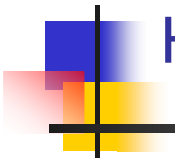
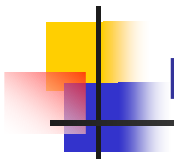


Hilos de ejecución POSIX

Ampliación de Sistemas Operativos (prácticas)
E.U. Informática en Segovia
Universidad de Valladolid

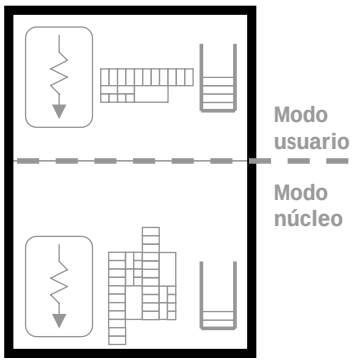


Hilos de ejecución

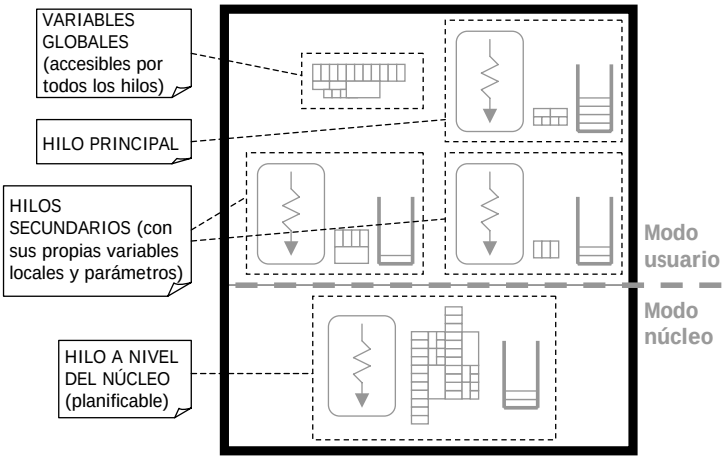
- Concepto
  - Un proceso convencional se caracteriza por:
    - Ser la *unidad de propiedad de recursos*, es decir, un conjunto de recursos asignados a un proceso, en particular:
      - Espacio de direcciones virtuales que contiene la imagen del proceso
      - Otros recursos que puede solicitar (dispositivos de E/S, archivos, etc.)
    - Ser la *unidad de ejecución*:
      - Es una determinada secuencia de instrucciones, ejecutadas dentro de la imagen de memoria de un proceso (en su contexto)
      - Esta secuencia se puede intercalar con las secuencias de otros procesos (mediante cambios de contexto) y alcanzándose de este modo concurrencia a nivel de procesos
  - La noción de hilo de ejecución (*thread*) surge cuando el sistema operativo gestiona independientemente las dos características fundamentales de un proceso tradicional

Hilos de ejecución

- Procesos tradicionales (pesados)
  - Concurrencia fuera del proceso (ínter procesos)
  - Unidad de propiedad de recursos
  - Tiene muchos atributos
  - Es costoso su creación, destrucción, cambio de contexto, etc.



- Procesos multihilo
  - Actividad concurrente dentro de un proceso pesado (intra proceso): pueden existir varios hilos
  - Comparten los recursos del proceso (variables globales, ficheros, etc.)
  - Tienen pocos atributos (contexto de hilo) y son menos costosos de gestionar



Hilos de ejecución

- ¿Cómo se programa con hilos? Se tienen básicamente dos posibilidades:
  - Utilizando un lenguaje de programación convencional y llamadas al sistema (o funciones de biblioteca)
    - Ejemplo: Hilos POSIX utilizados desde C, (enlazando con la librería `pthreads`, opción `-lptrhead`)
  - Utilizando construcciones lingüísticas (o clases) de un lenguaje de programación concurrente
    - Ejemplo: Tareas en Ada95, threads en Java
    - En este caso, el compilador traduce las construcciones lingüísticas a:
      - Llamadas al sistema (hilos a nivel de núcleo)
      - Llamadas a bibliotecas propias de soporte de hilos (hilos a nivel de usuario)

Hilos POSIX (pthreads)

- Dentro de un proceso POSIX convencional:
  - Existe un **hilo inicial** que ejecuta la función **`main()`**
  - Este hilo puede crear más hilos para ejecutar otras funciones dentro del espacio de direcciones del proceso
  - Todos los hilos de un proceso se encuentran al mismo nivel
    - Esto significa que son “hermanos”, a diferencia de los proceso cuya relación es “padre-hijo”
  - Los hilos de un proceso comparten las variables y recursos globales (archivos, manejadores de señales, etc.) del proceso
    - Además, cada uno tiene una **copia privada** de sus parámetros iniciales y de las variables locales de la función que ejecuta (almacenados en su pila particular)
- El estándar POSIX define, entre otras, funciones para:
  - Creación de hilos
  - Creación/destrucción de atributos de creación de hilos
  - Terminación/espera a la terminación de un hilo
  - Identificación de hilos

Hilos POSIX (pthreads): creación

- **`pthread_create`**: Creación de hilos

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(* start_routine)(void *),
                   void *arg);
```

- Descripción
  - Crea inmediatamente el hilo en estado **preparado**, por lo que el hilo creado y su hilo creador **compitan** por la CPU según la política de planificación del sistema
  - Puede ser invocada por cualquier hilo del proceso (no sólo por el “hilo inicial”) para crear otro hilo
  - Parámetros:
    - **`attr`** es el atributo que contiene las características del hilo creado (véanse atributos de un hilo en las siguientes transparencias)
    - **`start_routine`** es la función que ejecutará el hilo
    - **`arg`** es un puntero a los parámetros iniciales del hilo
    - En **`thread`** se devuelve el identificador del hilo creado si la llamada tiene éxito
  - Valor de retorno:
    - 0 si éxito y un valor negativo si hay error

Hilos POSIX (pthreads): atributos

- **`pthread_attr_init/destroy`**: Manipulación atributos de un hilo

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- Descripción
  - **`pthread_attr_init`** inicializa el objeto de atributos de un hilo **`attr`** y establece los valores por defecto
  - Posteriormente, este objeto, con los atributos por defecto de un hilo, se puede utilizar para crear múltiples hilos
  - **`pthread_attr_destroy`**, destruye el objeto de atributos de un hilo, **`attr`**, y éste no puede volver a utilizarse hasta que no se vuelva a inicializar

Hilos POSIX (pthreads): atributos

- Descripción (continuación)
  - Atributos (más relevantes) de un hilo POSIX:
    - **`detachstate`**: controla si otro hilo hilo podrá esperar por la terminación de este hilo (mediante la invocación a **`pthread_join`**):
      - **`PTHREAD_CREATE_JOINABLE`** (valor por defecto)
      - **`PTHREAD_CREATE_DETACHED`**
    - **`schedpolicy`**: controla cómo se planificará el hilo
      - **`SCHED_OTHER`** (valor por defecto, planificación normal + no tiempo real)
      - **`SCHED_RR`** (*Round Robin* + tiempo real + privilegios *root*)
      - **`SCHER_FIFO`**(*First In First Out* + tiempo real + privilegios *root*)
    - **`scope`**: controla a qué nivel es reconocido el hilo
      - **`PTHREAD_SCOPE_SYSTEM`** (valor por defecto, el hilo es reconocido por el núcleo)
      - **`PTHREAD_SCOPE_PROCESS`** (no soportado en la implementación LinuxThreads de hilos POSIX)

Hilos POSIX (pthreads): atributos

- **`pthread_attr_get/getxxxxx`**: Establecimiento/Consulta atributos particulares de un objeto con los atributos de un hilo

```
#include <pthread.h>

int pthread_attr_setdetachstate (pthread_attr_t *attr,
                                 int detachstate);
int pthread_attr_getdetachstate (const pthread_attr_t *attr,
                                 int *detachstate);
int pthread_attr_setschedpolicy (pthread_attr_t *attr,
                                 int policy);
int pthread_attr_getdetachstate (const pthread_attr_t *attr,
                                 int *policy);
int pthread_attr_setscope       (pthread_attr_t *attr,
                                 int contentionscope);
int pthread_attr_getscope       (const pthread_attr_t *attr,
                                 int *contentionscope);
...
```


Hilos POSIX (pthreads): terminación

- **`pthread_exit`**: Terminación de un hilo

```
#include <pthread.h>

void pthread_exit(void *status);
```

- Descripción
  - **`pthread_exit`** finaliza explícitamente la ejecución del hilo que la invoca
    - La finalización de un hilo también se hace cuando finaliza la ejecución de las instrucciones de su función
  - La finalización del último hilo de un proceso finaliza la ejecución del proceso
  - Si el hilo es sincronizable (*joinable*) el identificador del hilo y su valor de retorno puede examinarse por otro hilo mediante la invocación a **`pthread_join`** a través del parámetro **`status`**

Hilos POSIX (pthreads): espera por terminación

- **`pthread_join`**: Esperar por la terminación de un hilo

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **status);
```

- Descripción
  - Esta función suspende la ejecución del hilo que la invoca hasta que el hilo identificado por el valor **`tid`** finaliza, bien por la invocación a la función **`pthread_exit`** o por estar cancelado
  - Si **`status`** no es **`NULL`**, el valor devuelto por el hilo (el argumento de la función **`pthread_exit`**, cuando el hilo hijo finaliza) se almacena en la dirección indicada por **`status`**
  - El valor devuelto es o bien el argumento de la función **`pthread_exit`** o el valor **`PTHREAD_CANCELED`** si el hilo **`tid`** está cancelado
  - El hilo por el que se espera su terminación debe estar en estado sincronizable (*joinable state*)
    - Cuando un hilo en este estado termina, no se liberan sus propios recursos (descriptor del hilo y pila) hasta que otro hilo espere por él
    - La espera por la terminación de un hilo para el cual ya hay otro hilo esperando, genera un error

Hilos POSIX (pthreads): cancelación

- **`pthread_cancel`**: Solicitar la cancelación de un hilo

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

- Descripción
  - La cancelación es el mecanismo por el cual un hilo puede solicitar la terminación de la ejecución de otro
  - Dependiendo de la configuración del hilo al que se solicita su cancelación, puede aceptar peticiones de cancelación (**`PTHREAD_CANCEL_ENABLE`**, estado por defecto) o rechazarlas (**`PTHREAD_CANCEL_DISABLE`**)
  - En caso de aceptar peticiones de cancelación, un hilo puede completar la cancelación de dos formas diferentes:
    - De forma asíncrona (**`PTHREAD_CANCEL_ASYNCHRONOUS`**), o
    - De forma diferida (**`PTHREAD_CANCEL_DEFERRED`**, valor por defecto) hasta que se alcance un punto de cancelación
      - Un punto de cancelación (*cancelation point*) es un punto en el flujo de control de un hilo en el que se comprueba si hay solicitudes de cancelación pendientes
  - Cuando un hilo acepta una petición de cancelación, el hilo actúa como si se hubiese realizado la siguiente invocación **`pthread_exit(PTHREAD_CANCELED)`**

Hilos POSIX (pthreads): identificación

- **`pthread_self`**: devuelve el identificador de un hilo

```
#include <pthread.h>

pthread_t pthread_self(void);
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

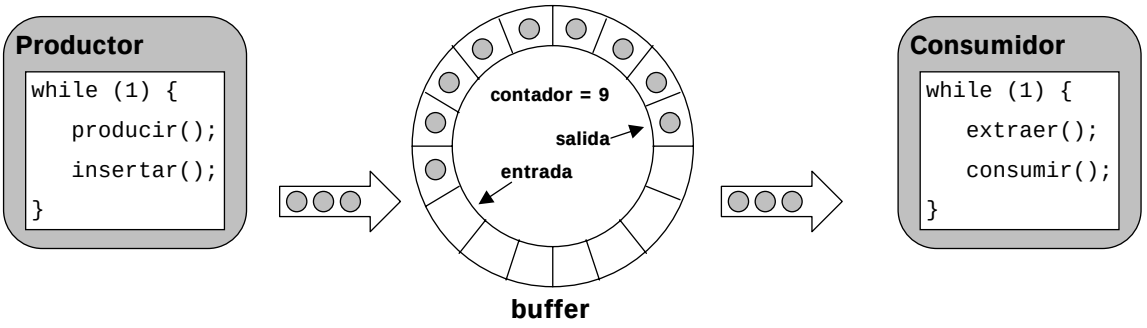
- Descripción
  - La función **`pthread_self`** devuelve el identificador de hilo (tid, *thread identifier*) del hilo que la invoca
  - Para comparar diferentes identificadores de hilo debe utilizarse la función **`pthread_equal`** que:
    - Devuelve 0 si los identificares no son iguales
    - Otro valor si los identificadores sí son iguales

Hilos POSIX (pthreads): Ejemplo 1

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

void *func_hilo (void *arg) {
   printf("Hilo creado: ¡Hola mundo!\n");
}

int main(void) {
   pthread_t      tid;
   pthread_attr_t atrib;

   printf("Hilo principal: INICIO\n");
   pthread_attr_init(&atrib);
   pthread_create(&tid, &atrib, func_hilo, NULL);
   printf("Hilo principal: HILO CREADO\n");
   pthread_join(tid, NULL);
   printf("Hilo principal: FIN\n");
}
```

Sincronización de Hilos: Problemática

- Ejemplo: el problema del “productor/consumidor” (o del “buffer acotado”)
  - Existen dos tipos de entidades: productores y consumidores (de ciertos “items” o elementos de datos
  - Existe un buffer acotado (cola circular) que acomoda la diferencia de velocidad entre productores y consumidores:
    - Si el buffer se llena, los productores deben suspenderse
    - Si el buffer se vacía, los consumidores deben suspenderse



Sincronización de Hilos: Problemática

- Ejemplo: implementación en POSIX
  - Variables globales

```
#define N  20
int        buffer[N]
int        entrada, salida, contador;
```

  - Programa principal

```
int main(void) {
   pthread_attr_t atrib;
   pthread_t      hcons, hprod;

   pthread_attr_init(&atrib);
   entrada= 0; salida= 0; contador= 0;
   pthread_create(&hprod, &atrib, func_proc, NULL);
   pthread_create(&hcons, &atrib, func_cons, NULL);
   pthread_join(hprod, NULL);
   pthread_join(hcons, NULL);
}
```

Sincronización de Hilos: Problemática

- Ejemplo: implementación en POSIX (continuación)
  - Código de los hilos productor y consumidor:

```
void *func_prod(void *arg) {
   int item;

   while (1) {
      item= producir();

      while (contador == N)
         /*bucle vacío: espera */ ;
      buffer[entrada]= item;
      entrada= (entrada + 1) % N;
      contador= contador + 1;
}
```

```
void *func_cons(void *arg) {
   int item;

   while (1) {
      while (contador == 0)
         /*bucle vacío: espera */ ;
      item= buffer[entrada];
      salida= (salida + 1) % N;
      contador= contador - 1;

      consumir(item);
}
```

  - En este código:
    - **`contador`** y **`buffer`** son compartidos por el hilo productor y consumidor
    - Con varios hilos productores y consumidores, **`entrada`** sería compartida por todos los productores y **`salida`** por todos los consumidores

Sincronización de Hilos: Problemática

- En este ejemplo, los hilos que ejecutan el código de productor y consumidor se ejecutan de forma **concurrente**
  - Cada hilo puede ser elegido para ejecución independientemente
  - Las decisiones de
    - **qué** hilo se ejecuta en cada momento, y
    - **cuándo** se produce cada cambio de contexto

    dependen de un **planificador** y no del programador de la aplicación (por lo general)
  - Esto puede acarrar que un código que es correcto si se ejecuta **secuencialmente**, pueda dejar de serlo cuando se ejecuta **concurrentemente**, debido a una situación conocida como **condición de carrera** (*race condition*)
  - **Ejemplo de condición de carrera**
    - Al compilar el código, instrucciones que aparentemente son individuales, pueden convertirse en secuencias de instrucciones en ensamblador
    - Por ejemplo, supóngase las siguientes instrucciones que ejecutan un productor y un consumidor

Productor

```
contador= contador + 1;
```

```
mov reg1, contador;
inc reg1;
mov contador, reg1;
```

Consumidor

```
contador= contador – 1;
```

```
mov reg2, contador;
dec reg2;
mov contador, reg2;
```

Sincronización de Hilos: Problemática

- **Ejemplo de condición de carrera (continuación)**
  - Si se supone que:
    - Inicialmente **`contador`** vale 5
    - Un productor ejecuta la instrucción **`contador = contador + 1;`**
    - Un consumidor ejecuta la instrucción **`contador = contador – 1;`**

      entonces, el resultado de **`contador`** debería ser otra vez 5
  - Pero, qué pasa si se produce un cambio de contexto en un momento “inoportuno”:

Cambio de contexto

t	Hilo	Operación	reg1	reg2	contador
0	productor	**mov** contador, reg1	5	?	5
1	productor	**inc** reg1	6	?	5
2	consumidor	**mov** contador, reg2	?	5	5
3	consumidor	**dec** reg2	?	4	5
4	consumidor	**mov** reg2, contador	?	4	4
5	productor	**mov** reg1, contador	6	?	**6**

¡Incorrecto!

Sincronización de Hilos: Problemática

- Así pues, una **condición de carrera**
  - Se produce cuando la ejecución de un conjunto de operaciones sobre una variable compartida deja la variable en un estado inconsistente con las especificaciones de corrección
  - Además, el resultado final almacenado de la variable depende de la velocidad relativa en que se ejecutan las operaciones
- El problema de las condiciones de carrera es difícil de resolver, porque
  - El programador se preocupa de la corrección secuencial de su programa, pero no sabe cuándo van a producirse cambios de contexto
  - El sistema operativo no sabe qué están ejecutando los procesos/hilos, ni si es conveniente o no realizar un cambio de contexto en un determinado momento
  - El error es muy difícil de depurar, porque
    - El código de cada hilo es correcto por separado
    - La inconsistencia suele producirse muy de vez en cuando, porque sólo ocurre si hay un cambio de contexto en un lugar preciso (e inoportuno) del código

Semáforos POSIX

- POSIX.1b introdujo el tipo de variable semáforo **`sem_t`**
  - Los semáforos se pueden compartir entre procesos y pueden ser accedidos por parte de todos los hilos del proceso. Los semáforos se heredan de padre a hijo igual que otros recursos (como por ejemplo los descriptores de archivo)
- Las operaciones que soporta son:

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);              /* Operación P(sem) */
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);              /* Operación V(sem) */
int sem_getvalue(sem_t *sem, int *sval);
```


Semáforos POSIX

```
#include <semaphore.h>
sem_t mutex, lleno, vacio;
```

Productor/Consumidor revisado

```
void *func_prod(void *p) {
   int item;

   while(1) {
      item= producir();
      sem_wait(&vacio);
      sem_wait(&mutex);
      buffer[entrada]= item;
      entrada= (entrada + 1) % N;
      contador= contador + 1;
      sem_post(&mutex);
      sem_post(&lleno);
   }
}
```

```
void *func_cons(void *p) {
   int item;

   while(1) {
      sem_wait(&lleno);
      sem_wait(&mutex);
      item= buffer[entrada];
      salida= (salida + 1) % N;
      contador= contador - 1;
      sem_post(&mutex);
      sem_post(&vacio);
      consumir(item);
   }
}
```

```
   ...
   sem_init(&mutex, 0, 1);
   sem_init(&vacio, 0, N);
   sem_init(&lleno, 0, 0);
   ...
```

Semáforos POSIX

- El uso inadecuado de semáforos
  - Puede dejar a un conjunto de procesos/hilos bloqueados
  - Si el único proceso/hilo que puede despertar a otro(s) nunca lo hace, o también se suspende, todos pueden quedar suspendidos indefinidamente
- La más grave de estas situaciones se da cuando se produce interbloqueo
  - Un interbloqueo es una situación en la que un conjunto de procesos/hilos quedan suspendidos, cada uno de ellos esperando a que otro del grupo lo despierte
  - Ejemplo
    - Dos hilos, “hilos1” e “hilo2” manipulan dos semáforos (S1 y S2), inicializados a 1, de acuerdo con el siguiente código

```
void *hilo1(void *arg) {
   ...
   sem_wait(&S1);
   sem_wait(&S2);
   ...
}
```

```
void *hilo2(void *arg) {
   ...
   sem_wait(&S2);
   sem_wait(&S1);
   ...
}
```


Sincronización de *pthreads*

- POSIX.1c ofrece dos objetos de sincronización
  - **Cerrojos de exclusión mutua** (*mutexes*): para la gestión de exclusión mutua. Son equivalentes a un semáforo que sólo puede tomar valor inicial 1
  - **Variables de condición** (*contition variables*): para la espere de sucesos de duración ilimitada. Los hilos se suspenden voluntariamente en estas variables
  - Los cerrojos de exclusión mutua y las variables de condición se pueden usar en todos los hilos del proceso que los crea. Están pensados para sincronizar hilos de un mismo proceso, aunque excepcionalmente se pueden compartir entre procesos
- Como se mostrará, POSIX impone además un estilo de programación
  - Las variables de condición se usan en combinación con los cerrojos de exclusión mutua
  - Su uso resulta más intuitivo que el de los semáforos

Sincronización de *pthreads*: *mutexes*

- Los *mutex*
  - Son mecanismos de sincronización a nivel de hilos (*threads*)
  - Se utilizan únicamente para garantizar la exclusión mutua
  - Conceptualmente funcionan como un **cerrojo**
    - Existen dos operaciones básicas de acceso: **cierre** y **apertura**
  - Cada *mutex* posee
    - Estado: dos posibles estados internos: abierto y cerrado
    - Propietario: Un hilo es el propietario de un cerrojo de exclusión mutua cuando ha ejecutado sobre él una operación de cierre con éxito
  - Funcionamiento
    - Un *mutex* se crea inicialmente abierto y sin propietario
    - Cuando un hilo invoca a la operación de **cierre**
      - Si el *mutex* estaba abierto (y por tanto, sin propietario), lo cierra y pasa a ser su propietario
      - Si el *mutex* ya estaba cerrado, el hilo que invoca a la operación de cierre se suspende
    - Cuando el **propietario** del *mutex* invoca a la operación de apertura
      - Se abre el *mutex*
      - Si existían hilos suspendidos (esperando por el cerrojo), se selecciona uno y se despierta, con lo que puede cerrarlo (y pasar a ser el nuevo propietario)

Sincronización de *pthreads*: *mutexes*

- Creación/destrucción de *mutexes*

```
#include <pthread.h>

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

pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

  - Para crear el cerrojo de exclusión mutua, se tienen dos opciones
    - Con unos atributos **`attr`** específicos (previa invocación a **`pthread_mutexattr_init`**)
    - Con unos atributos por defecto (utilizando la macro **`PTHREAD_MUTEX_INITIALIZER`**)
  - La función **`pthread_mutex_destroy`** destruye el *mutex*
- Gestión atributos de creación de los *mutexes*

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

  - Permiten crear/destruir un objeto **`attr`** con los atributos con los que posteriormente se creará el *mutex*
  - Dichos atributos pueden modificarse mediante funciones específicas
    - Por ejemplo, utilizando la función **`pthread_mutexattr_setpshared`** puede modificarse el atributo **`pshared`**, para conseguir que el *mutex* pueda ser utilizado por otros procesos

Sincronización de *pthreads*: *mutexes*

- Operaciones de apertura y cierre sobre un mutex

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - La operaciones **`lock`** y **`trylock`** tratan de cerrar el cerrojo de exclusión mutua
    - Si está abierto, se cierra y el hilo invocador pasa a ser el propietario
    - Si está cerrado
      - La operación **`lock`** suspende al hilo invocador
      - La operación **`trylock`** retorna, pero devolviendo un error al hilo invocador
  - La operación **`unlock`** abre el cerrojo
    - Si hay hilos suspendidos, se selecciona el más prioritario y se permite que cierre el cerrojo de nuevo

Sincronización de *pthreads*: *mutexes*

- Ejemplo

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

int V= 1000;
              /* Variable global */
pthread_mutex_t m=
   PTHREAD_MUTEX_INITIALIZER;
 /* mutex que controla acceso a V */

void *hilo1(void *arg){
   int i;
   for (i=0; i<1000; i++) {
      pthread_mutex_lock(&m);
      V= V + 1;
      pthread_mutex_unlock(&m);
    }
}
```

```
void *hilo2(void *arg) {
   int i;
   for (i=0; i<1000; i++) {
      pthread_mutex_lock(&m);
      V= V - 1;
      pthread_mutex_unlock(&m);
   }
}

int main(void) {
   pthread_t        h1, h2;
   pthread_attr_t   atrib;
   pthread_attr_init(&atrib);
   pthread_create(&h1, &atrib, hilo1, NULL);
   pthread_create(&h2, &atrib, hilo2, NULL);
   pthread_join(h1, NULL);
   pthread_join(h2, NULL);
   printf("Hilo principal: V= %d\n", V);
}
```

Sincronización de *pthreads*: variables de condición

- Existen ocasiones en las que los hilos:
  - Deben suspenderse a la espera de que se produzca una condición lógica determinada
  - Esta condición lógica suele hacer referencia al estado de recursos (variables) que se estén compartiendo (por ejemplo, condición de buffer lleno o vacío)
  - El problema se tiene cuando la suspensión se lleva a cabo en medio de una sección de código en exclusión mutua
    - El hilo tiene uno (o varios) *mutexes* cerrados
    - Si el hilo se suspende sin más, los *mutexes* quedan cerrados y ningún otro hilo puede ejecutar su "sección crítica" (a pesar de que el mismo no está ejecutándola)
- Por este motivo:
  - La mejor solución es proporcionar un mecanismo general que combine la suspensión de hilos con los *mutex*
  - Este mecanismo se denomina **variable de condición**
    - Son un tipo abstracto de datos con tres operaciones básicas:
      - ESPERA, AVISO_SIMPLE y AVISO_MÚLTIPLE
    - Las variables de condición se llaman así por el hecho de que siempre se usan con una condición, es decir, un predicado. Un hilo prueba un predicado y llama a ESPERA si el predicado es falso. Cuando otro hilo modifica variables que pudieran hacer que se cumpla la condición, activa al hilo bloqueado invocando a AVISO

Sincronización de *pthreads*: variables de condición

- Las variables de condición están siempre asociadas a un *mutex*. POSIX establece que:
  - La **espera** debe invocarse desde **dentro de una sección crítica**, es decir, cuando el hilo es el **propietario** de un cerrojo de exclusión mutua específico
  - El **aviso** debe invocarse desde **dentro de una sección crítica**, cuando el hilo es el propietario del mismo *mutex* asociado a la condición
- Especificación de las **operaciones de acceso**
  - La invocación de **espera(c, m)** ejecuta una apertura del *mutex* **m** y suspende al hilo invocador en la variable de condición **c**
    - El hilo tiene que ser el propietario de **m** antes de invocar a esta operación
    - Las acciones de abrir el *mutex* y suspenderse en la condición se ejecutan de forma **atómica**
    - Cuando el hilo se despierte posteriormente, realizará automáticamente una operación de cierre sobre **m** antes de seguir con la ejecución del código tras la "espera"
  - La invocación de **aviso_simple(c)** despierta a un hilo suspendido en **c**
    - Si no hay hilos suspendidos, la operación no tiene efecto (el aviso se pierde)
  - La invocación de **aviso_multiple(c)** despierta a todos los hilos suspendidos en **c**
    - Si no hay hilo suspendidos, tampoco tiene efecto

Sincronización de pthreads: variables de condición

- En resumen, POSIX establece que el uso correcto de las variables de condición es el siguiente:
  - Dentro de una sección crítica, protegida por "**mutex**", un cierto hilo ("**hilo1**") comprueba si se cumple "**condicion**" y, si es el caso, se suspende en "**cond**"
  - Dentro de una sección crítica protegida por **el mismo** "**mutex**", otro hilo ("**hilo2**") despierta a un hilo (o a todos, con el AVISO_MÚLTIPLE) suspendidos en "**cond**", si se considera que se modifica alguna variable involucrada en el predicado "**condicion**"

```
hilo1:

   cierre(mutex);
   ...
   si (!condicion) entonces
      espera(cond, mutex);
   ...
   /* Acceso a variables comunes */
   apertura(mutex);
   ...
```

```
hilo2:

   cierre(mutex);
   ...
   /* modificacion variables
   involucradas en condicion */
   ...
   aviso_simple(cond);
   apertura(mutex);
   ...
```


Sincronización de *pthreads*: variables de condición

- Creación/destrucción de variables de condición

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *mutex, const pthread_condattr_t *attr);

pthread_cond_t mutex= PTHREAD_COND_INITIALIZER;

int pthread_cond_destroy(pthread_cond_t *mutex);
```

  - Para crear la variable de condición, se tienen dos opciones
    - Con unos atributos **attr** específicos (previa invocación a **pthread_condattr_init**)
    - Con unos atributos por defecto (utilizando la macro **PTHREAD_COND_INITIALIZER**)
  - La función **pthread_cond_destroy** destruye la variable de condición
- Gestión atributos de creación de variables de condición

```
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);

int pthread_condattr_destroy(pthread_condattr_t *attr);
```

  - Permiten crear/destruir un objeto **attr** con los atributos con los que posteriormente se creará la variable de condición
  - Dichos atributos pueden modificarse mediante funciones específicas
    - Por ejemplo, utilizando la función **pthread_condattr_setpshared** puede modificarse el atributo **pshared**, para conseguir que la variable de condición pueda ser utilizado por otros procesos

Sincronización de *pthreads*: variables de condición

- Espera sobre variables de condición

```
#include <pthread.h>
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);
```

  - La operación **`wait`** ejecuta de forma atómica la operación **`pthread_mutex_unlock`** sobre **`mutex`** y suspende al hilo invocador en la variable de condición **`cond`**
    - Al despertarse, realiza automáticamente una operación **`pthread_mutex_lock`** sobre **`mutex`**
  - La operación **`timedwait`** actúa como **`wait`**, salvo que el hilo se suspende como mucho hasta que se alcanza el instante de tiempo **`abstime`**
    - Si se alcanza **`abstime`** antes de que otro hilo ejecute un **`signal/broadcast`** sobre la variable de condición, el hilo dormido se despierta, y **`timedwait`** retorna devolviendo el error **`ETIMEDOUT`**

Sincronización de *pthreads*: variables de condición

- Aviso sobre variables de condición

```
#include <pthread.h>
int pthread_cond_signal   (pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

  - La operación **`signal`** selecciona el hilo más prioritario suspendido en cond y lo despierta
  - La operación **`broadcast`** despierta a todos los hilos que estén suspendidos en la variable de condición **`cond`**
    - Ambas operaciones no tienen efecto si no hay hilos suspendidos en la variable
  - En ambos casos, es **recomendable** que el hilo invocador sea el propietario del cerrojo de exclusión mutua (**`mutex`**) asociado a la variable de condición **`cond`** en los hilos suspendidos
    - De esta manera se garantiza que, si el hilo que va a suspenderse en un **`wait`**, comienza su sección crítica antes que el hilo que va a despertarlo, entonces el aviso **no se pierde**, porque hasta que el primer hilo no se duerme y libera el **`mutex`**, el segundo no puede alcanzar la instrucción **`signal/broadcast`**

Sincronización de *pthreads*: variables de condición

- Recomendaciones de cómo programar con variables de condición
  - Aunque es posible programar así

hilo 1

```
pthread_mutex_lock(&m);
if (!condicion)
   pthread_cond_wait(&c, &m);
/* Resto de la sección crítica */
pthread_mutex_unlock(&m);
```

hilo 2

```
pthread_mutex_lock(&m);
...
/* Se altera la condición */
pthread_cond_signal(&c);
pthread_mutex_unlock(&m);
```

  - Cuando hay más de un hilo del tipo “hilo 1”, se recomienda hacerlo de este otro modo

hilo 1

```
pthread_mutex_lock(&m);
while (!condicion)
   pthread_cond_wait(&c, &m);
/* Resto de la sección crítica */
pthread_mutex_unlock(&m);
```

hilo 2

```
pthread_mutex_lock(&m);
...
/* Se altera la condición */
pthread_cond_broadcast(&c);
pthread_mutex_unlock(&m);
```