

# SISTEMAS OPERATIVOS: COMUNICACIÓN Y SINCRONIZACIÓN ENTRE PROCESOS

Desarrollo de servidores concurrentes

# Contenido

2

- ❑ Servidores de peticiones.
- ❑ Solución basada en procesos.
- ❑ Solución basada en hilos bajo demanda.
- ❑ Solución basada en pool de hilos.

# Contenido

3

- ❑ **Servidores de peticiones.**
- ❑ Solución basada en procesos.
- ❑ Solución basada en hilos bajo demanda.
- ❑ Solución basada en pool de hilos.

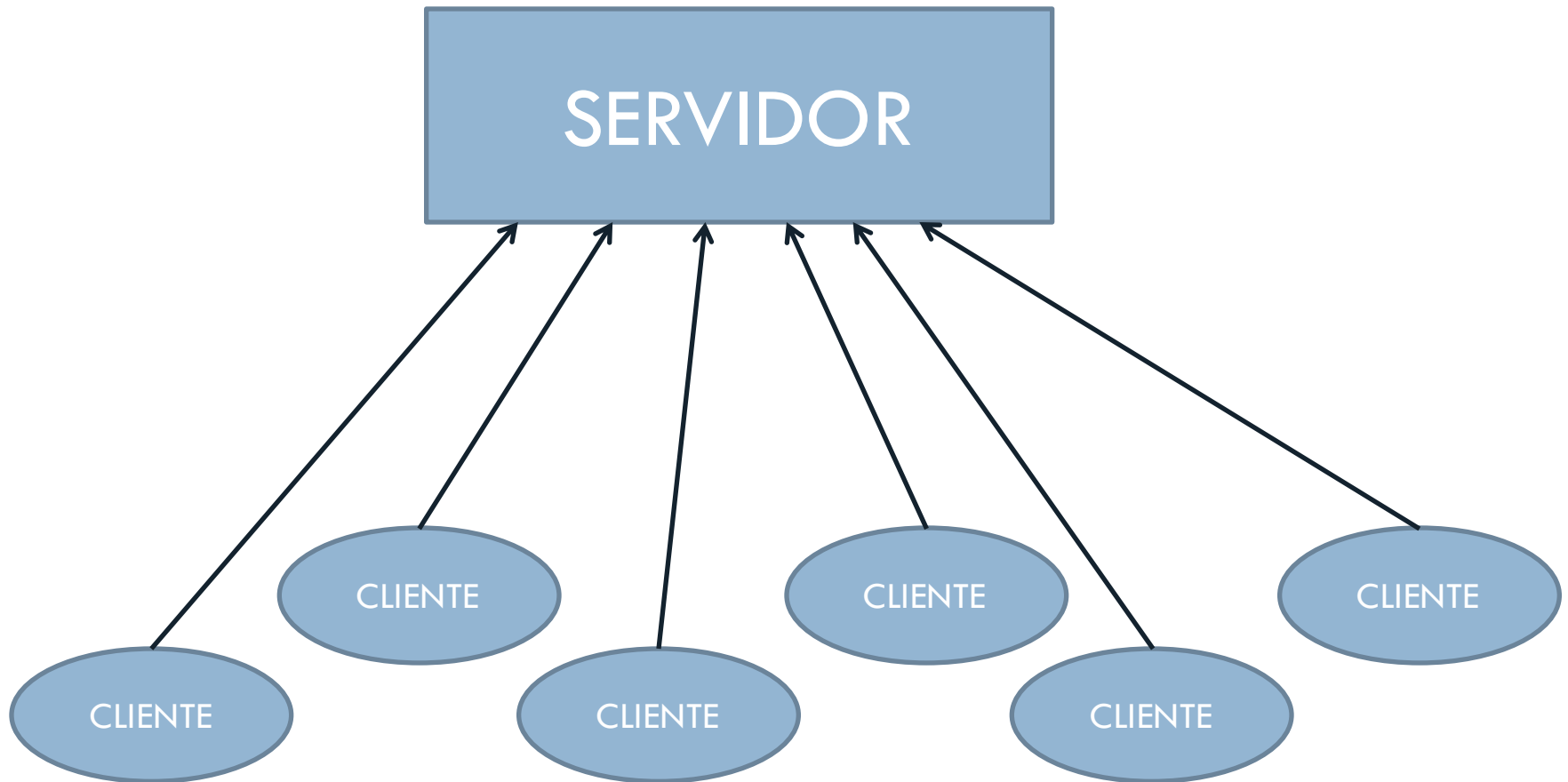
# Servidor de peticiones

4

- 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


5



# Problema:

## Servidor de peticiones

6

- 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.
- 

# Una biblioteca para pruebas

7

- Para poder evaluar las soluciones hoy vamos a usar una biblioteca sencilla como base.
  
- Ideas:
  - ▣ Simular la recepción de peticiones.
  - ▣ Simular el procesamiento de peticiones.
  - ▣ Simular el envío de respuestas.

# Biblioteca base

8

```
#ifndef PETICION_H
#define PETICION_H

    struct petition {
        long id;
        /* Resto de campos necesarios */
        int tipo;
        char url[80];
        /* ... */
    };

    typedef struct petition petition_t;

    void recibir_petition (petition_t * p);
    void responder_petition (petition_t * p);

#endif
```



# Recepción de peticiones

9

```
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

10

```
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);
```

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

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

```
}
```

# Envío de peticiones

11

```
void responder_peticion (peticion_t * p)
{
    int delay, i;
    double x;
    fprintf(stderr, "Enviando petición %d\n", p->id);

    /* Simulación de tiempo de procesamiento */
    for (i=0;i<1000000;i++) { x = 2.0 * i; }

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

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

# Envío de peticiones

12

```
void responder_peticion (peticion_t * p)
{
    int delay, i;
    double x;
    fprintf(stderr, "Enviando petición %d\n", p->id);

    /* Simulación de tiempo de procesamiento */
    for (i=0;i<1000000;i++) { x = 2.0 * i; }

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

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

Aquí iría el  
procesamiento  
de la petición

Aquí iría alguna llamada  
bloqueante para responder  
a la petición

# Una primera solución

13

- Ejecutar de modo indefinido la secuencia:
  - ▣ Recibir una petición.
  - ▣ Procesar la petición.

```
#include "peticion.h"

int main() {
    petition_t p;

    for (;;) {
        recibir_peticion(&p);
        responder_peticion(&p);
    }

    return 0;
}
```

# Problemas

14

- 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?

# Solución inicial con medición

15

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

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

```
#include <time.h>
```

```
int main()
```

```
{
```

```
    int i;
```

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

```
    time_t t1,t2;
```

```
    double dif;
```

```
    peticion_t p;
```



```
t1 = time(NULL);
```

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

```
    recibir_peticion(&p);
```

```
    responder_peticion(&p);
```

```
}
```

```
t2 = time(NULL);
```

```
dif = difftime(t2,t1);
```

```
printf("Tiempo: %lf\n",dif);
```

```
return 0;
```

```
}
```



# Ejecución

16

```
$ time ./ej1
Recibiendo petición
Petición 0 recibida después de 0 segundos
Enviando petición 0
Petición 0 enviada después de 13 segundos
Recibiendo petición
Petición 1 recibida después de 3 segundos
Enviando petición 1
Petición 1 enviada después de 2 segundos
Recibiendo petición
Petición 2 recibida después de 4 segundos
Enviando petición 2
Petición 2 enviada después de 0 segundos
Recibiendo petición
```

```
Petición 3 recibida después de 3 segundos
Enviando petición 3
Petición 3 enviada después de 12 segundos
Recibiendo petición
Petición 4 recibida después de 1 segundos
Enviando petición 4
Petición 4 enviada después de 16 segundos
Tiempo: 54.000000
```

```
real    0m54.164s
user    0m0.061s
sys     0m0.046s
```



# Comparación

17

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

# Contenido

18

- Servidores de peticiones.
- **Solución basada en procesos.**
- Solución basada en hilos bajo demanda.
- Solución basada en pool de hilos.

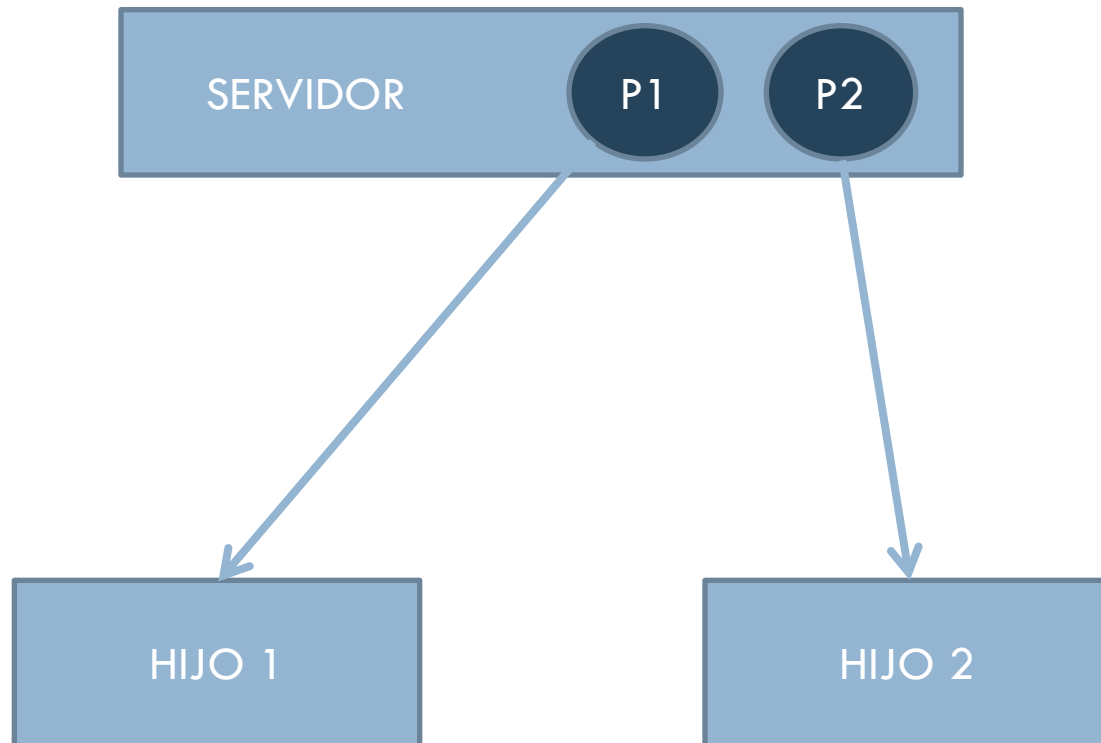
# Primera idea

19

- 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

20



# Implementación (1 / 3)

21

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

```
int main() {  
    const int MAX_PETICIONES = 5;  
    int i;  
    time_t t1,t2;  
    peticion_t p;  
    int pid, hijos=0;  
  
    t1 = time(NULL);
```

# Implementación (2/3)

22

```
for (i=0;i<MAX_PETICIONES;i++) {  
    recibir_peticion(&p);  
  
    do {  
        fprintf(stderr, "Comprobando hijos\n");  
        pid = waitpid(-1, NULL, WNOHANG);  
        if (pid>0) { hijos--; }  
    } while (pid > 0);  
  
    pid = fork();  
    if (pid<0) { perror("Error en la creación del hijo"); }  
    if (pid==0) { responder_peticion(&p); exit(0); } /* HIJO */  
    if (pid!=0) { hijos++; } /* PADRE */  
}
```

# Implementación (3/3)


23

```
fprintf(stderr, "Comprobando %d hijos\n", hijos);  
while (hijos>0) {  
    pid = waitpid(-1, NULL, WNOHANG);  
    if (pid>0) { hijos--; }  
};  
  
t2 = time(NULL);  
  
double dif = difftime(t2,t1);  
printf("Tiempo: %lf\n",dif);  
  
return 0;  
}
```

# Ejecución

24

```
$ time ./ej2
Recibiendo petición
Petición 0 recibida después de 0 segundos
Comprobando hijos
Recibiendo petición
Enviando petición 0
Petición 1 recibida después de 3 segundos
Comprobando hijos
Recibiendo petición
Enviando petición 1
Petición 2 recibida después de 3 segundos
Comprobando hijos
Petición 1 enviada después de 3 segundos
Recibiendo petición
Enviando petición 2
Petición 3 recibida después de 2 segundos
Comprobando hijos
```



```
Comprobando hijos
Recibiendo petición
Enviando petición 3
Petición 2 enviada después de 2 segundos
Petición 4 recibida después de 4 segundos
Comprobando hijos
Comprobando hijos
Comprobando 3Enviando petición hijos4
Petición 4 enviada después de 0 segundos
Petición 0 enviada después de 13 segundos
Petición 3 enviada después de 9 segundos
Tiempo: 17.000000
```

```
real    0m17.311s
user    0m0.872s
sys     0m3.092s
```





# Comparación

25

Normal	Procesos	Hilo x petición	<i>Pool</i> de hilos
54 seg.	17 seg.		

# Problemas

26

- ❑ 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

27

- 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.

# Contenido

28

- Servidores de peticiones.
- Solución basada en procesos.
- **Solución basada en hilos bajo demanda.**
- Solución basada en pool de hilos.

# Hilos bajo demanda

29

- 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

30

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

sem_t snhijos;

int main()
{
    time_t t1, t2;
    double dif;
    pthread_t thr;
```



```
t1 = time(NULL);
sem_init(&snhijos, 0, 0);
pthread_create(&thr, NULL,
               receptor, NULL);
pthread_join(thr, NULL);
sem_destroy(&snhijos);
t2 = time(NULL);

dif = difftime(t2,t1);
printf("Tiempo: %lf\n",dif);

return 0;
}
```

# Implementación: receptor

31

```
void * receptor (void * param)
{
    const int MAX_PETICIONES = 5; int nservicio = 0; int i;
    petition_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;
}
```

# Implementación: servicio

32

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

    copia_peticion(&pet,(petición_t*)p);
    fprintf(stderr, "Iniciando servicio\n");
    responder_peticion(&pet);
    sem_post(&snhijos);

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



# Reflexión

33

- ¿Puede darse una condición de carrera?

# Ejecución

34

```
$ time ./ej3  
Recibiendo petición  
Petición 0 recibida después de 0  
segundos  
Recibiendo petición  
Iniciando servicio  
Enviando petición 1  
Petición 1 enviada después de 0 segundos  
Terminando servicio  
Petición 1 recibida después de 3  
segundos  
Recibiendo petición  
Iniciando servicio  
Enviando petición 2  
Petición 2 enviada después de 0 segundos  
Terminando servicio
```

```
Petición 2 recibida después de 3  
segundos  
Recibiendo petición  
Iniciando servicio  
Enviando petición 3  
Petición 3 enviada después de 0 segundos  
Terminando servicio  
Petición 3 recibida después de 2  
segundos  
Recibiendo petición  
Iniciando servicio  
Enviando petición 4  
Petición 4 enviada después de 0 segundos  
Terminando servicio  
Petición 4 recibida después de 4  
segundos
```

# Ejecución

35

```
Haciendo wait
Iniciando servicio
Saliendo de wait
Enviando petición 4
Haciendo wait
Saliendo de wait
Haciendo wait
Saliendo de wait
Haciendo wait
Saliendo de wait
Haciendo wait
Petición 4 enviada después de 0 segundos
Terminando servicio
Saliendo de wait
Tiempo: 12.000000
```

```
real    0m12.132s
user    0m0.046s
sys     0m0.015s
```

# Comparación

36

Normal	Procesos	Hilo x petición	<i>Pool</i> de hilos
54 seg.	17 seg.	12 seg.	

# Problema

37

- 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

38

- ¿Puede darse una condición de carrera?



# Reflexión

39

□ ¿Puede darse una condición de carrera?

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

```
peticion_t p; ■■■
```

```
recibir_peticon(&p);
```

```
nservicio++;
```

```
pthread_create(&hijo, NULL, servicio, &p);
```

```
recibir_peticon(&p);
```

```
nservicio++;
```

```
pthread_create(&hijo, NULL, servicio, &p);
```



```
...
```

# Reflexión

40

- ¿Puede darse una condición de carrera?

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

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



# Reflexión

41

□ ¿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

42

□ ¿Puede darse una condición de carrera?

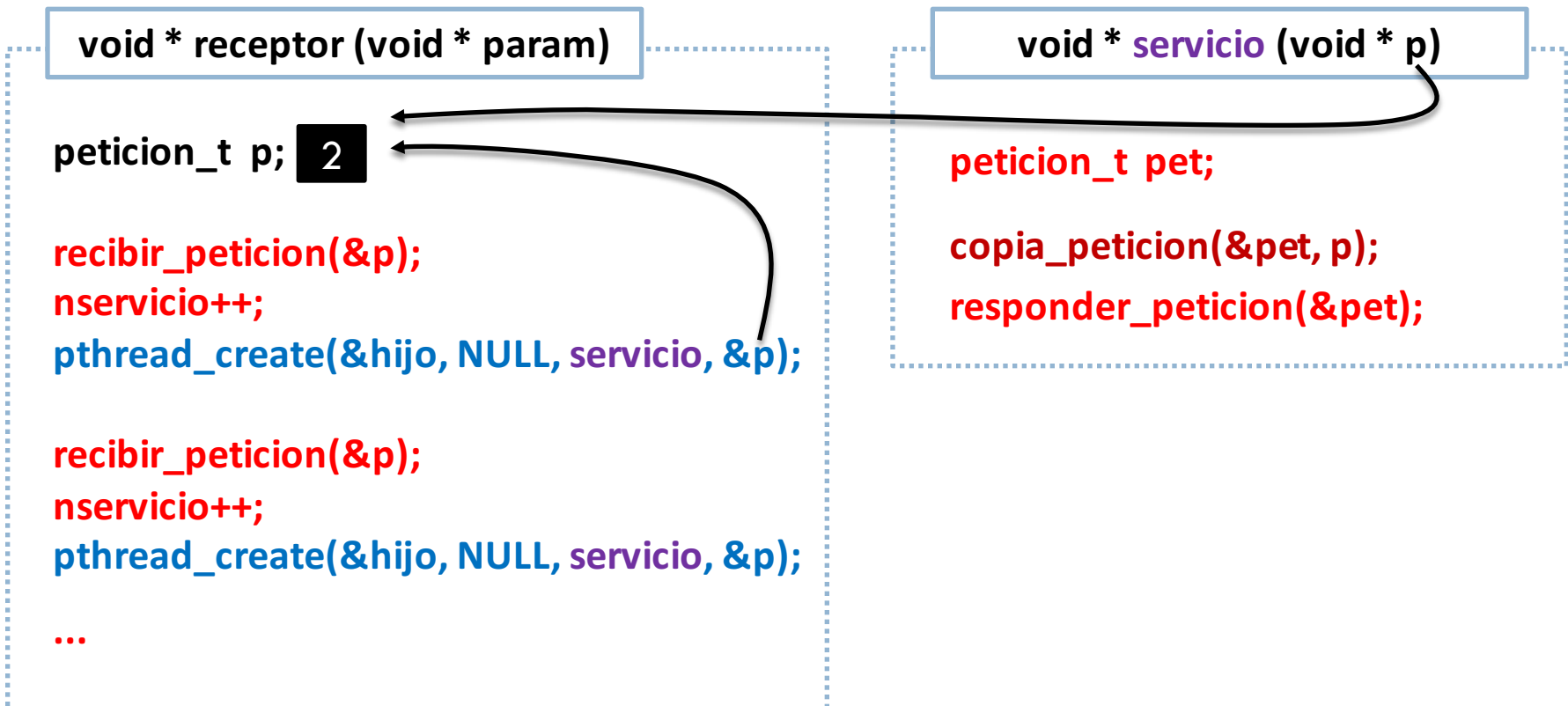
```
void * receptor (void * param)
```

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

# Reflexión

43

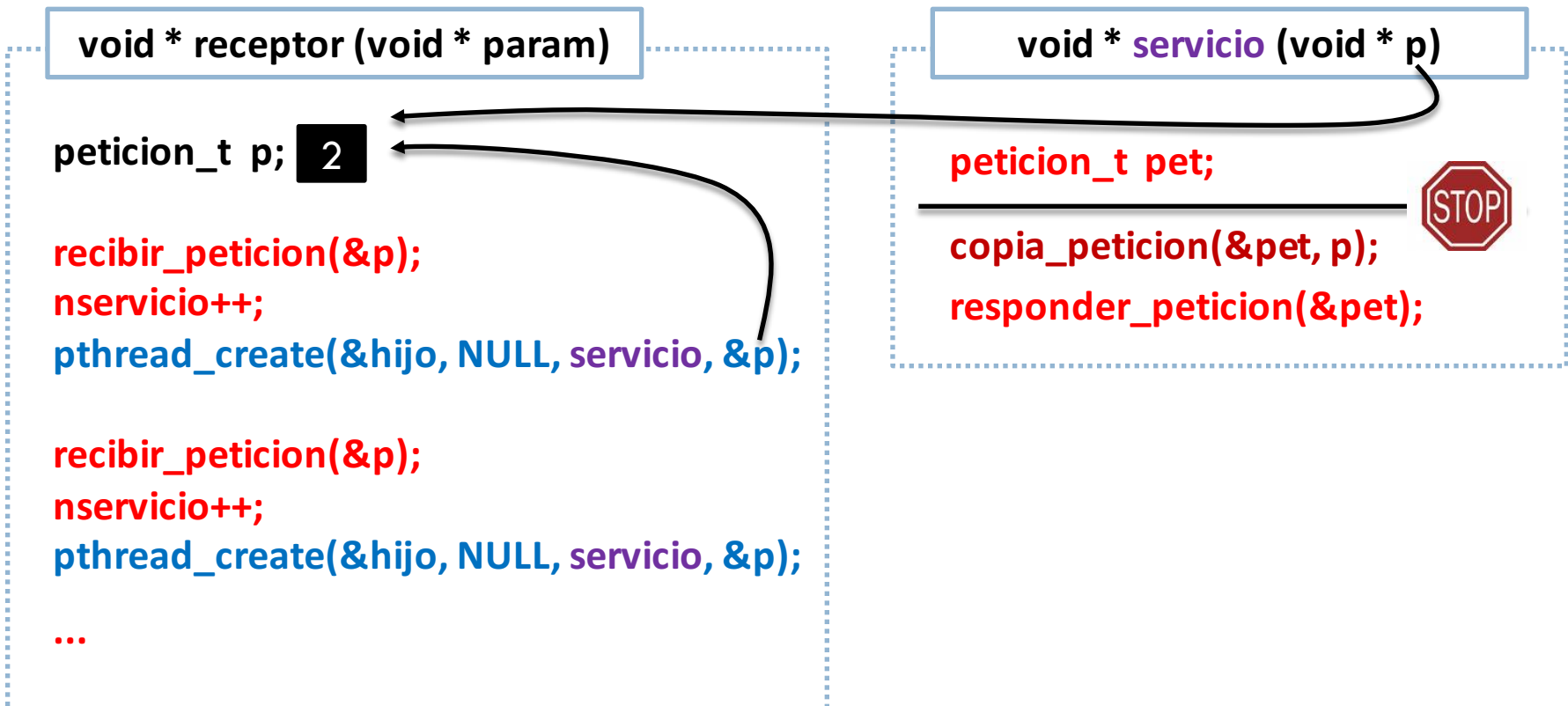
□ ¿Puede darse una condición de carrera?



# Reflexión

44

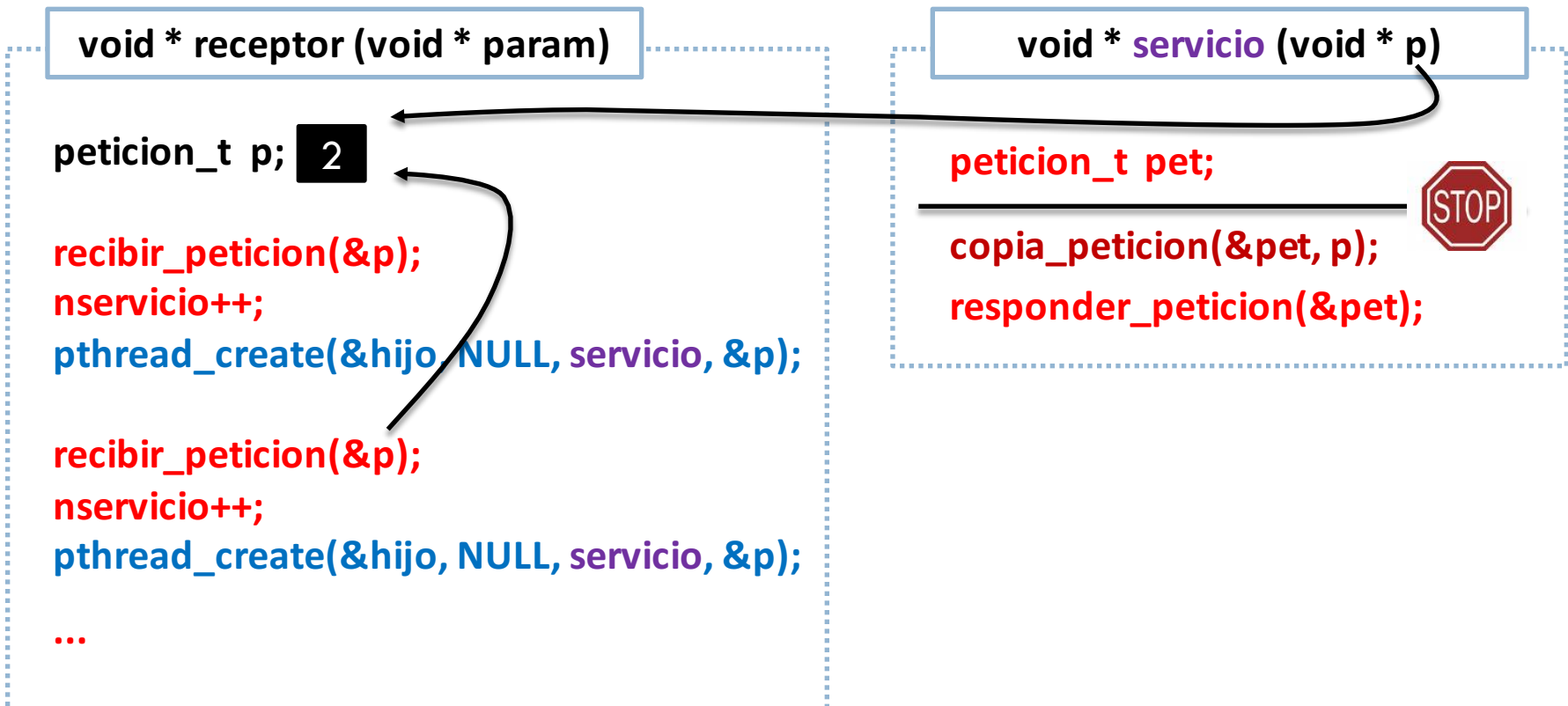
□ ¿Puede darse una condición de carrera?



# Reflexión

45

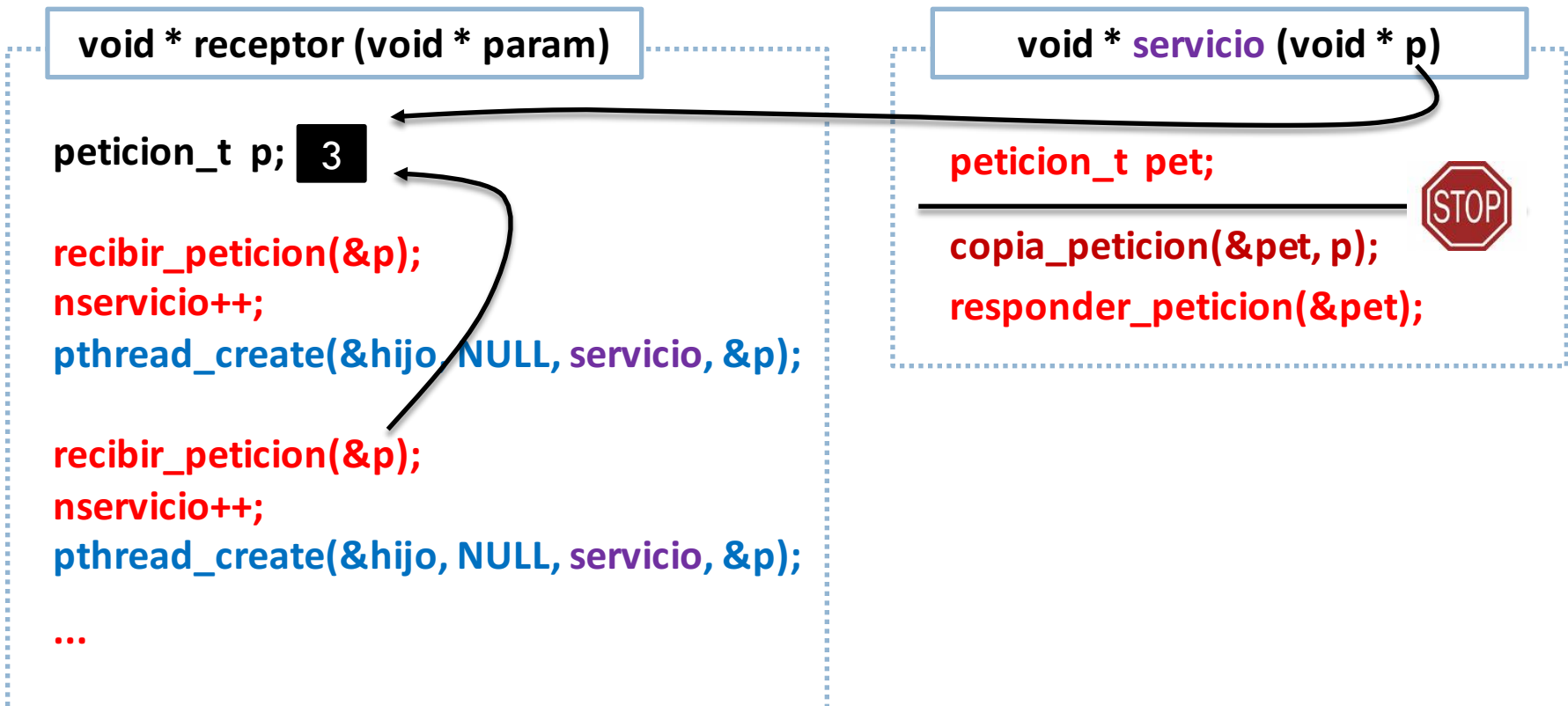
□ ¿Puede darse una condición de carrera?



# Reflexión

46

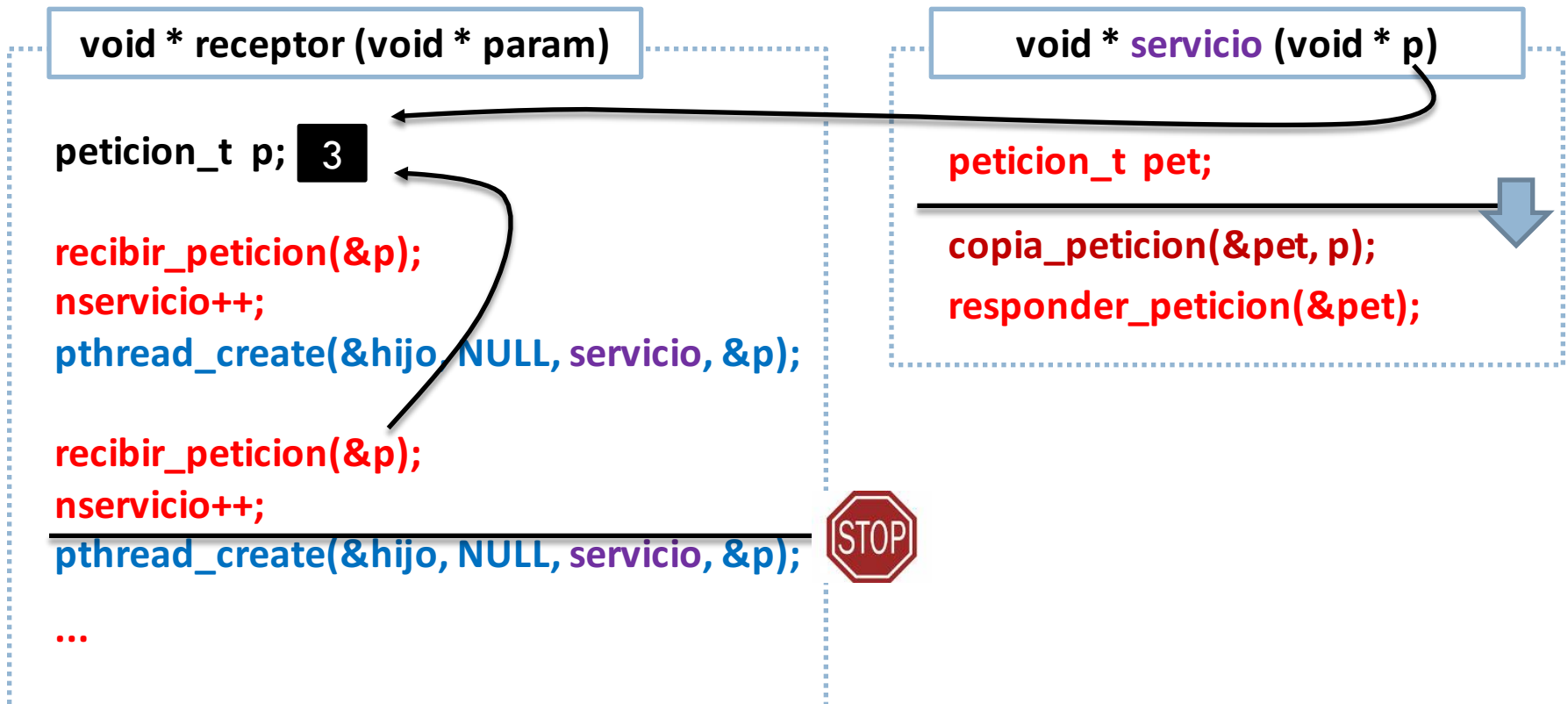
□ ¿Puede darse una condición de carrera?



# Reflexión

47

□ ¿Puede darse una condición de carrera?



# Contenido

48

- Servidores de peticiones.
- Solución basada en procesos.
- Solución basada en hilos bajo demanda.
- **Solución basada en pool de hilos.**



# Pool de threads

49

- 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)

50

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

#define MAX_BUFFER 128
peticion_t buffer[MAX_BUFFER];
int n_elementos;
int pos_servicio = 0;
pthread_mutex_t mutex;
pthread_cond_t no_lleno;
pthread_cond_t no_vacio;

pthread_mutex_t mfin;
int fin=0;
```

# Implementación: main (2/3)

51

```
int main()
{
    time_t t1, t2;
    double dif;
    pthread_t thr;
    pthread_t ths[MAX_SERVICIO];
    const int MAX_SERVICIO = 5; int i;

    t1 = time(NULL);

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

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

# Implementación: main (3/3)

52

```
pthread_join(thr, NULL);
for (i=0;i<MAX_SERVICIO;i++) {
    pthread_join(thr[i],NULL);
}

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

t2 = time(NULL);

dif = difftime(t2,t1);
printf("Tiempo: %lf\n",dif);

return 0;
}
```

# Implementación: receptor (1 / 2)

53

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

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

# Implementación: receptor (2/2)

54

```
fprintf(stderr,"Finalizando receptor\n");  
pthread_mutex_lock(&mfin);  
fin=1;  
  
pthread_mutex_unlock(&mfin);  
pthread_mutex_lock(&mutex);  
pthread_cond_broadcast(&no_vacio);  
pthread_mutex_unlock(&mutex);  
  
fprintf(stderr, "Finalizado receptor\n");  
pthread_exit(0);  
return NULL;  
} /* receptor */
```

# Implementación: servicio (1 / 2)

55

```
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
    }
```

# Implementación: servicio (2/2)

56

```
fprintf(stderr, "Sirviendo posicion %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_peticion(&p);  
}  
  
pthread_exit(0);  
return NULL;  
}
```



# Comparación

57

Normal	Procesos	Hilo x petición	<i>Pool</i> de hilos
54 seg.	17 seg.	12 seg.	?