Bloque I: Principios de sistemas operativos

Notas:

Tema 4. Sincronización y programación dirigida por eventos

- Necesidad de la sincronización y principales métodos.
- Sincronización de acceso mutuamente exclusivo.
- Sincronización de espera mediante semáforos.
- Sincronización de espera mediante variables condicionales.
- Interbloqueos (*deadlocks*).
- Generación, bloqueo y aceptación de señales.

1. Necesidad de la sincronización y principales métodos

Los procesos o threads pueden:

- ser independientes de los demás
- cooperar entre ellos
- competir por el uso de recursos compartidos

En los dos últimos casos los procesos deben sincronizarse para:

- intercambiar datos o eventos, mediante algún mecanismo de señalización y espera, o de paso de mensajes
- usar recursos compartidos de manera mutuamente exclusiva

Exclusión mutua

Se usa para asegurar la *congruencia* de los datos o dispositivos compartidos

Mecanismos habituales:

- semáforos (binarios o contadores)
- mutexes y regiones críticas
- monitores
- regiones críticas condicionales
- variables condicionales

Semáforos

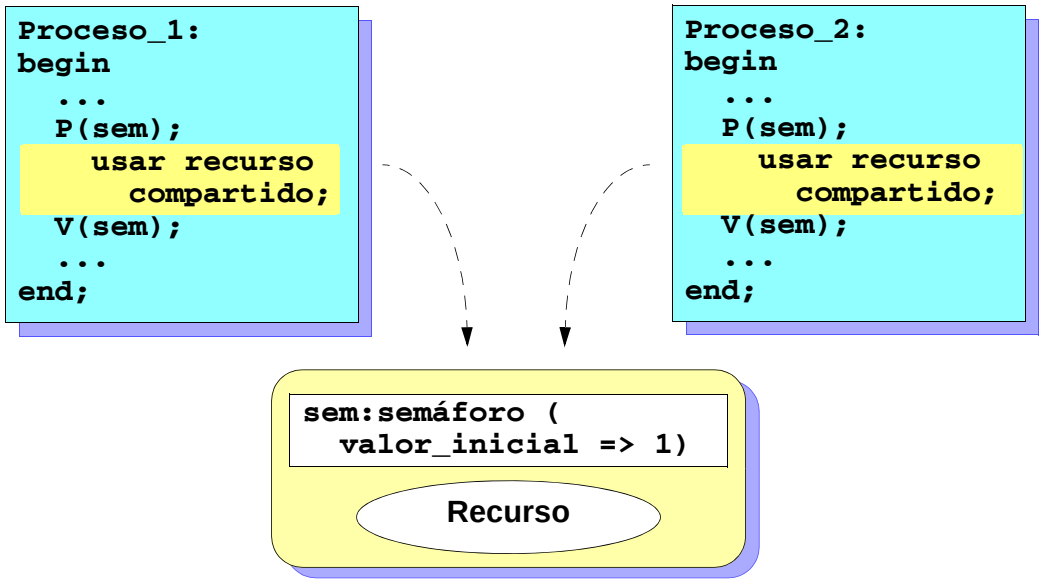
Tienen una *cola* de procesos en espera, y un *valor* que es:

- un entero positivo o cero para los semáforos contadores
- un valor binario (0,1) para los semáforos binarios

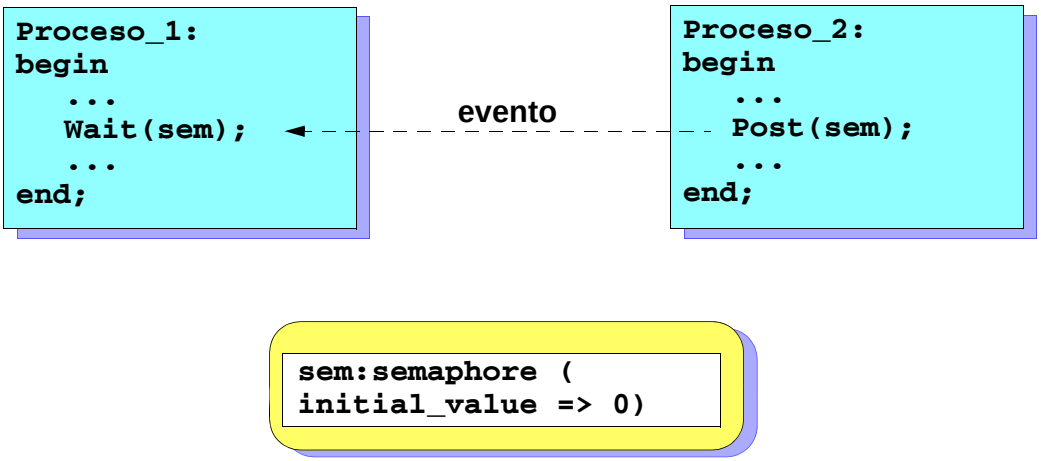
Tienen dos operaciones definidas

- "*P*" o *esperar*:
  - si valor>0 entonces valor:=valor-1
  - si valor=0 entonces suspender el proceso (metiéndolo en la cola)
- "*V*" o *señalizar*:
  - si hay un proceso esperando, activarlo
  - si no, valor:=valor+1 (con límite 1 para los binarios)

Semáforos: exclusión mutua



Semáforos: sincronización de espera a un evento



Mutexes

Muy similares a los semáforos, pero con un nuevo concepto:

- cada recurso tiene un *propietario*

Su estado puede ser *libre*, o *tomado*. Tienen dos operaciones:

- *tomar* (*lock*):
  - si libre, el estado se cambia a tomado
  - si no, espera
  - el proceso que toma el mutex es el propietario
- *liberar* (*unlock*), sólo permitido al propietario:
  - si hay procesos esperando, se activa uno, que se convierte en el nuevo propietario
  - si no, el mutex se libera

Regiones críticas

Son instrucciones estructuradas en algunos lenguajes concurrentes

Funcionan como un mutex, pero de manera más fiable

- ya que la operación de liberar está implícita en la instrucción

Monitores

Un monitor encapsula el código relativo a un recurso compartido en un solo módulo de programa; ventajas:

- mantenimiento más simple
- menos errores de programación

La interfaz del monitor es un conjunto de funciones que representan las diferentes operaciones que pueden hacerse con el recurso

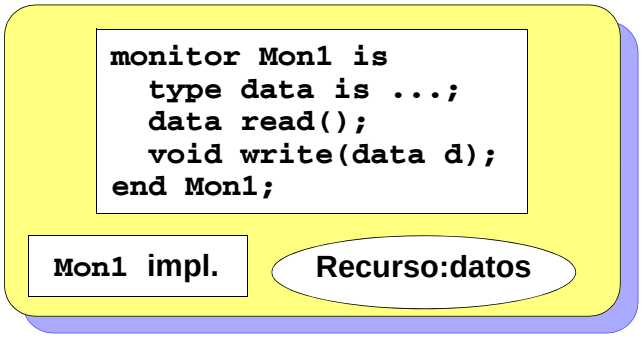
La implementación del monitor garantiza la exclusión mutua

- mediante semáforos o algún otro mecanismo
- o implícitamente en los lenguajes concurrentes

Monitor: ejemplo

```
Process_1:
begin
   ...
   d1=Mon1.read();
   ...
   Mon1.write(d2);
   ...
end;
```

```
Process_2:
begin
   ...
   Mon1.write(d3);
   ...
   d4=Mon1.read();
   ...
end;
```



Monitor: implementación

```
monitor body Mon1 is
  mutex m;
  data shared_data;

  data read () is
  begin
    lock(m);
    d:=shared_data;
    unlock (m);
    return d;
  end read;

  void write(data d) is
  begin
    lock(m);
    shared_data=d;
    unlock(m);
  end write;
end Mon1;
```

Paso de mensajes

El paso de mensajes es un mecanismo de sincronización común en programas concurrentes. Tiene dos operaciones:

- *esperar* (*wait*): un proceso espera hasta poder recibir un mensaje
- *señalizar* (*signal*): un proceso envía un mensaje a otro(s) proceso(s); tiene tres alternativas:
  - *asíncrona*: el proceso que señaliza no espera
  - *síncrona*: el proceso que señaliza espera hasta que el mensaje se recibe
  - *punto de encuentro* o *rendezvous*: el proceso que señaliza espera; cuando el receptor está listo ejecutan un fragmento de código en común, en el que pueden intercambiar datos en ambas direcciones

Paso de mensajes (cont.)

Hay dos formas de especificar el destino de un mensaje:

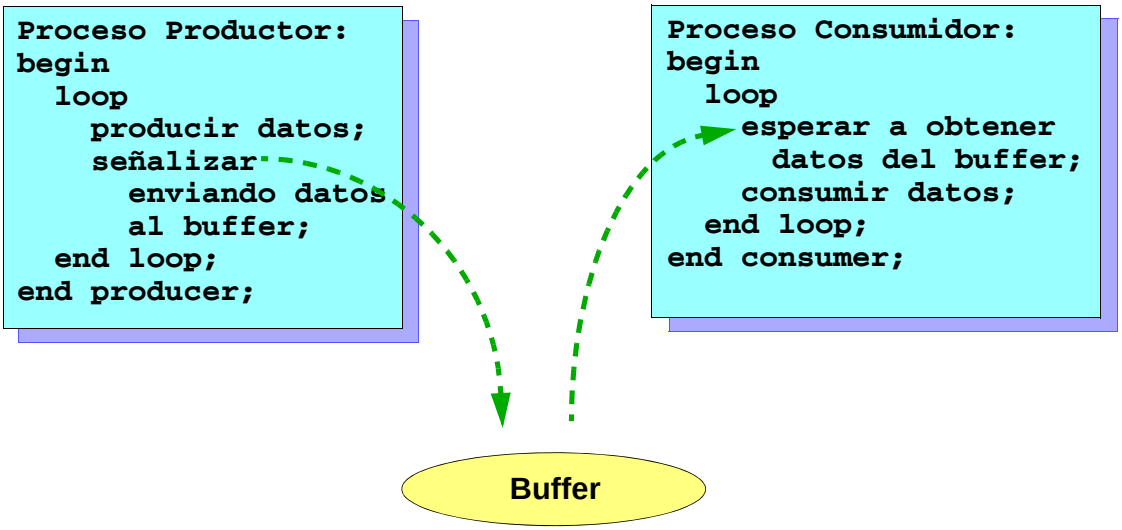
- especificando un *proceso* destino
- especificando un objeto "*buzón*" (también llamado *canal* o *cola de mensajes*)

Muchos sistemas ofrecen la espera selectiva, que se puede usar para esperar a uno de entre varios eventos

Otras operaciones: espera condicional, espera temporizada, y espera con barreras o condiciones extra

Se suelen usar monitores para encapsular todo el código relativo a una clase particular de eventos o mensajes

Paso de mensajes: ejemplo



2. Sincronización de acceso mutuamente exclusivo

***Mutex*:**

- objeto de sincronización por el que múltiples threads acceden de forma mutuamente exclusiva a un recurso compartido
- operaciones:
  - ***tomar o bloquear***: si el mutex está libre, se toma y el thread correspondiente se convierte en propietario; si no el thread se suspende y se añade a una cola
  - ***liberar***: si hay threads esperando en la cola, se elimina uno de la cola y se convierte en el nuevo propietario; si no, el mutex queda libre; sólo el propietario del mutex puede invocar esta operación
- los mutex permiten opcionalmente la herencia de prioridad

Mutexes en POSIX

Atributos de inicialización:

- *pshared*: indica si es compartido o no entre procesos:
  - PTHREAD_PROCESS_SHARED
  - PTHREAD_PROCESS_PRIVATE
- hay otros atributos relativos a la planificación, que veremos más adelante

Estos atributos se almacenan en un objeto de atributos

Objetos de atributos para mutex

```
#include <pthread.h>

int pthread_mutexattr_init
      (pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy
      (pthread_mutexattr_t *attr);

int pthread_mutexattr_getpshared
      (const pthread_mutexattr_t *attr,
       int *pshared);
int pthread_mutexattr_setpshared
      (pthread_mutexattr_t *attr,
       int pshared);
```

Inicializar y destruir un mutex

Inicializar un Mutex:

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

Destruir un mutex:

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


Tomar y liberar un mutex

Tomar un mutex y suspenderse si no está libre:

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

Tomar un mutex, sin suspenderse:

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

Liberar un mutex

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

Ejemplo: come-tiempos

```
////////////////////////////////////////////////////
///                    LOAD.H                    ///
////////////////////////////////////////////////////
//    La función eat() permite consumir el tiempo   //
//    de CPU indicado por for_seconds               //

//    La función adjust() debe ser llamada una vez  //
//    antes de llamar a eat(). No es necesario      //
//    llamarla posteriormente.                      //

void eat (float for_seconds);

void adjust (void);
```


Ejemplo con un mutex

```
#include <pthread.h>
#include <stdio.h>
#include "load.h"
struct shared_data {
   pthread_mutex_t mutex;
   int a,b,c,i;
} data;
// Este thread incrementa a, b y c 20 veces
void * incrementer (void *arg)
{
   for (data.i=1;data.i<20;data.i++) {
      pthread_mutex_lock(&data.mutex);
      data.a++; eat(0.1);
      data.b++; eat(0.1);
      data.c++; eat(0.1);
      pthread_mutex_unlock(&data.mutex);
   }
   pthread_exit (NULL);
}
```

Ejemplo con un mutex (cont.)

```
// Este thread muestra el valor de a, b y c
// hasta que i vale 20
void * reporter (void *arg)
{
   do {
      pthread_mutex_lock(&data.mutex);
      printf("report a=%d, b=%d, c=%d\n",data.a,data.b,data.c);
      pthread_mutex_unlock(&data.mutex);
      eat(0.2);
   } while (data.i<20);
   pthread_exit (NULL);
}

// Programa principal: crea el mutex y los threads
// Luego, espera a que los threads acaben
int main()
{
   pthread_mutexattr_t mutexattr;
   pthread_t t1,t2;
```


Ejemplo con un mutex (cont.)

```
   adjust();
   data.a = 0; data.b =0; data.c = 0; data.i = 0;
   pthread_mutexattr_init(&mutexattr);
   if (pthread_mutex_init(&data.mutex,&mutexattr)!= 0) {
        printf ("mutex_init\n"); exit(1);
   }

   if (pthread_create (&t1,NULL,incrementer,NULL)!= 0) {
        printf ("pthread_create\n");
        exit(1);
   }
   if (pthread_create (&t2,NULL,reporter,NULL)!= 0) {
        printf ("pthread_create\n");
        exit(1);
   }
   if (pthread_join(t1,NULL)!= 0) printf ("in join\n");
   if (pthread_join(t2,NULL)!= 0) printf ("in join\n");
   exit (0);
}
```

Monitor de sincronización

Para acceder de forma mutuamente exclusiva a un objeto de datos es preferible crear un monitor

- con operaciones que encapsulan el acceso mutuamente exclusivo
- con una interfaz separada del cuerpo

Esto evita errores en el manejo del mutex, que podrían bloquear la aplicación

A continuación se muestra un ejemplo con el mismo objeto compartido del ejemplo anterior, y con tres operaciones:

- inicializar, modificar, y leer

Ejemplo con monitor

```
//////////////////////////////////////////////////
// File shared_object.h
// Object Prototype

typedef struct {
   int a,b,c,i;
 } shared_t;

#define OK    0
#define ERROR 1

int sh_create (const shared_t *initial_value);
     // sh_create must finish before the object is used

int sh_modify (const shared_t *new_value);
    // store the new value of a, b and c, and increment i

int sh_read (shared_t *current_value);
```

Ejemplo con monitor (cont.)

```
////////////////////////////////////////////////////
// File shared_object.c: Object Implementation

#include <sched.h>
#include <pthread.h>
#include "shared_object.h"

shared_t object;
pthread_mutex_t the_mutex;

int sh_create (const shared_t *initial_value)
{
      if (pthread_mutex_init(&the_mutex,NULL)!=0) {
           return ERROR;
      }
      object=*initial_value;
      return OK;
}
```


Ejemplo con monitor (cont.)

```
int sh_modify (const shared_t *new_value)
{
    if (pthread_mutex_lock(&the_mutex)!=0) {
          return ERROR;
    }
    object=*new_value;
    if (pthread_mutex_unlock(&the_mutex)!=0) {
         return ERROR;
    }
    return OK;
}
```

Ejemplo con monitor (cont.)

```
int sh_read (shared_t *current_value)
{
    if (pthread_mutex_lock(&the_mutex)!=0) {
        return ERROR;
    }
    *current_value=object;
    if (pthread_mutex_unlock(&the_mutex)!=0) {
        return ERROR;
    }
    return OK;
}
```


Ejemplo con monitor (cont.)

```
#include <pthread.h>
#include <stdio.h>
#include "load.h"
#include "shared_object.h"

static int error =-1;

// Este thread incrementa 'a', 'b', y 'c' 20 veces

void * incrementer (void *arg)
{
  int i;
  shared_t data;

  for (i=1;i<=20;i++) {
    if (sh_read(&data)!=OK) {
      pthread_exit(&error);
    }
```

Ejemplo con monitor (cont.)

```
    data.a++; eat(0.1);
    data.b++; eat(0.1);
    data.c++; eat(0.1);
    data.i=i;
    if (sh_modify(&data)!=OK) {
      pthread_exit(&error);
    }
  }
  pthread_exit (NULL);
}

// Este thread muestra el valor de 'a', 'b' y 'c'
// hasta que i vale 20

void * reporter (void *arg)
{
  shared_t data;
```


Ejemplo con monitor (cont.)

```
  do {
    if (sh_read(&data)!=OK) {
      pthread_exit(&error);
    }
    printf("a=%d, b=%d, c=%d\n",data.a,data.b,data.c);
    eat(0.2);
  } while (data.i<20);
  pthread_exit (NULL);
}

// Programa principal: crea el mutex y los threads
// Luego, espera a que los threads acaben

int main() {

  pthread_t t1,t2;
  pthread_attr_t attr;
  void * st;
  shared_t data;
```

Ejemplo con monitor (cont.)

```
  adjust();
  data.a = 0; data.b =0; data.c = 0; data.i = 0;
  if (sh_create(&data)!=0) {
    printf("error en sh_create\n"); exit(1);
  }
  if (pthread_attr_init(&attr) != 0) {
    printf("error en pthread_attr_create\n"); exit(1);
  }
  if (pthread_create (&t1,&attr,incrementer,NULL) != 0) {
    printf("error en pthread_create\n"); exit(1);
  }
  if (pthread_create (&t2,&attr,reporter,NULL) != 0) {
    printf("error en pthread_create\n"); exit(1);
  }
  if (pthread_join(t1,&st) != 0) printf ("error en pthread_join");
  if (pthread_join(t2,&st) != 0) printf ("error en pthread_join");
  exit (0);
}
```


3. Sincronización de espera mediante semáforos

Semáforo Contador:

- es un recurso compartible con un valor entero no negativo
- cuando el valor es cero, el semáforo no está disponible
- se utiliza tanto para sincronización de acceso mutuamente exclusivo, como de espera
- operaciones que se aplican al semáforo:
  - *esperar*: si el valor es 0, el proceso o thread se añade a una cola; si es >0, se decrementa
  - *señalizar*: si hay procesos o threads esperando, se elimina uno de la cola y se le activa; si no, se incrementa el valor
- existen semáforos con nombre, y sin nombre

Semáforos contadores

Inicializar un semáforo sin nombre:

```
#include <semaphore.h>
int sem_init (sem_t *sem, int pshared,
              unsigned int value);
```

- inicializa el semáforo al que apunta `sem`
- si `pshared` = 0 el semáforo sólo puede ser usado por threads del mismo proceso
- si `pshared` no es 0, el semáforo se puede compartir entre diferentes procesos
- `value` es el valor inicial del semáforo

Semáforos contadores (cont.)

Destruir un semáforo sin nombre:

```
int sem_destroy (sem_t *sem);
```

Abrir/inicializar un semáforo con nombre:

```
sem_t *sem_open (const char *name, int oflag,
                 [,mode_t mode, unsigned int value]);
```

- devuelve un puntero al semáforo
- `name` debe tener el formato `"/nombre"`
- `oflag` indica las opciones:
  - `O_CREAT`: si el semáforo no existe, se crea; en este caso se requieren los parámetros `mode`, que indica permisos, y `value`, que es el valor inicial
  - `O_EXCL`: si el semáforo ya existe, error

Semáforos contadores (cont.)

Cerrar un semáforo con nombre:

```
int sem_close (sem_t *sem);
```

Borrar un semáforo con nombre:

```
int sem_unlink (const char *name);
```

Esperar en un semáforo:

```
int sem_wait (sem_t *sem);
```

- decrementa el semáforo si está libre; si no, se suspende hasta que se libera, o se ejecuta un manejador de señal

```
int sem_trywait (sem_t *sem);
```

- decrementa el semáforo si está libre; si no, retorna indicando un error

Semáforos contadores (cont.)

Señalizar un semáforo:

```
int sem_post (sem_t *sem);
```

Leer el valor de un semáforo:

```
int sem_getvalue (sem_t *sem, int *sval);
```

Ejemplo: espera entre procesos

```
#include <stdio.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include "load.h"
int main()
{
      sem_t *sem; pid_t childpid; int i;
      adjust();
      if ((sem = sem_open ("/my_sem",O_CREAT,S_IRUSR|S_IWUSR,0))
         == SEM_FAILED) {
         // valor inicial =0 para sincronización de espera
            perror("error in sem_open\n");
      }
      if ((childpid = fork()) == -1) {
            perror("fork failed\n");
      }
```


Ejemplo de espera (cont.)

```
// El proceso hijo señaliza el semáforo 20 veces
      if (childpid == 0) {
               for (i=0;i<20;i++) {
                  eat(1.0);
                  printf("child process writes i=%d\n",i);
                  sem_post(sem);
               }
               sem_close(sem); exit(0);
      } else {
// El proceso padre espera en el semáforo 20 veces
               for (i=0;i<20;i++) {
                  eat(0.5);
                  sem_wait(sem);
                  printf("parent process writes i=%d\n",i);
               }
               sem_close(sem);
               sem_unlink("/my_sem"); exit(0);
      }
}
```

4. Sincronización de espera con variables condicionales

Variable Condicional

- objeto de sincronización que permite a un thread suspenderse hasta que otro thread lo reactiva y se cumple una condición lógica
- operaciones:
  - *esperar* a una condición: se suspende la tarea hasta que otro thread señaliza esa condición; entonces, generalmente se comprueba una condición lógica, y se repite la espera si la condición es falsa
  - *señalizar* una condición: se reactiva uno o más de los threads suspendidos es espera de esa condición
  - "*broadcast*" de una condición: se reactivan todos los threads suspendidos en espera de esa condición

Variables condicionales

Las variables condicionales se utilizan para sincronización de espera a una condición

La condición se representa mediante una condición booleana

La evaluación de la condición booleana se hace protegida mediante un mutex

Pseudocódigo del thread que señaliza:

```
pthread_mutex_lock(un_mutex);
   condicion=TRUE;
   pthread_cond_signal(cond);
pthread_mutex_unlock(un_mutex);
```

Variables condicionales (cont.)

Pseudocódigo del thread que espera:

```
pthread_mutex_lock(un_mutex);
  while (condicion == FALSE) {
       pthread_cond_wait (cond, un_mutex);
  }
pthread_mutex_unlock(un_mutex);
```

- la condición se cambia o evalúa con el mutex tomado
- la operación de espera está ligada al mutex:
  - para esperar a la condición es necesario tomar el mutex
  - mientras se espera, el mutex se libera de forma automática
  - al retornar la función, el mutex está tomado

Atributos de variables condicionales

Atributos definidos:

- `pshared`: indica si se puede compartir entre procesos:
  - `PTHREAD_PROCESS_SHARED`
  - `PTHREAD_PROCESS_PRIVATE`

Funciones:

```
int pthread_condattr_init (pthread_condattr_t *attr);
int pthread_condattr_destroy (pthread_condattr_t *attr);

int pthread_condattr_getpshared
      (const pthread_condattr_t *attr,
       int *pshared);
int pthread_condattr_setpshared (
      (pthread_condattr_t *attr,
       int pshared);
```

Inicializar y destruir variables condicionales

Inicializar:

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

Destruir:

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


Operaciones con variables condicionales

Señalizar:

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

Broadcast:

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

Esperar:

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

Espera con timeout

```
int pthread_cond_timedwait
     (pthread_cond_t *cond,
      pthread_mutex_t *mutex,
      const struct timespec *abstime);
```

Especificación del tiempo con alta resolución

Se usa la estructura timespec:

```
struct timespec {
    time_t tv_sec;     // segundos
    long int tv_nsec; // nanosegundos
}
```

El tipo `time_t` es un entero de al menos 32 bits

El número de nanosegundos debe estar comprendido entre 0 y 999.999.999

Ejemplo de espera

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

// Se define un buffer para almacenar los datos producidos
// (Hasta un máximo de 20)

struct buffer {
      int num_consumed;
      int num_produced;
      pthread_cond_t cond;
      pthread_mutex_t mutex;
      int data[20];
};
```

Ejemplo de espera (cont.)

```
void * producer (void *arg)
{
      struct buffer *my_buffer;
      int i;

      my_buffer = (struct buffer *)arg;
      for (i=40;i>20;i--) {
            eat(1.0);
            printf("producer produces i=%d\n",i);
            pthread_mutex_lock(&my_buffer->mutex);
            my_buffer->num_produced++;
            my_buffer->data[my_buffer->num_produced-1]=i;
            pthread_cond_signal(&my_buffer->cond);
            pthread_mutex_unlock(&my_buffer->mutex);
      }
      pthread_exit (NULL);
}
```


Ejemplo de espera (cont.)

```
void * consumer (void *arg)
{
   struct buffer *my_buffer;
   int i,consumed;

   my_buffer = (struct buffer *)arg;
   for (i=0;i<20;i++) {
      pthread_mutex_lock(&my_buffer->mutex);
      while (my_buffer->num_produced <= my_buffer->num_consumed) {
            pthread_cond_wait(&my_buffer->cond,&my_buffer->mutex);
      }
      my_buffer->num_consumed++;
      consumed=my_buffer->data[my_buffer->num_consumed-1];
      pthread_mutex_unlock(&my_buffer->mutex);
      eat(0.5);
      printf("consumer consumes i=%d\n", consumed);
   }
   pthread_exit (NULL);
}
```

Ejemplo de espera (cont.)

```
int main()
{
   pthread_mutexattr_t mutexattr;
   pthread_condattr_t condattr;
   pthread_t t1,t2;
   struct buffer my_buffer;

   adjust();
   my_buffer.num_produced = 0;
   my_buffer.num_consumed = 0;
   pthread_mutexattr_init(&mutexattr);
   if (pthread_mutex_init(&my_buffer.mutex,&mutexattr)!= 0) {
        printf ("error in mutex_init\n");
   }

   pthread_condattr_init(&condattr);
   if (pthread_cond_init(&my_buffer.cond,&condattr)!= 0) {
        printf ("error in cond_init\n");
   }
```


Ejemplo de espera (cont.)

```
   if (pthread_create (&t1,NULL,producer,&my_buffer)!= 0) {
        printf("error in pthread_create\n");
        exit(1);
   }
   if (pthread_create (&t2,NULL,consumer,&my_buffer)!= 0) {
        printf("error in pthread_create\n");
        exit(1);
   }
   if (pthread_join(t1,NULL)!= 0) printf ("error in join\n");
   if (pthread_join(t2,NULL)!= 0) printf ("error in join\n");
   exit (0);
}
```

Monitor de espera

Para realizar una sincronización de espera sobre un objeto determinado es preferible crear un monitor

- con operaciones que encapsulan el uso de las primitivas de sincronización
- con una interfaz separada del cuerpo

Esto evita errores en el manejo del mutex y la variable condicional, que podrían bloquear la aplicación.

A continuación se muestra un ejemplo con un buffer implementado con una cola de tamaño limitado y las siguientes operaciones:

- crear, insertar, extraer

Ejemplo con monitor de espera: tipo elemento

```
//////////////////////////////////////////
// File message_object.h: Object Prototype

#ifndef BUFDATA

#define BUFDATA

typedef struct {
  int finish;
  int number;
  char name[15];
} buf_data_t;

#endif

// Nota el símbolo BUFDATA y las directivas ifndef y define
// se usan para asegurar que no se repite la definición del
// tipo de datos
```

Ejemplo con monitor de espera: cola, cabeceras

```
//////////////////////////////////////
// File buffer_queue.h: Object Prototype

#include "message_object.h"
#define MAX_DATA 100
#define ERROR    1

int queue_insert  (const buf_data_t *message);
      // returns 0 if OK, ERROR if full

int queue_extract (buf_data_t *message);
      // returns 0 if OK, ERROR if empty

int queue_isfull  (void);
      // returns 1 if full, 0 if not full

int queue_isempty (void);
      // returns 1 if empty, 0 if not empty
```


Ejemplo con monitor de espera: cola, cuerpo

```
//////////////////////////////////////////////
// File buffer_queue.c: Object Implementation

#include "buffer_queue.h"

int queue_head=0;
int queue_tail=MAX_DATA-1;
buf_data_t queue_item[MAX_DATA];

int queue_isfull(void)
{
   return (queue_tail+2)% MAX_DATA == queue_head;
}

int queue_isempty(void)
{
   return (queue_tail+1)% MAX_DATA == queue_head;
}
```

Ejemplo con monitor de espera: cola, cuerpo

```
int  queue_insert (const buf_data_t *message)
{
     if (queue_isfull()) {
          return ERROR;
     } else {
         queue_tail=(queue_tail+1)% MAX_DATA;
         queue_item[queue_tail]=*message;
         return 0;
     }
}
```


Ejemplo con monitor de espera: cola, cuerpo

```
int queue_extract (buf_data_t *message)
{
     if (queue_isempty()) {
         return ERROR;
     } else {
         *message=queue_item[queue_head];
         queue_head=(queue_head+1)% MAX_DATA;
         return 0;
     }
}
```

Ejemplo con monitor: sincronización, cabecera

```
//////////////////////////////////////////
// File one_way_buffer.h: Object Prototype

#include "message_object.h"
#define CREATION_ERROR  -3
#define SYNCH_ERROR     -2
#define BUFFER_FULL     -1

int buf_create (void);
     // buf_create must finish before the object is used
     // buf_create returns 0 if OK, or CREATION_ERROR
     // if an error occurs
int buf_insert (const buf_data_t *message);
     // buf_insert returns 0 if OK or BUFFER_FULL or
     // SYNCH_ERROR
int buf_extract (buf_data_t *message);
     // buf_extract suspends the calling task until data
     // is available
     // buf_extract returns 0 if OK or SYNCH_ERROR
```


Ejemplo con monitor: sincronización, cuerpo

```
//////////////////////////////////////////////
// File one_way_buffer.c: Object Implementation
#include <pthread.h>
#include "buffer_queue.h"
#include "one_way_buffer.h"
pthread_mutex_t the_mutex;
pthread_cond_t the_condition;

int buf_create (void){
  // Initialize the mutex
  if (pthread_mutex_init(&the_mutex,NULL)!=0) {
    return CREATION_ERROR;
  }
  // Initialize the condition variable with default attributes
  if (pthread_cond_init(&the_condition,NULL)!=0) {
    return CREATION_ERROR;
  }
  return (0);
}
```

Ejemplo con monitor: sincronización, cuerpo (cont.)

```
int  buf_insert (const buf_data_t *message) {
  int ret_value;

  if (pthread_mutex_lock(&the_mutex)!=0) {
    return SYNCH_ERROR;
  }
  if (queue_isfull()) {
    ret_value=BUFFER_FULL;
  } else {
    ret_value=0;
    if (queue_insert(message)!=0) return BUFFER_FULL;
    if (pthread_cond_signal(&the_condition)!=0)
       return SYNCH_ERROR;
  }
  if (pthread_mutex_unlock(&the_mutex)!=0) {
    return SYNCH_ERROR;
  }
  return (ret_value);
}
```


Ejemplo con monitor: sincronización, cuerpo (cont.)

```
int buf_extract (buf_data_t *message)
{
  if (pthread_mutex_lock(&the_mutex)!=0) return SYNCH_ERROR;
  while (queue_isempty()) {
    if (pthread_cond_wait(&the_condition,&the_mutex)!=0) {
      return SYNCH_ERROR;
    }
  }
  if (queue_extract(message)!=0) return ERROR;
  if (pthread_mutex_unlock(&the_mutex)!=0) {
    return SYNCH_ERROR;
  }
  return 0;
}
```

Ejemplo con monitor: programa principal

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "one_way_buffer.h"
#include "load.h"

static int error=-1;
```


Ejemplo con monitor: programa principal (cont.)

```
// Productor
void * producer (void *arg)
{
  buf_data_t msg;  int i;
  // Produce 20 mensajes y consume un tiempo aleatorio
  for (i=0;i<20;i++) {
    eat(((float)rand())/((float)RAND_MAX)*1.0);
    printf("producer sends i=%d\n",i);
    msg.finish=0;
    msg.number=i;
    strcpy(msg.name,"I am a message\0");
    if (buf_insert(&msg)!=0) pthread_exit(&error);
  }
  // Dos mensajes de finalizacion, uno por consumidor
  msg.finish=1; msg.number=0; msg.name[0]=0;
  if (buf_insert(&msg)!=0) pthread_exit(&error);
  if (buf_insert(&msg)!=0) pthread_exit(&error);
  pthread_exit (NULL);
}
```

Ejemplo con monitor: programa principal (cont.)

```
// Consumidor
void * consumer1 (void *arg)
{
  buf_data_t msg;

  do { // Recibe mensajes y los muestra en pantalla
    if (buf_extract(&msg)!=0) pthread_exit(&error);
    printf("consumer 1 receives i=%d, name=%s\n",msg.number,
           msg.name);
    eat(((float)rand())/((float)RAND_MAX)*1.0);
  } while (msg.finish == 0);
  pthread_exit (NULL);
}

// El consumidor 2 es igual
void * consumer2 (void *arg)
{
  ...
}
```


Ejemplo con monitor: programa principal (cont.)

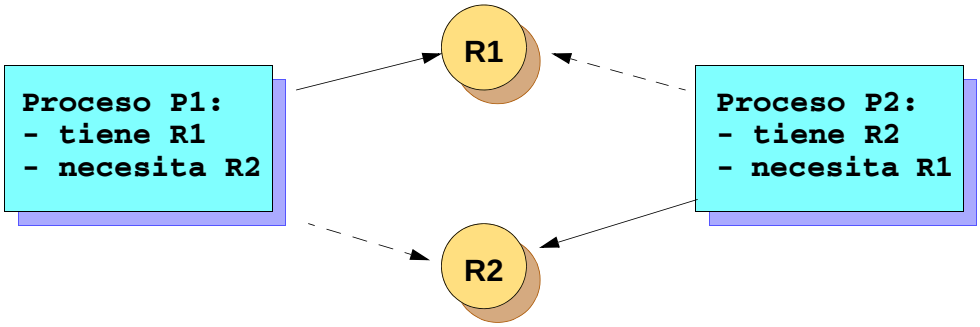
```
int main()
{
  pthread_t t1,t2,t3;
  pthread_attr_t th_attr;
  void * st;

  adjust();
  if (buf_create()!=0) {
    printf("error al crear buffer\n");
    exit (1);
  }
  if (pthread_attr_init(&th_attr)!=0) {
    printf("error al crear atributos\n");
    exit (1);
  }
  if (pthread_create (&t1,&th_attr,producer,NULL) != 0) {
    printf("error en pthread_create\n");
    exit(1);
  }
```

Ejemplo con monitor: programa principal (cont.)

```
  if (pthread_create (&t2,&th_attr,consumer1,NULL) != 0) {
    printf("error en pthread_create\n");
    exit(1);
  }
  if (pthread_create (&t3,&th_attr,consumer2,NULL) != 0) {
    printf("error en pthread_create\n");
    exit(1);
  }
  if (pthread_join(t1,&st) != 0) printf ("error in join\n");
  if (pthread_join(t2,&st) != 0) printf ("error in join\n");
  if (pthread_join(t3,&st) != 0) printf ("error in join\n");
  exit (0);
}
```


5. Interbloqueos (deadlocks)

Ejemplo de una situación de interbloqueo :



Gestión de interbloqueos:

- evitación
- detección y recuperación

Evitar Interbloqueos

Cuando son debidos al uso de mutexes:

- uso de un protocolo de techo o protección de prioridad (lo veremos más adelante)
- toma de recursos siempre en el mismo orden
  - el orden puede ser arbitrario

Cuando son debidos a la sincronización de espera o combinaciones de otras

- Existen técnicas matemáticas para verificar si hay posibilidad de interbloqueos o no

6. Generación, bloqueo y aceptación de señales

Señal:

- Mecanismo por el que un proceso puede ser notificado de, o afectado por, un evento que se produce en el sistema
  - excepciones detectadas por hardware
  - expiración de una alarma o temporizador
  - finalización de una operación de I/O asíncrona
  - llegada de un mensaje
  - invocación de la función `kill`() y similares
  - actividad del terminal, etc.

Número de señal:

- Entero positivo que identifica una señal. Existen también constantes predefinidas para dar nombre a las señales

Señales existentes:

Señales no "fiables"	
Nombre de Señal	Descripción
SIGABRT	Terminación anormal, p.e. con *abort*()
SIGALRM	Timeout, p.e. con *alarm*()
SIGFPE	Operación aritmética errónea, p.e. división por cero u overflow
SIGHUP	Desconexión del terminal controlador, o terminación del proceso controlador
SIGILL	Instrucción hardware inválida
SIGINT	Señal de atención interactiva (ctrl-C)
SIGKILL	Señal de terminación (no puede ser "cazada" ni ignorada)
SIGPIPE	Escritura en una *tubería* sin lectores
SIGQUIT	Señal de terminación interactiva

Señales existentes (cont.):

Señales no "fiables" (cont.)	
Nombre de Señal	Descripción
SIGSEGV	Referencia a memoria inválida
SIGTERM	Señal de terminación
SIGUSR1	Reservada para la aplicación
SIGUSR2	Reservada para la aplicación
SIGBUS	Acceso a una porción indefinida de un objeto de memoria (opcional)

Asimismo existen señales de control de sesiones, que sólo se requieren si se soporta la opción de control de sesiones

Todas las señales anteriores son "no fiables", pues se pueden perder si ya hay una señal del mismo número pendiente

Señales de tiempo real

Si se soporta la opción de señales de tiempo real, existen señales con números comprendidos entre `SIGRTMIN` y `SIGRTMAX`, que son señales de tiempo real, o "fiables"

Las señales de tiempo real:

- no se pierden (se encolan)
- se aceptan en orden de prioridad (la prioridad es el número de señal)
- tienen un campo adicional de información
- hay como mínimo 8

Generación, entrega y aceptación

Una señal se *genera*:

- cuando el evento que causa la señal ocurre

Cada thread tiene una máscara de señales, que define las señales bloqueadas

- la máscara se hereda del thread padre, y se puede modificar

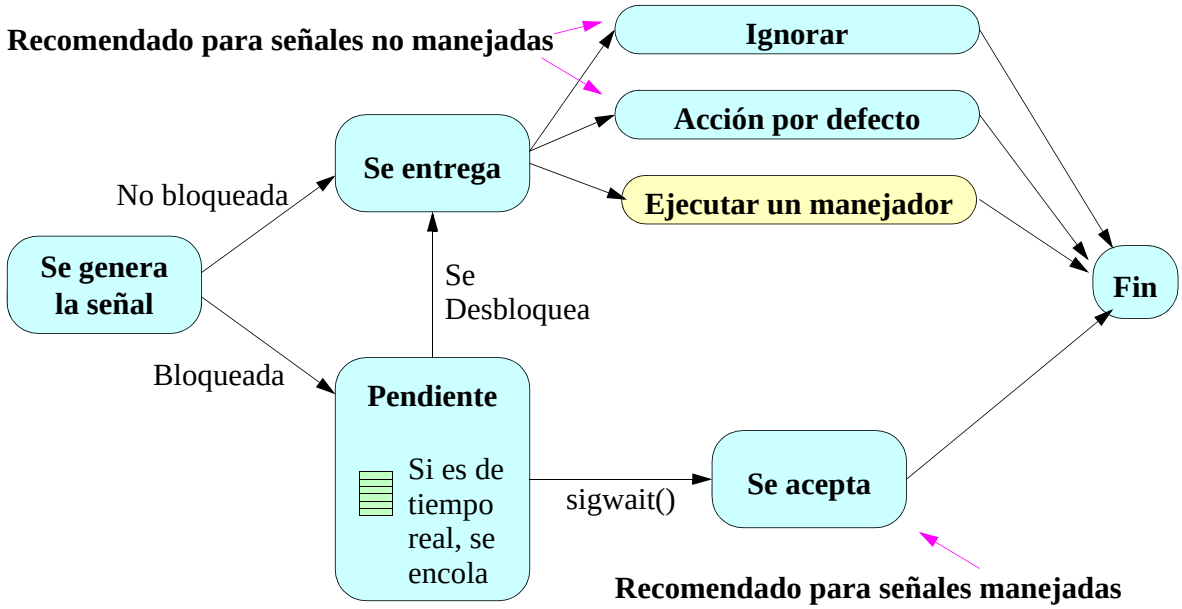
Una señal no bloqueada se *entrega*:

- cuando la señal causa al proceso la acción asociada: ignorar, terminar el proceso o ejecutar un manejador

Una señal bloqueada se *acepta*:

- cuando se selecciona y devuelve por una función *sigwait*()

Diagrama de funcionamiento



Señales pendientes

Señal *pendiente:*

- estado entre su generación y su entrega o aceptación
- observable cuando la señal está ***bloqueada*** (enmascarada)
- si ocurre un nuevo evento de una señal pendiente:
  - si es una señal “no fiable”, el evento puede perderse
  - si es una señal “de tiempo real”, y la opción `SA_SIGINFO` ha sido especificada, la señal se encola

Acciones asociadas a una señal

Hay tres tipos de acciones que se pueden asociar a una señal:

- **`SIG_DFL`**: acción por defecto (generalmente, terminar el proceso)
- **`SIG_IGN`**: ignorar la señal
- ***puntero a función***: capturar (o "cazar") la señal, ejecutando un manejador de señal
  - al entregarse la señal, se ejecuta el manejador indicado
  - una vez finalizado el manejador, se continúa el proceso en el lugar interrumpido

Manejadores de señales

Deben corresponder a los siguientes prototipos:

- si la opción **`SA_SIGINFO`** no está especificada para esa señal:

```
void func (int signo)
```

- si la opción **`SA_SIGINFO`** está especificada la especificación es distinta: tiene tres parámetros

Los manejadores sólo pueden llamar a funciones "seguras"

Notificación de eventos

Algunos servicios pueden notificar eventos mediante señales de tiempo real: finalización de I/O asíncrona, expiración de un temporizador, llegada de un mensaje, la función `sigqueue`()

En todos estos servicios la notificación se especifica con la estructura `sigevent`, que tiene entre otros los siguientes campos:

- Tipo de notificación: `int sigev_notify`
  - `SIGEV_NONE`: no hay notificación
  - `SIGEV_SIGNAL`: se genera una señal (de número `sigev_signo`)
- Número de señal: `int sigev_signo`

Señales en procesos multi-thread

Una señal se puede generar para un proceso o un thread

- si el evento es a causa de un thread, se envía a ese thread (p.e. una excepción hardware)
- si el evento es asíncrono al proceso, se envía al proceso (p.e. finalización de una operación de I/O asíncrona)
- las señales generadas en asociación a un *pid* se envían al proceso; en asociación a un *tid*, se envían al thread

Un señal generada para un thread

- si no está bloqueada: se entrega
- si está bloqueada y la acción no es ignorar: se queda pendiente hasta que se acepta con *sigwait* o se desbloquea
- si está bloqueada y la acción es ignorar: no especificado

Señales en procesos multi-thread (cont.)

Una señal generada para un proceso:

- **si la acción no es ignorar, se envía a un único thread que esté esperando en un *sigwait*, o a un thread que no tenga la señal bloqueada**
- **si no se entrega, la señal queda pendiente hasta que un thread llama a *sigwait*, un thread desbloquea la señal, o la acción asociada se pone en ignorar**

Uso recomendado de señales en procesos multithread

El thread principal enmascara todas las señales que se vayan a usar (salvo las destinadas a finalizar el proceso).

Los demás threads heredan esta máscara.

Las señales se aceptan con funciones *sigwait*().

No se usan manejadores de señal; en su lugar, se crean threads que manejan las señales deseadas.

Ningún thread debe desenmascarar ninguna de las señales que se vayan a usar.

Funciones para configurar señales

Manipular conjuntos de señales:

```
#include <signal.h>
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigdelset (sigset_t *set, int signo);
int sigismember (const sigset_t *set,
                 int signo);
```

Estas funciones no afectan de forma directa al comportamiento del sistema

Funciones para configurar señales (cont.)

Examinar y cambiar la acción asociada a una señal:

```
int sigaction
      (int sig,
       const struct sigaction *act,
       struct sigaction *oact);
```

- si `act` es NULL, la acción actual no se cambia
- si `oact` no es NULL, la acción actual se devuelve en `*oact`

Funciones para configurar señales (cont.)

- `sigaction` tiene entre otros los siguientes miembros:
  - `void (*) (int) sa_handler`: vale `SIG_DFL`, `SIG_IGN`, o puntero a función manejadora de señal
  - `sigset_t sa_mask`: conjunto de señales a ser bloqueadas adicionalmente mientras se ejecuta el manejador de señal (además, se añade automáticamente la señal manejada)
  - `int sa_flags`: opciones; valores:
    - `SA_SIGINFO`: la señal se encola y lleva información asociada; el manejador de señal es de tres argumentos
    - `SA_NOCLDSTOP`: no generar `SIGCHLD` cuando un hijo se detiene

Funciones para configurar señales (cont.)

Examinar y cambiar señales bloqueadas:

```
int pthread_sigmask (int how,
     const sigset_t *set, sigset_t *oset);
```

- `how` indica la forma de operar:
  - `SIG_BLOCK`: señales bloqueadas = anteriores ∪ nuevas
  - `SIG_UNBLOCK`: señales bloq. = anteriores ∩ nuevas)
  - `SIG_SETMASK`: señales bloqueadas = nuevas
- `set`: señales nuevas; si vale NULL, no se cambian
- `oset`: si no es NULL, devuelve señales bloq. anteriores

Funciones para envío y aceptación de señales

Enviar una señal a un proceso:

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig);
int sigqueue (pid_t pid, int signo,
              const union sigval value);
```

Enviar una señal a un thread:

```
pthread_kill(pthread_t thread, int sig);
```


Envío y aceptación de señales (cont.)

Aceptar una señal bloqueada:

```
int sigwait (const sigset_t *set, int *sig);

int sigwaitinfo
     (const sigset_t *set,
      siginfo_t *info);

int sigtimedwait
     (const sigset_t *set,
      siginfo_t *info,
      const struct timespec *timeout);
```

Ejemplo del uso de señales: thread que acepta una señal

```
// El thread acepta las señales SIGINT o SIGUSR1. Cuando recibe
// una señal escribe un mensaje en pantalla. A la cuarta señal
// recibida termina el proceso.

#include <signal.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

static int error_status=-1;

void * sigwaiter (void * arg)
{
  sigset_t set;
  int sig;
  int counter = 0;
  siginfo_t info;
  // La mascara de señales se hereda del padre
  // Debe tener SIGINT y SIGUSR1 enmascarada
```


Ejemplo del uso de señales (cont.)

```
  sigemptyset(&set);
  sigaddset(&set,SIGINT);
  sigaddset(&set,SIGUSR1);

  while (1) {
    if ((sig=sigwaitinfo(&set, &info)) == -1) {
      //sigwait error
      printf("Error en sigwait:%d \n",sig);
      pthread_exit ((void*)&error_status);
    }
    // Maneja la señal de terminación
    counter ++;
    if (counter < 4) {
      if (info.si_signo == SIGINT) {
        printf ("SIGINT %d recibida\n", counter);
      } else { // es SIGUSR1
        printf ("SIGUSR1 %d recibida\n", counter);
      }
    } else {
```

Ejemplo del uso de señales (cont.)

```
      printf ("Señal final recibida\n");
      exit(0); // mata el proceso completo
    }
  }
}

// Thread principal

int main ()
{
  sigset_t set;
  pthread_t th;
```


Ejemplo del uso de señales (cont.)

```
  // Coloca la mascara para él y para el hijo
  sigemptyset(&set);
  sigaddset(&set,SIGINT);
  sigaddset(&set,SIGUSR1);
  if (pthread_sigmask(SIG_BLOCK, &set, NULL) !=0) {
    printf ("Error al poner la máscara de señales\n"); exit (1);
  }

  if (pthread_create(&th,NULL,sigwaiter,NULL) != 0) {
    printf ("Error de creacion del thread\n"); exit (1);
  }

  while (1) {
    printf ("Main thread executing \n");
    sleep(1);
  }
}
```