

# SISTEMAS OPERATIVOS: COMUNICACIÓN Y SINCRONIZACIÓN ENTRE PROCESOS



Desarrollo de servidores concurrentes

# A recordar...

Antes de clase

Clase

Después de clase

Preparar los pre-requisitos.

Estudiar el material asociado a la **bibliografía**:  
las transparencias solo no son suficiente.  
Preguntar dudas (especialmente tras estudio).

Ejercitar las competencias:

- ▶ Realizar todos los **ejercicios**.
- ▶ Realizar los **cuadernos de prácticas** y las **prácticas** de forma progresiva.

# Lecturas recomendadas

## Base



1. Carretero 2020:
  1. Cap. 6
2. Carretero 2007:
  1. Cap. 6.1 y 6.2

## Recomendada



1. Tanenbaum 2006:
  1. (es) Cap. 5
  2. (en) Cap. 5
2. Stallings 2005:
  1. 5.1, 5.2 y 5.3
3. Silberschatz 2006:
  1. 6.1, 6.2, 6.5 y 6.6

# Contenidos

- Introducción (definiciones):
  - ▣ Procesos concurrentes.
  - ▣ Concurrencia, comunicación y sincronización
  - ▣ Sección crítica y condiciones de carrera
  - ▣ Exclusión mutua y sección crítica.
- Mecanismos de sincronización (I):
  - ▣ Primitivas básicas iniciales
  - ▣ Semáforos.
- Problemas clásicos de concurrencia (I):
  - ▣ Productor-consumidor
  - ▣ Lectores-escriitores
- Mecanismos de sincronización de threads (II)
  - ▣ Semáforos
    - Llamadas al sistema para semáforos.
    - Problemas clásicos de concurrencia.
  - ▣ Mutex y variables condición
    - Llamadas al sistema para mutex.
    - Problemas clásicos de concurrencia.

- **Desarrollo de servidores concurrentes**
  - ▣ Servidores de peticiones.
  - ▣ Solución basada en procesos.
  - ▣ Solución basada en hilos bajo demanda.
  - ▣ Solución basada en pool de hilos.

# Contenidos

- Introducción (definiciones):
  - ▣ Procesos concurrentes.
  - ▣ Concurrencia, comunicación y sincronización
  - ▣ Sección crítica y condiciones de carrera
  - ▣ Exclusión mutua y sección crítica.
- Mecanismos de sincronización (I):
  - ▣ Primitivas básicas iniciales
  - ▣ Semáforos.
- Problemas clásicos de concurrencia (I):
  - ▣ Productor-consumidor
  - ▣ Lectores-escriptores
- Mecanismos de sincronización de threads (II)
  - ▣ Semáforos
    - Llamadas al sistema para semáforos.
    - Problemas clásicos de concurrencia.
  - ▣ Mutex y variables condición
    - Llamadas al sistema para mutex.
    - Problemas clásicos de concurrencia.

## □ Desarrollo de servidores concurrentes

### ▣ **Servidores de peticiones.**

### ▣ Solución basada en procesos.

### ▣ Solución basada en hilos bajo demanda.

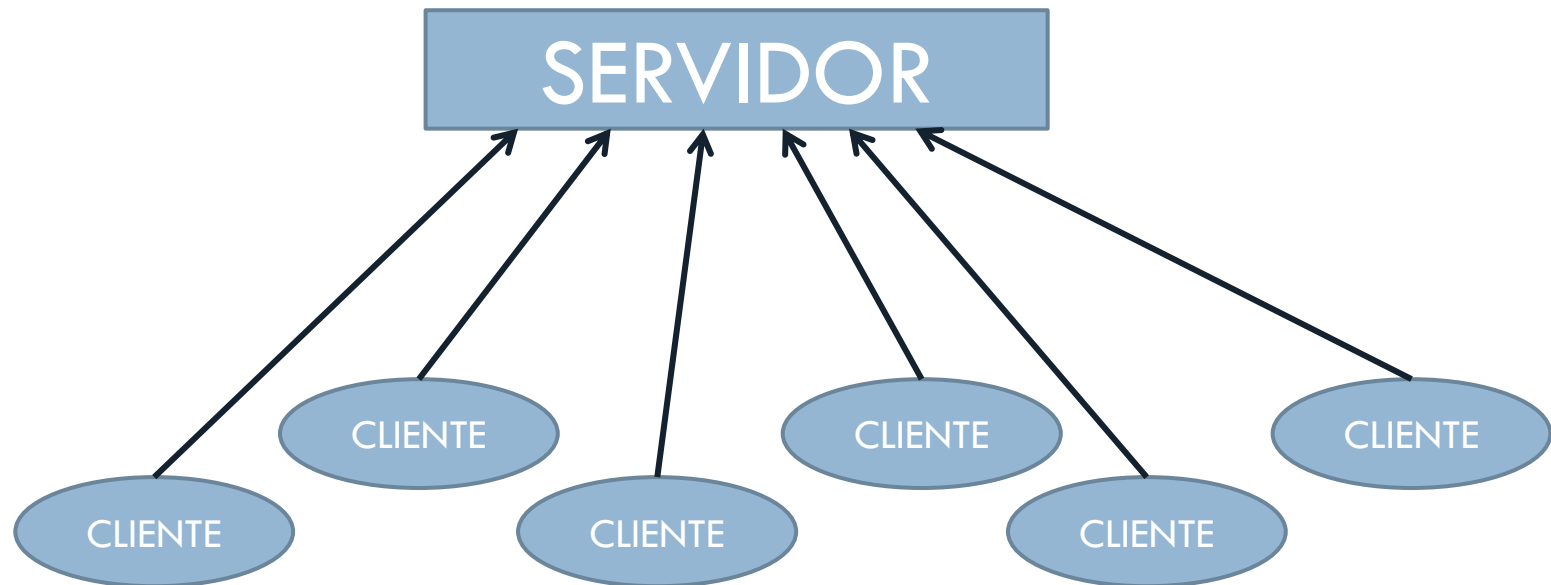
### ▣ Solución basada en pool de hilos.

# Servidor de peticiones

- Un servidor recibe peticiones que debe procesar.
- En muchos contextos se desarrollan servidores de peticiones:
  - ▣ Servidor Web.
  - ▣ Servidor de Base de datos.
  - ▣ Servidor de aplicaciones.
  - ▣ Programa de intercambio de ficheros.
  - ▣ Aplicaciones de mensajería.
  - ▣ ...

# Servidor de peticiones

- Un servidor recibe peticiones que debe procesar.



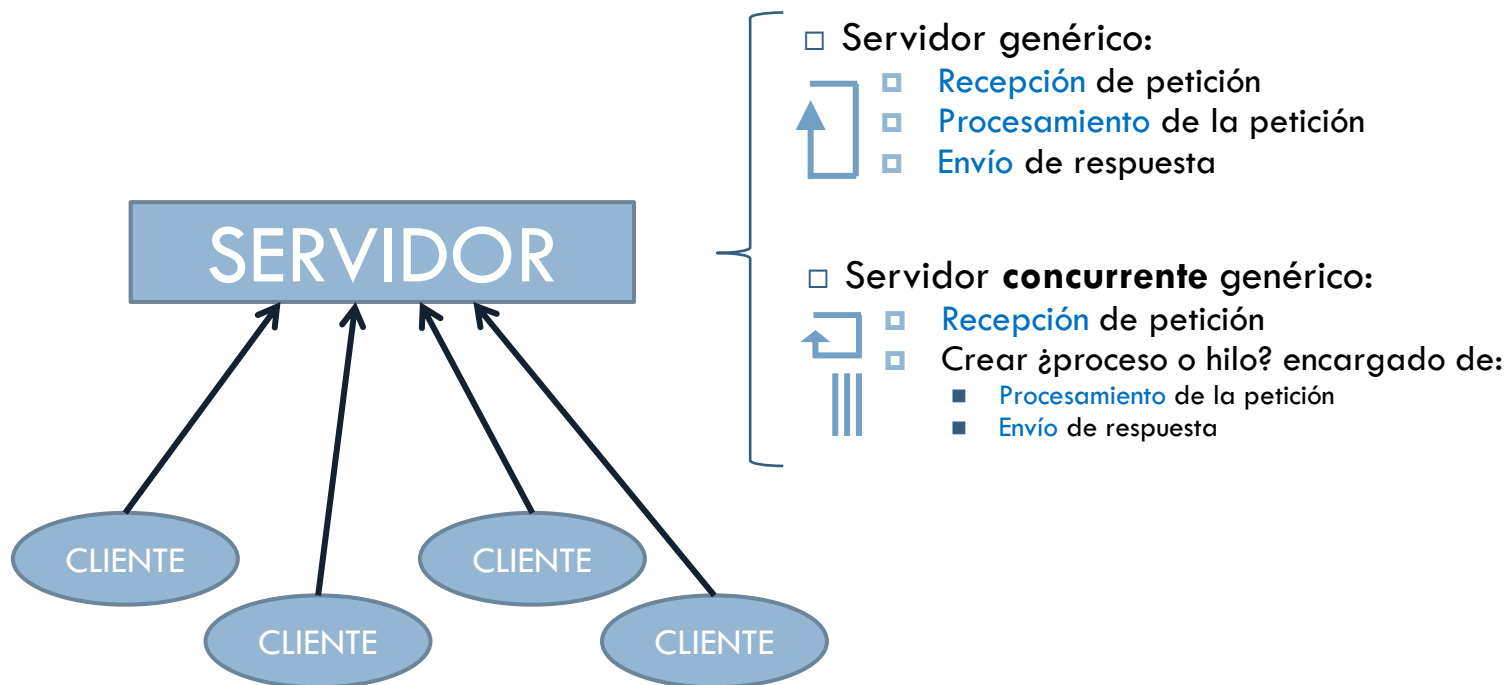
# Servidor de peticiones

- Un servidor recibe peticiones que debe procesar.
- Estructura de un servidor genérico:
  - **Recepción** de petición:
    - Cada petición requiere un cierto tiempo en operaciones de entrada/salida para ser recibida.
  - **Procesamiento** de la petición:
    - Un cierto tiempo de procesamiento en CPU.
  - **Envío** de respuesta:
    - Un cierto tiempo de entrada/salida para contestar.



# ¿Servidor de peticiones concurrente?

- Un servidor recibe peticiones que debe procesar.



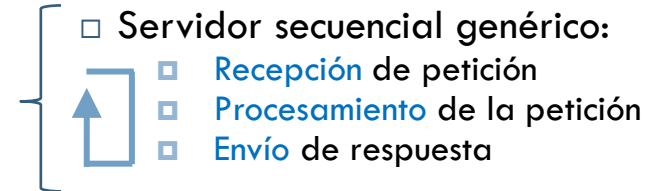
# Entorno de pruebas: aplicación de prueba

10

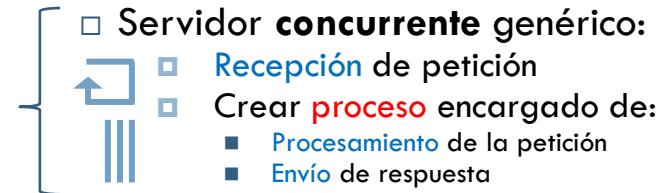
Sistemas operativos: una visión aplicada

Alejandro Calderón Mateos 

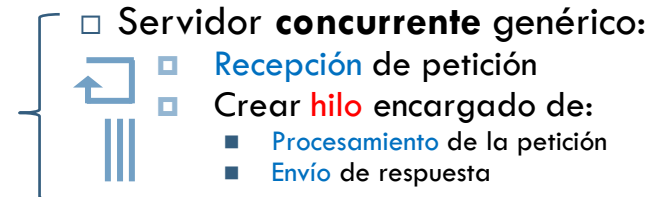
- Secuencial



- Concurrente con procesos



- Concurrente con hilos bajo demanda



- Concurrente con hilos en un *pool* de hilos (precreados)

# Entorno de pruebas: aplicación de prueba

## □ Para evaluar las soluciones se usará:

### □ Programa que:

#### ■ Atiende NPET peticiones:

- Recepción de petición.
- Envío de respuesta.

#### ■ Mide el tiempo en atender las peticiones

```
#include "peticion.h"
#define NPET 20

int main()
{
    peticion_t p;

    t1=tomar_tiempo();
    for (i=0; i<NPET; i++) {
        recibir_peticion(&p);
        responder_peticion(&p);
    }
    t2=tomar_tiempo();

    printf("tiempo: %d", t2-t1);
    return 0;
}
```

### □ Una biblioteca que simulará:

- La recepción de peticiones.
- El procesamiento y envío de respuestas.

```
#ifndef PETICION_H
#define PETICION_H

typedef struct peticion {
    /* ... */
} peticion_t;

void recibir_peticion ( peticion_t * p );
void responder_peticion ( peticion_t * p );

#endif
```

# Biblioteca base

## peticiones.h

12

Alejandro Calderón Mateos 

```
#ifndef PETICION_H
#define PETICION_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

struct petition {
    long id;
    /* Resto de campos necesarios */
    int tipo;
    /* ... */
};

typedef struct petition petition_t;

void recibir_peticion    ( petition_t * p );
void responder_peticion ( petition_t * p );

#endif
```

# Recepción de peticiones

## peticiones.c

13

Alejandro Calderón Mateos 

```
static long petid = 0;

void recibir_peticion (peticion_t * p)
{
    int delay;

    fprintf(stderr, "Recibiendo petición\n");
    p->id = petid++;

    /* Simulación de tiempo de E/S */
    delay = rand() % 5;
    sleep(delay);

    fprintf(stderr, "Petición %d recibida después de %d segundos\n",
            p->id, delay);
}
```

# Recepción de peticiones

## peticiones.c

14

Alejandro Calderón Mateos 

```
static long petid = 0;

void recibir_peticion (peticion_t * p)
{
    int delay;

    fprintf(stderr, "Recibiendo petición\n");
    p->id = petid++;

    /* Simulación de tiempo de E/S */
    delay = rand() % 5;
    sleep(delay);

    fprintf(stderr, "Petición %d recibida después de %d segundos\n",
            p->id, delay);
}
```

Aquí iría alguna llamada bloqueante para recibir la petición (por ejemplo de la red)

# Procesamiento y envío de peticiones

## peticiones.c

15

Alejandro Calderón Mateos 

```
void responder_peticion (peticion_t * p)
{
    int delay, i;
    char * mz;

    fprintf(stderr, "Enviando petición %d\n", p->id);

    /* Simulación de tiempo de procesamiento */
    mz = malloc(1000000);
    for (i=0; i<1000000; i++) { mz[i] = 0; }
    free(mz) ;

    /* Simulación de tiempo de E/S */
    delay = rand() % 20;
    sleep(delay);

    fprintf(stderr, "Petición %d enviada después de %d segundos\n",
            p->id, delay);
}
```

# Procesamiento y envío de peticiones

## peticiones.c

16

Alejandro Calderón Mateos 

```
void responder_peticion (peticion_t * p)
```

```
{
```

```
    int delay, i;
```

```
    char * mz;
```

```
    fprintf(stderr, "Enviando petición %d\n", p->id);
```

```
    /* Simulación de tiempo de procesamiento */
```

```
    mz = malloc(1000000);
```

```
    for (i=0; i<1000000; i++) { mz[i] = 0; }
```

```
    free(mz) ;
```

Aquí se procesaría  
la petición

```
    /* Simulación de tiempo de E/S */
```

```
    delay = rand() % 20;
```

```
    sleep(delay);
```

Aquí iría una llamada bloqueante  
para responder a la petición

```
    fprintf(stderr, "Petición %d enviada después de %d segundos\n",  
            p->id, delay);
```

```
}
```



# Solución inicial con medición

```
#include "peticion.h"

const int MAX_PETICIONES = 5;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;
    peticion_t p;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec * 1000 +
        (long)ts.tv_usec / 1000 ;
    for (int i=0; i<MAX_PETICIONES; i++) {
        recibir_peticion(&p);
        responder_peticion(&p);
    }
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec * 1000 +
        (long)ts.tv_usec / 1000 ;

    printf("Tiempo: %lf\n", (t2-
        t1)/1000.0);
    return 0;
}
```

```
#include "peticion.h"
```

```
const int MAX_PETICIONES = 5;
```

```
int main ( int argc, char *argv[] )
{
```

```
    struct timeval ts;
    long t1, t2;
    peticion_t p;
```

```
    gettimeofday(&ts, NULL) ;
```

```
    t1 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;
```

```
    for (int i=0; i<MAX_PETICIONES; i++) {
```

```
        recibir_peticion(&p);
        responder_peticion(&p);
    }
```

```
    gettimeofday(&ts, NULL) ;
```

```
    t2 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;
```

```
    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
```

```
}
```

# Ejecución de la solución inicial

```
$ time ./ej1
```

```
Recibiendo petición  
Petición 0 recibida después de 3 segundos  
Enviando petición 0  
Petición 0 enviada después de 6 segundos  
Recibiendo petición  
Petición 1 recibida después de 2 segundos  
Enviando petición 1  
Petición 1 enviada después de 15 segundos  
Recibiendo petición  
Petición 2 recibida después de 3 segundos  
Enviando petición 2  
Petición 2 enviada después de 15 segundos  
Recibiendo petición  
Petición 3 recibida después de 1 segundos  
Enviando petición 3  
Petición 3 enviada después de 12 segundos  
Recibiendo petición  
Petición 4 recibida después de 4 segundos  
Enviando petición 4  
Petición 4 enviada después de 1 segundos
```

```
Tiempo: 62.110000
```

```
real    1m2.053s  
user    0m0.047s  
sys     0m0.000s
```

# Problemas

- Llegada de peticiones:
  - Si dos peticiones llegan al mismo tiempo ...
  - Si una petición llega mientras otra se está procesando ...
  
- Utilización de los recursos.
  - ¿Cómo será la utilización de la CPU?

# Comparación

Normal	Procesos	Hilo x petición	<i>Pool</i> de hilos
62.11 seg.			

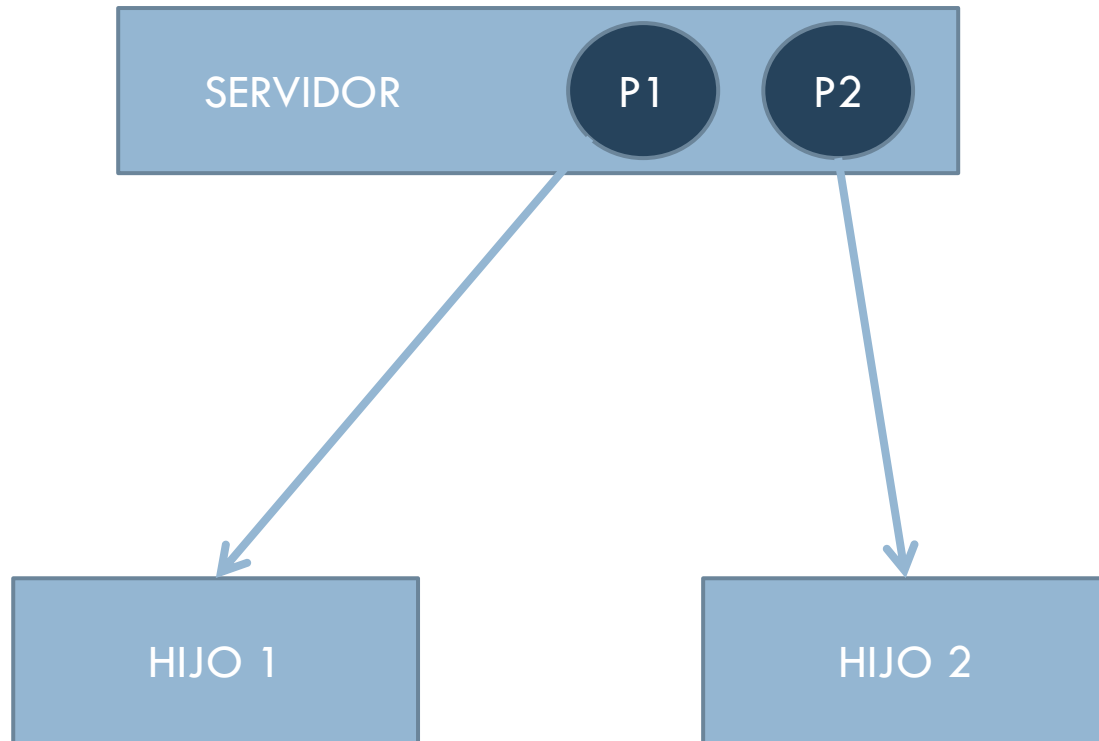
# Contenidos

- Introducción (definiciones):
  - ▣ Procesos concurrentes.
  - ▣ Concurrencia, comunicación y sincronización
  - ▣ Sección crítica y condiciones de carrera
  - ▣ Exclusión mutua y sección crítica.
- Mecanismos de sincronización (I):
  - ▣ Primitivas básicas iniciales
  - ▣ Semáforos.
- Problemas clásicos de concurrencia (I):
  - ▣ Productor-consumidor
  - ▣ Lectores-escriitores
- Mecanismos de sincronización de threads (II)
  - ▣ Semáforos
    - Llamadas al sistema para semáforos.
    - Problemas clásicos de concurrencia.
  - ▣ Mutex y variables condición
    - Llamadas al sistema para mutex.
    - Problemas clásicos de concurrencia.
- Desarrollo de servidores concurrentes
  - ▣ Servidores de peticiones.
  - ▣ **Solución basada en procesos.**
  - ▣ Solución basada en hilos bajo demanda.
  - ▣ Solución basada en pool de hilos.

# Primera idea

- Cada vez que llega una petición se crea un proceso hijo:
  - ▣ El **proceso hijo** realiza el **procesamiento** de la **petición**.
  - ▣ El **proceso padre** pasa a esperar la **siguiente petición**.

# Servidor basado en procesos



# Implementación (1 / 2)

```
#include <sys/types.h>
#include <sys/wait.h>
#include "peticion.h"

const int MAX_PETICIONES = 5;
void * receptor ( void ) ;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;
    receptor() ;
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;

    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * receptor ( void )
{
    struct peticion t p;
    int pid, hijos=0;

    for (int i=0; i<MAX_PETICIONES; i++)
    {
        recibir_peticion(&p);

        pid = fork();
        if (pid<0) { perror("Error en la creación del hij");}
        if (pid==0) { responder_peticion(&p); exit(0); } /* HIJO
        */
        if (pid!=0) { hijos++; } /* PADRE
        */
    }

    fprintf(stderr, "Esperando fin de %d hijos\n", hijos);
    while (hijos > 0)
    {
        pid = waitpid(-1, NULL, WNOHANG);
        if (pid > 0) { hijos--; }
    } ;

    return NULL ;
}
```

```
#include <sys/types.h>
#include <sys/wait.h>
#include "peticion.h"
```

```
const int MAX_PETICIONES = 5;
void * receptor ( void ) ;
```

```
int main ( int argc, char *argv[] )
{
```

```
    struct timeval ts;
    long t1, t2;
```

```
    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;
    receptor() ;
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;
```

```
    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
```

```
}
```



# Implementación (2/2)

25

Alejandro Calderón Mateos 

```
#include <sys/types.h>
#include <sys/wait.h>
#include "peticion.h"

const int MAX_PETICIONES = 5;
void * receptor ( void ) ;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;
    receptor() ;
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;

    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
}
```

```
void * receptor ( void )
{
    peticion_t p;
    int pid, hijos=0;

    for (int i=0; i<MAX_PETICIONES; i++)
    {
        recibir_petición(&p);

        pid = fork();
        if (pid<0) { perror("Error en la creación del hij"); }
        if (pid==0) { responder_petición(&p); exit(0); } /* HIJO
        */
        if (pid!=0) { hijos++; } /* PADRE
        */
    }

    fprintf(stderr, "Esperando fin de %d hijos\n", hijos);
    while (hijos > 0)
    {
        pid = waitpid(-1, NULL, WNOHANG);
        if (pid > 0) { hijos--; }
    } ;

    return NULL ;
}
```

```
void * receptor ( void )
{
    peticion_t p;
    int pid, hijos=0;

    for (int i=0; i<MAX_PETICIONES; i++)
    {
        recibir_petición(&p);

        pid = fork();
        if (pid<0) { perror("Error al crear hijo"); }
        if (pid==0) { responder_petición(&p); exit(0); }
        if (pid!=0) { hijos++; }
    }

    fprintf(stderr, "Esperando fin de %d hijos\n", hijos);
    while (hijos > 0)
    {
        pid = waitpid(-1, NULL, WNOHANG);
        if (pid > 0) { hijos--; }
    } ;

    return NULL ;
}
```

# Ejecución

```
$ time ./ej2
```

```
Recibiendo petición  
Petición 0 recibida después de 3 segundos  
Recibiendo petición  
Enviando petición 0  
Petición 1 recibida después de 1 segundos  
Recibiendo petición  
Enviando petición 1  
Petición 2 recibida después de 2 segundos  
Recibiendo petición  
Enviando petición 2  
Petición 3 recibida después de 0 segundos  
Recibiendo petición  
Enviando petición 3  
Petición 4 recibida después de 3 segundos  
Esperando fin de 5 hijos  
Petición 0 enviada después de 6 segundos  
Enviando petición 4  
Petición 3 enviada después de 13 segundos  
Petición 1 enviada después de 17 segundos  
Petición 2 enviada después de 15 segundos  
Petición 4 enviada después de 15 segundos
```

```
Tiempo: 24.086000
```

```
real    0m24.012s  
user    0m9.569s  
sys     0m5.459s
```

# Comparación

Normal	Procesos	Hilo x petición	<i>Pool</i> de hilos
62.11 seg.	24.08 seg.		

# Problemas

- ❑ Hace falta arrancar un proceso (fork) por cada petición que llega.
- ❑ Hace falta terminar un proceso (exit) por cada petición que termina.
- ❑ Excesivo consumo de recursos del sistema.
- ❑ No hay control de admisión.
  - ❑ Problemas de calidad de servicio.

# Soluciones con hilos

- Hilos bajo demanda.
  - ▣ Cada vez que se recibe una petición se crea un hilo.
  
- *Pool* de hilos.
  - ▣ Se tiene un número fijo de hilos creados.
  - ▣ Cada vez que se recibe una petición se busca un hilo libre ya creado para que atienda la petición.
    - Comunicación mediante una cola de peticiones.

# Contenidos

- Introducción (definiciones):
  - ▣ Procesos concurrentes.
  - ▣ Concurrencia, comunicación y sincronización
  - ▣ Sección crítica y condiciones de carrera
  - ▣ Exclusión mutua y sección crítica.
- Mecanismos de sincronización (I):
  - ▣ Primitivas básicas iniciales
  - ▣ Semáforos.
- Problemas clásicos de concurrencia (I):
  - ▣ Productor-consumidor
  - ▣ Lectores-escritores
- Mecanismos de sincronización de threads (II)
  - ▣ Semáforos
    - Llamadas al sistema para semáforos.
    - Problemas clásicos de concurrencia.
  - ▣ Mutex y variables condición
    - Llamadas al sistema para mutex.
    - Problemas clásicos de concurrencia.

- **Desarrollo de servidores concurrentes**
  - ▣ Servidores de peticiones.
  - ▣ Solución basada en procesos.
  - ▣ **Solución basada en hilos bajo demanda.**
  - ▣ Solución basada en pool de hilos.

# Hilos bajo demanda

- Se tiene un hilo receptor encargado de recibir las peticiones.
- Cada vez que llega una petición se crea un hilo y se le pasa **una copia** la petición al hilo recién creado.
  - Tiene que ser una copia de la petición porque la petición original se podría modificar.

# Implementación (1 / 3 main)

32

Alejandro Calderón Mateos 

```
#include <pthread.h>
#include <semaphore.h>
#include "peticion.h"

sem_t snhijos;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
    sem_init(&snhijos, 0, 0);
    pthread_create(&thr, NULL, receptor, NULL);
    pthread_join(thr, NULL);
    sem_destroy(&snhijos);
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;

    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * servicio (void * p)
{
    peticion_t pet;

    copia_peticion(&pet, (peticion_t*)p);
    fprintf(stderr, "Iniciando servicio\n");
    responder_peticion(&pet);
    sem_post(&snhijos);
    fprintf(stderr, "Terminando servicio\n");
    pthread_exit(0); return NULL;
}

void * receptor (void * param)
{
    const int MAX_PETICIONES = 5; int nservicio = 0; int i;
    peticion_t p; pthread_t th_hijo;

    for (i=0;i<MAX_PETICIONES;i++) {
        recibir_peticion(&p); nservicio++;
        pthread_create(&th_hijo, NULL, servicio, &p);
    }

    for (i=0;i<nservicio;i++) {
        fprintf(stderr, "Haciendo wait\n");
        sem_wait(&snhijos);
        fprintf(stderr, "Saliendo de wait\n");
    }

    pthread_exit(0); return NULL;
}
```

```
#include <pthread.h>
#include <semaphore.h>
#include "peticion.h"
```

**sem\_t snhijos;**

```
int main ( int argc, char *argv[] )
{
```

```
    struct timeval ts;
    long t1, t2;
pthread_t thr;
```

```
    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
sem_init(&snhijos, 0, 0);
pthread_create(&thr, NULL, receptor, NULL);
pthread_join(thr, NULL);
sem_destroy(&snhijos);
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
```

```
    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
}
```



# Implementación (2/3 receptor)

33

Alejandro Calderón Mateos 

```
#include <pthread.h>
#include <semaphore.h>
#include "peticion.h"

sem_t snhijos;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
    sem_init(&snhijos, 0, 0);
    pthread_create(&thr, NULL, receptor, NULL);
    pthread_join(thr, NULL);
    sem_destroy(&snhijos);
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * servicio (void * p)
{
    peticion_t pet;
    copia_peticion(&pet, (peticion_t*)p);
    fprintf(stderr, "Iniciando servicio\n");
    responder_peticion(&pet);
    sem_post(&snhijos);
    fprintf(stderr, "Terminando servicio\n");
    pthread_exit(0); return NULL;
}

void * receptor (void * param)
{
    const int MAX_PETICIONES = 5; int nservicio = 0; int i;
    peticion_t p; pthread_t th_hijo;

    for (i=0; i<MAX_PETICIONES; i++) {
        recibir_peticion(&p); nservicio++;
        pthread_create(&th_hijo, NULL, servicio, &p);
    }

    for (i=0; i<nservicio; i++) {
        fprintf(stderr, "Haciendo wait\n");
        sem_wait(&snhijos);
        fprintf(stderr, "Saliendo de wait\n");
    }

    pthread_exit(0); return NULL;
}
```

```
const int MAX_PETICIONES = 5;
```

```
void * receptor ( void * param )
{
```

```
    int nservicio = 0;
    peticion_t p;
    pthread_t th_hijo;
```

```
    for (int i=0; i<MAX_PETICIONES; i++) {
        recibir_peticion(&p);
        nservicio++;
        pthread_create(&th_hijo, NULL, servicio, &p);
    }
```

```
    for (int i=0; i<nservicio; i++) {
        fprintf(stderr, "Haciendo wait\n");
        sem_wait(&snhijos);
        fprintf(stderr, "Saliendo de wait\n");
    }
```

```
    pthread_exit(0);
    return NULL;
```

```
}
```

# Implementación (3/3 servicio)

34

Alejandro Calderón Mateos 

```
#include <pthread.h>
#include <semaphore.h>
#include "peticion.h"

sem_t snhijos;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
    sem_init(&snhijos, 0, 0);
    pthread_create(&thr, NULL, receptor, NULL);
    pthread_join(thr, NULL);
    sem_destroy(&snhijos);
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;

    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
}
```

```
void * servicio (void * p)
{
    peticion_t pet;

    copia_peticion(&pet, (peticion_t *)p);
    fprintf(stderr, "Iniciando servicio\n");
    responder_peticion(&pet);
    sem_post(&snhijos);
    fprintf(stderr, "Terminando servicio\n");
    pthread_exit(0); return NULL;
}
```

```
void * receptor (void * param)
{
    const int MAX_PETICIONES = 5; int nservicio = 0; int i;
    peticion_t p; pthread_t th_hijo;

    for (i=0; i<MAX_PETICIONES; i++) {
        recibir_peticion(&p); nservicio++;
        pthread_create(&th_hijo, NULL, servicio, &p);
    }

    for (i=0; i<nservicio; i++) {
        fprintf(stderr, "Haciendo wait\n");
        sem_wait(&snhijos);
        fprintf(stderr, "Saliendo de wait\n");
    }

    pthread_exit(0); return NULL;
}
```

```
void * servicio ( void * p )
{
    peticion_t pet;

    memmove(&pet, (peticion_t *)p, sizeof(pet));
    fprintf(stderr, "Iniciando servicio\n");
    responder_peticion(&pet);
    sem_post(&snhijos);

    fprintf(stderr, "Terminando servicio\n");
    pthread_exit(0);
    return NULL;
}
```

## □ ¿Puede darse una condición de carrera?

```
void * receptor ( void * param )
{
    const int MAX_PETICIONES = 5;
    int nservicio = 0;
    petition_t p;
    pthread_t th_hijo;

    for (int i=0; i<MAX_PETICIONES; i++) {
        recibir_peticion(&p);
        nservicio++;
        pthread_create(&th_hijo, NULL, servicio, &p);
    }

    for (int i=0; i<nservicio; i++) {
        fprintf(stderr, "Haciendo wait\n");
        sem_wait(&snhijos);
        fprintf(stderr, "Saliendo de wait\n");
    }

    pthread_exit(0);
    return NULL;
}
```

```
void * servicio ( void * p )
{
    petition_t pet;

    memmove(&pet, (petition_t *)p, sizeof(pet));
    fprintf(stderr, "Iniciando servicio\n");
    responder_peticion(&pet);
    sem_post(&snhijos);

    fprintf(stderr, "Terminando servicio\n");
    pthread_exit(0);
    return NULL;
}
```

# Ejecución

```
$ time ./ej3
Recibiendo petición
Petición 0 recibida después de 3 segundos
Recibiendo petición
Iniciando servicio
Enviando petición 0
Petición 1 recibida después de 1 segundos
Recibiendo petición
Iniciando servicio
Enviando petición 1
Petición 2 recibida después de 0 segundos
Recibiendo petición
Iniciando servicio
Enviando petición 3
Petición 3 recibida después de 3 segundos
Recibiendo petición
Iniciando servicio
Enviando petición 4
Petición 4 recibida después de 2 segundos
Haciendo wait
...
Saliendo de wait
Haciendo wait
Petición 1 enviada después de 15 segundos
Terminando servicio
Saliendo de wait
Haciendo wait
Petición 0 enviada después de 17 segundos
Terminando servicio
Saliendo de wait
Tiempo: 20.012000

real    0m20.012s
user    0m0.033s
sys     0m0.000s
```

# Comparación

Normal	Procesos	Hilo x petición	<i>Pool</i> de hilos
62.11 seg.	24.08 seg.	20.01 seg.	

# Problema

- La creación y terminación de hilos tiene un coste menor que la de procesos, pero sigue siendo un coste.
- No hay control de admisión:
  - ¿Que pasa si llegan muchas peticiones o las peticiones recibidas no terminan?

# Reflexión

- ¿Puede darse una condición de carrera?



# Reflexión

40

Alejandro Calderón Mateos 

□ ¿Puede darse una condición de carrera?

```
void * receptor (void * param)
```

```
peticion_t p; 
```

```
recibir_peticion(&p);  
nservicio++;  
pthread_create(&hijo, NULL, servicio, &p);
```

```
recibir_peticion(&p);  
nservicio++;  
pthread_create(&hijo, NULL, servicio, &p);
```

```
...
```





# Reflexión

41

Alejandro Calderón Mateos 

□ ¿Puede darse una condición de carrera?

```
void * receptor (void * param)
```

```
peticion_t p;    
recibir_petición(&p);  
nservicio++;  
pthread_create(&hijo, NULL, servicio, &p);  
  
recibir_petición(&p);  
nservicio++;  
pthread_create(&hijo, NULL, servicio, &p);  
...
```

# Reflexión

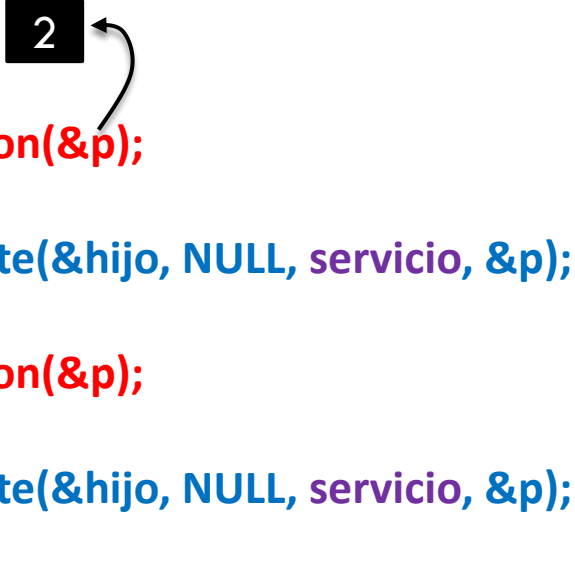
42

Alejandro Calderón Mateos 

□ ¿Puede darse una condición de carrera?

```
void * receptor (void * param)
```

```
peticion_t p; 2  
recibir_petición(&p);  
nservicio++;  
pthread_create(&hijo, NULL, servicio, &p);  
  
recibir_petición(&p);  
nservicio++;  
pthread_create(&hijo, NULL, servicio, &p);  
...
```



# Reflexión

43

Alejandro Calderón Mateos 

□ ¿Puede darse una condición de carrera?

```
void * receptor (void * param)
```

```
peticion_t p; 2
```

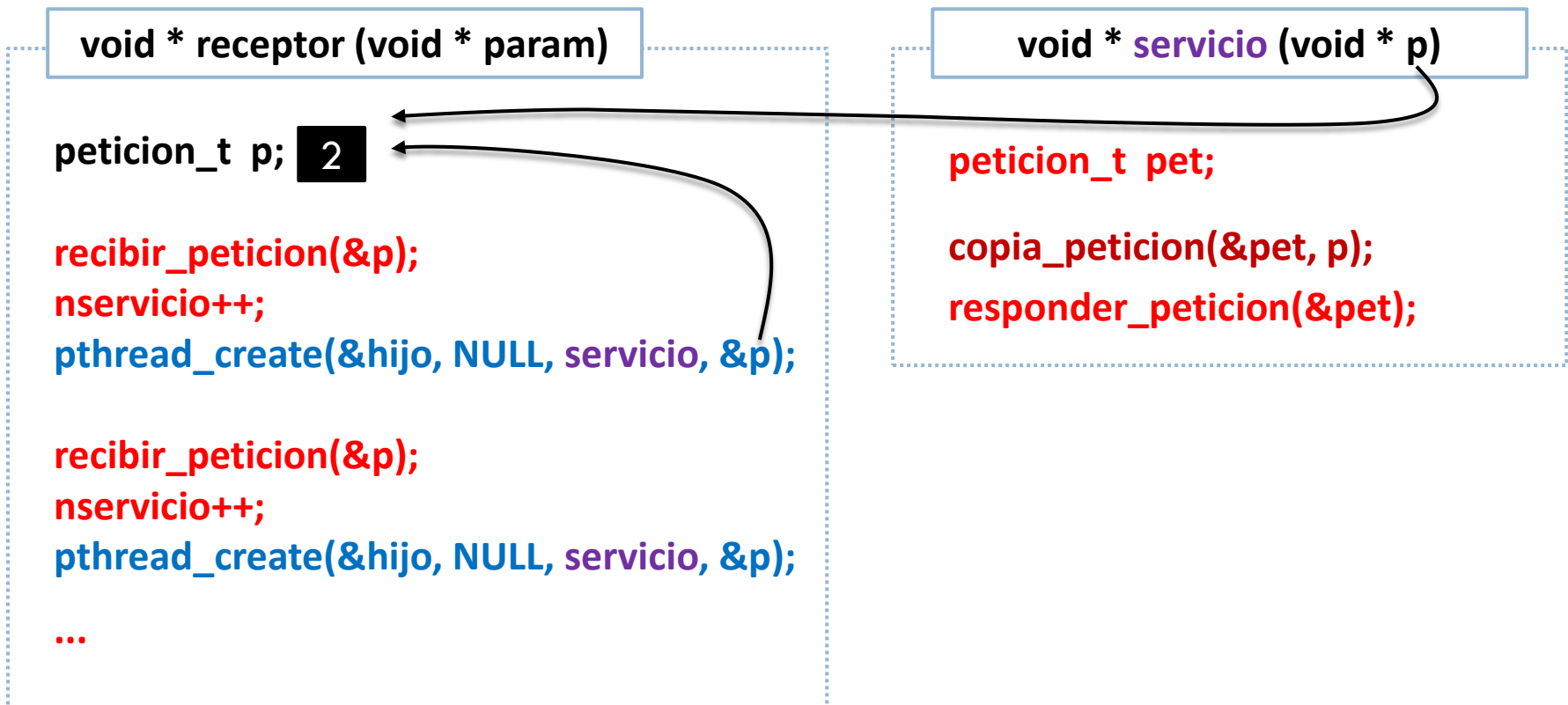
```
recibir_peticion(&p);  
nservicio++;  
pthread_create(&hijo, NULL, servicio, &p);
```

```
recibir_peticion(&p);  
nservicio++;  
pthread_create(&hijo, NULL, servicio, &p);
```

```
...
```

# Reflexión

□ ¿Puede darse una condición de carrera?

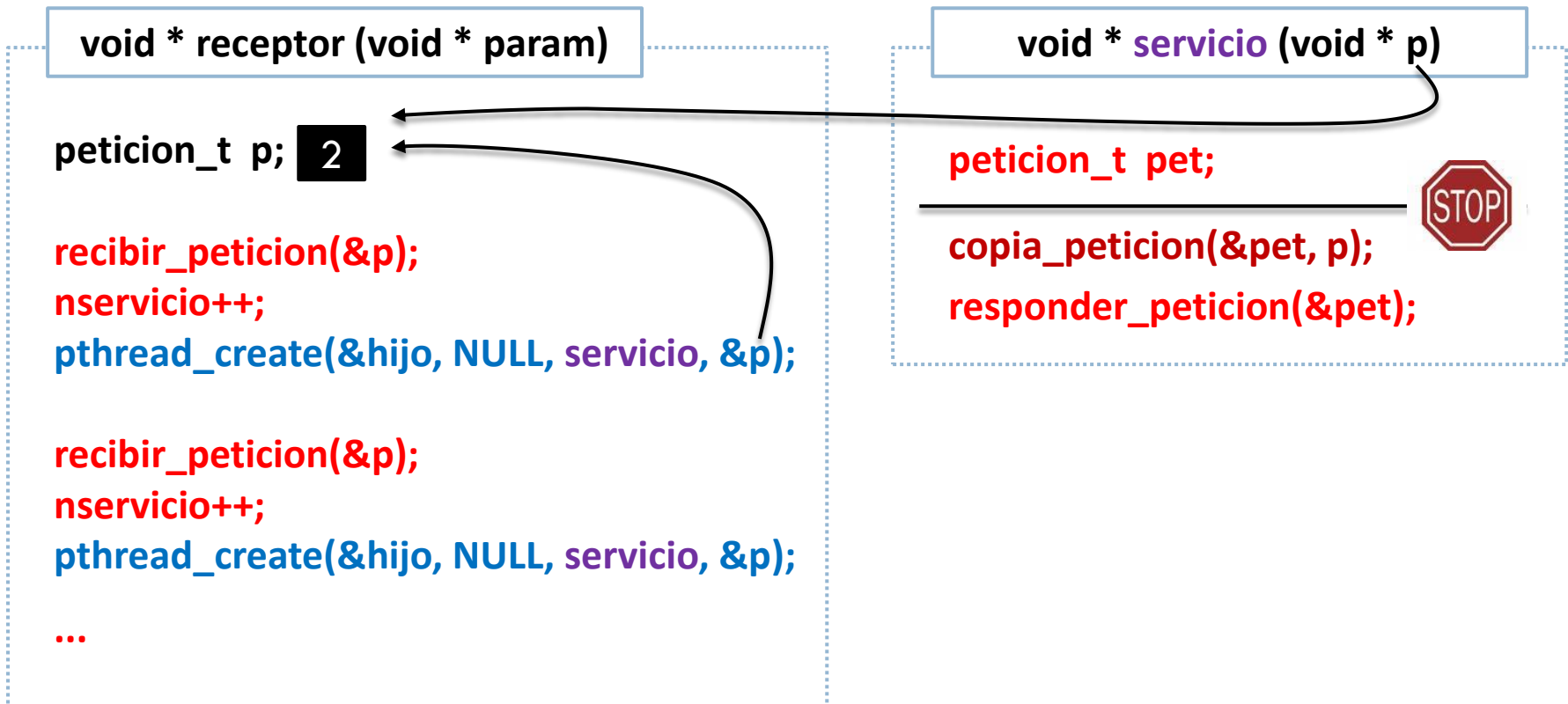


# Reflexión

45

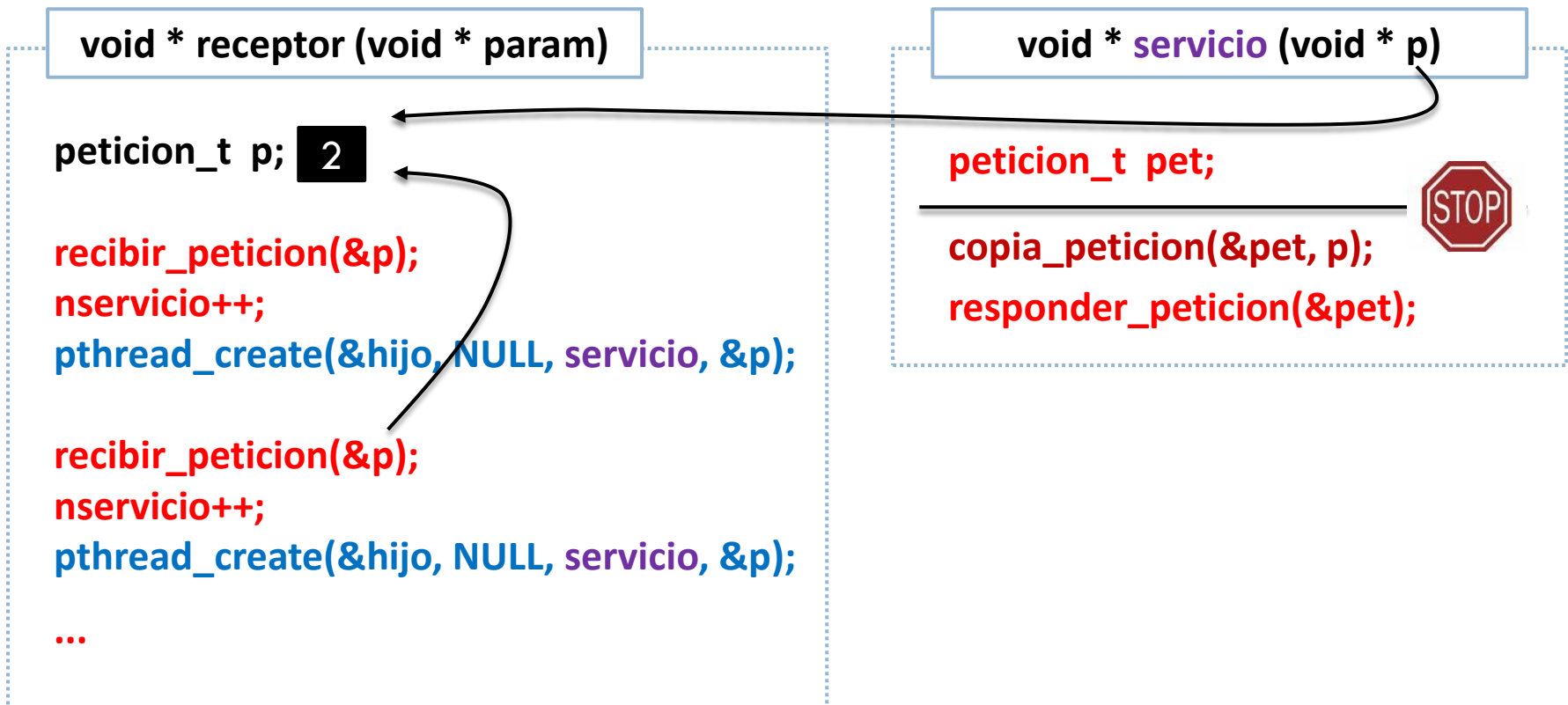
Alejandro Calderón Mateos 

□ ¿Puede darse una condición de carrera?



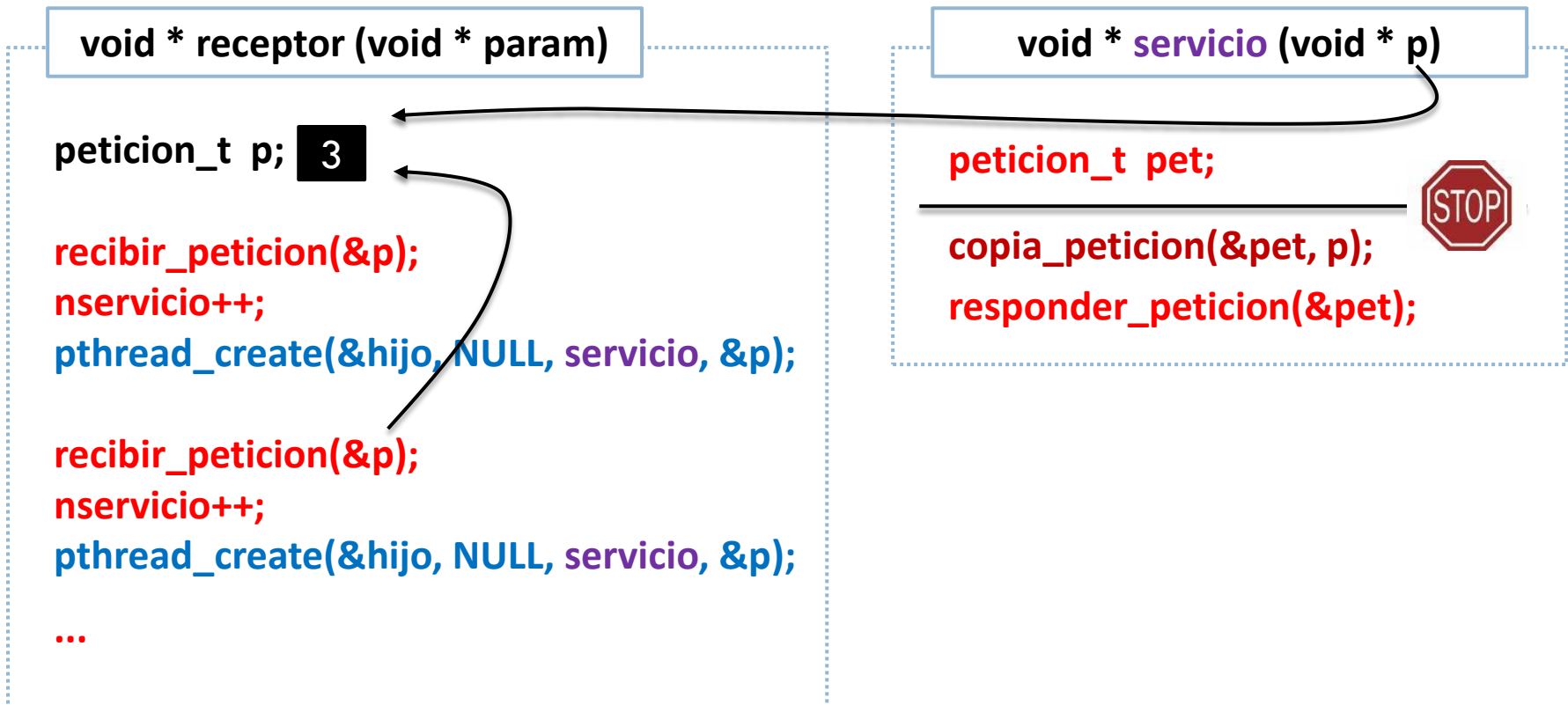
# Reflexión

□ ¿Puede darse una condición de carrera?



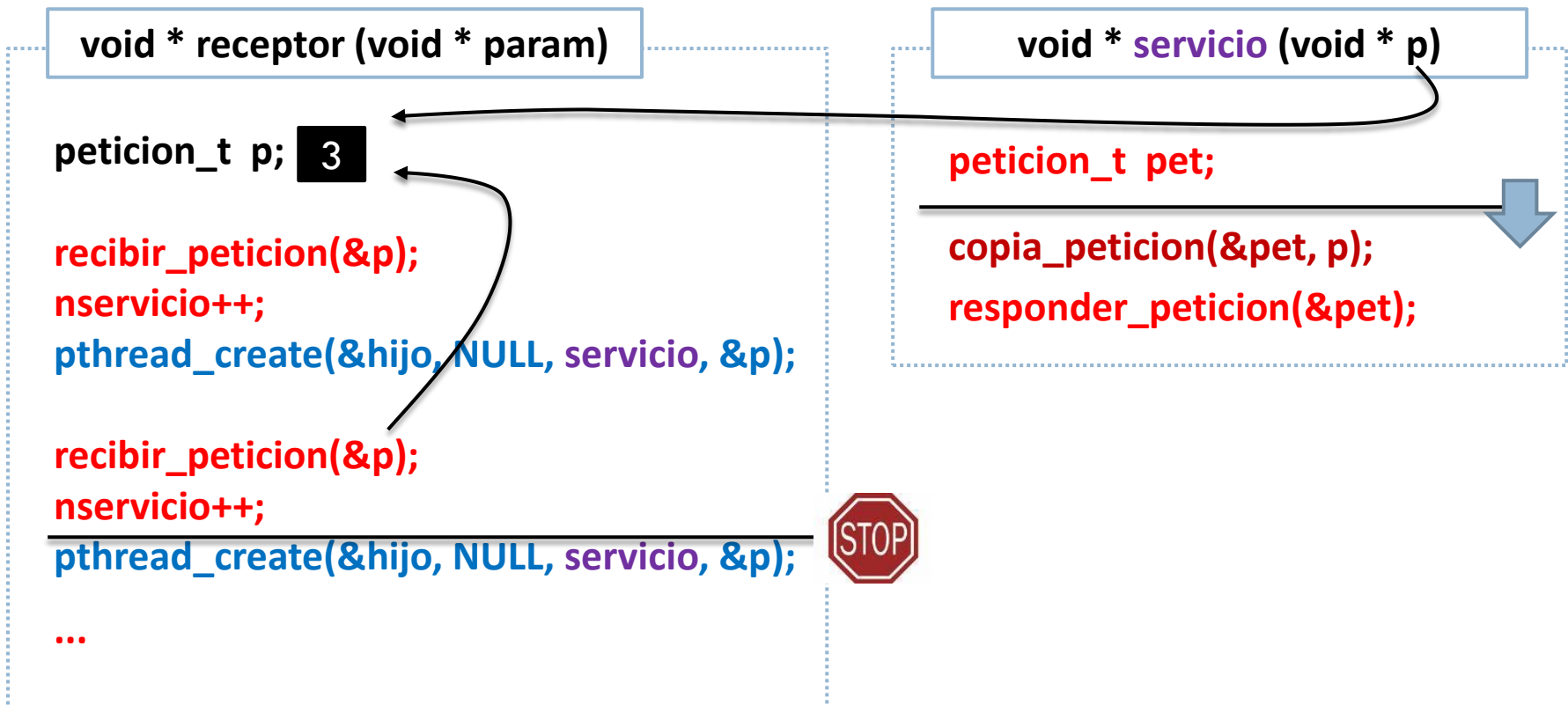
# Reflexión

□ ¿Puede darse una condición de carrera?



# Reflexión

□ ¿Puede darse una condición de carrera?





# Posible solución (mutex + condition)

## □ Proceso ligero padre

```
lock(mutex); /* acceso al recurso */  
while (petición no copiada)  
    wait(condition, mutex);  
marcar el recurso como ocupado;  
unlock(mutex);
```

## □ Proceso ligero hijo

```
lock(mutex);  
copiar la petición  
marcar la petición como copiada;  
signal(condition);  
unlock(mutex);
```

# Contenidos

- Introducción (definiciones):
  - Procesos concurrentes.
  - Concurrencia, comunicación y sincronización
  - Sección crítica y condiciones de carrera
  - Exclusión mutua y sección crítica.
- Mecanismos de sincronización (I):
  - Primitivas básicas iniciales
  - Semáforos.
- Problemas clásicos de concurrencia (I):
  - Productor-consumidor
  - Lectores-escriores
- Mecanismos de sincronización de threads (II)
  - Semáforos
    - Llamadas al sistema para semáforos.
    - Problemas clásicos de concurrencia.
  - Mutex y variables condición
    - Llamadas al sistema para mutex.
    - Problemas clásicos de concurrencia.
- **Desarrollo de servidores concurrentes**
  - **Servidores de peticiones.**
  - **Solución basada en procesos.**
  - **Solución basada en hilos bajo demanda.**
  - **Solución basada en pool de hilos.**

# Pool de threads

- Un *pool* de hilos es un conjunto de hilos que se tiene creados desde el principio para ejecutar un servicio:
  - ▣ Cada vez que llega una petición se pone en una cola de peticiones pendientes.
  - ▣ Todos los hilos esperan a que haya alguna petición en la cola y la retiran para procesarla.

# Implementación: main (1 / 3)

52

Alejandro Calderón Mateos



```
#include "peticion.h"
#include <pthread.h>
#include <semaphore.h>

#define MAX_BUFFER 128
peticion_t buffer[MAX_BUFFER];

int n_elementos = 0;
int pos_servicio = 0;
int fin = 0;
pthread_mutex_t mutex;
pthread_cond_t no_lleno;
pthread_cond_t no_vacio;

void * receptor (void * param) ;
void * servicio (void * param) ;

int main()
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
    pthread_t ths[MAX_SERVICIO];
    const int MAX_SERVICIO = 5;

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

    for (int i=0; i<MAX_SERVICIO; i++) {
        pthread_create(&ths[i], NULL, servicio, NULL);
    }
    sleep(1);

    gettimeofday(&ts, NULL);
    t1 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;
    pthread_create(&thr, NULL, receptor, NULL);
    pthread_join(thr, NULL);
    for (int i=0; i<MAX_SERVICIO; i++) {
        pthread_join(ths[i], NULL);
    }
    gettimeofday(&ts, NULL);
    t2 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&no_lleno);
    pthread_cond_destroy(&no_vacio);

    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * receptor (void * param)
{
    const int MAX_PETICIONES = 5;
    peticion_t p;
    int pos=0;

    for (int i=0; i<MAX_PETICIONES; i++)
    {
        receive_peticion(&p);
        pthread_mutex_lock(&mutex);
        while (n_elementos == MAX_BUFFER)
            pthread_cond_wait(&no_lleno, &mutex);
        buffer[pos] = p;
        pos = (pos+1) % MAX_BUFFER;
        n_elementos++;
        pthread_cond_signal(&no_vacio);
        pthread_mutex_unlock(&mutex);
    }
    fprintf(stderr, "Finalizando receptor\n");
    pthread_mutex_lock(&mutex);
    time_t t;
    pthread_cond_broadcast(&no_vacio);
    pthread_mutex_unlock(&mutex);
    fprintf(stderr, "Finalizado receptor\n");
    pthread_exit(0);
    return NULL;
}

void * servicio (void * param)
{
    peticion_t p;

    for (;;)
    {
        pthread_mutex_lock(&mutex);
        while (n_elementos == 0)
        {
            if (fin==1) {
                fprintf(stderr, "Finalizando servicio\n");
                pthread_mutex_unlock(&mutex);
                pthread_exit(0);
            }
            pthread_cond_wait(&no_vacio, &mutex);
        }
        // while
        printf(stderr, "Sirviendo pos. %d\n", pos_servicio);
        p = buffer[pos_servicio];
        pos_servicio = (pos_servicio + 1) % MAX_BUFFER;
        n_elementos--;
        pthread_cond_signal(&no_lleno);
        pthread_mutex_unlock(&mutex);
        responde_peticion(&p);
    }
    pthread_exit(0);
    return NULL;
}
```

```
#include "peticion.h"
#include <pthread.h>
#include <semaphore.h>
```

```
#define MAX_BUFFER 128
peticion_t buffer[MAX_BUFFER];
```

```
int n_elementos = 0;
int pos_servicio = 0;
int fin=0;
```

```
pthread_mutex_t mutex;
pthread_cond_t no_lleno;
pthread_cond_t no_vacio;
```

```
void * receptor (void * param) ;
void * servicio (void * param) ;
```

# Implementación: main (2/3)

53

Alejandro Calderón Mateos



```
#include "petición.h"
#include <pthread.h>
#include <semaphore.h>

#define MAX_BUFFER 128
petición_t *buffer[MAX_BUFFER];

int n_elementos = 0;
int pos_servicio = 0;
int fin = 0;
pthread_mutex_t mutex;
pthread_cond_t no_lleno;
pthread_cond_t no_vacio;

void *receptor(void *param);
void *servicio(void *param);

int main()
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
    pthread_t ths[MAX_SERVICIO];
    const int MAX_SERVICIO = 5;

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

    for (int i=0; i<MAX_SERVICIO; i++) {
        pthread_create(&ths[i], NULL, servicio, NULL);
    }
    sleep(1);

    gettimeofday(&ts, NULL);
    t1 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;
    pthread_create(&thr, NULL, receptor, NULL);
    pthread_join(thr, NULL);
    for (int i=0; i<MAX_SERVICIO; i++) {
        pthread_join(ths[i], NULL);
    }
    gettimeofday(&ts, NULL);
    t2 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&no_lleno);
    pthread_cond_destroy(&no_vacio);
    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void *receptor(void *param)
{
    const int MAX_PETICIONES = 5;
    petición_t p;
    int pos=0;
    for (int i=0; i<MAX_PETICIONES; i++)
    {
        receive_petition(&p);
        pthread_mutex_lock(&mutex);
        while (n_elementos == MAX_BUFFER)
            pthread_cond_wait(&no_lleno, &mutex);
        buffer[pos] = p;
        pos = (pos+1) % MAX_BUFFER;
        n_elementos++;
        pthread_cond_signal(&no_vacio);
        pthread_mutex_unlock(&mutex);
    }
    fprintf(stderr, "Finalizando receptor\n");
    pthread_mutex_lock(&mutex);
    fin = 1;
    pthread_cond_broadcast(&no_vacio);
    pthread_mutex_unlock(&mutex);
    fprintf(stderr, "Finalizado receptor\n");
    pthread_exit(0);
    return NULL;
}

void *servicio(void *param)
{
    petición_t p;
    for (;;)
    {
        pthread_mutex_lock(&mutex);
        while (n_elementos == 0)
        {
            if (fin==1) {
                fprintf(stderr, "Finalizando servicio\n");
                pthread_mutex_unlock(&mutex);
                pthread_exit(0);
            }
            pthread_cond_wait(&no_vacio, &mutex);
        }
        while
        printf(stderr, "Sirviendo pos. %d\n", pos_servicio);
        p = buffer[pos_servicio];
        pos_servicio = (pos_servicio + 1) % MAX_BUFFER;
        n_elementos--;
        pthread_cond_signal(&no_lleno);
        pthread_mutex_unlock(&mutex);
        respond_peticion(&p);
    }
    pthread_exit(0);
    return NULL;
}
```

```
int main()
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
    pthread_t ths[MAX_SERVICIO];
    const int MAX_SERVICIO = 5;

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

    for (int i=0; i<MAX_SERVICIO; i++) {
        pthread_create(&ths[i], NULL, servicio, NULL);
    }
    sleep(1);
```

# Implementación: main (3/3)

54

Alejandro Calderón Mateos



```
#include "petición.h"
#include <pthread.h>
#include <semaphore.h>

#define MAX_BUFFER 128
petición_t *buffer[MAX_BUFFER];

int n_elementos = 0;
int pos_servicio = 0;
int fin = 0;
pthread_mutex_t mutex;
pthread_cond_t no_lleno;
pthread_cond_t no_vacio;

void *receptor (void * param) ;
void *servicio (void * param) ;

int main()
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
    pthread_t ths[MAX_SERVICIO];
    const int MAX_SERVICIO = 5;

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

    for (int i=0; i<MAX_SERVICIO; i++) {
        pthread_create(&ths[i], NULL, servicio, NULL);
    }
    sleep(1);

    gettimeofday(&ts, NULL);
    t1 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000;
    pthread_create(&thr, NULL, receptor, NULL);
    pthread_join(thr, NULL);
    for (int i=0; i<MAX_SERVICIO; i++) {
        pthread_join(ths[i], NULL);
    }
    gettimeofday(&ts, NULL);
    t2 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000;

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&no_lleno);
    pthread_cond_destroy(&no_vacio);
    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void *receptor (void * param)
{
    const int MAX_PETICIONES = 5;
    petición_t p; int pos=0;
    for (int i=0; i<MAX_PETICIONES; i++)
    {
        recibir_petición(&p);
        pthread_mutex_lock(&mutex);
        while (n_elementos == MAX_BUFFER)
            pthread_cond_wait(&no_lleno, &mutex);
        buffer[pos] = p; MAX_BUFFER;
        pos = (pos+1) % MAX_BUFFER;
        n_elementos++;
        pthread_cond_signal(&no_vacio);
        pthread_mutex_unlock(&mutex);
    }
    fprintf(stderr, "Finalizando receptor\n");
    pthread_mutex_lock(&mutex);
    time_t;
    pthread_cond_broadcast(&no_vacio);
    pthread_mutex_unlock(&mutex);
    fprintf(stderr, "Finalizado receptor\n");
    pthread_exit(0);
    return NULL;
}

void *servicio (void * param)
{
    petición_t p;
    for (;;)
    {
        pthread_mutex_lock(&mutex);
        while (n_elementos == 0)
            if (fin==1) {
                fprintf(stderr, "Finalizando servicio\n");
                pthread_mutex_unlock(&mutex);
                pthread_exit(0);
            }
        pthread_cond_wait(&no_vacio, &mutex);
        // while
        printf(stderr, "Sirviendo pos. %d\n", pos_servicio);
        p = buffer[pos_servicio];
        pos_servicio = (pos_servicio + 1) % MAX_BUFFER;
        n_elementos--;
        pthread_cond_signal(&no_lleno);
        pthread_mutex_unlock(&mutex);
        responder_petición(&p);
    }
    pthread_exit(0);
    return NULL;
}
```

```
gettimeofday(&ts, NULL) ;
t1 =(long)ts.tv_sec*1000+(long)ts.tv_usec/1000;

pthread_create(&thr, NULL, receptor, NULL);
pthread_join(thr, NULL);
for (int i=0; i<MAX_SERVICIO; i++) {
    pthread_join(ths[i], NULL);
}

gettimeofday(&ts, NULL) ;
t2 =(long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;

pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&no_lleno);
pthread_cond_destroy(&no_vacio);

printf("Tiempo: %lf\n", (t2-t1)/1000.0);
return 0;
}
```

# Implementación: receptor

55

Alejandro Calderón Mateos



```
#include "peticion.h"
#include <pthread.h>
#include <semaphore.h>

#define MAX_BUFFER 128
peticion_t buffer[MAX_BUFFER];

int n_elementos = 0;
int pos_servicio = 0;
int fin = 0;
pthread_mutex_t mutex;
pthread_cond_t no_lleno;
pthread_cond_t no_vacio;

void * receptor (void * param);
void * servicio (void * param);

int main()
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
    pthread_t thr_servicio;
    const int MAX_SERVICIO = 5;

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

    for (int i=0; i<MAX_SERVICIO; i++) {
        pthread_create(&thr[i], NULL, servicio, NULL);
    }
    sleep(1);

    gettimeofday(&ts, NULL);
    t1 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;
    pthread_create(&thr, NULL, receptor, NULL);
    pthread_join(thr, NULL);
    for (int i=0; i<MAX_SERVICIO; i++) {
        pthread_join(thr[i], NULL);
    }
    gettimeofday(&ts, NULL);
    t2 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&no_lleno);
    pthread_cond_destroy(&no_vacio);
    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * receptor (void * param)
{
    const int MAX_PETICIONES = 5;
    peticion_t p; int pos=0;
    for (int i=0; i<MAX_PETICIONES; i++)
    {
        recibir_peticon(&p);
        pthread_mutex_lock(&mutex);
        while (n_elementos == MAX_BUFFER)
            pthread_cond_wait(&no_lleno, &mutex);
        buffer[pos] = p;
        pos = (pos+1) % MAX_BUFFER;
        n_elementos++;
        pthread_cond_signal(&no_vacio);
        pthread_mutex_unlock(&mutex);
    }
    fprintf(stderr, "Finalizando receptor\n");
    pthread_mutex_lock(&mutex);
    sleep(1);
    pthread_cond_broadcast(&no_vacio);
    pthread_mutex_unlock(&mutex);
    fprintf(stderr, "Finalizado receptor\n");
    pthread_exit(0);
    return NULL;
}

void * servicio (void * param)
{
    peticion_t p;
    for (i=0; i<MAX_SERVICIO; i++)
    {
        pthread_mutex_lock(&mutex);
        while (n_elementos == 0)
        {
            if (fin==1)
            {
                fprintf(stderr, "Finalizando servicio\n");
                pthread_mutex_unlock(&mutex);
                pthread_exit(0);
            }
            pthread_cond_wait(&no_vacio, &mutex);
        }
        // while
        printf("Sirviendo pos. %d\n", pos_servicio);
        p = buffer[pos_servicio];
        pos_servicio = (pos_servicio + 1) % MAX_BUFFER;
        n_elementos--;
        pthread_mutex_unlock(&mutex);
        pthread_mutex_lock(&mutex);
        responder_peticon(&p);
    }
    pthread_exit(0);
    return NULL;
}
```

```
void * receptor (void * param)
{
    const int MAX_PETICIONES = 5;
    peticion_t p; int pos=0;

    for (int i=0; i<MAX_PETICIONES; i++)
    {
        recibir_peticon(&p);
        pthread_mutex_lock(&mutex);
        while (n_elementos == MAX_BUFFER)
            pthread_cond_wait(&no_lleno, &mutex);
        buffer[pos] = p;
        pos = (pos+1) % MAX_BUFFER;
        n_elementos++;
        pthread_cond_signal(&no_vacio);
        pthread_mutex_unlock(&mutex);
    }
    fprintf(stderr, "Finalizando receptor\n");
    pthread_mutex_lock(&mutex);
    fin=1;
    pthread_cond_broadcast(&no_vacio);
    pthread_mutex_unlock(&mutex);
    fprintf(stderr, "Finalizado receptor\n");
    pthread_exit(0);
    return NULL;
}
```

# Implementación: servicio

56

Alejandro Calderón Mateos 

```
#include "peticion.h"
#include <pthread.h>
#include <semaphore.h>

#define MAX_BUFFER 128
peticion_t buffer[MAX_BUFFER];

int n_elementos = 0;
int pos_servicio = 0;
int fin = 0;
pthread_mutex_t mutex;
pthread_cond_t no_lleno;
pthread_cond_t no_vacio;

void * receptor (void * param) ;
void * servicio (void * param) ;

int main()
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
    pthread_t thr1[MAX_SERVICIO];
    const int MAX_SERVICIO = 5;

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

    for (int i=0; i<MAX_SERVICIO; i++) {
        pthread_create(&thr1[i], NULL, servicio, NULL);
    }
    sleep(1);

    gettimeofday(&ts, NULL);
    t1 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;
    pthread_create(&thr, NULL, receptor, NULL);
    pthread_join(thr, NULL);
    for (int i=0; i<MAX_SERVICIO; i++) {
        pthread_join(thr1[i], NULL);
    }
    gettimeofday(&ts, NULL);
    t2 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&no_lleno);
    pthread_cond_destroy(&no_vacio);
    printf("Tiempo: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * receptor (void * param)
{
    const int MAX_PETICIONES = 5;
    peticion_t p; int pos=0;
    for (int i=0; i<MAX_PETICIONES; i++)
    {
        receive_petition(&p);
        pthread_mutex_lock(&mutex);
        while (n_elementos == MAX_BUFFER)
            pthread_cond_wait(&no_lleno, &mutex);
        buffer[pos] = p; pos++;
        n_elementos++;
        pthread_cond_signal(&no_vacio);
        pthread_mutex_unlock(&mutex);
    }
    fprintf(stderr, "Finalizando receptor\n");
    pthread_mutex_lock(&mutex);
    time_t t;
    pthread_cond_broadcast(&no_vacio);
    pthread_mutex_unlock(&mutex);
    fprintf(stderr, "Finalizado receptor\n");
    pthread_exit(0);
    return NULL;
}

void * servicio (void * param)
{
    peticion_t p;
    for (;;)
    {
        pthread_mutex_lock(&mutex);
        while (n_elementos == 0)
        {
            if (fin==1) {
                fprintf(stderr, "Finalizando servicio\n");
                pthread_exit(0);
            }
            pthread_cond_wait(&no_vacio, &mutex);
        } // while
        printf(stderr, "Sirviendo pos. %d\n", pos_servicio);
        p = buffer[pos_servicio];
        pos_servicio = (pos_servicio + 1) % MAX_BUFFER;
        n_elementos--;
        pthread_cond_signal(&no_lleno);
        pthread_mutex_unlock(&mutex);
        responder_petition(&p);
    }
    pthread_exit(0);
    return NULL;
}
```

```
void * servicio (void * param)
{
    peticion_t p;

    for (;;)
    {
        pthread_mutex_lock(&mutex);
        while (n_elementos == 0)
        {
            if (fin==1) {
                fprintf(stderr, "Finalizando servicio\n");
                pthread_mutex_unlock(&mutex);
                pthread_exit(0);
            }
            pthread_cond_wait(&no_vacio, &mutex);
        } // while
        printf(stderr, "Sirviendo pos. %d\n", pos_servicio);
        p = buffer[pos_servicio];
        pos_servicio = (pos_servicio + 1) % MAX_BUFFER;
        n_elementos--;
        pthread_cond_signal(&no_lleno);
        pthread_mutex_unlock(&mutex);
        responder_petition(&p);
    }
    pthread_exit(0);
    return NULL;
}
```



# Comparación

Normal	Procesos	Hilo x petición	<i>Pool</i> de hilos
62.11 seg.	24.08 seg.	20.01 seg.	¿ ?

# SISTEMAS OPERATIVOS: COMUNICACIÓN Y SINCRONIZACIÓN ENTRE PROCESOS



Desarrollo de servidores concurrentes