Master en Computación

# Programación Concurrente

## Bloque II: Programación concurrente en POSIX


Tema 1. Introducción al estándar POSIX
Tema 2. Sistema Operativo MaRTE OS
Tema 3. Gestión de Threads
Tema 4. Gestión del Tiempo
Tema 5. Planificación de Threads

**Tema 6. Sincronización**

Tema 7. Señales
Tema 8. Temporizadores y Relojes de Tiempo de Ejecución



© M. Aldea, M. González
oct-13




# Tema 6. Sincronización


6.1. Mecanismos de sincronización POSIX
6.2. Semáforos Contadores
6.3. Mutexes
6.4. Variables Condicionales
6.5. Implementación de secciones críticas y mutexes en MaRTE OS



© M. Aldea, M. González
oct-13




# 6.1 Mecanismos de sincronización POSIX

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 intercambio de mensajes
- usar recursos compartidos (datos o dispositivos) de manera mutuamente exclusiva
  - para asegurar su *congruencia* ante accesos concurrentes
  - el acceso al recurso puede ser condicional, de forma que sólo se pueda acceder a él si se encuentra en el estado apropiado


© M. Aldea, M. González
oct-13

**Mecanismos de sincronización proporcionados por el POSIX:**

- **Sincronización por intercambio de mensajes:**
  - **señales**
  - **semáforos contadores**
  - **colas de mensajes (no vistas en este curso)**
- **Exclusión mutua:**
  - **semáforos contadores**
  - **mutexes**
  - **variables condicionales**

**El POSIX no proporciona monitores ni regiones críticas**

- **aunque pueden construirse utilizando semáforos o mutexes y variables condicionales**





# Variables atómicas compartidas

**Un mecanismo primitivo de sincronización consiste en la utilización de variables atómicas compartidas**

- **variable atómica: puede ser leída o escrita con una sola instrucción del procesador**
  - **su tamaño máximo está limitado por la anchura del bus (16/32/64 bits)**
- **la congruencia ante accesos concurrentes está asegurada**

**Una variable compartida entre threads, manejadores de señal o manejadores de interrupción debe ser marcada como “volátil”**

- **indica al compilador que no debe optimizar basándose en su valor puesto que puede cambiar “inesperadamente”**
- **en C se utiliza la palabra reservada `volatile`:**

```
volatile int flag;
```





# 6.2 Semáforos Contadores

**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 señalización y espera (paso de mensajes)**

- **tanto entre procesos como entre threads**

**Operaciones que se aplican al semáforo:**

- **esperar (*wait*): si el valor es 0, el proceso o thread se añade a una cola; si el valor es mayor que 0, se decrementa**
- **señalizar (*post*): 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**

# Inicializar/destruir un semáforo sin nombre

- **Inicializar/destruir 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` es 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**

- **Destruir un semáforo sin nombre:**

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





# Abrir/cerrar un semáforo con nombre

- **Abrir (e 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**

- **Cerrar y borrar un semáforo con nombre:**

```
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```





# Esperar en un semáforo

- **Espera incondicional**

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

  - **si `valor>0` entonces `valor:=valor-1`**
  - **si `valor=0` el thread se bloquea y encola en la cola del semáforo**

- **Operación no bloqueante**

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

  - **si `valor>0` entonces `valor:=valor-1`**
  - **si `valor=0` retorna `-1` inmediatamente y `errno` vale `EAGAIN`**

# Esperar en un semáforo (cont.)

- Espera con tiempo límite

```
int sem_timedwait(sem_t *sem,
      const struct timespec *abs_timeout);
```

  - si `valor>0` entonces `valor:=valor-1`
  - si `valor=0` el thread se bloquea y encola en la cola del semáforo
  - si se alcanza el tiempo límite la función retorna `-1` y `errno` vale `ETIMEDOUT`
  - el tiempo límite está basado en el reloj del sistema o en el `CLOCK_REALTIME` (en el caso de que el S.O. lo soporte)





# Señalizar un semáforo

- Señalizar un semáforo:

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

  - si hay algún thread esperando en la cola: se activa el de mayor prioridad
  - si no, `valor:=valor+1`

- Leer el valor de un semáforo:

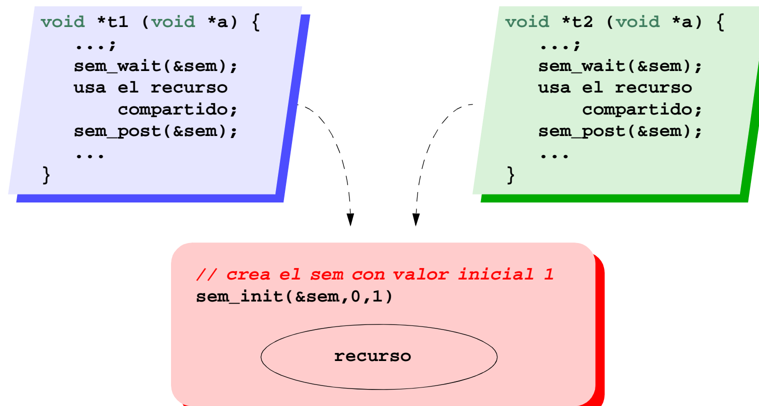
```
int sem_getvalue(sem_t *sem, int *sval);
```

  - retorna en `sval` el valor que tenía el semáforo en el momento de la llamada a la función

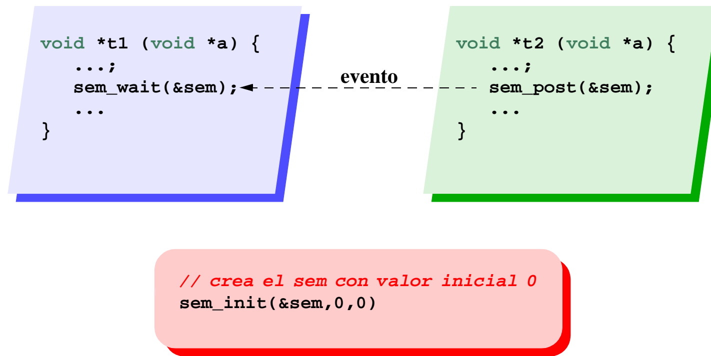




# Uso de semáforos para exclusión mutua



# Uso de semáforos para señalización y espera




```
// crea el sem con valor inicial 0
sem_init(&sem,0,0)
```


© M. Aldea, M. González
oct-13




# Ejemplo: Señalización con semáforos

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <misc/error_checks.h>

sem_t sem;

// Thread que espera el semáforo
void * waiter(void *arg)
{
  while (1) {
    printf("  -Waiter antes de esperar en el semáforo \n");
    CHKE( sem_wait(&sem) );
    printf("  -Waiter después de esperar \n");
  }
  return NULL;
}
```


© M. Aldea, M. González
oct-13




```
// Thread que señaliza el semáforo
void * signaler(void *arg)
{
  printf ("Signaler antes de señalizar 3 veces \n");
  CHKE( sem_post(&sem) );
  CHKE( sem_post(&sem) );
  CHKE( sem_post(&sem) );
  sleep(10);  // deja ejecutar al thread señalizado

  while (1) {
    printf ("Signaler antes de señalizar \n");
    CHKE( sem_post(&sem) );
    sleep(10); // deja ejecutar al thread señalizado
  }
  return NULL;
}
```


© M. Aldea, M. González
oct-13

```
int main()
{
  pthread_t t1,t2;
  int ret;

  CHKE( sem_init(&sem, 0, 1) );

  // Crea los threads
  CHK( pthread_create(&t1, NULL, waiter, NULL) );
  CHK( pthread_create(&t2, NULL, signaler, NULL) );

  // deja ejecutar un rato y termina
  sleep(40);
  exit(0);
}
```





# 6.3 Mutexes

**Objeto de sincronización parecido a los semáforos**

- **por el que múltiples threads/procesos acceden de forma *mut*uamente *ex*clusiva a un recurso**
- **pero con un nuevo concepto: cada mutex tiene un *propietario***

**Operaciones:**

- **tomar o bloquear (*lock*): si el mutex está libre, se toma y el thread se convierte en propietario; si no se suspende y se añade a una cola**
- **liberar (*unlock*): si hay threads esperando en la cola, el de mayor prioridad toma el mutex y se convierte en su nuevo propietario; si no, el mutex queda libre; sólo el propietario del mutex puede invocar esta operación**





# Protocolos de sincronización

**Patologías relacionadas con la sincronización:**

- **Interbloqueos**
- **Falta de vivacidad: Inversión de prioridad no acotada (ver transparencia siguiente)**

**Los mutexes permiten los protocolos:**

- ***Herencia de Prioridad***
- ***Protección por Prioridad* (o “Techo de Prioridad Inmediato”)**

**Ambos *acotan* la inversión de prioridad**

- **a la duración de una o varias secciones críticas**

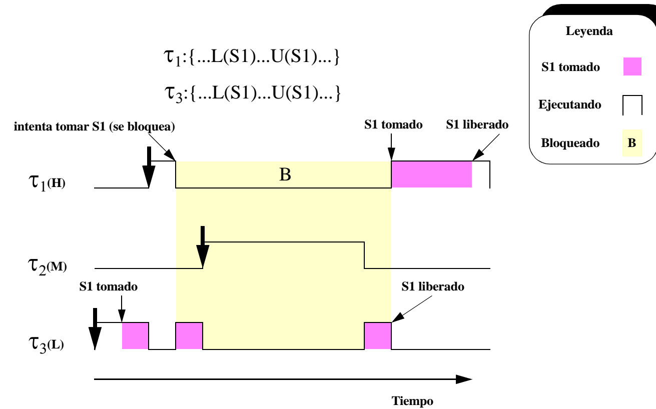
**Protección por Prioridad evita los interbloqueos**

- **siempre que las tareas no se suspendan dentro de las regiones críticas**

# Inversión de prioridad no acotada

Ocurre cuando no se usa ningún protocolo de sincronización

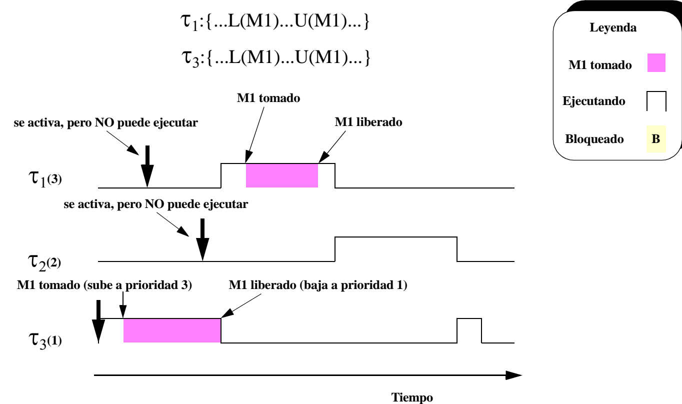
- usando semáforos o mutexes sin protocolo








# Evitando la inversión de prioridad con el protocolo de Protección por Prioridad







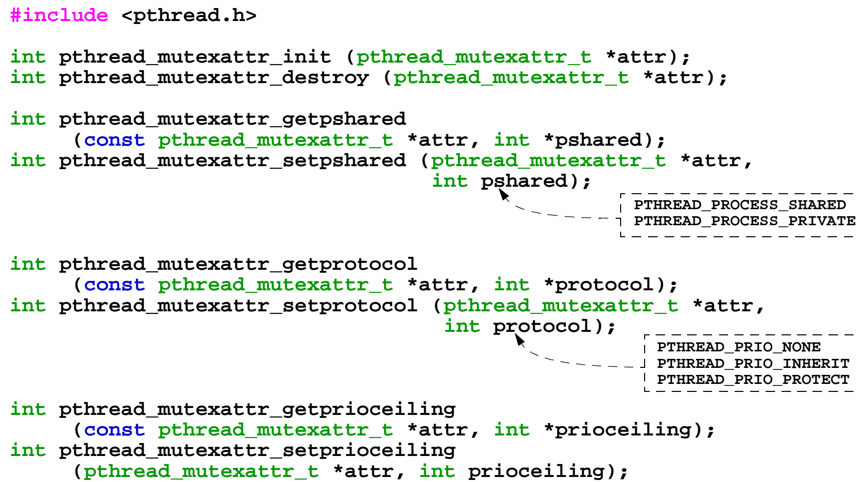

# Atributos de inicialización

- ***pshared***: indica si es compartido o no entre procesos:
  - `PTHREAD_PROCESS_SHARED`
  - `PTHREAD_PROCESS_PRIVATE`
- ***protocol***: protocolo utilizado
  - `PTHREAD_PRIO_NONE`: no hay herencia de prioridad
  - `PTHREAD_PRIO_INHERIT`: herencia de prioridad
  - `PTHREAD_PRIO_PROTECT`: protección de prioridad
- ***prioceiling***: techo de prioridad (valor entero)

Estos atributos se almacenan en un objeto de atributos

- `pthread_mutexattr_t`
- Los valores por defecto en el objeto de atributos son definidos por la implementación
  - MaRTE OS: `PTHREAD_PRIO_NONE` y techo 0

# 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);
                                      PTHREAD_PROCESS_SHARED
                                      PTHREAD_PROCESS_PRIVATE

int pthread_mutexattr_getprotocol
      (const pthread_mutexattr_t *attr, int *protocol);
int pthread_mutexattr_setprotocol (pthread_mutexattr_t *attr,
                                   int protocol);
                                      PTHREAD_PRIO_NONE
                                      PTHREAD_PRIO_INHERIT
                                      PTHREAD_PRIO_PROTECT

int pthread_mutexattr_getprioceiling
      (const pthread_mutexattr_t *attr, int *prioceiling);
int pthread_mutexattr_setprioceiling
      (pthread_mutexattr_t *attr, int prioceiling);
```





# Inicializar y Destruir un Mutex

- Inicializar un Mutex:

```
#include <pthread.h>

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

- Destruir un mutex:

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```





# Tomar un Mutex

- Tomar un mutex o suspenderse si no está libre:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

  - Retorna el error `EINVAL` si el mutex es de protocolo `PTHREAD_PRIO_PROTECT` el thread tiene prioridad mayor que el techo

- Tratar de tomar un mutex, sin suspenderse:

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

  - Retorna el error `EBUSY` si el mutex está tomado

# Tomar un Mutex (cont.)

- Tratar de tomar un mutex, con tiempo límite (`CLOCK_REALTIME`):

```
#include <pthread.h>

int pthread_mutex_timedlock
      (pthread_mutex_t *mutex,
       const struct timespec *abstime);
```

  - Retorna el error `ETIMEDOUT` si se alcanza el tiempo límite
  - el tiempo límite está basado en el reloj del sistema o en el `CLOCK_REALTIME` (en el caso de que el S.O. lo soporte)





# Liberar un mutex

```
#include <pthread.h>

int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

  - Si hay threads esperando en la cola, el de mayor prioridad se convierte en el nuevo propietario
  - Si no, el mutex queda libre
  - Retorna el error EPERM si el thread llamante no es el propietario del mutex





# Cambio Dinámico del Techo de Prioridad

```
#include <pthread.h>

int pthread_mutex_getprioceiling
     (const pthread_mutex_t *mutex,
      int *prioceiling);


int pthread_mutex_setprioceiling
     (pthread_mutex_t *mutex,
      int prioceiling,
      int *old_ceiling);
```

  - Toma el mutex antes de cambiar el techo

# Ejemplo con un Mutex

```
#include <pthread.h>
#include <stdio.h>
#include <misc/error_checks.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++) {
      CHK( pthread_mutex_lock(&data.mutex) );
      data.a++;
      data.b++;
      data.c++;
      CHK( pthread_mutex_unlock(&data.mutex) );
   }
   return NULL;
}
```





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



// Programa principal: crea el mutex y los threads
int main()
{
   pthread_mutexattr_t mutexattr;
   pthread_t t1,t2;
   struct sched_param sch_param;
   pthread_attr_t attr;

   data.a = 0; data.b =0; data.c = 0; data.i = 0;
```





```
   // crea el mutex
   CHK( pthread_mutexattr_init(&mutexattr) );
   CHK( pthread_mutexattr_setprotocol(&mutexattr,
                                      PTHREAD_PRIO_PROTECT) );
   CHK( pthread_mutexattr_setprioceiling(&mutexattr,
                               sched_get_priority_min(SCHED_RR) ) );
   CHK( pthread_mutex_init(&data.mutex, &mutexattr) );

      // Prepara atributos para planificación round-robin
   CHK( pthread_attr_init(&attr) );
   CHK( pthread_attr_setinheritsched(&attr,
                                     PTHREAD_EXPLICIT_SCHED) );
   CHK( pthread_attr_setschedpolicy(&attr, SCHED_RR) );
   sch_param.sched_priority = sched_get_priority_min(SCHED_RR);
   CHK( pthread_attr_setschedparam(&attr, &sch_param) );

   // crea los threads
   CHK( pthread_create(&t1,&attr,incrementer,NULL) );

   CHK( pthread_create(&t2,&attr,reporter,NULL) );

   // espera a que acaben
   CHK( pthread_join(t1,NULL) );
   CHK( pthread_join(t2,NULL) );
   return 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, int ceiling);
              // sh_create must finish before the object is used

int sh_modify (const shared_t *new_value);

int sh_read (shared_t *current_value);
```





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

#include <sched.h>
#include <pthread.h>
#include <misc/error_checks.h>
#include "shared_object.h"

shared_t object;
pthread_mutex_t the_mutex;
pthread_mutexattr_t mutexattr;

int sh_create (const shared_t *initial_value, int ceiling)
{
   CHK( pthread_mutexattr_init(&mutexattr) );
   CHK( pthread_mutexattr_setprotocol(&mutexattr,
                                    PTHREAD_PRIO_PROTECT) );
   CHK( pthread_mutexattr_setprioceiling(&mutexattr, ceiling) );
   CHK( pthread_mutex_init(&the_mutex,&mutexattr) );
   object=*initial_value;
   return OK;
}
```

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



int sh_read (shared_t *current_value)
{
    CHK( pthread_mutex_lock(&the_mutex) );
    *current_value=object;
    CHK( pthread_mutex_unlock(&the_mutex) );
    return OK;
}
```


© M. Aldea, M. González
oct-13




```
////////////////////////////
// programa principal

#include <pthread.h>
#include <stdio.h>
#include <misc/error_checks.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);
    data.a++;
    data.b++;
    data.c++;
    data.i=i;
    if (sh_modify(&data)!=OK) pthread_exit(&error);
  }
  return NULL;
}
```


© M. Aldea, M. González
oct-13




```
// Este thread muestra el valor de a, b y c
// hasta que i vale 20
void * reporter (void *arg)
{
  shared_t data;

  do {
    if (sh_read(&data)!=OK) pthread_exit(&error);
    printf("a=%d, b=%d, c=%d\n", data.a, data.b, data.c);
  } while (data.i<20);
  return 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;
  struct sched_param sch_param;
  void * st;
  shared_t data;
```


© M. Aldea, M. González
oct-13

```
   adjust();

   data.a = 0; data.b =0; data.c = 0; data.i = 0;
   if (sh_create(&data, sched_get_priority_min(SCHED_RR))!=0) {
     printf("error en sh_create \n"); exit(1);
   }

   // Prepara atributos para planificación round-robin
   CHK( pthread_attr_init(&attr) );
   CHK( pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED) );
   CHK( pthread_attr_setschedpolicy (&attr, SCHED_RR) );
   sch_param.sched_priority = sched_get_priority_min(SCHED_RR);
   CHK( pthread_attr_setschedparam (&attr, &sch_param) );

   // Crea dos threads
   CHK( pthread_create (&t1,&attr,incrementer,NULL) );
   CHK( pthread_create (&t2,&attr,reporter,NULL) );

   // Espera la finalización de los dos threads
   CHK( pthread_join(t1,&st) );
   CHK( pthread_join(t2,&st) );
   return (0);
}
```





# 6.4 Variables Condicionales

Permite a un thread/proceso suspenderse hasta que otro lo reactiva y se cumple un predicado lógico

- permiten implementar la sincronización condicional

Operaciones:

- esperar (*wait*): se suspende el thread hasta que otro señaliza esa condición; entonces, generalmente se comprueba un predicado lógico, y se repite la espera si el predicado es falso
- señalizar (*signal*): se reactiva al menos el thread de mayor prioridad de los suspendidos en espera de esa condición
  - puede activarse más de un thread (podría ser inevitable en multiprocesadores y entonces el POSIX lo permite también en monoprocesadores)
- señalización global (*broadcast*): se reactivan todos los threads suspendidos en espera de esa condición





# Variables Condicionales (cont.)

La condición se representa mediante un predicado booleano

La evaluación del predicado booleano se hace protegida mediante un mutex

Pseudocódigo del thread que señaliza:

```
pthread_mutex_lock(un_mutex);
predicate=TRUE;
actualiza los datos compartidos;
pthread_cond_signal(cond);
pthread_mutex_unlock(un_mutex);
```

# Variables Condicionales (cont.)

**Pseudocódigo del thread que espera:**

```
pthread_mutex_lock(un_mutex);
while (predicate == FALSE) {
  pthread_cond_wait (cond, un_mutex);
}
utiliza los datos compartidos;
pthread_mutex_unlock(un_mutex);
```

- el predicado 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





# Variables Condicionales (cont.)

**Ejemplo: esquema productor/consumidor con CV**

**Thread Productor**

```
...
pthread_mutex_lock(mutex);

// produce el dato
guarda el nuevo dato en una
          variable compartida;
indica que hay dato disponible;

// señaliza al thread consumidor
pthread_cond_signal(cond);

pthread_mutex_unlock(mutex);
...
```

**Thread Consumidor**

```
...
pthread_mutex_lock(mutex);

// espera que haya dato disponible
while (no hay dato disponible) {
   pthread_cond_wait (cond, mutex);
}

// consume el dato
utiliza el dato compartido;
indica que NO hay dato disponible;

pthread_mutex_unlock(mutex);
...
```





# Atributos de Variables Condicionales

**Atributos definidos:**

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

# Objeto de atributos

```
#include <pthread.h>

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);
```

PTHREAD_PROCESS_SHARED
PTHREAD_PROCESS_PRIVATE


© M. Aldea, M. González
oct-13




# Inicializar y Destruir Variables Condicionales

- **Inicializar:**

```
#include <pthread.h>

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

  - **Puede pasarse `NULL` en `attr` para utilizar los atributos por defecto**

- **Destruir:**

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
```


© M. Aldea, M. González
oct-13




# Señalización de Variables Condicionales

- **Señalizar:**

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

- **Broadcast:**

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

**Ambas llamadas pueden realizarse aunque el thread no sea el propietario del mutex asociado con la variable condicional**

- **pero si se quiere lograr un comportamiento predecible el thread debe ser el propietario del mutex**


© M. Aldea, M. González
oct-13

# Espera en Variables Condicionales

- Esperar:

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

- Espera con tiempo límite:

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

  - Retorna el error `ETIMEDOUT` si se alcanza el tiempo límite
  - el tiempo límite está basado en el reloj del sistema
    - o en el reloj deseado si el S.O. soporta la opción "*Clock Selection*" (no soportada en MaRTE OS)





# Ejemplo de espera con variable condicional

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h> // para usar NULL
#include <misc/error_checks.h>
#include <misc/load.h>

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

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





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

  for (i=40; i>20; i--) {
    eat(0.02);

    // produce new data
    CHK( pthread_mutex_lock(&my_buffer->mutex) );
    printf("producer produces i=%d\n",i);

    // put data in the buffer
    my_buffer->num_produced++;
    my_buffer->data[my_buffer->num_produced-1]=i;

    // signal consumer thread
    CHK( pthread_cond_signal(&my_buffer->cond) );

    CHK( pthread_mutex_unlock(&my_buffer->mutex) );
  }
  return NULL;
}
```

```
void * consumer(void *arg)
{
  struct buffer *my_buffer = (struct buffer *)arg;
  int i,consumed;
  for (i=0; i<20; i++) {
    eat(0.01);
    // consume data
    CHK( pthread_mutex_lock(&my_buffer->mutex) );

    // wait for new data
    while (my_buffer->num_produced <= my_buffer->num_consumed) {
      printf("consumer waits for new data\n");
      pthread_cond_wait(&my_buffer->cond,&my_buffer->mutex);
    }

    // consume the new data
    my_buffer->num_consumed++;
    consumed=my_buffer->data[my_buffer->num_consumed-1];
    printf("consumer consumes i=%d\n", consumed);

    CHK( pthread_mutex_unlock(&my_buffer->mutex) );
  }
  return NULL;
}
```





```
int main() {
  pthread_attr_t attr;
  struct sched_param sch_param;
  pthread_t t1,t2;
  struct buffer my_buffer;
  adjust();
  my_buffer.num_produced = 0;
  my_buffer.num_consumed = 0;
  CHK( pthread_mutex_init(&my_buffer.mutex, NULL) );
  CHK( pthread_cond_init(&my_buffer.cond, NULL) );
  // Create attributes for round-robin threads
  CHK( pthread_attr_init (&attr) );
  CHK( pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED) );
  CHK( pthread_attr_setschedpolicy(&attr, SCHED_RR) );
  sch_param.sched_priority = sched_get_priority_min(SCHED_RR);
  CHK( pthread_attr_setschedparam(&attr, &sch_param) );
  // create threads
  CHK( pthread_create (&t1, &attr, producer, &my_buffer) );
  CHK( pthread_create (&t2, &attr, consumer, &my_buffer) );
  // wait for thread termination
  CHK( pthread_join(t1, NULL) );
  CHK( pthread_join(t2, NULL) );
  return 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 _MESSAGE_OBJECT_H_
#define _MESSAGE_OBJECT_H_

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

#endif

// Nota el símbolo _MESSAGE_OBJECT_H_ 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;
     }
}


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 <stdlib.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 <misc/error_checks.h>
#include "one_way_buffer.h"
#include <misc/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\n");
    if (buf_insert(&msg)!=0) pthread_exit(&error);
  }
  // Dos mensajes de finalización, 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);
  }

  CHK( pthread_attr_init(&th_attr) );
  CHK( pthread_create (&t1,&th_attr,producer,NULL) );
  CHK( pthread_create (&t2,&th_attr,consumer1,NULL) );
  CHK( pthread_create (&t3,&th_attr,consumer2,NULL) );

  CHK( pthread_join(t1,&st) );
  CHK( pthread_join(t2,&st) );
  CHK( pthread_join(t3,&st) );
  exit (0);
}
```

# 6.5 Implementación de secciones críticas y mutexes en MaRTE OS

**Las estructuras del SO deben estar protegidas ante accesos concurrentes desde los threads**

- las secciones críticas se logran *deshabilitando interrupciones*

```
enter_critic_section
   guarda registro de flags
   deshabilita interrupciones
```

```
leave_critic_section
   recupera registro de flags
```

**Es suficiente con deshabilitar interrupciones puesto que un thread en ejecución sólo puede ser expulsado de la CPU si:**

- crea un thread de mayor prio. o le activa al liberar un recurso
  - eso nunca ocurre en una sección crítica del SO
- un thread de mayor prioridad finaliza su suspensión
  - implica una *interrupción del timer HW* que no será atendida





# Implementación de mutexes

```
Mutex
owner: thread_t
blocked_q: threads queue
           ordered by prio.
```

```
lock (m : mutex)
   enter_critic_section
   while(1) { // exit with mutex locked
      if (m.owner != null) { // Mutex locked
         enqueue(pthread_self, m.blocked_q)
         do_scheduling // leave CPU
      } else { // Mutex free
         m.owner = pthread_self
         change self's sched params
         break // leave loop
      }
   }
   leave_critic_section
```

```
unlock (m : mutex)
   enter_critic_section
   change self's sched params
   m.owner = null
   if (m.blocked_q not empty) {
      t = dequeue_head(m.blocked_q)
      activate(t)
   }
   do_scheduling
   leave_critic_section
```