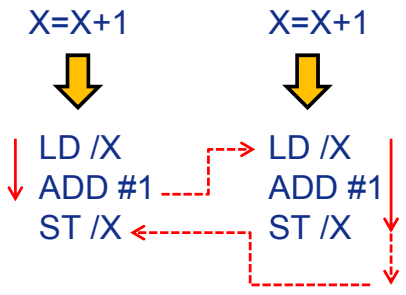
Comunicación por variables compartidas y sincronización

- Introducción
  - Sincronización con condiciones
  - Exclusión mutua
- Sincronización
  - Espera activa
  - Semáforos
  - Problemas de bloqueo
- Semáforos en POSIX 1003.1b
  - Creación, acceso y destrucción de semáforos
  - Sincronización con semáforos POSIX 1003.1b
- Memoria compartida en POSIX 1003.1b
- Sincronización para múltiples hilos en POSIX 1003.1c
  - Mutex
  - Variables de condición
- Variables compartidas y conceptos de alto nivel
  - Regiones críticas condicionales
  - Monitores
  - Objetos protegidos de ADA

Introducción

- Variables compartidas y sincronización
  - La comunicación por variables compartidas no incluye sincronización
- Sincronización
  - Sincronización con condiciones: Esperar determinado acontecimiento que sucede en otra tarea
  - Sincronización por exclusión mutua: Esperar a que los datos compartidos estén disponibles (exclusión mutua)
- Exclusión mutua:
  - Necesaria para que las operaciones sobre los datos compartidos sean atómicas (indivisibles)
  - Asegura la coherencia de los datos compartidos
  - Código en exclusión mutua: Sección crítica
  - Ejemplo:



1. A_1 =5, X=5
2. A_1=6, X=5
3. A_2=5, X=5
4. A_2=6, X=5
5. X=6
6. X=6

Espera activa ("busy wait")

- **Inconveniente básico:** El proceso que espera realiza contínuamente una actividad que no es útil.
- **Sincronización con condiciones:**

P1: (consumidor)
Hacer mientras flag = 0
Fin hacer
<realizar operaciones>

P2: (productor)
<realizar operaciones>
flag = 1

- **Exclusión mútua:** No pueden evitarse facilmente los fallos
  - **Primer caso**: si los dos detectan el flag contrario a 1, **nunca accederán a los datos (“livelock”)**:

P1:
flag1 = 1
/* El proceso 1 quiere entrar */
Mientras flag2 = 1 /* Primero espera al 2 */
Fin hacer
<sección crítica>
flag1 = 0 /* P1 ha acabado */
<resto de operaciones>

P2:
flag2 = 1
Mientras flag1 = 1
Fin hacer
<sección crítica>
flag2 = 0 /* P2 ha acabado */
<resto de operaciones>

  - **Segundo caso**: si los dos detectan el flag contrario a 0, **no hay exclusión mútua**:

P1:
Mientras flag2 = 1 /* Primero espera al 2 */
Fin hacer
flag1 = 1 /* El proceso 1 toma posesión */
<sección crítica>
flag1 = 0 /* P1 ha acabado */
<resto de operaciones>

P2:
Mientras flag1 = 1
Fin Hacer
flag2 = 1
<sección crítica>
flag2 = 0
<resto de operaciones>

Semáforos

- Concepto teórico, con diversas implementaciones
- Elementos:
  - **Valor entero no negativo**, no accesible directamente
  - Operación de **espera**, **wait**: Si el valor es cero, el proceso es suspendido hasta que otro proceso lo incremente con **signal**. Cuando el valor del semáforo es mayor que cero, es decrementado y el proceso continúa.
  - Una operación de **señalización**, **signal**: Incrementa el valor del semáforo en una unidad (y por tanto, puede desbloquear procesos que están esperando en **wait**).

→ **Importante**: Las operaciones se realizan de manera **atómica**, indivisible.

Semáforos

- Sincronización con una condición (sem_cond inicializado a 0):

P1: /* Espera la condición generada por P2 */
<.....>
wait(sem_cond)
<realizar operaciones>
<.....>

P2: /* Genera la condición que espera P1 */
<.....>
<realizar operaciones>
signal(sem_cond)
<.....>

- Exclusión mútua (sem_cond inicializado a 1):

P1:
<....>
wait(sem_excl) /* P1 espera a que P2 salga de su sección crítica */
<.sección crítica>
signal(sem_excl) /* P1 permite acceso a P2 */
<.....>

P2:
<....>
wait(sem_excl) /* P2 espera a que P1 salga de su sección crítica */
<.sección crítica>
signal(sem_excl) /* P2 permite acceso a P1 */
<.....>

Semáforos: Problemas de bloqueo

- Bloqueo mútuo (**"deadlock"**):

P1:
wait(sem1)
<....>
  wait(sem2)
  <.....>
  signal(sem2)
<....>
signal(sem1)

P2:
wait(sem2)
<...>
  wait(sem1)
  <....>
  signal(sem1)
<...>
signal(sem2)

  - P1 es expulsado antes de ejecutar el segundo **wait**, y después de ejecutar el primero.
  - P2 pondría sem2 a 0, y quedaría bloqueado en su segundo **wait**.
  - Cuando P1 continue, no podrá pasar del segundo **wait** tampoco.
  - La condición de bloqueo permanece indefinidamente.

- Inanición (**"starvation"**):
  - El algoritmo utilizado por el sistema operativo para escoger a el proceso que es desbloqueado lleva a algunos a la suspensión por **tiempo ilimitado** o con **límites indefinidos**

Semáforos POSIX (1003.1b)

- Semáforos “con nombre” (“named”):
  - Semáforos con cuenta
  - Objetos de sistema operativo, accesibles a través de su nombre por cualquier proceso con permiso adecuado

```
#include <semaphore.h>

/* Creación, acceso y destrucción */
sem_t *sem;
sem = sem_open(“/sem27”,
        O_CREAT | O_EXCL, S_IRWXU, 1);
<.....>
sem_close(sem);
sem_unlink(“/sem27”);
<....>
```

- Semáforos en memoria:
  - Semáforos con cuenta
  - Objetos creados en memoria compartida, sin nombre
  - Accesibles por procesos que comparten tal zona de memoria

```
/* Creación, acceso y destrucción */
#include <semaphore.h>

sem_t *psem;

memoria = <inicio de memoria compartida>;
psem = (sem_t *)memoria;
sem_init(psem, 1, 1);
<.....>
sem_destroy(psem);
```

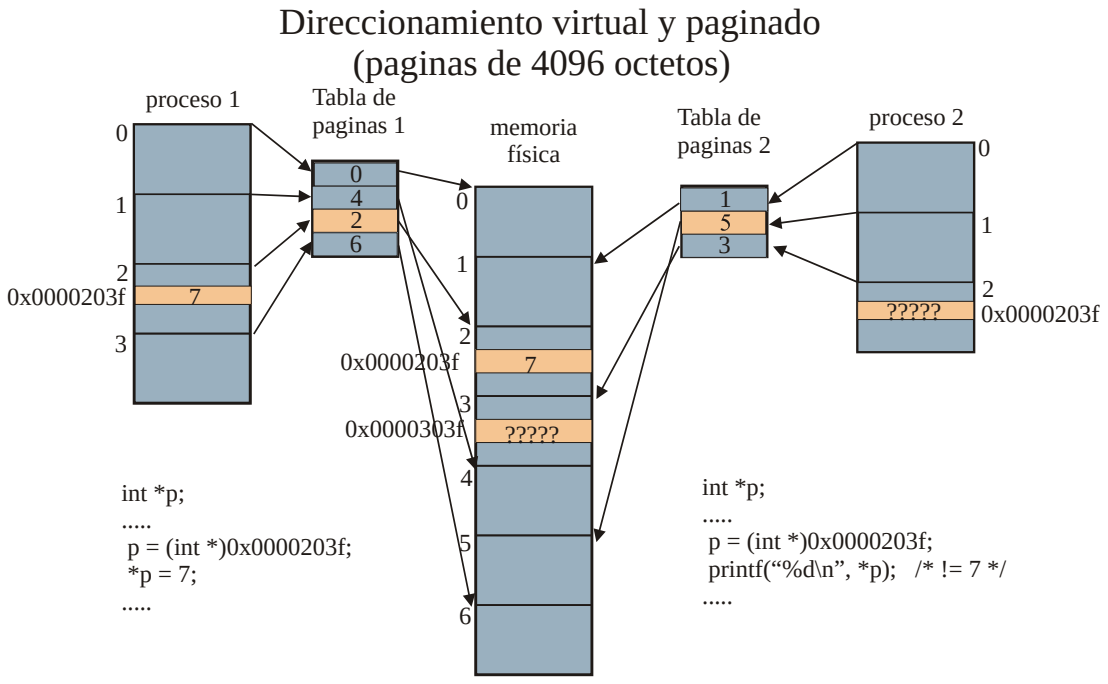

Semáforos POSIX (1003.1b)

- Operaciones básicas:
  - Equivalente de “**wait**”: **sem_wait**
  - Equivalente de **“signal”: sem_post**
  - El funcionamiento es **el mismo** para ambos tipos:

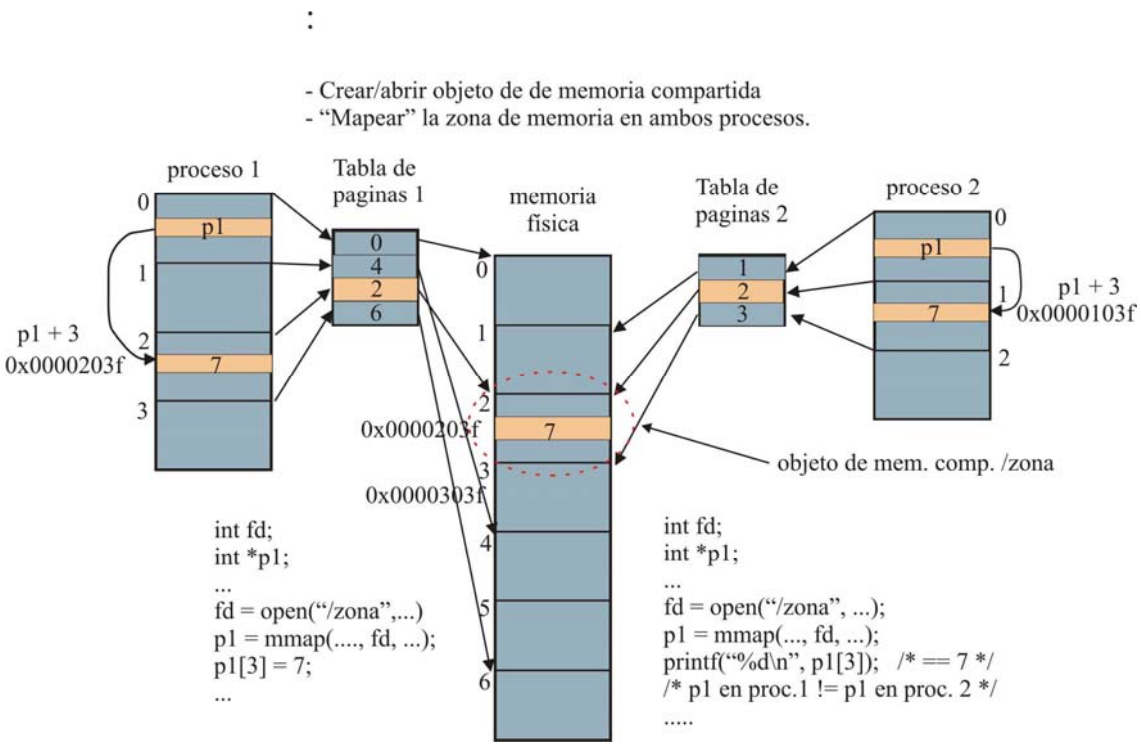
```
#include <semaphore.h>

sem_t *semafo;

semafo = sem_open(.....);
<....>
sem_wait(semafo);
<sección crítica>
sem_post(semafo);
<...>
sem_close(semafo);
```

- Otras operaciones:
  - Espera no bloqueante: **sem_trywait**
  - Obtención de la cuenta actual**: sem_getvalue**

Direccionamiento virtual y paginado



Memoria compartida en POSIX 1003.1b



Memoria compartida en POSIX 1003.1b

- Características básicas:
  - La zona de memoria debe aparecer (ser "mapeada") en el espacio de direccionamiento virtual de los procesos que la comparten
  - Mecanismo común con los archivos **(**_POSIX_MAPPED_FILES y _POSIX_SHARED_MEMORY_OBJECTS)
  - Conviene manejar siempre un número entero de páginas de memoria
- Operaciones a realizar:
  - Abrir un **objeto de memoria compartida** (**shm_open**)
  - Fijar la longitud del objeto (**ftruncate**)
  - "Mapear" el objeto sobre el espacio de direcciones de proceso (**mmap)**
  - El objeto puede cerrarse, pues es accesible de manera directa **(close**)
  - Cuando un proceso no la utilice, puede eliminarse el "mapeo" (**munmap**)
  - Cuando el objeto ya no sirva, puede destruirse (**shm_unlink)**

```
/* Ejemplo: */
#include <sys/mman.h>
#include <sys/stat.h> /* Modos de acceso, como
    S_IRWXU */
#include <fcntl.h>      /* Flags, como O_CREAT */
#include <limits.h>

int fd;
int *pmem;
off_t tamano;

tamano = sysconf(_SC_PAGESIZE);
fd = shm_open("/memcom",
            O_CREAT | O_RDWR, S_IRWXU);
ftruncate(fd, tamano);
pmem = (int *)mmap(0, tamano,
          PROT_READ | PROT_WRITE,
          MAP_SHARED, fd, 0);
close(fd);
<....>
```


Memoria compartida en POSIX 1003.1b

- Prototipo de **mmap**:

  void *mmap(void *donde, size_t long, int protec, int mapflags, int fd, off_t offset);

- Argumentos de **mmap**:
  - **donde**: Dirección de "mapeo" propuesta (puede ser rechazada). Normalmente: 0 (ninguna dirección en particular)
  - **longitud**: Tamaño de la zona "mapeada"; conviene que sea un número entero de páginas. Normalmente: tamaño total del objeto de memoria.
  - **protec**: Combinación de permisos "hardware", compatibles con los permisos del objeto de memoria. Normalmente: PROT_READ | PROT_WRITE.
  - **fd**: Descriptor de archivo asignado al objeto de memoria.
  - **mapflags**: Opciones de "mapeo": MAP_SHARED habilita el acceso desde varios procesos. MAP_FIXED obliga a aceptar **donde**, o fallar. Normalmente: MAP_SHARED.
  - **offset**: Desplazamiento de la zona mapeada, tomando como referencia el principio del objeto. Normalmente: 0.
  - Valor de retorno:
  - Dirección virtual donde el proceso puede "ver" el objeto de memoria, o -1, si **mmap** fracasa. Puede ser igual a **donde**, pero no necesariamente.

Sincronización para hilos (POSIX 1003.1c)

- Sincronización para múltiples hilos (POSIX 1003.1c)
  - Se comparten automáticamente todo el espacio de direccionamiento
  - Sincronización para exclusión mútua: mutex
  - Sincronización para espera de condiciones: variable de condición
  - `Opcionalmente el sistema operativo puede permitir usarlos entre hilos de procesos distintos
- Mutex:
  - Similar a un semáforo binario, que puede estar “abierto” o “cerrado”
  - No se puede:
    - Cerrar un mutex no inicializado
    - Cerrar un mutex que ya se ha cerrado
    - Abrir un mutex que no se ha cerrado antes
  - A diferencia de un semáforo, el hilo que cierra el mutex es el que tiene que abrirlo

Mutex

- Inicialización de un mutex:
  - Representado por una variable **pthread_mutex_t** (global, reservada con **malloc** en “heap”)
  - Inicialización:
    - Estática (sólo globales): Macro PTHREAD_MUTEX_INITIALIZER
    - Dinámica: int **pthread_mutex_init**(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
- Cierre de un mutex:

  int **pthread_mutex_lock**(pthread_mutex_t *pmut);
  - Es bloqueante
  - Si un hilo intenta cerrar un mutex que ya ha sido cerrado por otro hilo, debe esperar a que sea abierto
- Apertura de un mutex:

  pthread_mutex_unlock(pthread_mutex_t *pmut);
  - Si varios hilos están esperando por el mutex, sólo uno lo consigue cerrar y abandona la espera; el resto sigue esperando
  - Criterio: prioridad y algoritmo de planificación, cola de espera…

Mutex

```
#include <pthread.h>

/* Inicialización estática */
pthread_mutex_t mut =
        PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t *pmut;
(...)
/* Inicialización dinámica */
pmut = (pthread_mutex_t *)malloc
                (sizeof(pthread_mutex_t));
pthread_mutex_init(pmut, NULL);
(...)
```

```
/* Sección crítica de hilo 1 */
pthread_mutex_lock(&mut);
<código de la sección crítica>
pthread_mutex_unlock(&mut);

(…)

/* Sección crítica de hilo 2*/
pthread_mutex_lock(&mut);
<código de la sección crítica>
pthread_mutex_unlock(&mut);
```

- Otras llamadas:
  - **pthread_destroy** libera los recursos asociados a un mutex.
  - **pthread_mutex_trylock**, no es bloqueante: Si el mutex está cerrado, devuelve un error EBUSY y no espera

Variables de condición

- Condición: Expresión con resultado binario que depende de datos compartidos protegidos por un mutex (A==B, A>5, ….)
- Variable de condición:
  - No es la condición, no es cierta ni falsa, no tiene estado que indique si se cumple la condición
  - Es un medio de sincronización. Se utiliza para notificar el cumplimiento de una condición
  - Está asociada implícitamente al mutex que protege las variables compartidas de las que depende la condición
- Creación e inicialización:
  - Representada por una variable de tipo **pthread_cond_t** (global, reservada con **malloc** en “heap”)
  - Inicialización:
    - Estática (sólo globales): Macro PTHREAD_COND_INITIALIZER
    - Dinámica: int **pthread_cond_init**(pthread_cond_t *cond, pthread_condattr_t *attr);

```
#include <pthread.h>
/* Inicialización estática */
pthread_cond_t condicion =
    PTHREAD_COND_INITIALIZER;
pthread_cond_t *pcond;
(...)
```

```
/* Inicialización dinámica */
pcond = (pthread_cond_t
    *)malloc(sizeof(pthread_cond_t));
pthread_cond_init(pmut, NULL);
(...)
```

Variables de condición

- Invalidación y liberación de recursos:

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

- Operaciones básicas:
  - Esperar activación:
    - El hilo se bloquea a la espera de activación de la variable de condición
    - Una activación anterior a la espera no sirve para desbloquear

    int **pthread_cond_wait**(pthread_cond_t *cond,
    pthread_mutex_t *mutex);
    int **pthread_cond_timedwait**(pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *abstime);
  - Activar:
    - Al activar la variable uno o varios de los hilos que esperan activación se desbloquean (si hay alguno)
    - La activación no se memoriza

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

Espera de una condición

- Procedimiento recomendado:

  Pedir mutex para acceder a los datos compartidos
  Repetir mientras no se cumpla la condición binaria:
  Esperar la activación de la variable de condición (**pthread_cond_wait**).
  Fin Repetir
  Realizar el procesamiento necesario.
  Liberar el mutex
- Funcionamiento de **pthread_cond_wait**:
  - Argumentos: Una variable de condición (**cond**) y un mutex **(mutex**), que protege los datos de los que depende la condición
  - El hilo libera el mutex y espera la activación de **cond**
  - Cuando se activa **cond** algunos de los hilos bloqueados abandonan la espera pero han de **"competir" con el resto para cerrar el mutex**
  - Al salir de **pthread_cond_wait** el hilo tiene de nuevo el mutex
- ¿Por qué el bucle?
  - Si la condición se cumple de entrada, **no debe esperarse la activación**. La espera es inútil y podría no terminar nunca
  - Cuando **pthread_cond_wait** vuelve, **no hay garantías** de que la condición se siga cumpliendo; es necesario comprobarla de nuevo

Espera de una condición

- Código de la espera:

```
#include <pthread.h>
.............
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t   cond = PTHREAD_COND_INITIALIZER;
.............
pthread_mutex_lock(&mut);
while(<condición: expresion que depende de datos compartidos> == 0)
{
    pthread_cond_wait(&cond, &mut);
}
< procesamiento necesario >
pthread_mutex_unlock(&mut);
```

- Espera con sobretiempo:
  - **pthread_cond_timedwait** incorpora un “timeout” expresado en tiempo absoluto, no relativo
  - Si vence el sobretiempo el hilo ya no espera la condición, y el retorno es ETIMEDOUT
  - Sin embargo el hilo siempre ha esperado hasta “cerrar” el mutex de nuevo, y debe liberarlo en cualquier caso

Activación de una variable de condición

- Consideraciones generales:
  - Si un hilo está bloqueado en **pthread_cond_wait**, otro debe activar la variable para salir de tal situación
  - El hilo “activador” puede ser el causante de que se cumpla de nuevo la condición, o simplemente ha detectado su cumplimiento
  - Normalmente el “activador” ha **“**cerrado” el mutex, y debe liberarlo más tarde
  - Cuando un hilo es desbloqueado por la activación, aún tiene que competir por el mutex antes de salir de la espera – se mantiene la exclusión mútua
- Procedimiento habitual:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
(…)
pthread_mutex_lock(&mut);
<operaciones que originan el cumplimiento de la condición>
pthread_cond_broadcast(&cond); /* o pthread_cond_signal */
(…)
pthread_mutex_unlock(&mut);
```

Activación de una variable de condición

- Opciones para activar:
  - **pthread_cond_broadcast:** Todos los hilos son desbloqueados.
    - →Ningún hilo queda bloqueado cuando se cumple la condición, pero muchos pueden "despertarse" inútilmente
  - **pthread_cond_signal:** Al menos uno de los hilos es desbloqueado
    - →No se desbloquean hilos inútilmente, pero es posible que algunos queden bloqueados aunque la condición se cumpla
    - →Solución: realizar varias activaciones o hacer que cualquier hilo activa la variable cuando comprueba que la condición se cumple, aunque no sea el causante

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
(…)
/* En cualquier hilo que acceda a los datos compartidos */
pthread_mutex_lock(&mut);
<operaciones necesarias>
<comprobar condiciones;
  para las que se cumplen, utilizar pthread_cond_signal>
pthread_mutex_unlock(&mut);
```


Regiones críticas condicionales

- Regiones críticas condicionales
  - Los datos compartidos se agrupan en recursos, con nombre conocido
  - El lenguaje permite definir las regiones críticas, relacionándolas con un determinado recurso
  - El sistema asegurará la ejecución de las regiones críticas relacionadas con el mismo recurso en exclusión mútua
  - La sincronización con condiciones se resuelve con "guardas"

```
Recurso A:
    <definición de variables compartidas>
    int i;
Fin A.

Tarea P1:
    <....>
    Region A, cuando i!=0
    <región crítica>
    Fin Region.
    <...>
Fin P1
```

- Solución poco estructurada
- Reevaluación de condiciones: elevado coste en conmutaciones de procesos

Monitores

- Principios de funcionamiento:
  - Datos compartidos, inaccesibles de manera directa
  - Funciones que permiten realizar operaciones con los datos compartidos con acceso exclusivo
  - Sincronización con condiciones: operaciones **wait** y **signal**
- Conclusiones:
  - Encapsulamiento de datos y procedimientos
  - Mezcla de características de alto y bajo nivel (**wait** y **signal**) poco deseable
  - Servicios de mutex y variables de condición (POSIX 1003.1c) pensados para construir monitores

```
monitor A:
    condiciones:
        noescero;
        nomax;
    datos:
        <...>
        int MAX = 50;
        int i;
    fin datos;
entrada dec:
    if(i==0) wait(noescero);
    i=i-1;
    signal(nomax);
    <...>
fin dec;
entrada inc:
    if(i==MAX) wait(nomax);
    <...>
    i = i + 1;
    signal(noescero);
fin inc;
```


Objetos protegidos (ADA)

- Principios de funcionamiento:
  - Los datos compartidos solo son accesibles a través de procedimientos.
  - El acceso a los procedimientos del objeto protegido se realiza con garantías de exclusión mutua.
  - La sincronización con condiciones se realiza mediante barreras de entrada (“barriers”)
- Acceso al objeto protegido:
  - **function**: Acceso concurrente de varias tareas, con “read lock”: no se pueden modificar los datos compartidos; no se permite el acceso a **procedure** o **entry** hasta que quede libre el recurso.
  - **procedure**: Pueden modificarse los datos compartidos, y se garantiza el acceso exclusivo; el recurso está bajo “read/write lock”.
  - **entry**: Llevan asociada una condición (“barrier”) que debe verificarse, además de estar libre el recurso.

```
protected type ejemplo is
    function lee return Integer;
    procedure reset;
    entry dec(decre: in Integer);
    entry inc(incre:in Integer);
private
    midato: Integer := 0;
    maximo: Integer := 50;
end;
protected body ejemplo is
    function lee return Integer is
    begin
        return midato;
    end lee;
    procedure reset is
    begin
        midato := 0;
    end reset;
    entry dec(decre:in Integer)
        when midato > 0 is
    begin
        midato = midato -decre;
    end dec;
    entry inc(incre: in Integer)
        when midato <= maximo is
    begin
        midato = midato + incre;
    end dec;
end ejemplo;
```

Objetos protegidos (ADA)

- Evaluación de las condiciones de barrera:
  - Un proceso llame a una entry, y existe la posibilidad de que la condición se haya modificado desde su última evaluación.
  - Un proceso sale del objeto protegido, existen tareas encoladas a causa de una barrera que no se verifica y existe la posibilidad de que el proceso que abandona el objeto haya provocado el cambio de su condición asociada.