Tema 2

Comunicación y sincronización entre procesos

Sistemas Distribuidos

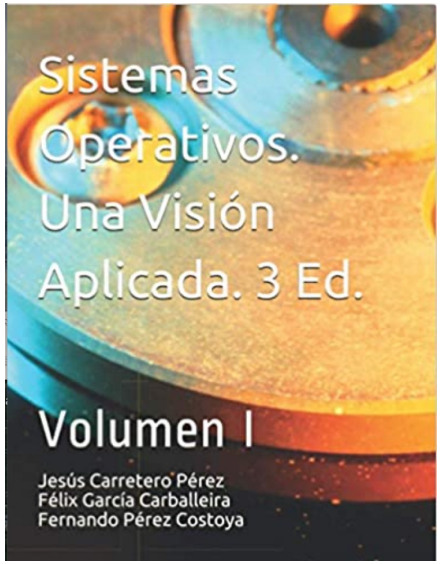
Grado en Ingeniería Informática

Universidad Carlos III de Madrid

Contenido

- Repaso de los conceptos de proceso y *threads*
- Concurrencia
- Mecanismos de comunicación
- Mecanismos de sincronización
- Llamadas al sistema POSIX
  - Para comunicación y sincronización de procesos
  - Gestión de threads
- Threads en C
- Threads en Python
- Threads en Java

Bibliografía

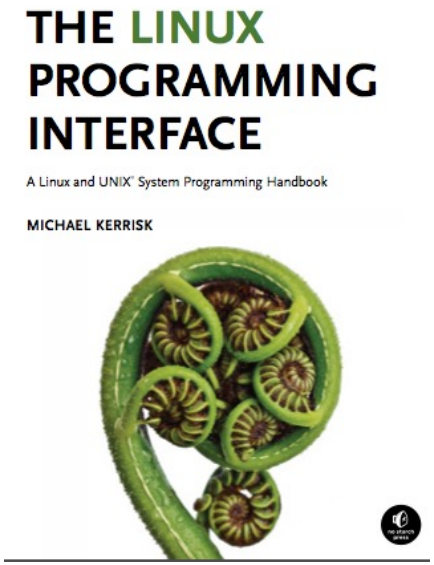


Sistemas Operativos

3ª edición.

Por Jesús Carretero, Félix García y Fernando Pérez

Independently published (Amazon)



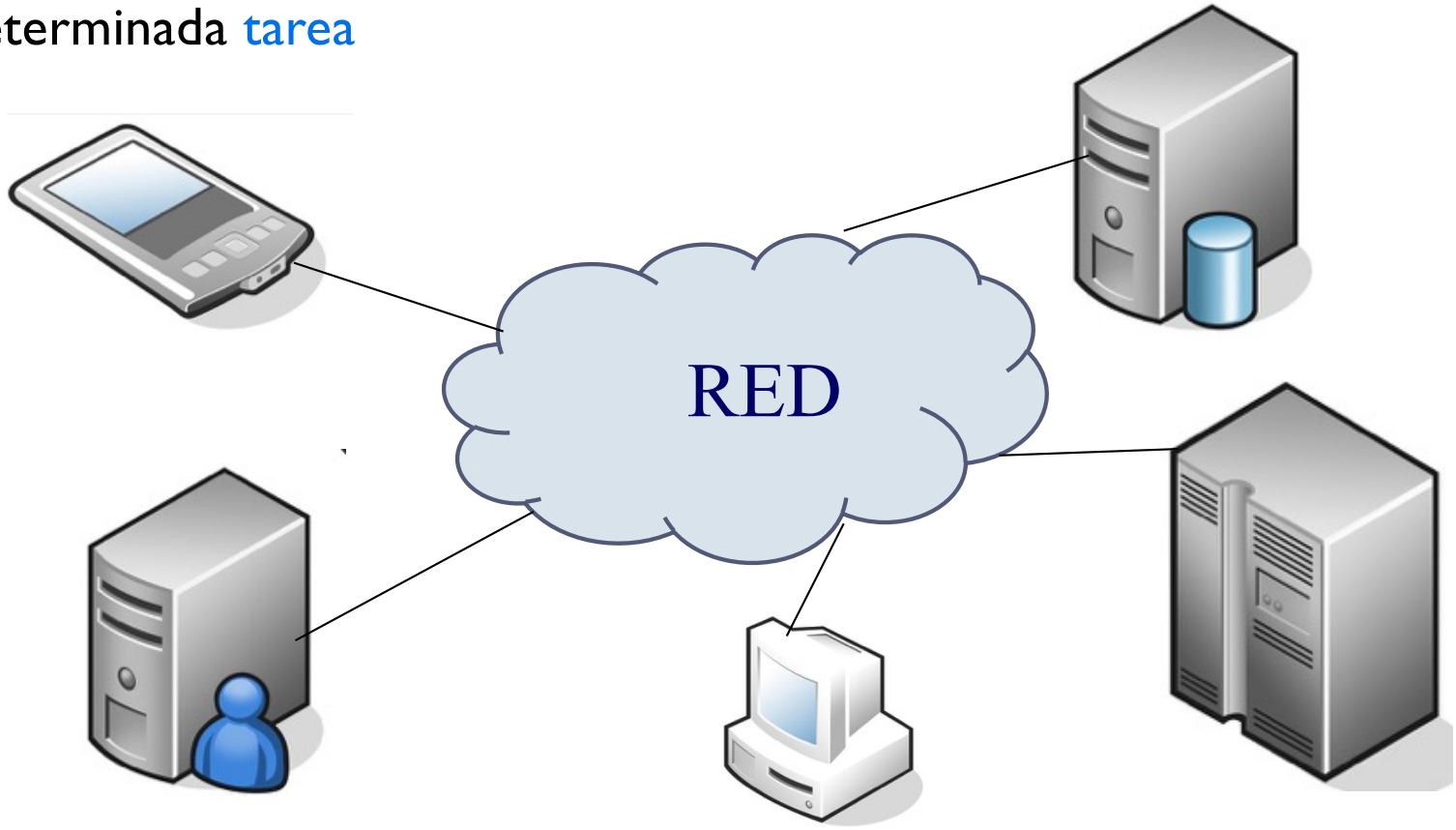
The Linux Programming Interface

Michael Kerrisk

William Pollock

Sistema distribuido

- Sistema formado por recursos de computación (hardware y software) físicamente distribuidos e interconectados a través de una red, que comunican mediante paso de mensajes y cooperan para realizar una determinada tarea



Fuente: Workstation and hardware icons y Ember Studio , CC BY-NC-SA 2.0

Conceptos previos

- Un programa es un conjunto de instrucciones
- Un proceso es un **programa en ejecución**

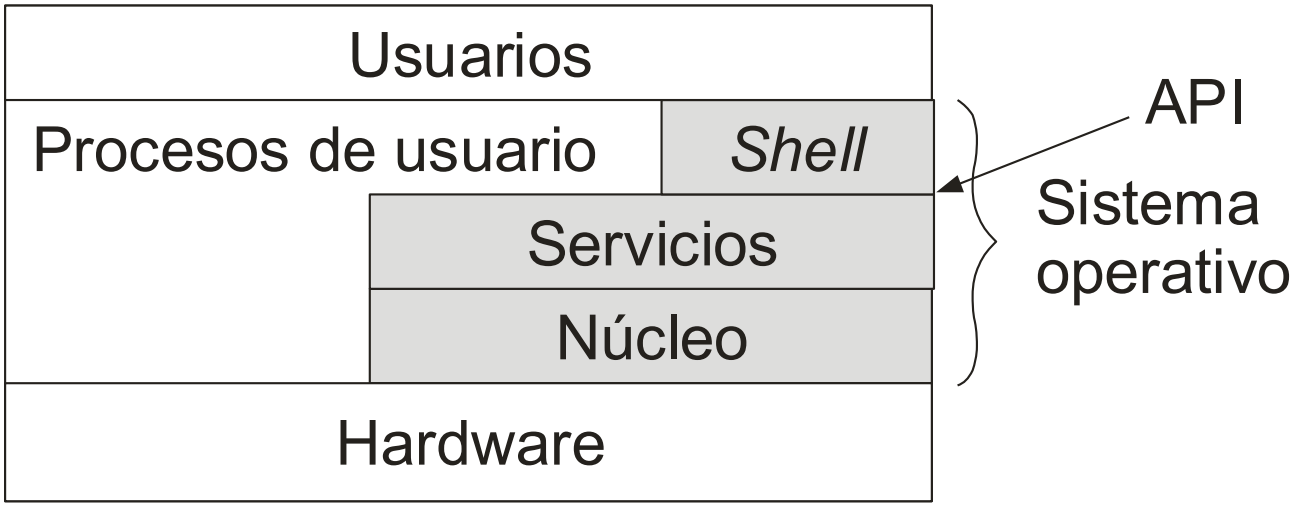
- Una red de computadores es un conjunto de computadores conectados por una red de interconexión

- Un sistema distribuido es un conjunto de computadores (sin memoria ni reloj común) conectados por una red
  - Aplicaciones distribuidas: conjunto de procesos que ejecutan en uno o más computadores que colaboran y comunican intercambiando *mensajes*

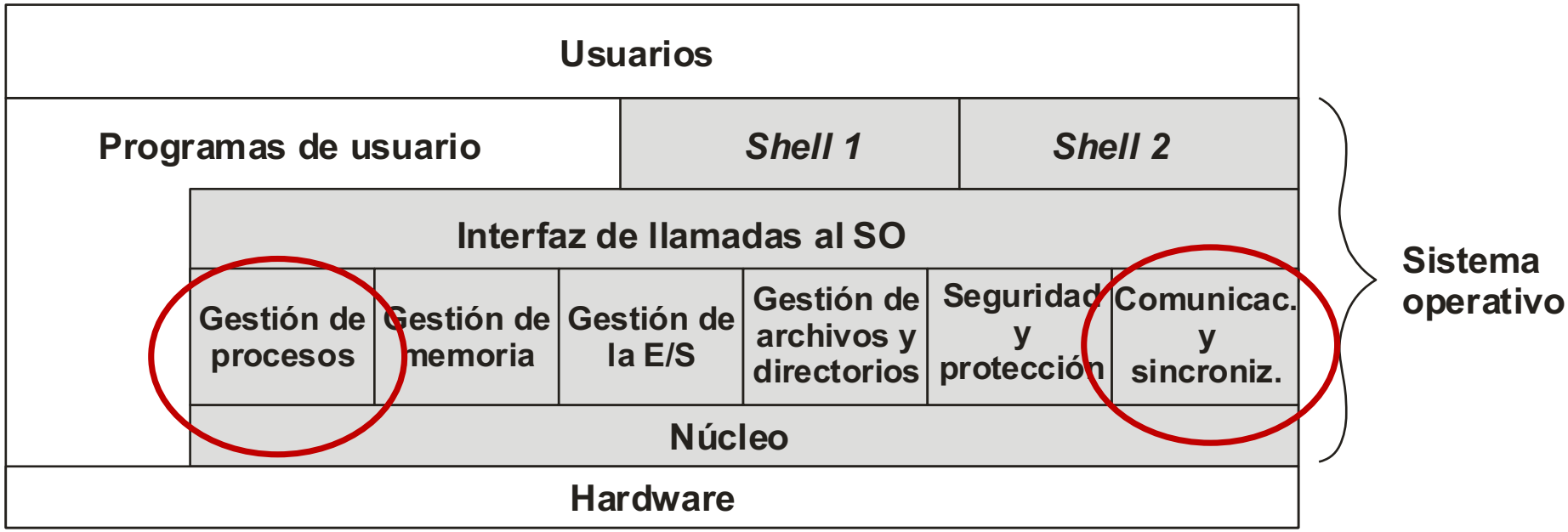
- Un protocolo es un conjunto de reglas e instrucciones que gobiernan la comunicación en un sistema distribuido, es decir, el intercambio y formato de los mensajes

Sistema Operativo

- Gestión de los recursos del computador
- Ejecución de servicios para los programas en ejecución
- Ejecución de los mandatos de los usuarios

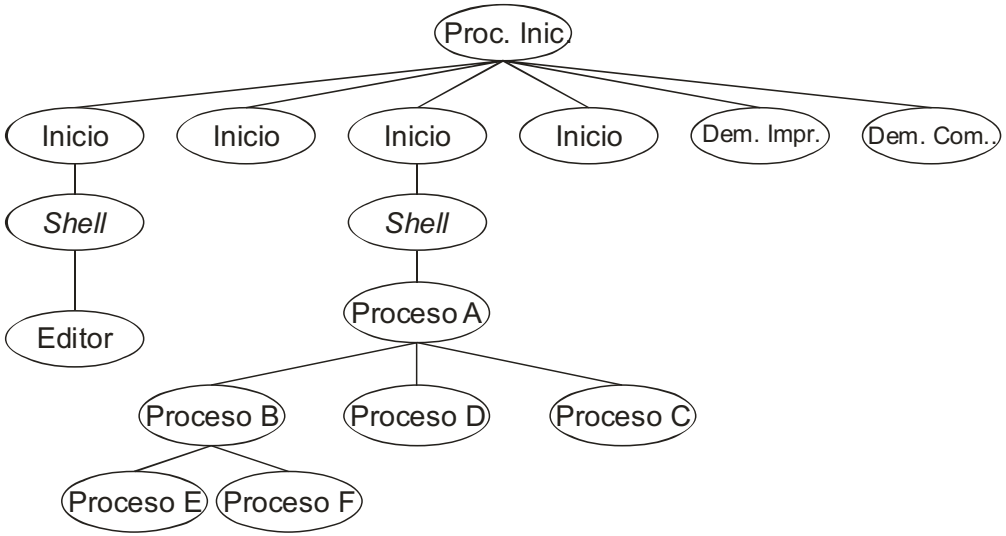


Componentes del Sistemas Operativo



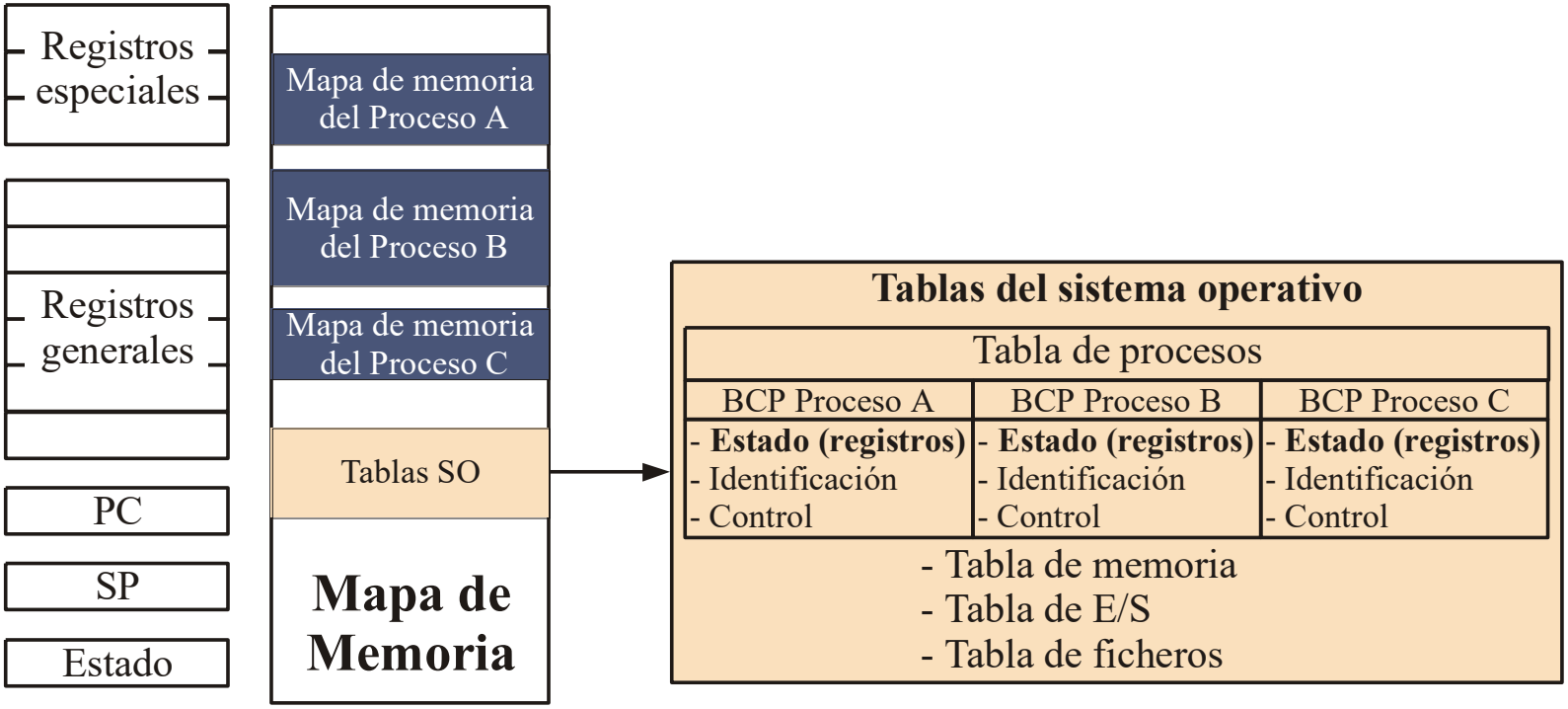
Proceso

- Definición
  - Programa en ejecución
  - Unidad de procesamiento gestionada por el SO
  - Para que un programa pueda ser ejecutado ha de residir en memoria principal
- Árbol de procesos
  - Relación de jerarquía



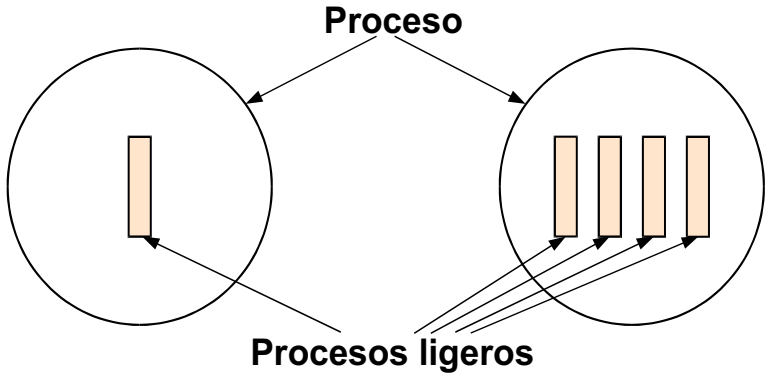
Información de los procesos

- El sistema operativo mantiene un conjunto de estructuras de información que permiten identificar y gestionar los procesos



Proceso ligero o thread

- Un proceso ligero, hilo o *thread* es un programa en ejecución (flujo de ejecución) que comparte la imagen de memoria (y otras informaciones) con otros procesos ligeros (dentro de un proceso)
  - Unidad de ejecución dentro de un proceso
  - Dependiendo del número de procesos que puedan ejecutar simultáneamente, un SO es monotarea o multitarea
- Un proceso puede contener un solo flujo de ejecución (*monothread*) o varios (*multithread*)
  - Por ejemplo, en el lenguaje C el hilo de ejecución primario se corresponde con la función principal `main`

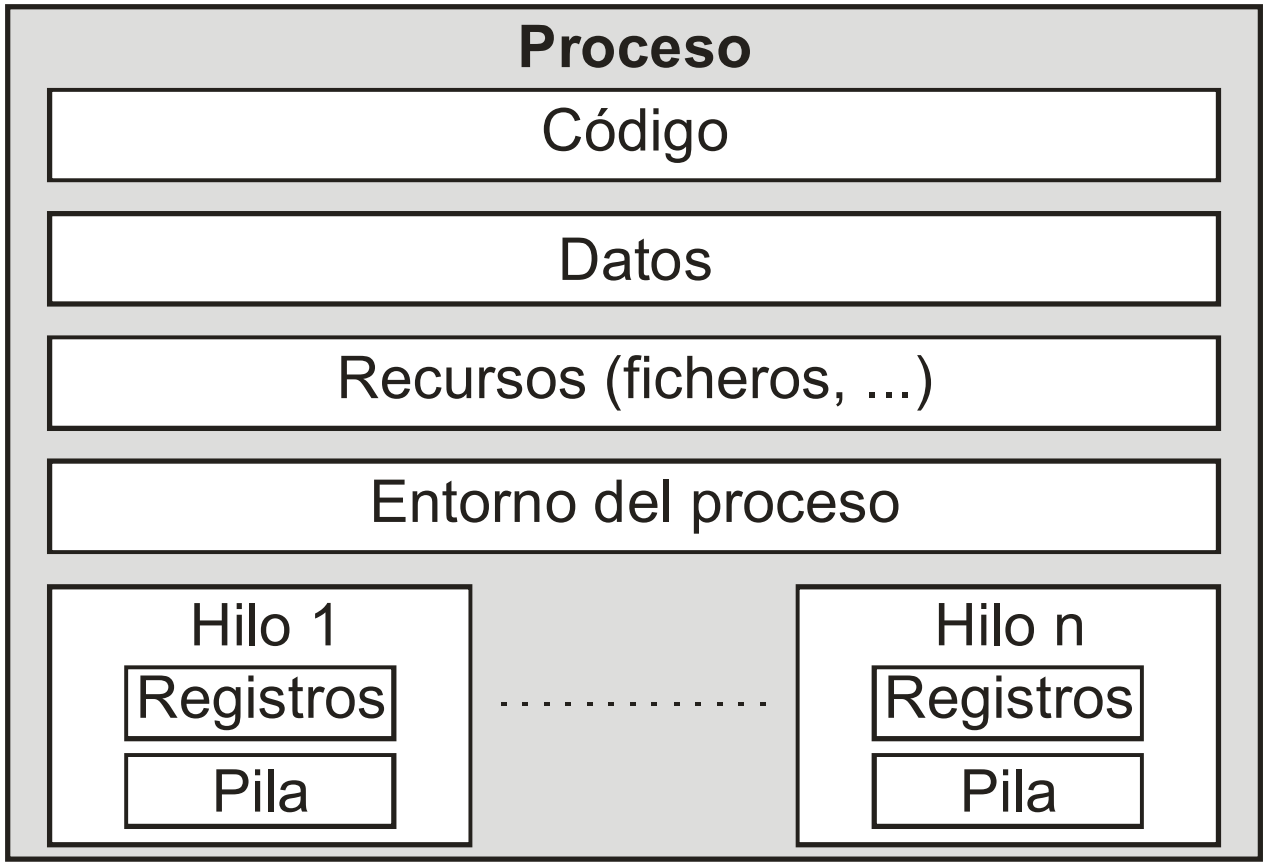


Información propia/compartida

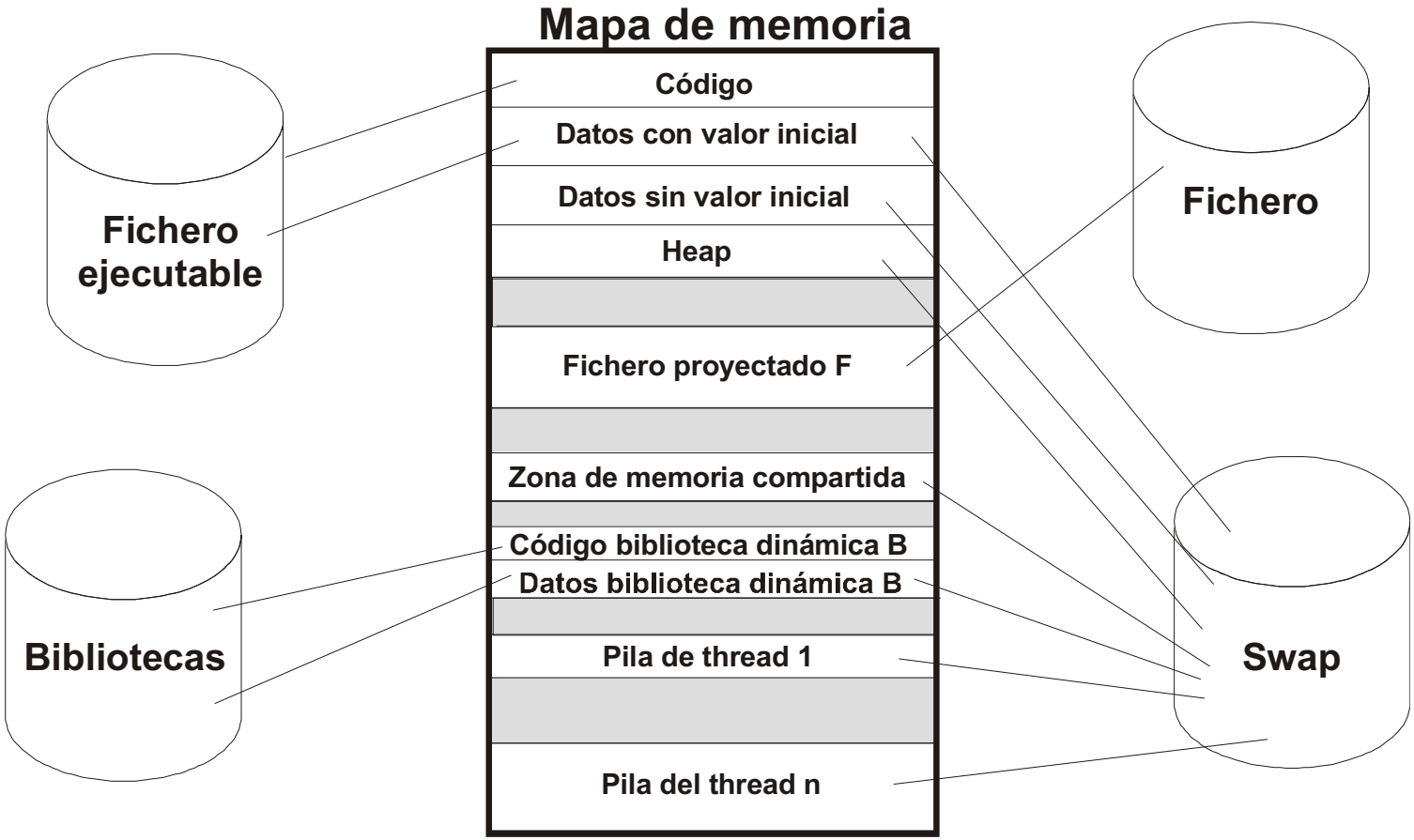
- Por *thread* (información propia)
  - Contador de programa, valores de los registros
  - Pila
  - Estado (ejecutando, listo o bloqueado)

- Por proceso (información compartida)
  - Espacio de memoria
  - Variables globales
  - Ficheros abiertos, descriptores de sockets
  - Procesos hijos
  - Temporizadores
  - Señales y semáforos
  - Contabilidad

Estructura de un proceso con *threads*

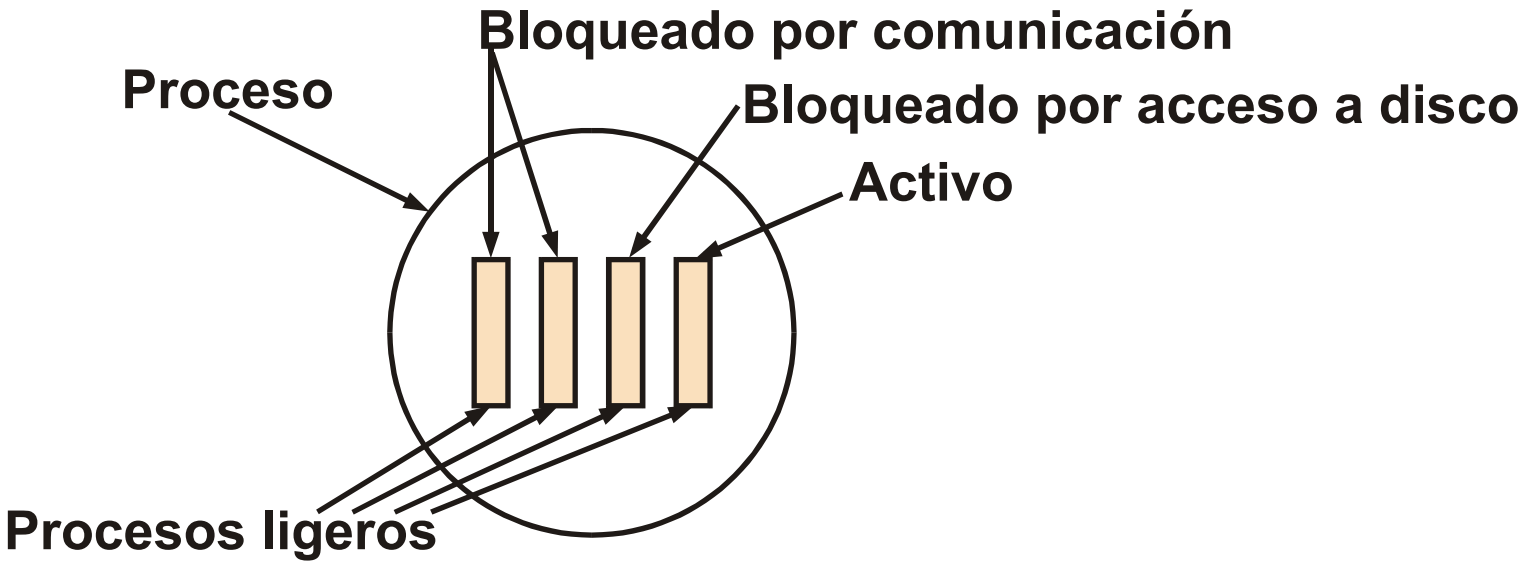


Mapa de memoria de un proceso con *threads*



Estados del proceso ligero

- Como los procesos convencionales, un proceso ligero puede estar en varias situaciones o estados: ejecutando, listo o bloqueado
  - El estado del proceso será la combinación de los estados de sus procesos ligeros.



Threads y paralelismo

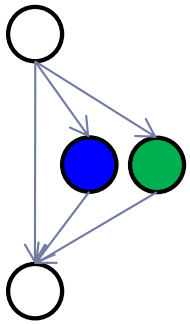
- Los procesos ligeros permiten paralelizar una aplicación
  - Un programa puede dividirse en procedimientos que pueden ejecutar de manera independiente mediante procesos ligeros
    - Los procesos ligeros pueden lanzar su ejecución de manera simultánea
  - Diseño modular de la aplicación
- La **base del paralelismo**:
  - Mantener algún proceso ligero siempre en estado de ejecución: *"mientras que un proceso está bloqueado, otro podría estar ejecutando"*

```
main(){
…
function1(args);
function2(args);
…

}
```

Diseño secuencial

```
main(){
…
Crear_thread(
   function1(args));
Crear_thread(
   function2(args));
}
```

Diseño paralelo

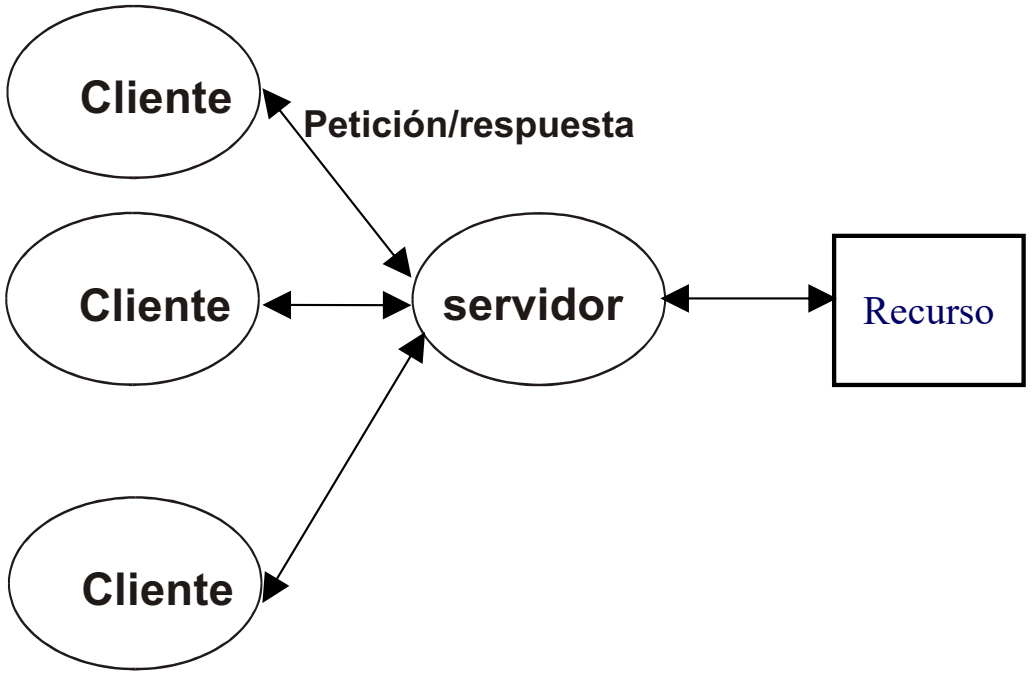


Escenarios de aplicaciones multithread

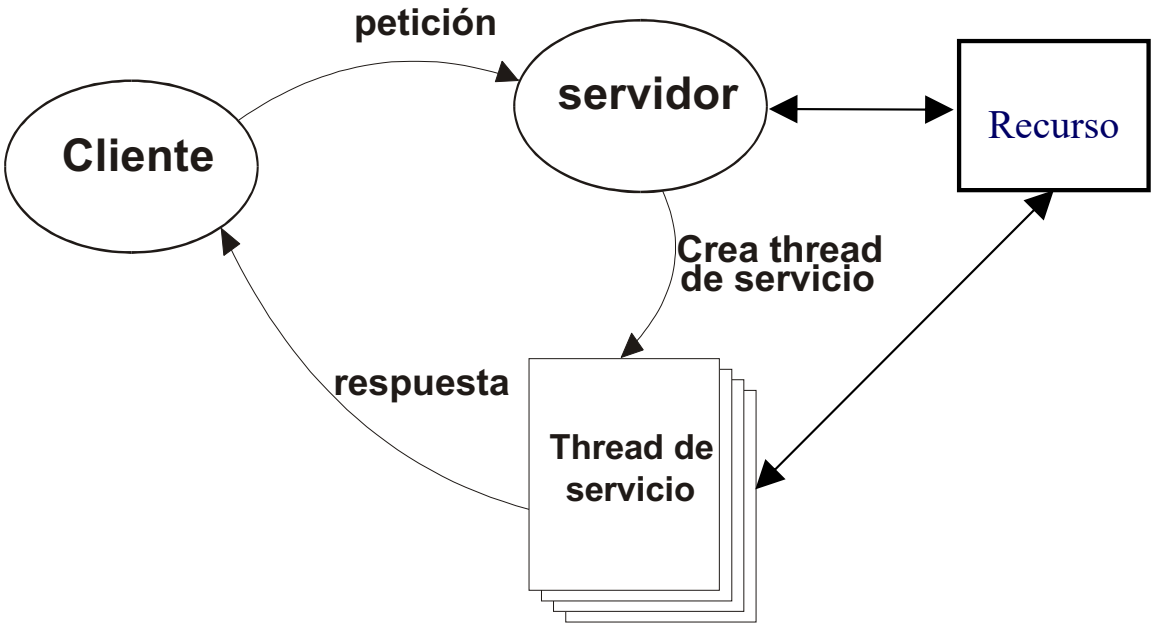
- Cuando se desea maximizar el uso de los recursos
  - Plataformas multicore

- Cuando se desea solapar la ejecución con actividades de E/S bloqueantes
  - En una aplicación *single-thread* el proceso se bloquea cuando se realiza una operación de E/S bloqueante (lectura de un fichero, espera de un mensaje,…)

Servidores secuenciales



Servidores concurrentes con *threads*

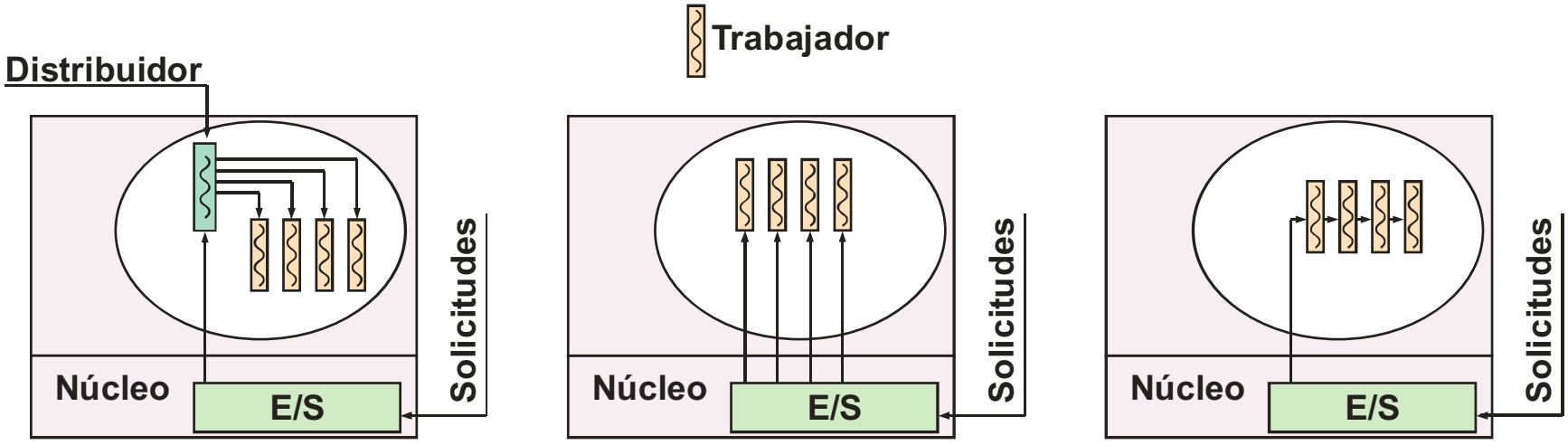


Diseño concurrente

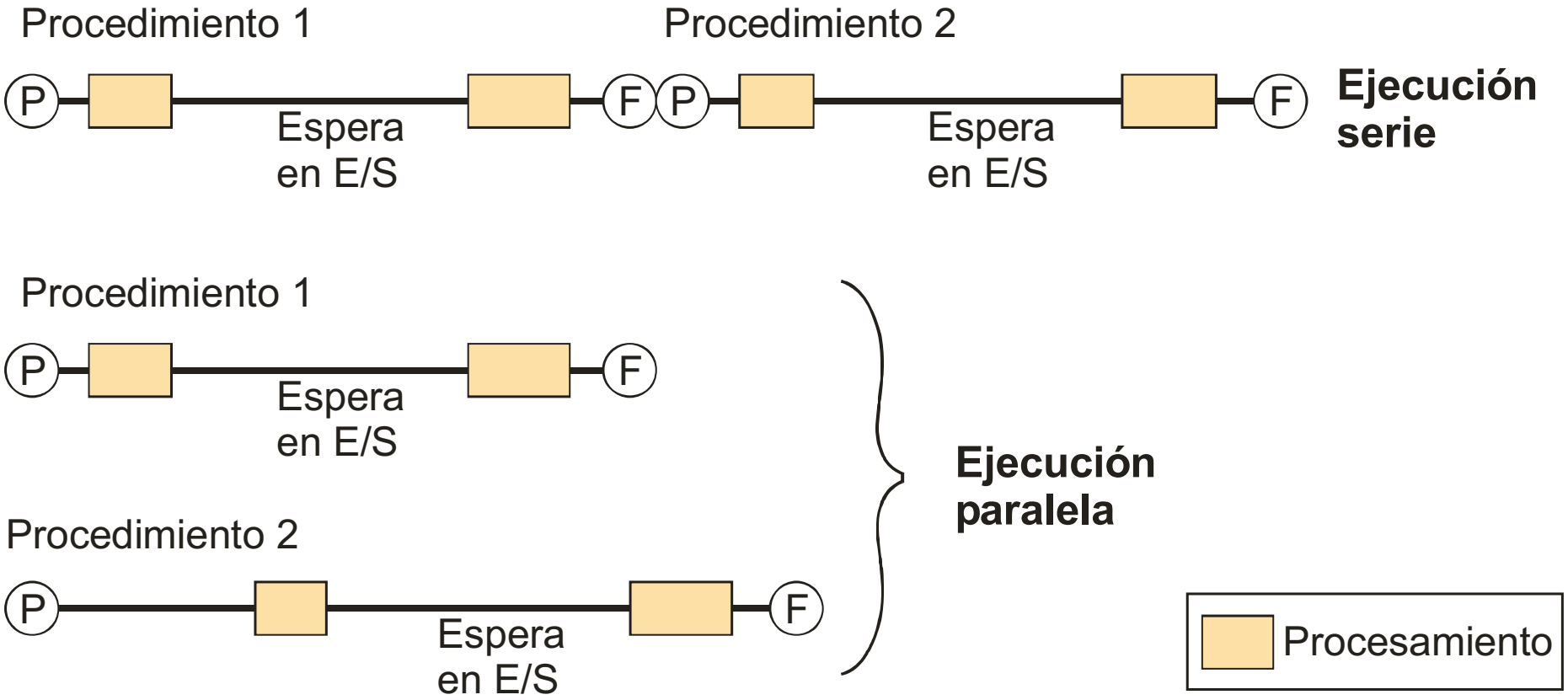
- Los procesos concurrentes deben comunicar y sincronizarse

Diseño de servidores mediante *threads*

- Distintas arquitecturas de SW para construir servidores paralelos:
  - Un proceso distribuidor que acepta peticiones y las distribuye entre un *pool* de procesos ligeros
  - Cada proceso ligero realiza las mismas tareas: aceptar peticiones, procesarlas y devolver su resultado
  - Segmentación: cada trabajo se divide en una serie de fases, cada una de ellas se procesa por un proceso ligero especializado



Ejemplo: ejecución serie vs. paralela



Ejercicio

- Se desea comparar el rendimiento de un servidor de ficheros secuencial con uno multiflujo que utiliza procesos ligeros (threads). En ambos casos el servicio de una petición implica la ejecución con 10 ms de tiempo de cómputo y en el caso de que los datos pedidos no esté en la cache de bloques del servidor, otros 40 ms de acceso al disco. En el servidor secuencial, durante el acceso al disco se bloquea el proceso servidor. Sin embargo, en el segundo caso sólo se bloquea el thread correspondiente. Teniendo en cuenta que el disco sólo puede llevar a cabo una operación en cada momento y suponiendo que el servidor se ejecuta en un sistema con un único core se pide:

  a) Suponiendo que no se producen aciertos en la cache, ¿Cuántas peticiones por segundo puede procesar el servidor secuencial? ¿y un servidor multiflujo?

  b) Igual que el anterior apartado pero con un 100% de aciertos en la cache de bloques.

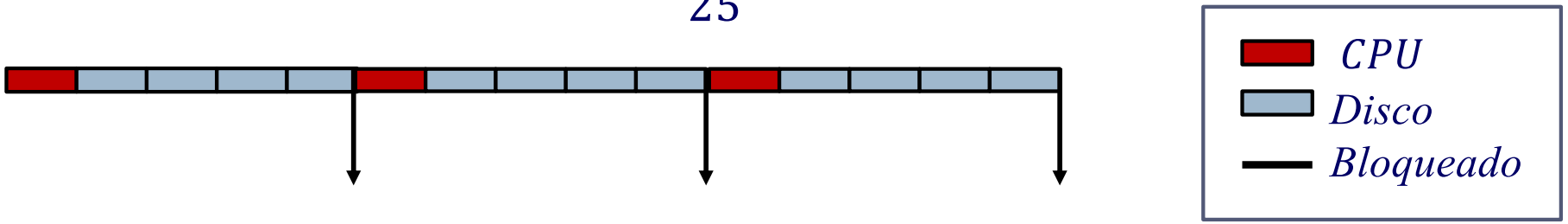
  c) Igual para un 50% de aciertos

  d) Considerar cómo afectaría a los resultados de los tres apartados anteriores el uso de un procesador con 4 núcleos para ejecutar el servidor.

Solución

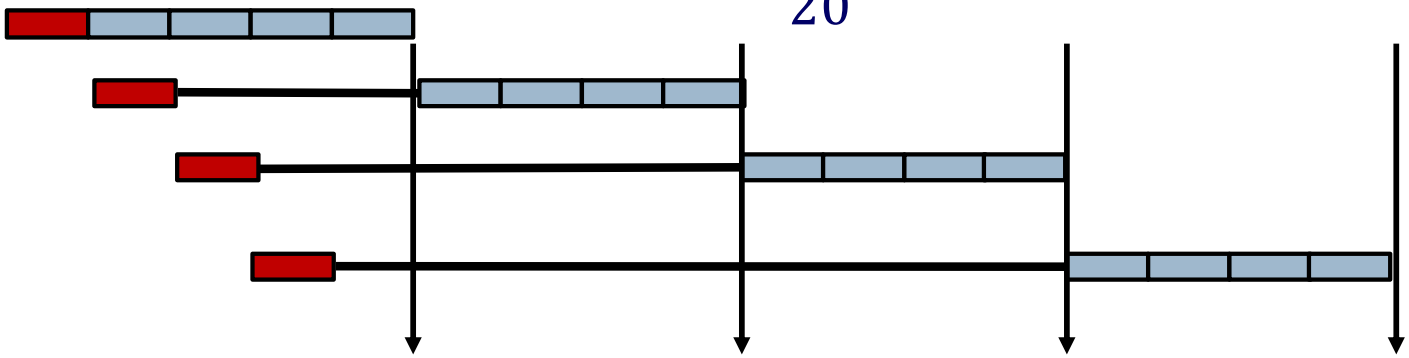
Escriba aquí la ecuación.*Análisis teniendo en cuenta máxima capacidad del servidor y sin limitación de recursos*

$$\frac{1000}{25} = 20\ peticiones/s$$



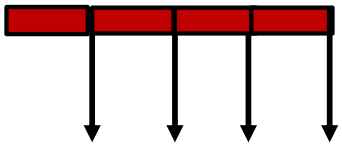
a) multithread:

$$\frac{1000}{20} = 25\ peticiones/s$$



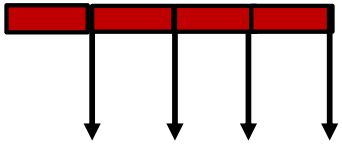
Solución

b) secuencial:



$$\frac{1000}{10} = 100\ peticiones/s$$

b) multithread:

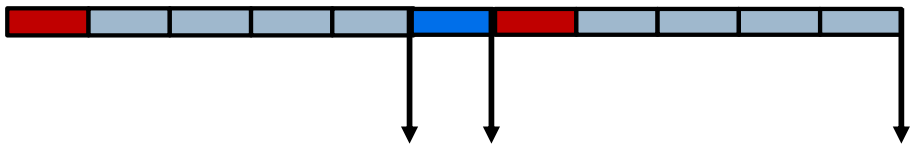


$$\frac{1000}{10} = 100\ peticiones/s$$

Solución

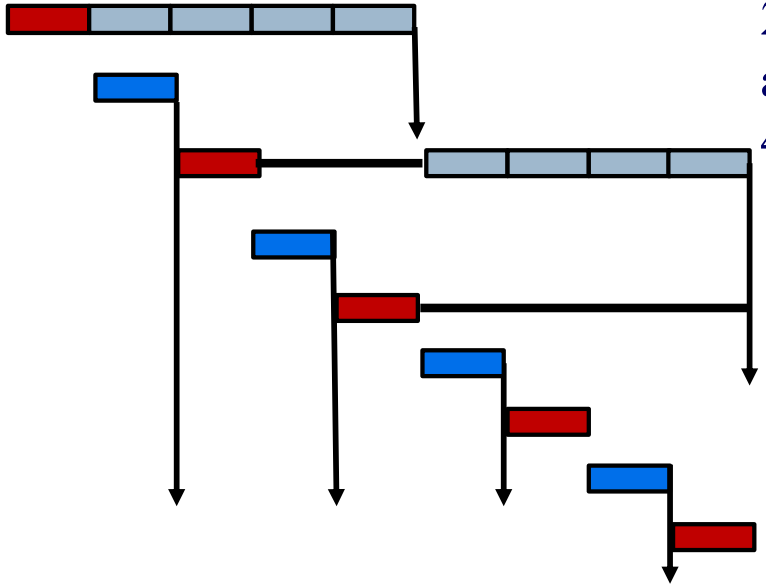
c) secuencial:

$$Tmedio = 50\times0{,}5 + 10\times05 = 30\ ms$$



$$\frac{1000}{30} = 33{,}33\ peticiones/s$$

c) multithread:



Cada 40 ms llegan 4 peticiones. En meia 2 son aciertos y finalizan. Al final de los 40 ms además, finaliza otra petición de disco. Cada 40 finalizan 3 peticiones

$$\frac{1000\times3}{40} = 75\ peticiones/s$$

Acierto en caché

Solución

d) 100 % de aciertos en caché:

Cada 40 ms se pueden atender en los 4 cores, 4 x 4 = 16 peticiones.
Todas son aciertos, por tanto cada 40 ms acaban 16 peticiones

$$\frac{1000 \times 16}{40} = 400\ peticiones/s$$

d) 50 % de aciertos en caché:

Cada 40 ms llegan 4 x 4 = 16 peticiones, la mitad son aciertos.
Por tanto, cada 40 ms acaban 8 peticiones más 1 de disco de los 40 ms anteriores. Cada 40 ms acaban 9 peticiones

$$\frac{1000 \times 9}{40} = 225\ peticiones/s$$

Programación con procesos ligeros

- Servicios POSIX incluyen:
  - Operaciones de gestión
    - Creación e identificación de procesos ligeros
    - Atributos de un proceso ligero
    - Planificación de procesos ligeros
      - Prioridades, políticas de planificación
    - Terminación de procesos ligeros

Creación e identificación de *threads*

- Crear un thread:

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

donde:

thread	Identificador del nuevo *thread* (se captura cuando se crea)
attr	Atributos a aplicar al nuevo *thread* o NULL
funcion	Función a ejecutar por el *thread* que es creado
arg	Puntero a el/los argumentos del *thread*

devuelve:

int	0 si éxito o -1 si error

- Obtener el identificador del *thread* que ejecuta:

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

Atributos de un proceso ligero

- Un hilo se crea con una serie de atributos por defecto
  - Por defecto, un hilo es "*joinable*" o No Independiente

- Es posible modificar los atributos por defecto del hilo:
- Inicia un objeto atributo que se puede utilizar para crear nuevos procesos ligeros

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

- Destruye un objeto de tipo atributo

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

- Cambiar el estado de terminación (Joinable / Detached)

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

Si `detachstate` es

  - PTHREAD_CREATE_DETACHED el proceso se crea independiente
  - PTHREAD_CREATE_JOINABLE el proceso se crea no independiente (por defecto)

- Otras atributos: tamaño de pila, planificación, prioridad, etc.

Terminación de procesos ligeros

- Finaliza la ejecución del proceso ligero:

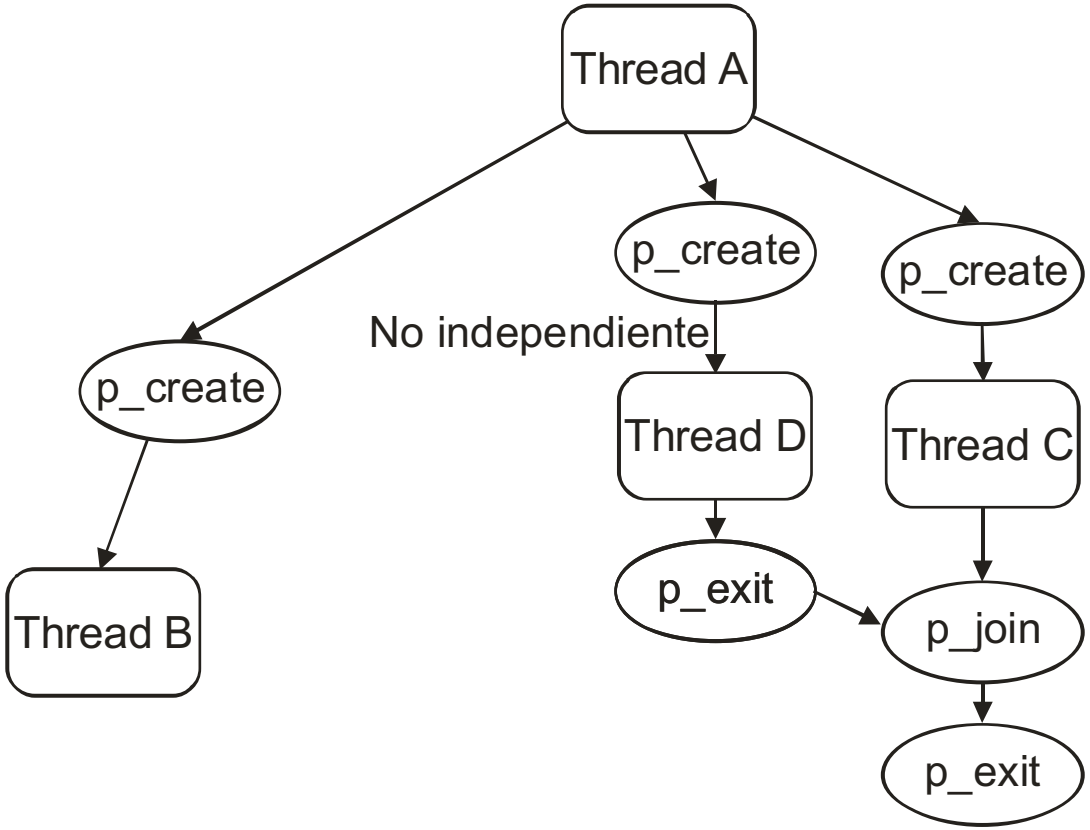
```
int pthread_exit(void *value);
```

- Para procesos creados como NO independientes es además necesario:

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

- Espera la terminación del proceso con identificador *thid*
- Devuelve en el segundo argumento el valor pasado en *pthread_exit*.
- Sólo se puede esperar la terminación de procesos ligeros no independientes (PTHREAD_CREATE_JOINABLE)

Ejemplo: Jerarquía de *threads*



Ejemplo: creación y terminación

```
#include <stdio.h>    /* printf */
#include <pthread.h> /* Para threads… */
#define NUM_THREADS 10
void funcion(int *idThread){
        int thid = *idThread;
        printf("Thread id = %ld\ti=%d: \n", pthread_self(),  thid);
        pthread_exit(NULL);
}
int main () {
        pthread_t arrayThread[NUM_THREADS]; /* Array de threads */
        int i;
        /* CREAR THREADS */
        for(i=0;i<NUM_THREADS;i++){
          if (pthread_create(&arrayThread[i],NULL,(void *)funcion, &i)==-1)
                printf("Error al crear proceso ligero\n");
        }
        /* ESPERAR TERMINACIÓN DE THREADS */
        for(i=0;i<NUM_THREADS;i++)
                pthread_join (arrayThread[i], NULL);
        return(0);
}
```

¿Pueden producirse estas salidas?

```
Thread id = 0
Thread id = 0
Thread id = 0
Thread id = 0
Thread id = 0
Thread id = 0
Thread id = 0
....
```

```
Thread id = 10
Thread id = 10
Thread id = 10
Thread id = 10
Thread id = 10
Thread id = 10
Thread id = 10
....
```

¿Pueden producirse estas salidas?

```
Thread id = 0
Thread id = 0
Thread id = 0
Thread id = 0
Thread id = 0
Thread id = 0
Thread id = 0
....
```

```
Thread id = 10
Thread id = 10
Thread id = 10
Thread id = 10
Thread id = 10
Thread id = 10
Thread id = 10
....
```

La respuesta es SI

Ejemplo: creación y terminación

```
#include <stdio.h>    /* printf */
#include <pthread.h> /* Para threads… */
#define NUM_THREADS 10
void funcion(int *idThread){
        int *thid = (int *) idThread;
        printf("Thread id = %ld\ti=%d: \n", pthread_self(),*thid);
        pthread_exit(NULL);
}
int main () {
        pthread_t arrayThread[NUM_THREADS]; /* Array de threads */
        int i;
        /* CREAR THREADS */
        for(i=0;i<NUM_THREADS;i++){
          if (pthread_create(&arrayThread[i],NULL,(void *)funcion, &i)==-1)
                printf("Error al crear proceso ligero\n");
        }
        /* ESPERAR TERMINACIÓN DE THREADS */
        for(i=0;i<NUM_THREADS;i++)
                pthread_join (arrayThread[i], NULL);
        return(0);
}
```

condición de carrera

Condiciones de carrera

- Un programa tiene una condición de carrera si varios procesos (threads) acceden a un recurso compartido sin control de forma que el resultado depende del orden de ejecución.
  - En el ejemplo anterior el recurso compartido es la dirección de memoria donde se almacena la variable i (de la función main).
  - A cada thread se le pasa la misma dirección de memoria y el contenido de la misma va cambiando en la ejecución del bucle for

Ejemplo: Modificar atributos

```
#include <stdio.h>    /* printf */
#include <pthread.h> /* Para threads… */
#define MAX_THREADS 10
void func(void) {
   printf("Thread %d \n", pthread_self());
   pthread_exit(0);
}
main() {
   int j;
   pthread_attr_t attr;
   pthread_t thid[MAX_THREADS];
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
   for(j = 0; j < MAX_THREADS; j ++)
       pthread_create(&thid[j], &attr,  func, NULL);
   sleep(5);
   pthread_attr_destroy(&attr);
}
```

Ejercicio

- Se desea desarrollar una aplicación que debe realizar dos tareas que se pueden ejecutar de forma independiente. Los códigos de estas dos tareas se encuentran definidos en dos funciones cuyos prototipos en lenguaje de programación C, son los siguientes:

  void tarea_A(void);

  void tarea_B(void);

  Programar la aplicación anterior utilizando tres modelos distintos: un programa secuencial ejecutado por un único proceso, un programa que crea procesos para desarrollar cada una de las tareas, y un programa que realiza las tareas anteriores utilizando procesos ligeros. En cualquiera de los tres casos la aplicación debe terminar cuando todas las tareas hayan acabado.

Solución

- Versión secuencial:

```
tarea_A();
tarea_B();
```

- Versión con fork():

```
pid = fork();
if (pid==0){
        tarea_A();
        exit(0);
}
else {
        tarea_B();
        wait(NULL);
}
```

Solución

- Versión con threads:

```
pthread_create(&t1, &attr,  tarea_A, NULL);
pthread_create(&t2, &attr,  tarea_B, NULL);
pthread_join (t1, NULL);
pthread_join (t2, NULL);
```

Programación concurrente

- En SSOO multitarea pueden coexistir varios procesos activos a la vez
  - Multiprogramación con un único procesador
  - Multiprocesador
  - Multicomputador

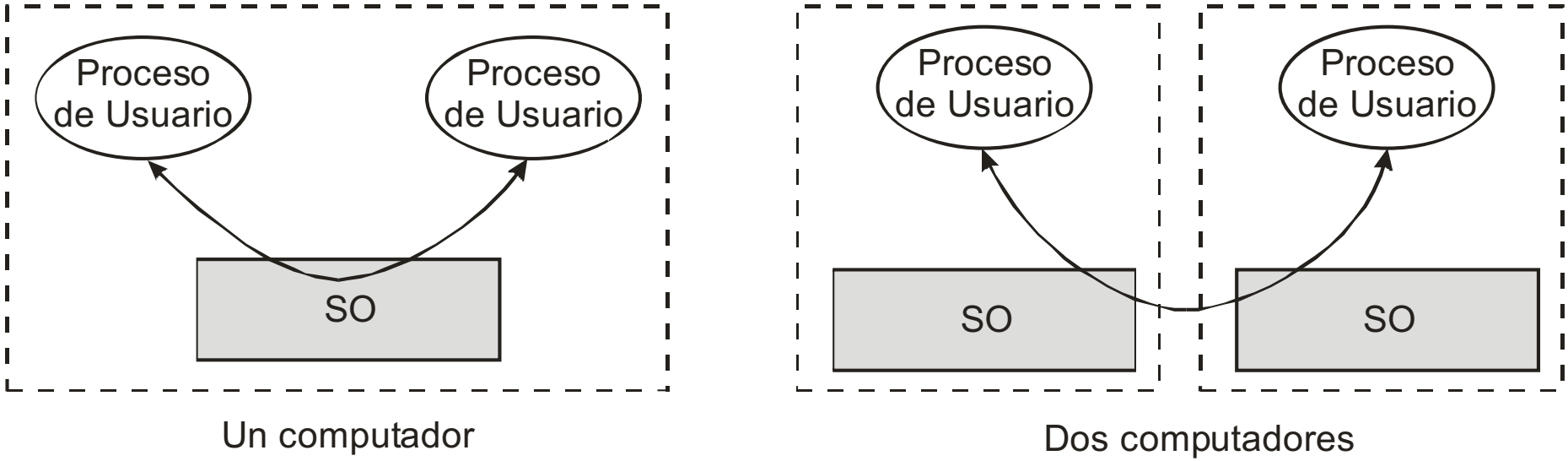
  - En general,

    para **p** procesadores y **n** procesos, la concurrencia es:
    - Aparente si n>p
    - Real si n<=p

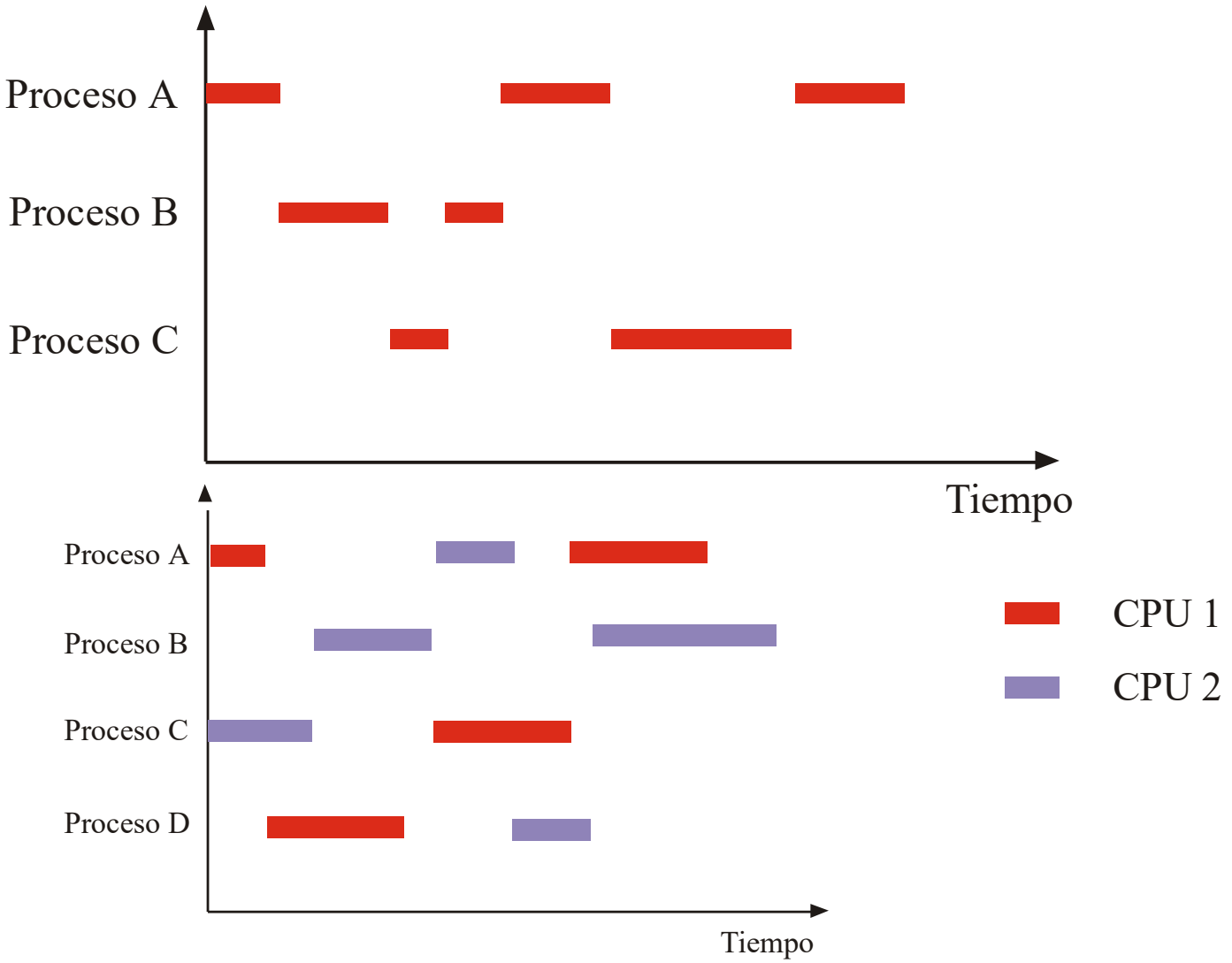
Concurrencia

- Modelos:
  - Multiprogramación en un único procesador
  - Multiprocesador
  - Multicomputador (proceso distribuido)
- Ventajas:
  - Facilita la programación
  - Acelera los cálculos
  - Posibilita el uso interactivo a múltiples usuarios
  - Mejor aprovechamiento de los recursos

Comunicación entre procesos



Monoprocesador vs. Multiprocesador



Tipos de procesos concurrentes

- Tipos de procesos:
  - Independientes: su ejecución no requiere la ayuda o cooperación con otros procesos
  - Cooperantes: su ejecución requiere trabajar conjuntamente con otros procesos para realizar una actividad

- En ambos casos se produce una interacción entre procesos
  - Compiten por recursos
  - Comparten recursos

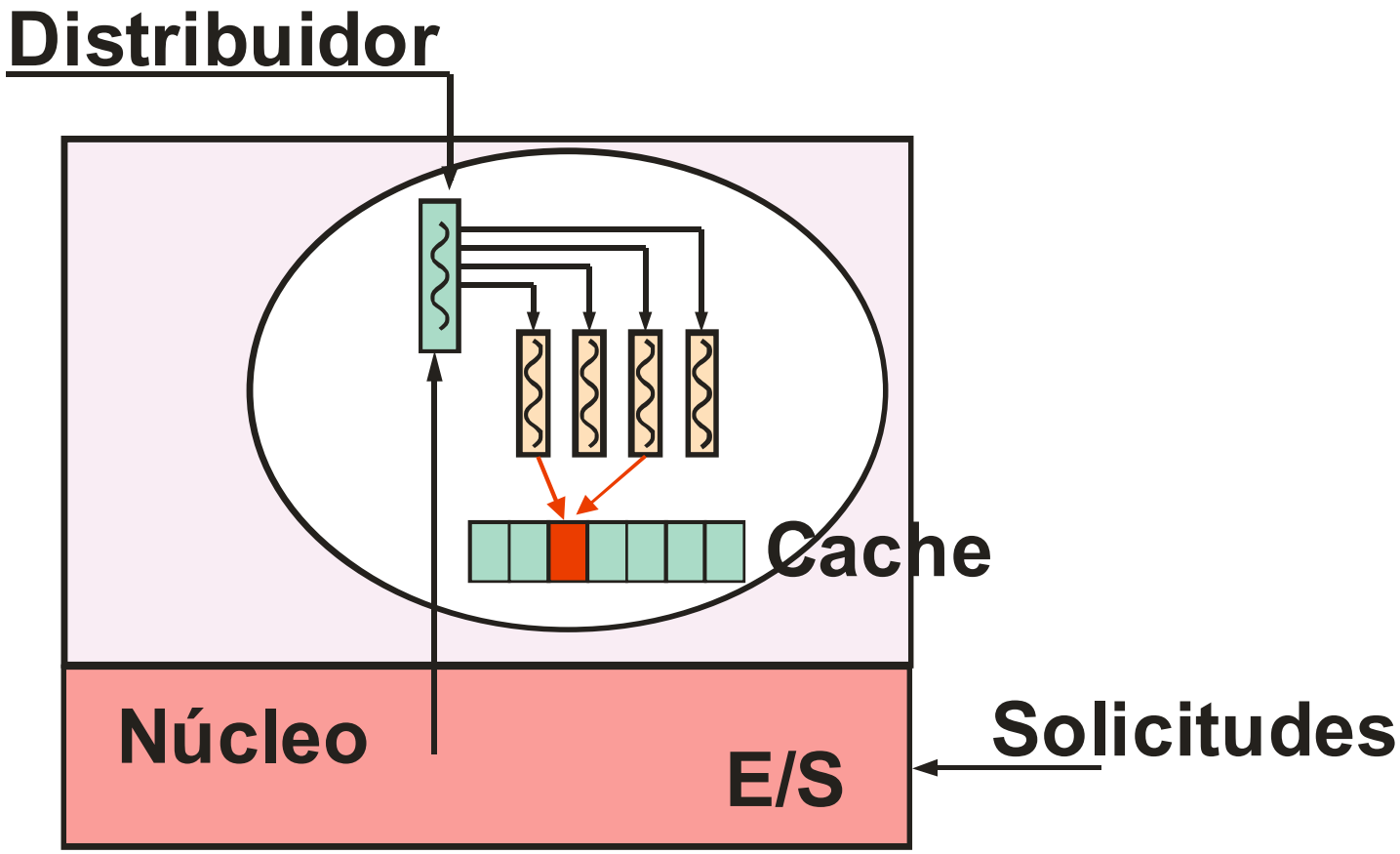
Conceptos previos

- Una condición de carrera es el acceso concurrente a los recursos compartidos sin coordinación
  - Puede producir resultados diferentes a los esperados

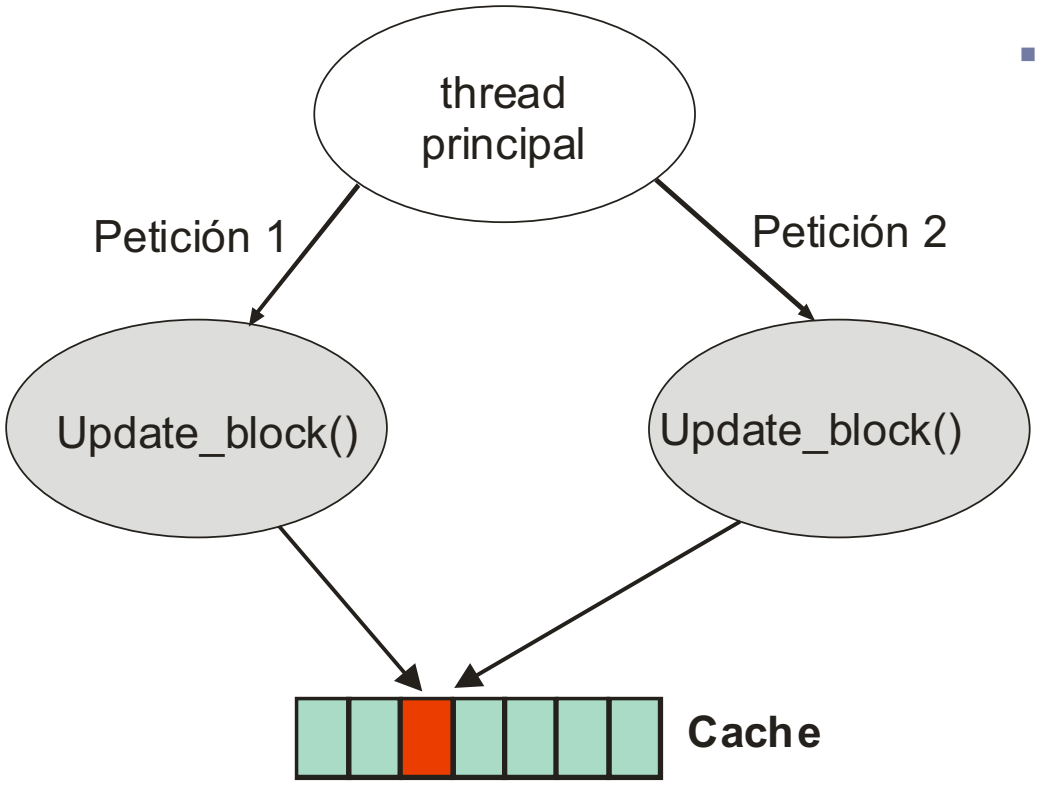
- Una sección crítica (SC) es un fragmento de código en el que los distintos procesos pueden acceder y modificar recursos compartidos

- Una sección es atómica si se ejecuta por un *thread* completamente de manera indivisible, de principio a fin, sin que su ejecución se pueda interrumpir por ningún otro *thread*
  - Tienen una semántica de éxito-o-fallo

Ejemplo: Servidor concurrente



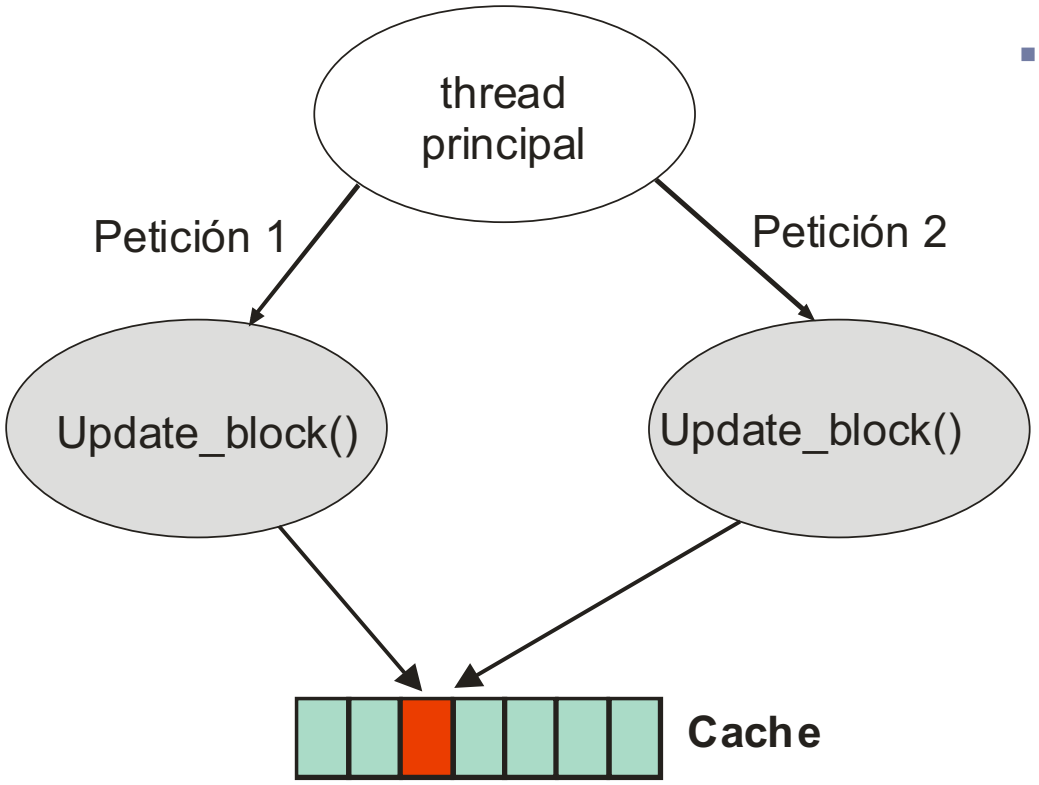
Ejemplo II:
Acceso a una caché compartida



- Condición de carrera en la sentencia: `Update_block`

```
Update_block(bloque) {
      b = Get_block();
      copy(b, bloque);
}

copy(b1, b2){
      for (i=0; i<length;i++)
        *b1++ = *b2++;
```

Acceso a una caché compartida



- Condición de carrera en la sentencia: `Update_block`

```
Update_block(bloque) {
      b = Get_block();
      copy(b, bloque);
}
```

```
copy(b1, b2){
      for (i=0; i<length;i++)
        *b1++ = *b2++;
```

Operación no atómica

El problema de la sección crítica

- Sistema compuesto por *n* procesos
  - Cada uno tiene un fragmento de código: sección crítica
  - Los procesos deben acceder a la SC con exclusividad

- **Estructura general** de cualquier mecanismo utilizado para resolver el problema de la sección crítica:

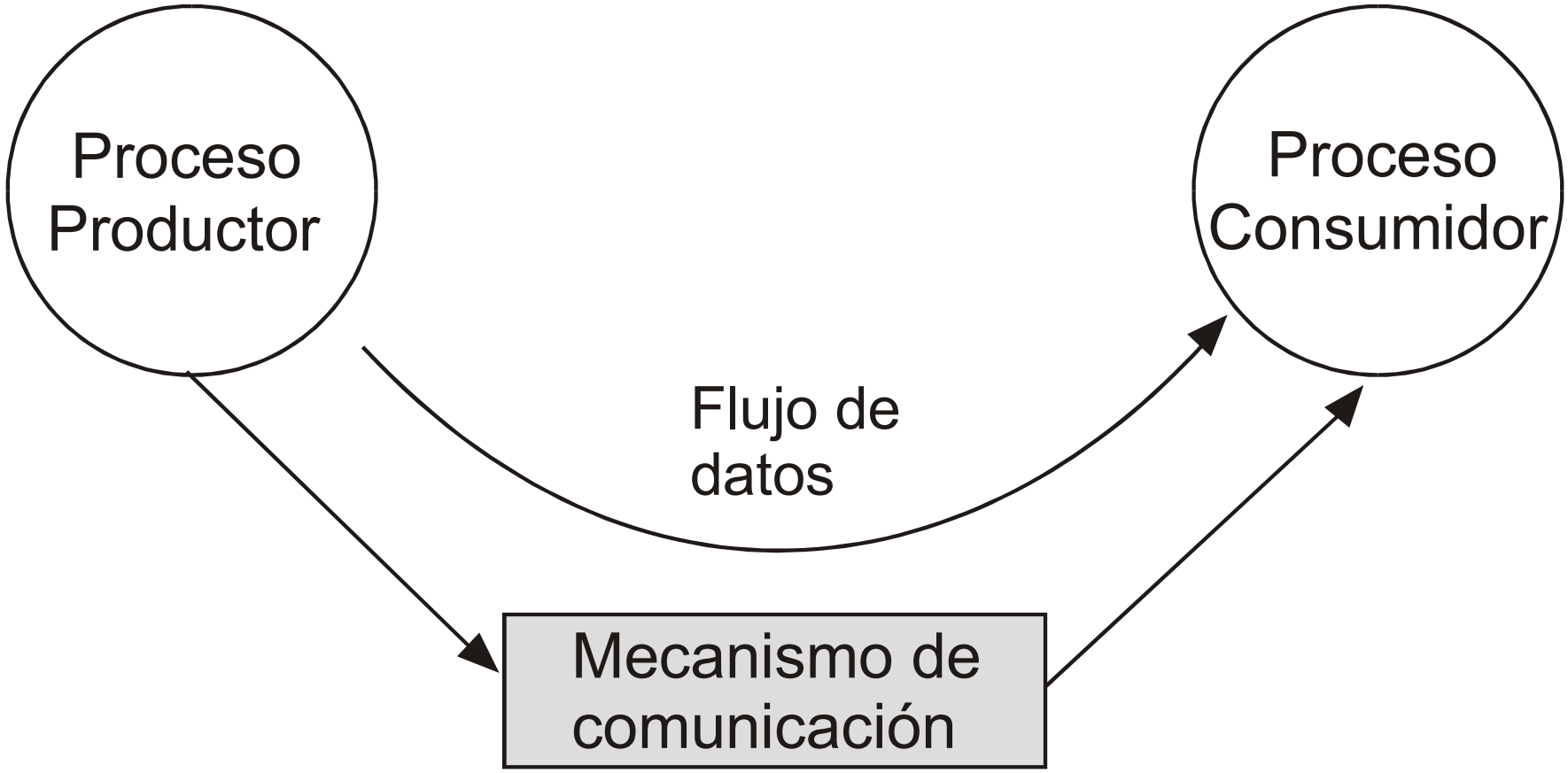
```
Entrada en la sección crítica
Código de la sección crítica
Salida de la sección crítica
```

- Requisitos que debe ofrecer cualquier solución para resolver el problema de la sección crítica:
  - Exclusión mutua
  - Progreso
  - Espera limitada

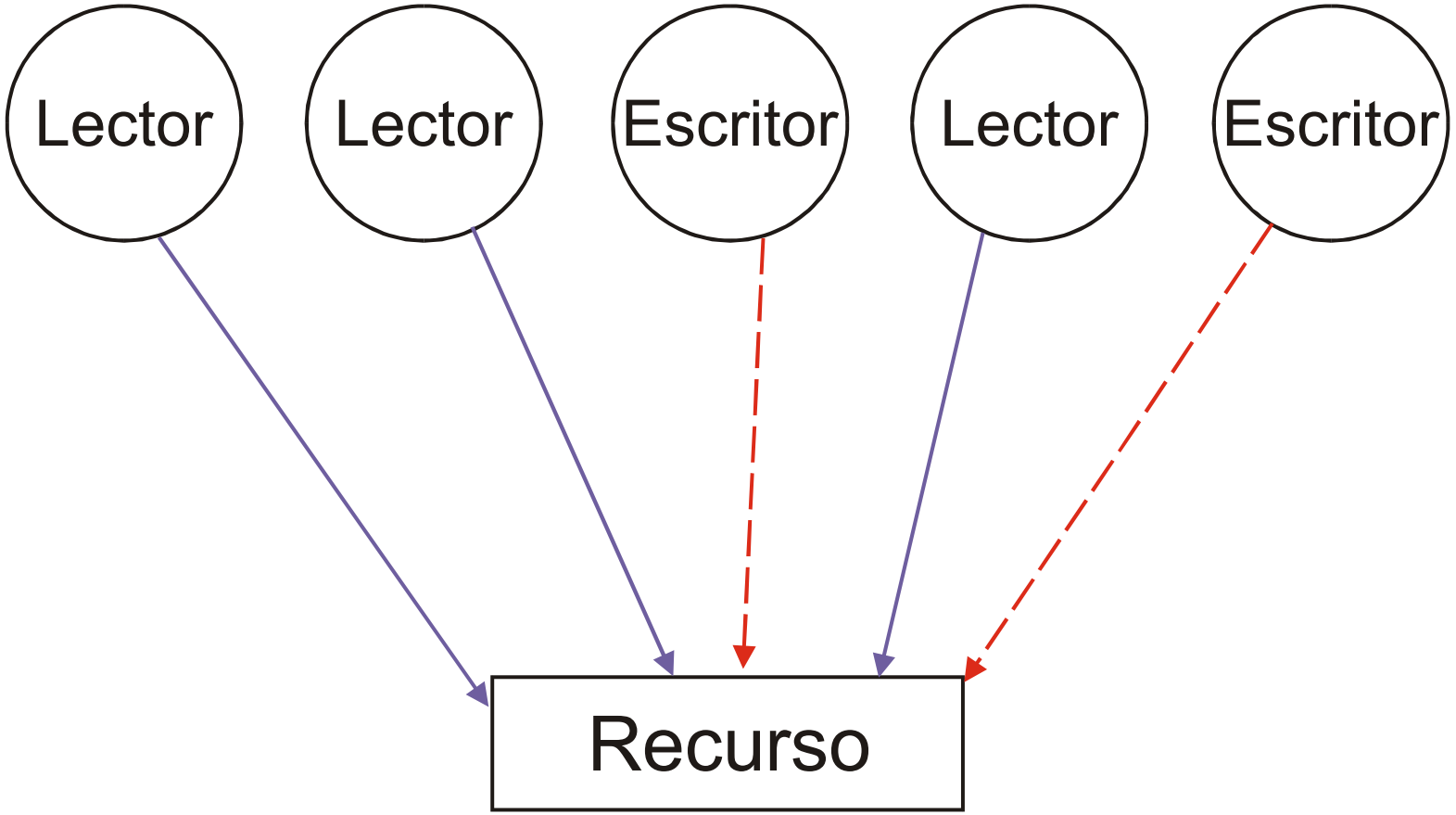
Modelos típicos de comunicación y sincronización

- Modelo productor-consumidor
- Modelo de lectores-escritores
- Modelo cliente-servidor

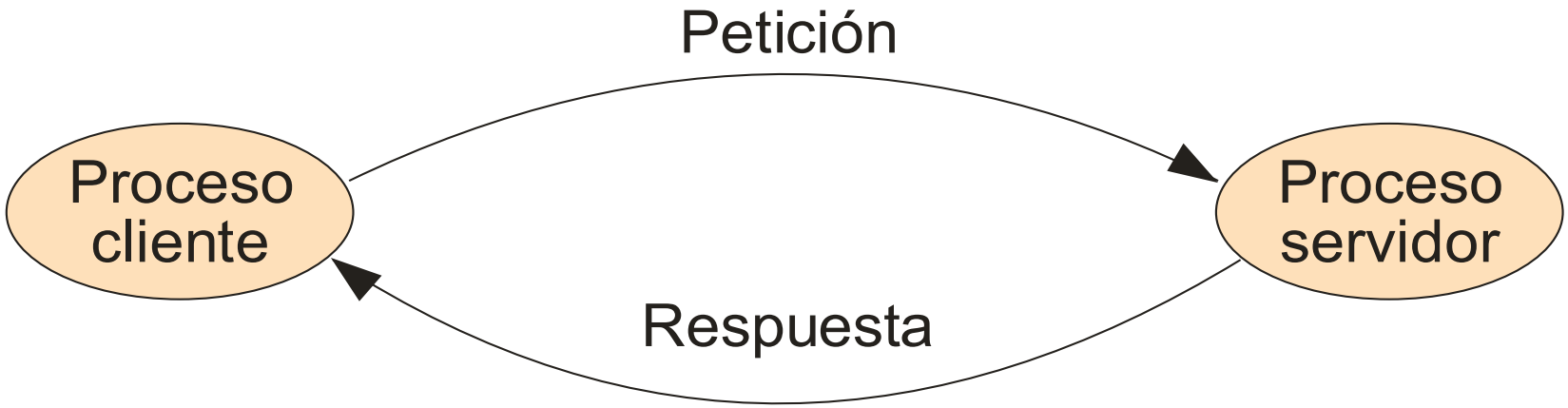
Modelo productor-consumidor



Modelo de lectores-escritores

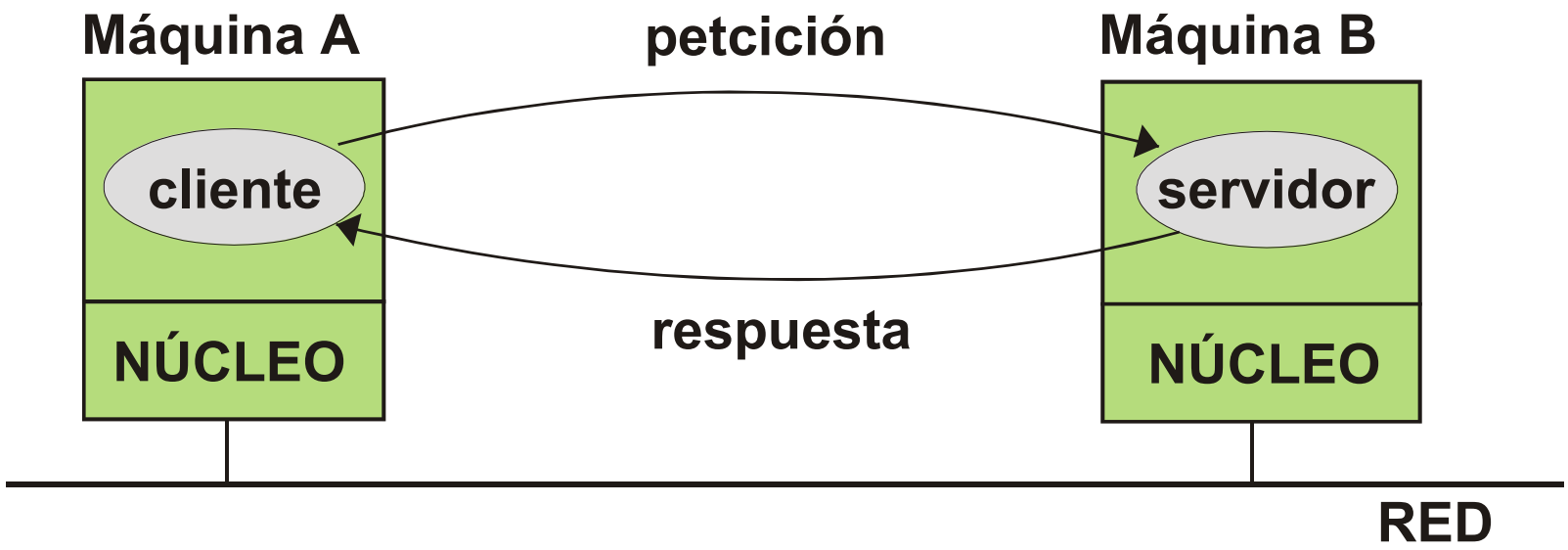


Modelo cliente-servidor



Comunicación cliente-servidor

- Muy utilizada en entornos distribuidos (más del **90%** de los sistemas distribuidos utilizan la arquitectura cliente-servidor)



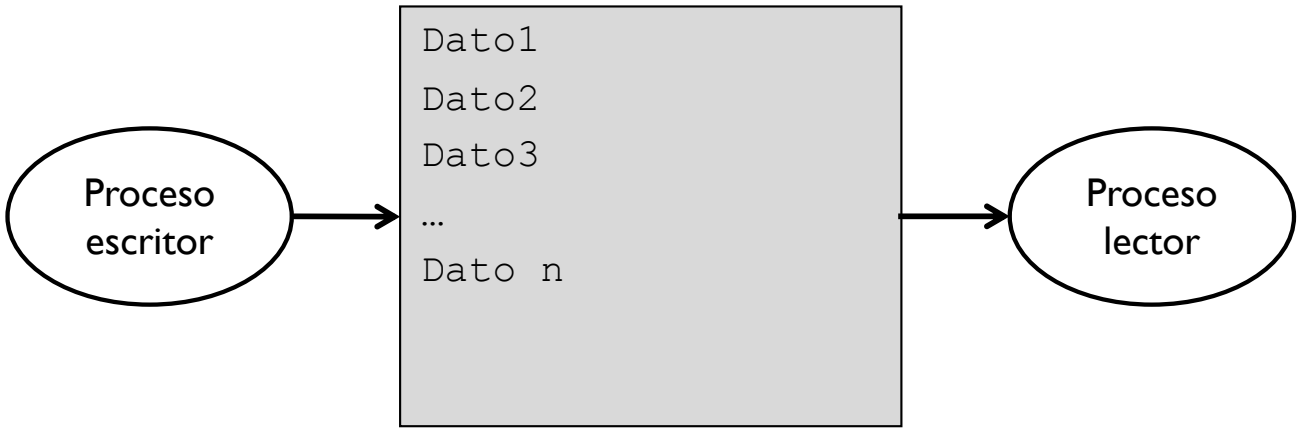
- Protocolo típico: petición-respuesta

Mecanismos de comunicación

- Ficheros
- Tuberías (*pipes*, FIFOS)
- Variables en memoria compartida
- Paso de mensajes

Comunicación mediante archivos

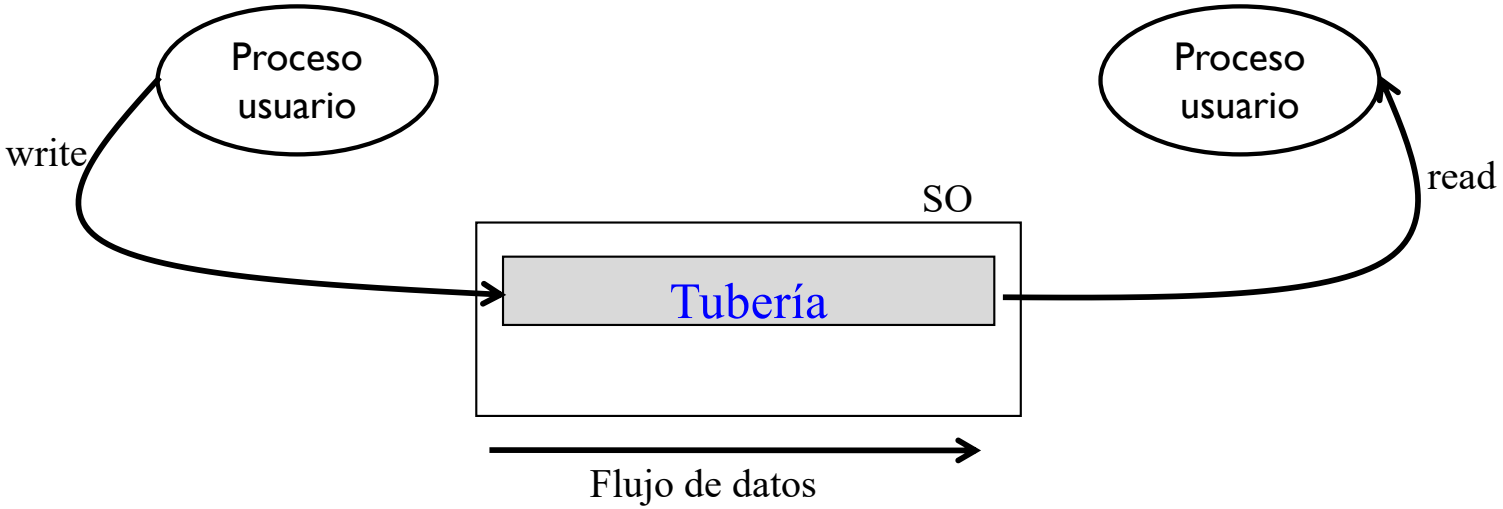
- Un archivo es un mecanismo que puede emplearse para comunicar procesos
  - Un proceso puede escribir datos y otro puede leerlos



Comunicación mediante tuberías

- Una tubería (pipe) es un mecanismo de comunicación y sincronización
  - Comunicación: archivo compartido
  - Sincronización: semántica bloqueante

- Puede verse como un *pseudoarchivo* identificado por dos descriptores, uno se usa para leer y el otro para escribir
  - Los servicios utilizados para lectura y escritura de *pipes* son los mismos que los usados sobre ficheros



Operaciones en un *pipe*

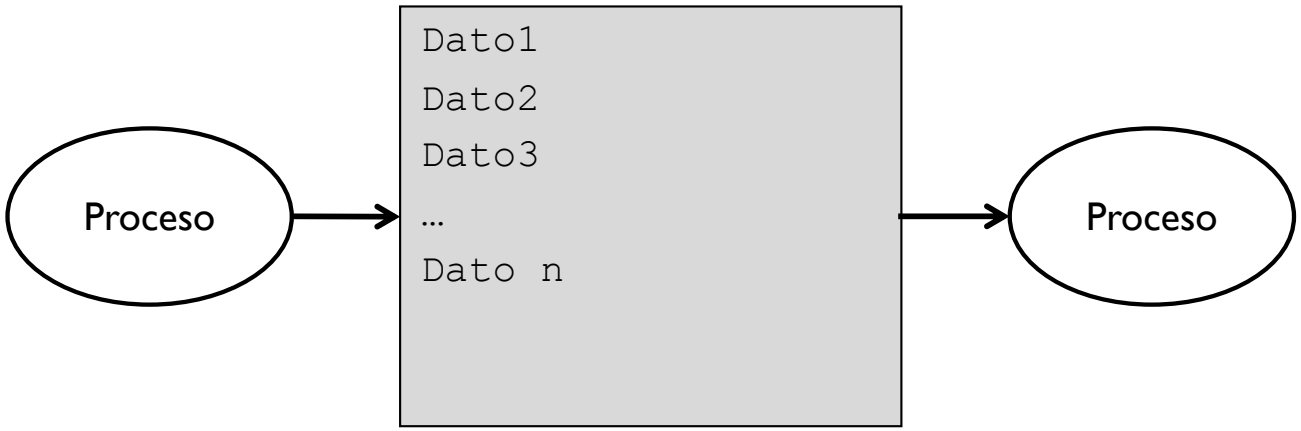
- **`read`**`(fildes[0], buffer, n)`
  - Pipe vacío se bloquea el lector
  - Pipe con `p` bytes
    - Si `p ≥ n` devuelve `n`
    - Si `p < n` devuelve `p`
  - Si pipe vacío y no hay escritores devuelve 0
- **`write`**`(fildes[1], buffer, n)`
  - Pipe lleno se bloquea el escritor
  - Si no hay lectores se recibe la señal SIGPIPE
- Lecturas y escrituras atómicas (cuidado con tamaños grandes)

Ejemplo III: Productor-consumidor con tuberías

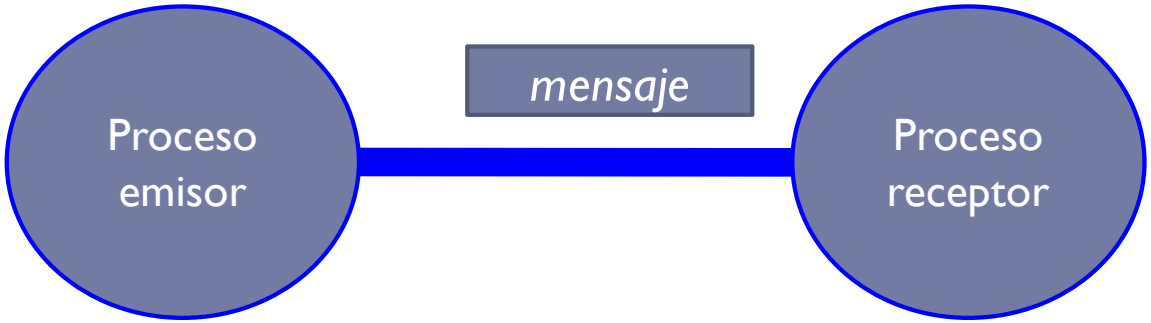
```
#include <stdio.h>    /* printf */
#include <unistd.h>
struct elemento dato;          /* dato a producir */
int fildes[2];
int main () {
   if (pipe(fildes)<0){
          printf("Error al crear la tubería);
          exit(1);
   }
   if (fork() == 0){
          for (;;){
                    <Producir dato>
                    write (fildes[1], (char*)&dato, sizeof (struct elemento));
          }
   }else{
          for (;;){
                    read (fildes[0], (char*)&dato, sizeof (struct elemento));
                    <Consumir dato>
          }
   }
```

Comunicación mediante memoria compartida

- Un proceso escribe un dato en una variable de memoria compartida y otro proceso accede a esa variable



Comunicación mediante paso de mensajes



Emisor:

send(destino, mensaje)

Receptor:

receive(origen, mensaje)

Mecanismos de sincronización

- Señales (asincronismo)
- Tuberías (pipes, FIFOs)
- Semáforos
- Mutex y variables condicionales
- Paso de mensajes

Semáforos

- Un semáforo [Dijkstra, 1965] es un mecanismo de sincronización de procesos que ejecutan en la misma máquina
- Idea muy simple:
  - Un semáforo es un objeto con un valor entero inicialmente no negativo
- Dos operaciones **atómicas**
  - `wait`
  - `signal`

Operaciones sobre semáforos

- Operación **wait**

```
wait(s)
{
  s = s - 1;
  if (s < 0) {
    <bloquear al proceso>
  }
}
```

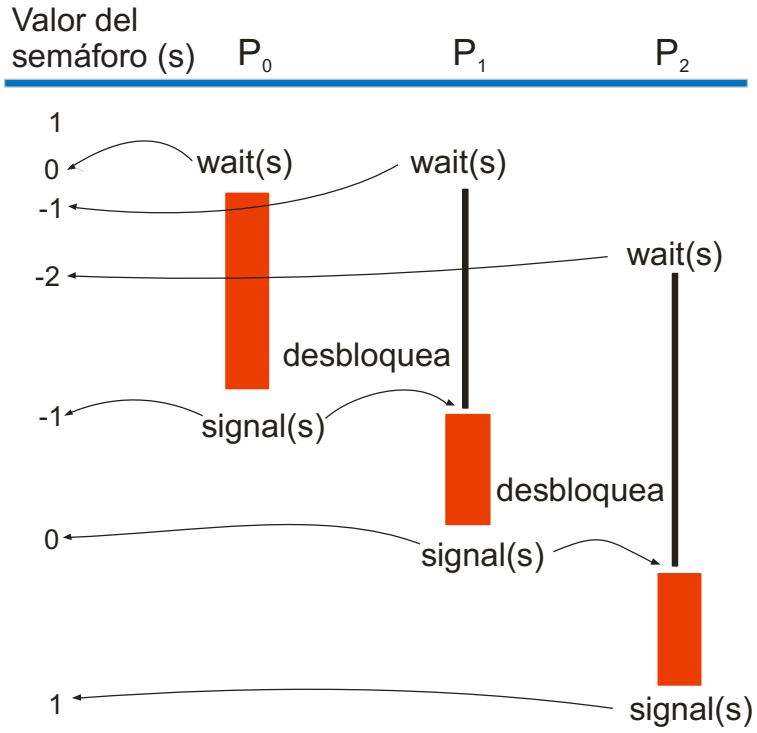
- Operación **signal**

```
signal(s)
 {
    s = s + 1;
    if (s <= 0)
       <Desbloquear a un
       proceso bloqueado por
       la operacion wait>
    }
}
```

Secciones críticas con semáforos

```
wait(s); /* entrada en la seccion critica */
< seccion critica >
signal(s);  /* salida de la seccion critica */
```

El semáforo debe tener valor inicial **1**



Ejecutando código de la sección crítica

Proceso bloqueado en el semáforo

s: valor entero asociado al semáforo

Estructura para projeger la SC:

```
wait(s)
Código de la sección crítica
signal(s)
```

Ejercicio

- ¿Cuál es el número máximo de procesos que pueden ejecutar una operación wait sobre un semáforo que se inicializó con un valor de 4? ¿Cuál es el número máximo de procesos que pueden bloquearse?

Solución

- ¿Cuál es el número máximo de procesos que pueden ejecutar una opeación wait sobre un semáforo que se inicializó con un valor de 4?
  - 4

- ¿Cuál es el número máximo de procesos que pueden bloquearse?
  - Potencialmente ilimitado. En la práctica tantos como procesos pueda soportar el sistema operativo

Ejercicio

- Utilizar un semáforo para asegurar que el inicio de una determinada actividad del proceso P2 comienza después de que finalice una actividad del proceso P1

Solución

```
sem = 0;   // semáforo
```

Proceso 1

```
    .
Acción1();
signal(sem);
    .
    .
```

Proceso 2

```
    .
wait(sem);
Accción2();
    .
    .
```

Ejercicio

- Escriba el código de dos procesos ligeros que deben alternar de forma estricta su ejecución. Resuelva el problema utilizando semáforos.

Solución

```
turno1 = 1;    //semáforos
turno2 = 0;
```

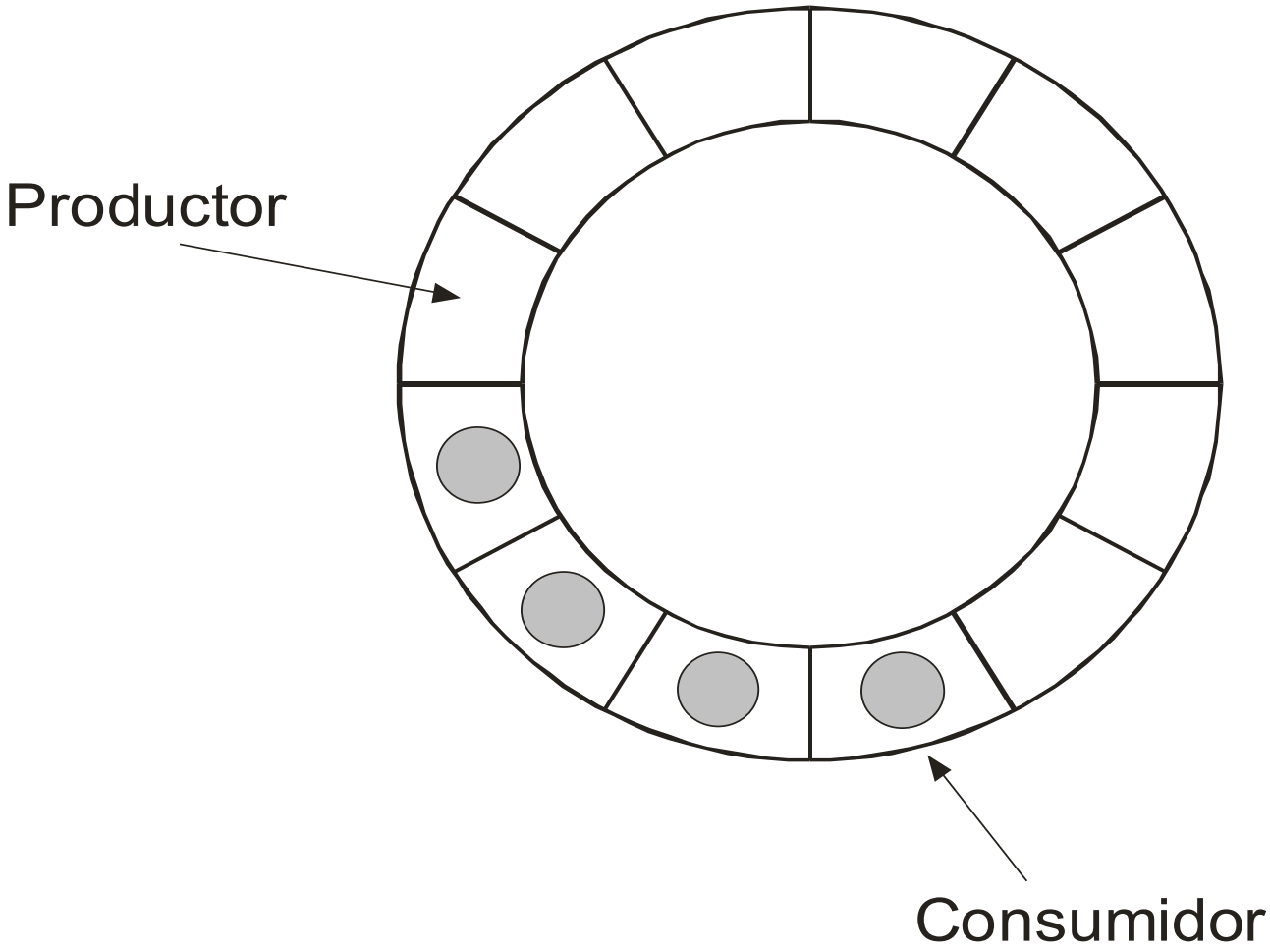
Proceso 1

```
while(1){
  wait(turno1);
  Acción1();
  signal(turno2);
}
```

Proceso 2

```
while (1){
     wait(turno2);
     Acción2();
     signal(turno1);
}
```

Productor-consumidor con semáforos



Productor-consumidor con semáforos

```
#define TAMAÑO_DEL_BUFFER 1024

Productor() {
   int posicion = 0;

   for(;;) {
         Producir un dato;
         wait(huecos);

         buffer[posicion] = dato;
         posicion = (posicion + 1)
            % TAMAÑO_DEL_BUFFER;

         signal(elementos);
   }
}
```

```
Consumidor() {
   int posición = 0;

   for(;;) {
      wait(elementos);

      dato = buffer[posicion];
      posicion = (posicion + 1) %
                 TAMAÑO_DEL_BUFFER;

      signal(huecos);
      Consumir el dato extraído;
   }
}
```

Lectores-escritores con semáforos

```
Lector() {

   wait(sem_lectores);
   n_lectores = n_lectores + 1;
   if (n_lectores == 1)
         wait(sem_recurso);
   signal(sem_lectores);

   Consultar el recurso compartido

   wait(sem_lectores);
   n_lectores = n_lectores - 1;
   if (n_lectores == 0)
         signal(sem_recurso);
   signal(sem_lectores);
}
```

```
Escritor(){

   wait(sem_recurso);

   /*se puede modificar el recurso*/

   signal(sem_recurso);
}
```

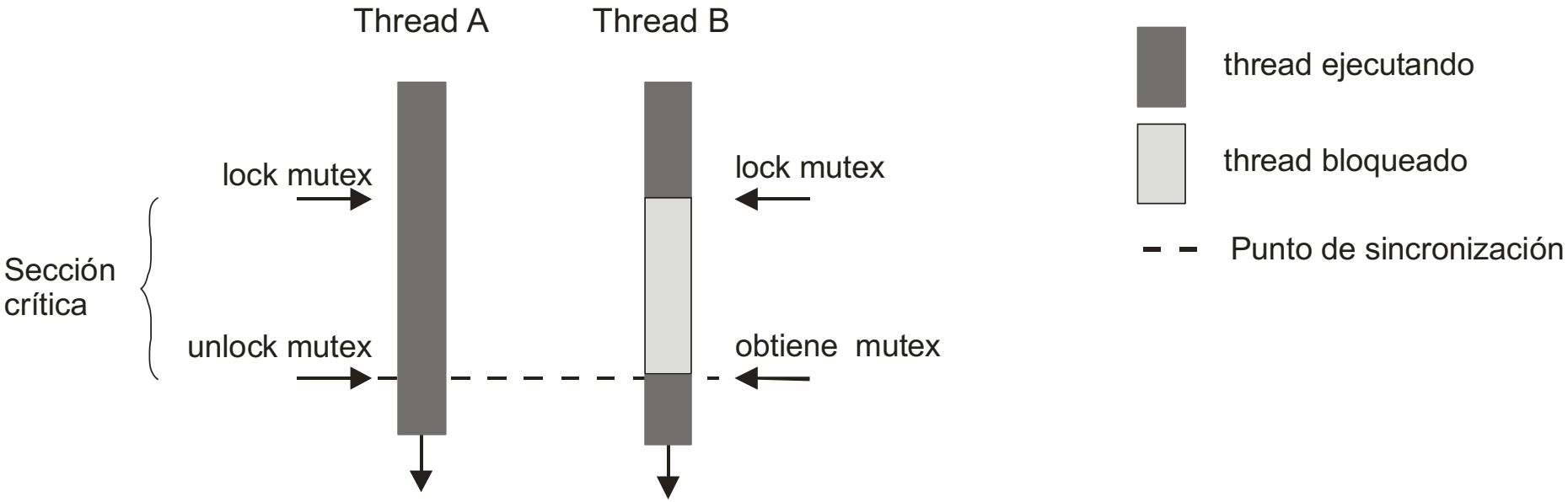
Mutex

- Un mutex es un mecanismo de sincronización especialmente indicado para procesos ligeros
- Se emplean para tener exclusión mutua sobre SC y tener acceso exclusivo a los recursos compartidos
- Dos operaciones atómicas:
  - **lock(m)**
    - Intenta bloquear el mutex.
    - Si el mutex ya está bloqueado el proceso se suspende
  - **unlock(m)**
    - Desbloquear el mutex.
    - Si existen procesos bloqueados en el mutex se desbloquea a uno
    - Esta operación la tiene que realizar el proceso que bloqueo el mutex (en los semáforos no es así)

Secciones críticas con mutex

```
lock(m);     /* entrada en la seccion critica */
< seccion critica >
unlock(m);   /* salida de la seccion critica */
```

- La operación `unlock` debe realizarla el proceso ligero que ejecutó `lock`



Ejercicio

- Utilizar un mutex para asegurar que el inicio de una determinada actividad del proceso P2 comienza después de que finalice una actividad del proceso P1

Ejercicio

- Utilizar un mutex para asegurar que el inicio de una determinada actividad del proceso P2 comienza después de que finalice una actividad del proceso P1

con mutex solo no se puede

Variables condicionales

- Una variable condicional es una variable de sincronización asociada a un mutex
  - Permite suspender la ejecución del proceso hasta que ocurra algún suceso
- Dos operaciones **atómicas**:
  - **c_wait**
    - Bloquea al proceso ligero que la ejecuta y libera el mutex
  - **c_signal**
    - Desbloquea a uno o varios procesos suspendidos en la variable condicional
    - El proceso que se despierta compite de nuevo por el mutex
- Estas dos operaciones hay que incluirlas entre un lock y un unlock

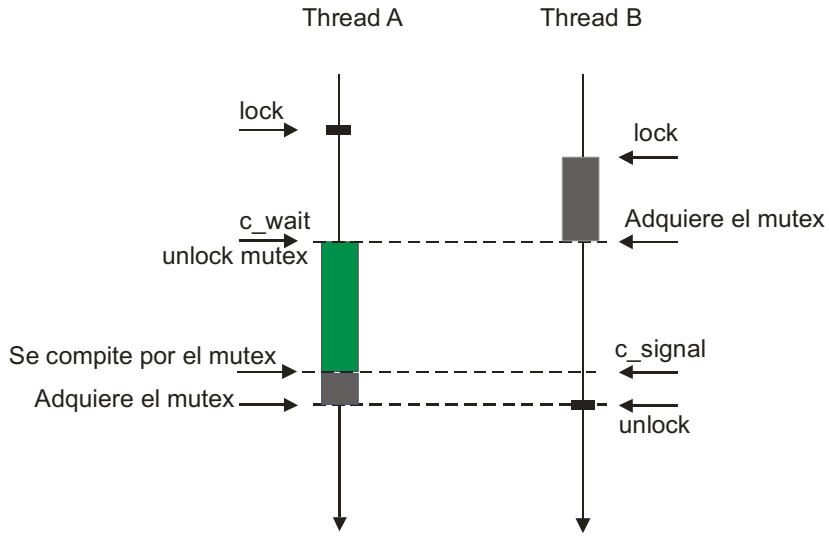
Uso de mutex y variables condicionales

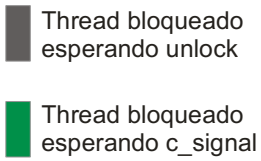
Thread A

```
lock(m);
/* Código de la sección crítica */
while (condición == FALSE)
  c_wait(c,m);   //SE SUSPENDE
/* Resto la sección crítica */
unlock(m); /* Salida de la SC */
```

Thread B

```
lock(m);
/* Código de la sección crítica */
/* Se modifica la condición */
condición = TRUE;
c_signal(c);   //DESPIERTA
/* Resto la sección crítica */
unlock(m); /* Salida de la SC */
```





Uso de mutex y variables condicionales

- Thread A

```
lock(mutex);  /* acceso al recurso */
comprobar las estructuras de datos;
while (recurso ocupado)
        wait(condition, mutex);
marcar el recurso como ocupado;
unlock(mutex);
```

- Thread B

```
lock(mutex);  /* acceso al recurso */
marcar el recurso como libre;
signal(condition);
unlock(mutex);
```

- Importante utilizar `while`

Ejercicio

- Utilizar un mutex y variables condicionales para asegurar que el inicio de una determinada actividad del proceso P2 comienza después de que finalice una actividad del proceso P1

Solución

```
mutex M;
cond  C;
listo = 0;
```

Thread 1

```
      .
Acción1();
lock(M);
listo = 1;
signal(c)
unlock(M);
      .
```

Thread 2

```
      .
lock(M);
      while(listo==0)
            wait(C,M);
unlock(M);
Accción2();
      .
```

Ejercicio

- Escriba el código de dos procesos ligeros que deben alternar de forma estricta su ejecución. Resuelva el problema utilizando mutex y variables condicionales.

Solución

```
mutex M;
cond  C;
turno = 1;
```

Thread 1

```
while(1){
    lock(M);
    while(turno != 1)
        wait(C,M);
    unlock(M);
    Acción1();
    lock(M);
    turno = 2;
    signal(C);
    unlock(M);
}
```

Thread 2

```
while(1) {
    lock(M);
    while(turno != 2)
        wait(C,M);
    unlock(M);
    Acción2();
    lock(M);
    turno=1;
    signal(C);
    unlock(M);
}
```

Ejercicio

- El siguiente fragmento de programa muestra el uso de una variable condicional, cond, y su mutex asociado, mutex. El objetivo de este fragmento de código es bloquear a un proceso ligero hasta que la variable ocupado tome valor *false*.

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

- Por otra parte, el código que permite bloquear al proceso ligero que ejecute el fragmento anterior es el siguiente:

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

Ejercicio (cont.)

- Considerando que el valor de la variable ocupado es *true*, y que existen dos procesos ligeros en el sistema (procesos ligeros A y B), se pide:

a) ¿Es posible la siguiente secuencia de ejecución: A1, A2, B6, B7, B8, A2, A3, B9, donde A1 indica que el proceso ligero A ejecuta la sentencia 1, y así sucesivamente.

Solución

- Considerando que el valor de la variable ocupado es *true*, y que existen dos procesos ligeros en el sistema (procesos ligeros A y B), se pide:

a) ¿Es posible la siguiente secuencia de ejecución: A1, A2, B6, B7, B8, A2, A3, B9, donde A1 indica que el proceso ligero A ejecuta la sentencia 1, y así sucesivamente.

No, porque el mutex está bloqueado en la acción A1 y no se podría adquirir en la acción B6

Productor-consumidor con mutex y variables condicionales

```
Productor() {
   int pos = 0;
   for(;;) {
         < Producir un dato >
         lock(mutex);
         /* acceder al buffer */
         while (n_elementos ==
                  TAMAÑO_DEL_BUFFER)
             c_wait(lleno, mutex);

         buffer[pos] = dato;
         pos = (pos + 1)
               % TAMAÑO_DEL_BUFFER;
         n_elementos ++;

         if (n_elementos == 1)
             c_signal(vacio);
         unlock(mutex);
   }
}
```

```
Consumidor() {
   int = 0;
   for(;;) {
         lock(mutex);

         while (n_elementos == 0)
             c_wait(vacío, mutex);

         dato = buffer[pos];
         pos = (pos + 1)
                % TAMAÑO_DEL_BUFFER;
         n_elementos --;

         if (n_elementos ==
           (TAMAÑO_DEL_BUFFER - 1));
            c_signal(lleno);

         unlock(mutex);
         < Consumir el dato >
   }
}
```

Lectores-escritores con mutex
Posible solución

```
Lector() {
   lock(mutex_lectores);
   n_lectores ++;
   if (n_lectores == 1)
         lock(mutex_recurso);
   unlock(mutex_lectores);

   // SE CONSULTA/LEE EL RECURSO

   lock(mutex_lectores);
   n_lectores -- ;
   if (mutex_lectores == 0)
      unlock(mutex_recurso);
   unlock(mutex_lectores);
}
```

```
Escritor() {

   lock(mutex_recurso);

   //  SE MODIFICA EL RECURSO

   unlock(mutex_recurso);
}
```

¿Qué problema tiene esta solución?

Lectores-escritores con mutex
¿Qué problema tiene esta solución?

```
Lector() {
   lock(mutex_lectores);
   n_lectores ++;
   if (n_lectores == 1)
         lock(mutex_recurso);
   unlock(mutex_lectores);

   // SE CONSULTA/LEE EL RECURSO

   lock(mutex_lectores);
   n_lectores -- ;
   if (mutex_lectores == 0)
      unlock(mutex_recurso);
   unlock(mutex_lectores);
}
```

```
Escritor() {

   lock(mutex_recurso);

   //  SE MODIFICA EL RECURSO

   unlock(mutex_recurso);
}
```

La operación marcada en rojo la puede ejecutar un thread que no ejecutó la correspondiente operación lock

Solución que da prioridad a los lectores

```
Lector() {
    lock(mutex_lectores);
    n_lectores ++;
    if (n_lectores == 1) {
          th_id = pthread_self();
          lock(mutex_recurso);
    }
    unlock(mutex_lectores);

    // SE CONSULTA/LEE EL RECURSO

    lock(mutex_lectores);
    n_lectores -- ;
    if (th_id == pthread_self() {
          while (n_lectores > 0)
                  wait(n_lectores, cond);
          unlock(mutex_recurso);
    }
    else
          signal(cond);
    unlock(mutex_lectores);
}
```

Servicios POSIX para mutex

- Inicializa un mutex

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

- Destruye un mutex

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

- Intenta obtener el mutex. Bloquea al proceso ligero si el mutex se encuentra adquirido por otro proceso ligero

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

- Desbloquea el mutex

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

Servicios POSIX para variables condicionales

- Inicializa una variable condicional

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

- Destruye una variable condicional

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

- Bloquea al proceso ligero y libera el mutex:
- Suspende al proceso ligero hasta que otro proceso señaliza la variable condicional `cond`.
- Automáticamente se libera el `mutex`. Cuando se despierta el proceso ligero vuelve a competir por el `mutex`.

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

Servicios POSIX para variables condicionales

- Desbloquea el/los proceso(s) ligero(s) y libera el mutex

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

- Se reactivan uno o más de los procesos ligeros que están suspendidos en la variable condicional `cond`

- No tiene efecto si no hay ningún proceso ligero esperando (diferente a los semáforos)

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

- Todos los threads suspendidos en la variable condicional `cond` se reactivan

- No tiene efecto si no hay ningún proceso ligero esperando

Otros servicios de sincronización

- Barreras de sincronización

```
int pthread_barrier_wait   (pthread_barrier_t *barrier);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_init (pthread_barrier_t *barrier,
                          const pthread_barrier attr_t *restrict attr,
                          unsigned count);
```

Productor-consumidor con mutex

```
#define MAX_BUFFER          1024      /* tamanio del buffer */
#define DATOS_A_PRODUCIR    100000    /* datos a producir */

pthread_mutex_t mutex;      /* mutex para controlar el acceso al buffer
   compartido */
pthread_cond_t no_lleno;    /* controla el llenado del buffer */
pthread_cond_t no_vacio;    /* controla el vaciado del buffer */
int n_elementos;            /* numero de elementos en el buffer */

int buffer[MAX_BUFFER];     /* buffer comun */

int main(int argc, char *argv[]){
    pthread_t th1, th2;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&no_lleno, NULL);
    pthread_cond_init(&no_vacio, NULL);
```

Productor-consumidor con mutex

```
    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);

    return 0;
}
```

Productor-consumidor con mutex

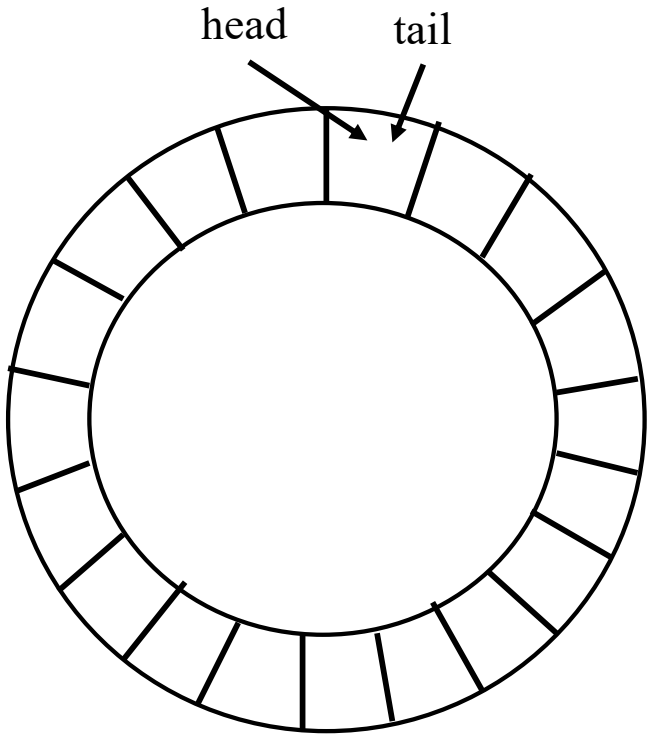
```
void Productor(void) {
   int dato, i ,pos = 0;

   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(&lleno, &mutex); /* se bloquea*/

         buffer[pos] = i;

         pos = (pos + 1) % MAX_BUFFER;
         n_elementos = n_elementos + 1;

         if (n_elementos == 1)
                  pthread_cond_signal(&vacio);    /* buffer no vacío*/

         pthread_mutex_unlock(&mutex);
   }
   pthread_exit(0);
}
```

Productor-consumidor con mutex

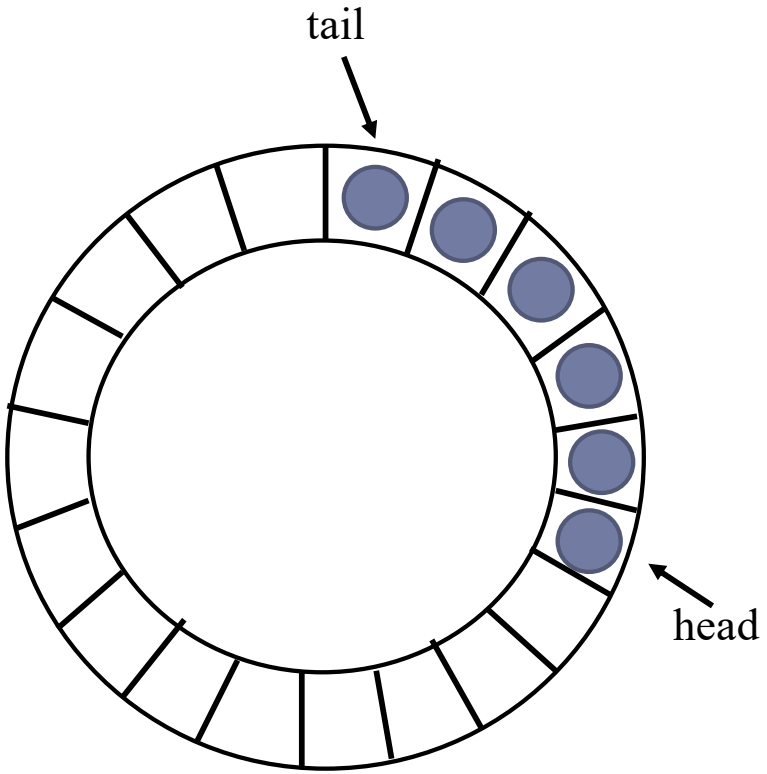
```
void Consumidor(void)  {
   int dato, i ,pos = 0;

   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(&vacio, &mutex); /* se bloquea */

        dato = buffer[pos];

        pos = (pos + 1) % MAX_BUFFER;
        n_elementos = n_elementos - 1 ;

        if (n_elementos == MAX_BUFFER - 1);
                pthread_cond_signal(&lleno);/* buffer no lleno */

        pthread_mutex_unlock(&mutex);

        printf("Consume %d \n", dato);  /* consume dato */
   }
   pthread_exit(0);
}
```

Programación concurrente libre de cerrojos

- Cola con un buffer circular para un único productor-único consumidor



cola vacía



cola con elementos

Programación concurrente libre de cerrojos

- Implementación de una cola con cerrojos para un único productor-único consumidor

```
enqueue(data){
   lock(queue);

   if (NEXT(head)==tail) {
        unlock(queue);
        return EQUEUEBLOCK;
   }
   buffer[head]= data;
   head = NEXT(head);

   unlock(queue);
   return 0;
}
```

```
dequeue (data){
   lock(queue);

   if (head==tail) {
        unlock(queue);
        return EQUEUEBLOCK;
   }
   data = buffer[tail];
   tail= NEXT(tail);

   unlock(queue);
   return 0;
}
```

Programación concurrente libre de cerrojos

- Implementación libre de cerrojos para un único productor-único consumidor

```
enqueue(data){
    if (NEXT(head)==tail){
         return EQUEUEBLOCK;
    }
    buffer[head]= data;
    head = NEXT(head);
    return 0;
}
```

```
dequeue (data){
    if (head==tail) {
         return EQUEUEBLOCK;
    }
    data = buffer[tail];
    tail= NEXT(tail);
    return 0;
}
```

- Bajo consistencia secuencial, para un único productor-único consumidor, la solución anterior no requiere bloqueos
- Reduce la sobrecarga de ejecución

Threads en C

- Similares a los threads de POSIX
- Definidos en <threads.h>
- Creación de un thread:
  - `int thrd_create( thrd_t *thr, thrd_start_tfunc, void *arg );`
- Terminación de un thread:
  - `void thrd_exit( int res );`
- Convierte a un thread en independiente. Sus recursos serán liberados cuando el thread finalice:
  - `int thrd_detach( thrd_t thr );`
- Espera la terminación de un thread:
  - `int thrd_join( thrd_t thr, int *res );`
- Obtener identificador del thread que está ejecutando:
  - `thrd_t thrd_current( );`

Ejemplo

```
#include <stdio.h>    /* printf */
#include <threads.h> /* Para threads… */

#define NUM_THREADS 10

void funcion(int *idThread){
        int thid = *idThread;
        printf("Thread id = %ld\ti=%d: \n", thrd_current(),  thid);
        thrd_exit(0);
}
int main () {
        thrd_t arrayThread[NUM_THREADS]; /* Array de threads */
        int i;
        /* CREAR THREADS */
        for(i=0;i<NUM_THREADS;i++){
          if (thrd_create(&arrayThread[i],(void *)funcion, &i)==-1)
                printf("Error al crear proceso ligero\n");
        }
        /* ESPERAR TERMINACIÓN DE THREADS */
        for(i=0;i<NUM_THREADS;i++)
                thrd_join (arrayThread[i], NULL);
        return(0);
}
```

Ejemplo

```
#include <stdio.h>    /* printf */
#include <threads.h> /* Para threads… */

#define NUM_THREADS 10

void funcion(int *idThread){
        int thid = *idThread;
        printf("Thread id = %ld\ti=%d: \n", thrd_current(),  thid);
        thrd_exit(0);
}
int main () {
        thrd_t th;
        int i;
        /* CREAR THREADS */
        for(i=0;i<NUM_THREADS;i++){
          if (thrd_create(&th,(void *)funcion, &i)==-1)
                printf("Error al crear proceso ligero\n");
          thrd_detach(th);
        }
        return(0);
}
```

Mutex en C

- Tipo indentificador de un mutex: **`mtx_t`**
- Creación de un mutex:
  - `int mtx_init( mtx_t* mutex, int type );`
  - Tipo:
    - `mtx_plain`: mutex normal
    - `mtx_recursive`: mutex recursivo
- Destrucción de un mutex:
  - `void mtx_destroy( mtx_t *mutex );`
- Operaciones lock y unlock:
  - `int mtx_lock( mtx_t * mutex );`
  - `int mtx_unlock( mtx_t *mutex );`

Variables condicionales en C

- Tipo de una variable condicional: **`cnd_t`**
- Inicialización:
  - `int cnd_init( cnd_t* cond );`
- Destrucción:
  - `void cnd_destroy( cnd_t * cond );`
- Operaciones:
  - `int cnd_signal( cnd_t  *cond );`
  - `int cnd_wait( cnd_t * cond, mtx_t * mutex );`
  - `int cnd_broadcast( cnd_t *cond );`