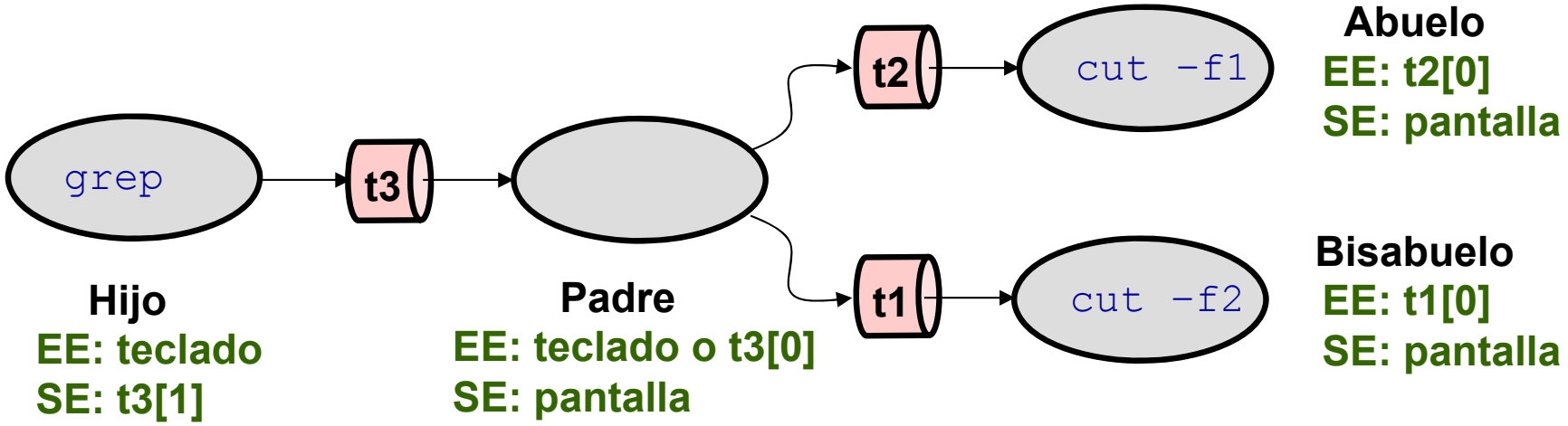
  y, a continuación, la salida de ejecutar

  ```
  grep argv[1] argv[2] | cut –d: -f2
  ```

  El acceso al fichero `argv[2]` se deberá realizar una sola vez.

Ejemplos

- **Ejercicio 8 (solución):**
  1. Flujo de información
  2. Jerarquía de procesos
  3. Redireccionamiento de entrada y salida estándar de procesos



EE = Entrada estándar
SE = Salida estándar

Tuberías: Ejercicios

- **Ejercicio 8 (solución):**

```
#include<fcntl.h>
#include<string.h>
#include<unistd.h>
main(int argc,char *argv[])
{
int fd,t1[2],t2[2],t3[2],estado; char car;
if (argc != 3)
   printf ("Uso incorrecto\n");
else
{  pipe(t1);
   if (fork()!=0)
   {  /* Bisabuelo */
      close(0);
      dup(t1[0]);
      close(t1[0]);
      close(t1[1]);
      wait(&estado);
      printf("### Segundo Campo\n");
      execlp("cut","cut","-d:","-f2",NULL);
      exit(-1);
```

Tuberías: Ejercicios

- **Ejercicio 8 (solución):**

```
}else{  /* Abuelo */
   pipe(t2);
   if (fork()!=0)
   {  close(0);
      dup(t2[0]);
      close(t2[0]);
      close(t1[0]);

      close(t1[1]);
      close(t2[1]);

      printf("### Primer Campo\n");
      execlp("cut","cut","-d:","-f1",NULL);
      exit(-1);
```

Tuberías: Ejercicios

- **Ejercicio 8 (solución):**

```
} else {   /* Padre */
  pipe(t3);
  if (fork()!=0)
  {  close(t1[0]);
     close(t2[0]);
     close(t3[1]);

     while (read(t3[0],&car,sizeof(char))>0)
        {   write(t1[1],&car,sizeof(char));
            write(t2[1],&car,sizeof(char));
        }

     close(t3[0]);
     close(t1[1]);
     close(t2[1]);

     exit (0);
```

Tuberías: Ejercicios

- **Ejercicio 8 (solución):**

```
            } else {   /* Hijo */
               close(t1[0]);
               close(t2[0]);
               close(t3[0]);
               close(t1[1]);
               close(t2[1]);

               close(1);
               dup(t3[1]);
               close(t3[1]);

               execlp("grep","grep",argv[1],argv[2],NULL);
               exit(-1);
            }
         }
      }
   }
}
```

Tuberías: Ejercicios

- **Ejercicio 8 (solución):**

  Una alternativa para el código del proceso padre sería:

```
} else {   /* Padre */
  pipe(t3);
  if (fork()!=0)
  {  close(t1[0]);
     close(t2[0]);
     close(t3[1]);

     close(0);
     dup(t3[0]);
     close(t3[0]);

     while (read(0,&car,sizeof(char))>0)
        {   write(t1[1],&car,sizeof(char));
            write(t2[1],&car,sizeof(char));
        }
     close(t1[1]);
     close(t2[1]);

     exit (0);
```

Tuberías: Ejercicios

- **Ejercicio 9:**

  Escribir un programa en C que cree dos procesos hijos, H1 y H2, que se ejecuten de forma concurrente. Posteriormente, el proceso padre leerá desde el teclado un número y se lo enviará a los dos hijos. H1 debe crear los números pares comprendidos entre 0 y el número que ha recibido del proceso padre y enviará dichos números al padre. H2 hará lo mismo con los números impares. El padre debe imprimir los números recibidos de forma ordenada: 0, 1, 2, 3, …

  La sincronización y comunicación necesaria entre los procesos se realizará mediante tuberías. Una posibilidad sería ajustándose al siguiente esquema: