Threads en Harbour

2016 Rafa Carmona

Índice

Threads con Harbour para novatos

Vamos a empezar a usar threads con Harbour, y lo primero que tenemos que saber que para ello tenemos que compilar con soporte para threads.
Haciendo uso de **hbmk2** , simplemente, le pasamos **-mt** a la hora de compilar.

Bien, ahora ya estamos preparados para tener acceso al uso de threads
¿Pero que un thread y que nos aporta en nuestras aplicaciones?
Definición simple: Los threads nos permiten hacer varias cosas a la vez. Threads Wikipedia

Un caso real de éxito sobre un software realizado en Clipper, pasarlo a Harbour 32 bits, y después a usar hilos, fue realizado en un par de días.

Inicialmente ese software era Clipper, que lo ha hace es procesar peticiones según se van dejando ficheros en un directorio, viendo por encima, es similar a esto;

```
while  .t.
  aFilesDirectory := adir()
   for n := 1 to len( aFilesDirectory )
       ? "Procesa fichero de peticion"
       Procesa( aFilesDirectory[ n ] )
   next
   deletefiles( aFilesDirectory )
end while
```

Al pasarlo a 32 bits, fue trivial, recompilar y punto.
Pero este sistema tiene un gran hándicap, y es que cuando se procesa una petición, el programa no puede atender a otras peticiones, por lo que el tiempo de las peticiones se suman.

Así si tenemos 5 peticiones, y cada una tarda 10 segundos, vemos que la última será procesada 10 x 5 = 50 segundos después.

¿Y si os digo que usando threads tardaremos 10 segundos en procesar todas las peticiones?
El código de arriba, se convierte usando threads;

```
while  .t.
  aFilesDirectory := adir()
   for n := 1 to len( aFilesDirectory )
       ? "Procesa fichero de peticion"
      hb_threadStart( @Procesa(),  aFilesDirectory[ n ] )
   next
   deletefiles( aFilesDirectory )
```

```
end while
```

Os veo la cara iluminada ;-)

Este software funciona hoy en día con un rendimiento excepcional.
Quien diría que un software en Clipper de 15 años, funcionando en 32bits con Threads.

Ahora, también nos interesa saber si el ejecutable cumple con esta condición, usando simplemente la función **hb_mtvm()**

```
IF hb_mtvm()
   ? "Soporte de Threads"
ENDIF
```

A continuación, vamos a ver la primera función , que no es ni más ni menos que como crear un hilo (intentaré usar los ejemplos que están en Harbour **/test/mt/**);

hb_threadStart(<@sStart()> | <bStart | "sStart" > [, <params,...>]) -> **<pThID>**

Esta función simplemente crea y ejecuta la función, @sStart()y nos devuelve un id.
Un ejemplo simple, seria poner un simple reloj; (**tests/mt/mttest11.prg** es similar)

```
proc main()

   cls
   hb_threadStart( @thFunc() )

   @1,1 SAY "Pulsa tecla para salir"
   inkey( 0 )

return

proc thFunc()

   while .t.
       DispOutAt( 2, 1, Time() )
       hb_idleSleep( 1 )
   end

return
```

- *usar PROCEDURE si la función no retorna nada, simplemente es más optimizado.*

hb_threadStart permite llamar igualmente de estas maneras a la función;

- hb_threadStart(@thFunc())
- hb_threadStart("thFunc")
- hb_threadStart({|| thFunc() })

Es exactamente lo mismo. Normalmente se usa @thFunc(), por ser la primera implementación y al ser un puntero, creo que es más óptimo a nivel de VM. (VM = Virtual Machine)

Ya tenemos nuestra aplicación usando threads. Como vemos, la cosa es bien sencilla, de momento ;-)

Ahora , también a la hora de crear un thread , podemos definir la visibilidad de las variables dentro del hilo. Fichero **hbthread.ch**

```
#define HB_THREAD_INHERIT_PUBLIC     1
```
Indica que las variables públicas son compartidas por todos los hilos

```
#define HB_THREAD_INHERIT_PRIVATE    2
```
Indica que las variables privadas son compartidas por todos los hilos

```
#define HB_THREAD_INHERIT_MEMVARS    3
```
Indica que las variables privadas y públicas son compartidas por todos los hilos

```
#define HB_THREAD_MEMVARS_COPY       4
```
Indica que lo que se envía es una copia , no la variable en sí.

```
hb_threadStart( HB_THREAD_INHERIT_PUBLIC, @Process(), Self ) )
```
otro ejemplo ;
```
hb_threadStart( HB_BITOR( HB_THREAD_INHERIT_MEMVARS + ;
                          HB_THREAD_MEMVARS_COPY ), ;
                          @thFunc() )
```
Un ejemplo de esto es el ejemplo en **harbour/tests/mt/mttest08.prg**

Una como a tener en cuenta, es que cuando creamos un hilo, las variables **PUBLIC** que se creen en ese hilo , solo es visible en ese hilo o en hilos hijos, el padre no sabe de su existencia.

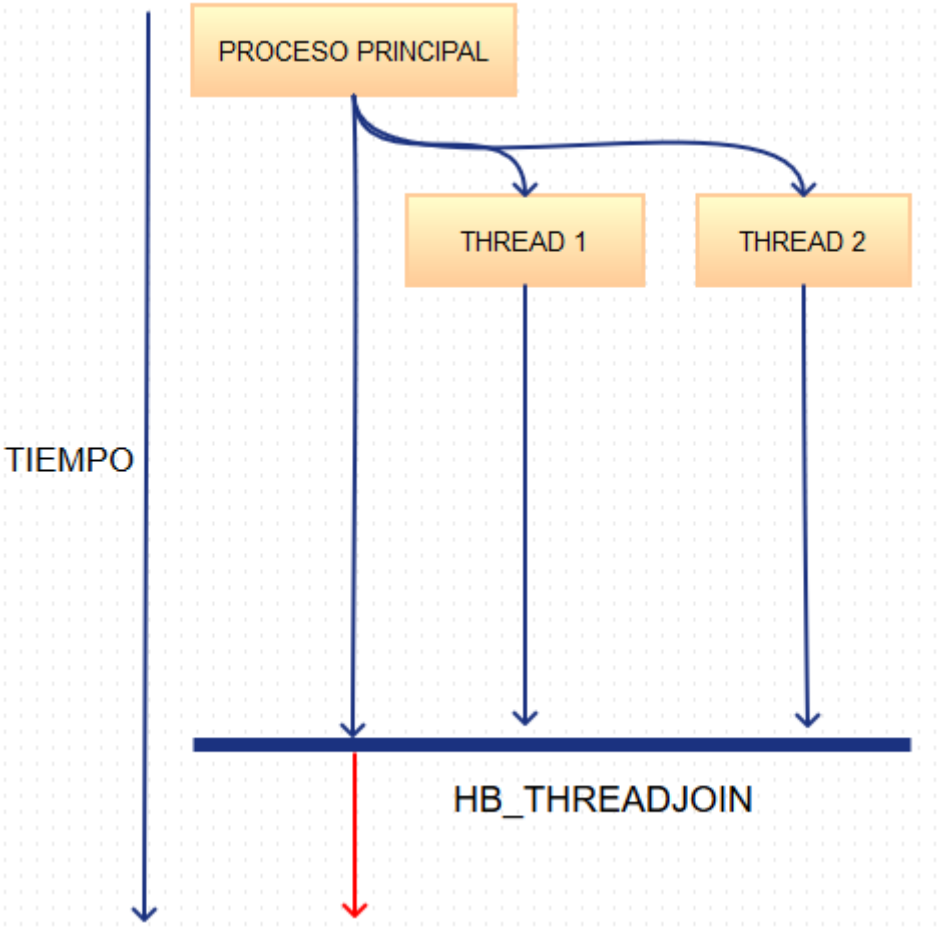
Y recordad que el acceso a escribir en variables compartidas por hilos debe ser protegido por el usuario, pero de momento, no te preocupes, lo veremos más adelante, con los mutex.

Ah, y otra cosa súper interesante , y es que el ámbito de una tabla dbf abierta, pertenece en el hilo que se abrió, usando el mismo alias en hilos separados.
Esto, a la hora de portar código existente , como el caso anteriormente explicado, nos ahorra horas y horas de portabilidad de un sistema monolítico a un sistema con threads.

hb_threadJoin

hb_threadJoin(<pThID> [, @<xRetCode>]) -> <lOK>
Bloquea el thread principal, hasta la finalización de pThID.
Se para una variable por referencia para obtener la respuesta de la función si fuese necesario.

Una descripción gráfica sería la siguiente, donde la llamada a **hb_threadJoin ,** para thread1 y thread2 , bloquea el proceso principal, hasta que los 2 threads se hayan realizado.



Un ejemplo simple es lanzar una serie de hilos para precalcular datos, y esperar a que todos terminen para poder continuar.

En este ejemplo, simplemente lo metemos en un for con un delay para ver el efecto en real de lo que hace **hb_threadJoin()**;

```
#include "hbthread.ch"
static s_p1, s_p2, s_p3

proc main()
   Local aThreads := {}
   Local oObject := Test():New()
```

```
   s_p1 := s_p2 := s_p3 := 0

   AAdd( aThreads, hb_threadStart( HB_THREAD_INHERIT_PUBLIC, @p1() ) )
   AAdd( aThreads, hb_threadStart( HB_THREAD_INHERIT_PUBLIC, @p2() ) )
   AAdd( aThreads, hb_threadStart( HB_THREAD_INHERIT_PUBLIC, @p3() ) )

      AAdd( aThreads, hb_threadStart( {|| oObject:&("Paint")( "From
Harbour" ) } ) )

   AEval( aThreads, {| h | hb_threadJoin( h ) } )

   ? "Total:", s_p1 + s_p2 + s_p3

return

proc p1()
   local x
   for x := 1 to 5
       s_p1 := x
       DispOutAt( 1, 1, "(Thread 1):"+ Str( x,2 ) )
       hb_idleSleep( 0.5 )
   next

return

proc p2()
   local x
   for x := 1 to 5
       s_p2 := x
       DispOutAt( 2, 1, "(Thread 2):"+ Str( x,2 ) )
       hb_idleSleep( 0.5 )
   next

return

proc p3()
   local x

   for x := 1 to 10
       s_p3 := x
       DispOutAt( 3, 1, "(Thread 3):"+ Str( x,2 ) )
       hb_idleSleep( 0.5 )
   next

return

#include "hbclass.ch"
CLASS TEST
      METHOD New( ) CONSTRUCTOR
      METHOD Paint( cValue ) INLINE DispOutAt( 2,20, cValue )
END CLASS

METHOD New() CLASS TEST
RETURN Self
```

También podéis observar cómo llamar al **method** de un **objeto;**

```
hb_threadStart( {|| oObject:&("Paint")( "From Harbour" ) } )
```

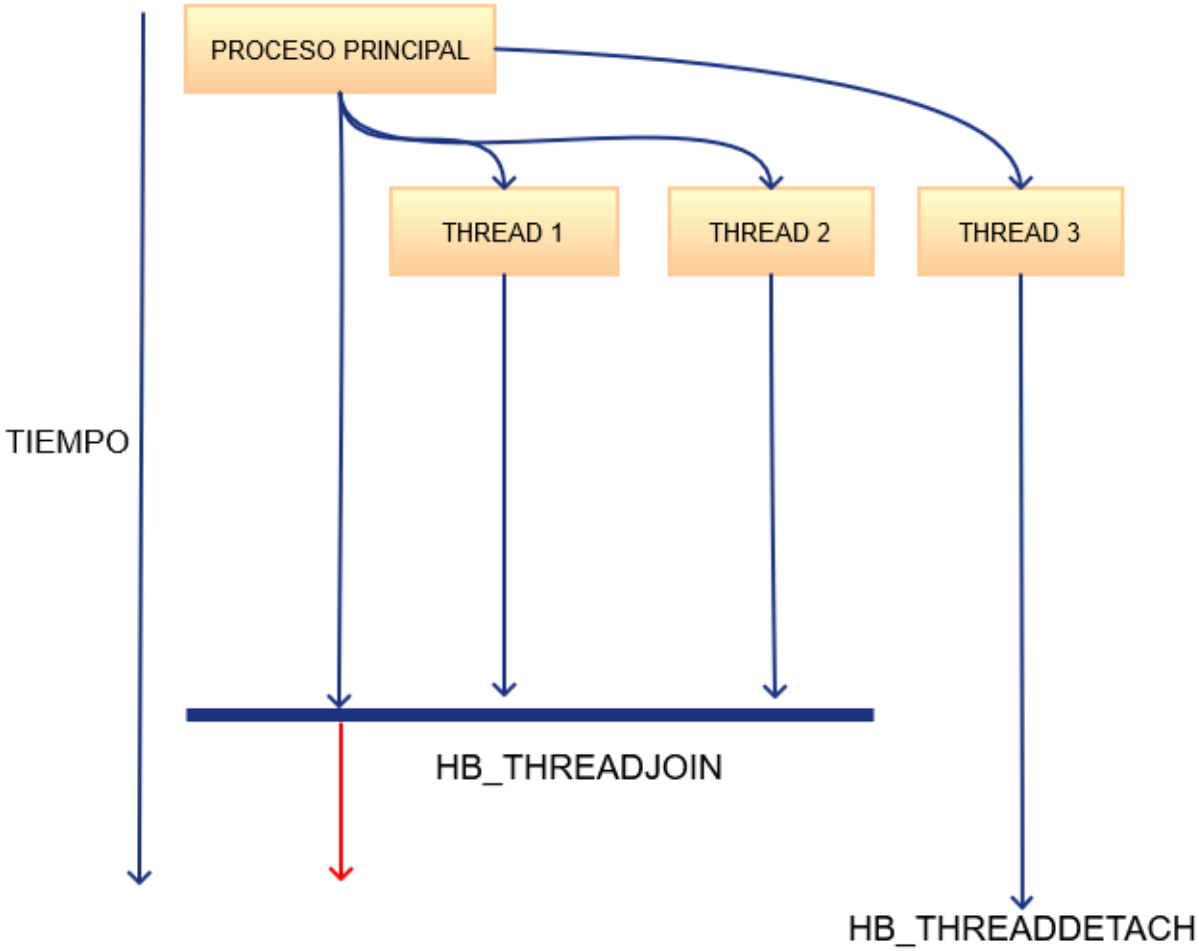
Este ejemplo de Harbour, **/tests/mt/mttest01.prg**, podemos observar cómo obtener la respuesta de la función thFunc(), en este caso, la variable **xResult.**

```
static s_var
proc main()
   local xResult
   ? Version()
   ? "join:", hb_threadJoin( hb_threadStart( @thFunc() ), @xResult
)
   ? "result:", xResult
   ? "static var type:", valtype( s_var )
   ? eval( s_var )
   ? eval( s_var )
return

func thFunc()
   local i := 12345.678
   s_var := {|| i++ }
return replicate( "Hello World!!! ", 3 )
```

hb_threadDetach

hb_threadDetach(<pThID>) -> <lOK>



¿Qué debo hacer cuando mi hilo no devuelve nada útil, y no tiene que esperar a su finalización?

hb_threadDetach indica al sistema: Estoy empezando este hilo, pero no estoy interesado en unirse a él. Esta operación se denomina **separar un thread**.

Las reglas más importantes para unirse (**hb_threadJoin)** / separarse(**hb_threadDetach**) son:

- No se una a un thread que ya se ha unido
- No puede unirse a un thread separado.
- Si extrae un thread, no se puede "volver a conectarse" al mismo.

De todas maneras, no sé exactamente cuál es el propósito del **detach en Harbour** si llamando a hb_threadStart() realiza la misma función, sin usar Join, al menos, en las pruebas que he realizado no logro ver diferencias entre crear un hilo y dejarlo morir y llamar a detach. ***(Si alguien lo sabe, bienvenida explicación)***

Un ejemplo, sería poner un reloj o ejecutar un sonido ante un evento, sin importar el resultado.

Ahora bien, hay que tener en cuento que los threads pueden no terminar, porque el proceso principal ha terminado, matando los threads pendientes.

En el siguiente código, su ejecución va a provocar la creación de un fichero o no, pero ojo, puede ser que el contenido esté o no.

Esto se debe al separar el hilo con **hb_threadDetach**, el terminar el proceso principal, hace que el hilo también se acabe, sin terminar su tarea.

El objetivo del ejemplo es mostrar que hace **hb_threadDetach,** lógicamente, el ejemplo no tiene sentido.

```
#include "hbthread.ch"

proc main()
   cls

   s_p1 := s_p2 := s_p3 := 0
   ? "Llama"
  hb_threadDetach(  hb_threadStart( @p1(), "sin nada" ) )

   ? "Salimos"

return

proc p1( ctext )
   Local h

    h := fcreate("test.txt")
    fwrite( h, ctext)
    fclose( h )

return
```

Entonces, ¿cómo podemos evitar esta situación?

hb_threadWaitForAll

La función **hb_threadWaitForAll**( ) sólo es eficaz cuando se le llama en el hilo principal de la aplicación.

Lo que hace es suspender el hilo principal hasta que el resto de hilos en ejecución terminen.

En el ejemplo anterior, podemos ver cómo solucionar el problema anterior;

```
#include "hbthread.ch"

proc main()
   cls

   s_p1 := s_p2 := s_p3 := 0
   ? "Llama"
  hb_threadDetach(  hb_threadStart( @p1(), "sin nada" ) )

   ? "Salimos"
   hb_threadWaitForAll()

return

proc p1( ctext )
   Local h

    h := fcreate("test.txt")
    fwrite( h, ctext)
    fclose( h )

return
```

Ahora sí que el fichero con su contenido estará disponible.

En el caso que explique en el post inicial, sobre la portabilidad de un sistema de mensajes, el salir del bucle principal, antes de salir de la aplicación, está la llamada a **hb_threadWaitForAll()** para que se terminen los hilos que están en marcha y evitar mensajes sin procesar porque la aplicación ha finalizado.

¿Pero, puedo esperar a un hilo en concreto a que termine su tarea?

hb_threadWait

hb_threadWait(<pThID>|<apThID>, [<nTimeOut>] [, <lAll>]) =><nThInd>|<nThCount> 0

Espera a que se termine **pThID** pasado, o en su caso, un array de threads **apThID.**
nTimerOut indica los segundos que tiene que esperar a que se terminen los threads pasados.
Si no pasamos, se espera hasta que todos los threads pasados finalicen si está activado el tercer parámetro **lAll.**

Lógicamente, no se usan hilos que están unidos a través de hb_threadJoin.

Devuelve el primer índice del hilo que ha terminado, **nThInd,** o si está activado **lAll**, la cantidad de hilos que han podido terminar en el periodo de tiempo que hemos establecido, **nThCount,** o **0 si no se terminó ninguno.**

Un ejemplo que muestra esto es el ejemplo de Harbour **/tests/mt/mttest10.prg**

```
#include "inkey.ch"

#ifdef __PLATFORM__WINDOWS
   REQUEST HB_GT_WVT_DEFAULT
   #define THREAD_GT hb_gtVersion()
#else
   REQUEST HB_GT_STD_DEFAULT
   #define THREAD_GT "XWC"
#endif

proc main( cGT )
   local i, aThreads, n

   if ! hb_mtvm()
      ? "This program needs HVM with MT support"
      quit
   endif

   if empty( cGT )
      cGT := THREAD_GT
   endif

   if  cGT == "QTC" .and. ! cGT == hb_gtVersion()
      /* QTC have to be initialized in main thread */
      hb_gtReload( cGT )
   endif

   ? "Starting threads..."
   aThreads := {}
   for i := 1 to 3
      aadd( aThreads, hb_threadStart( @thFunc(), cGT ) )
      ? i, "=>", atail( aThreads )
```

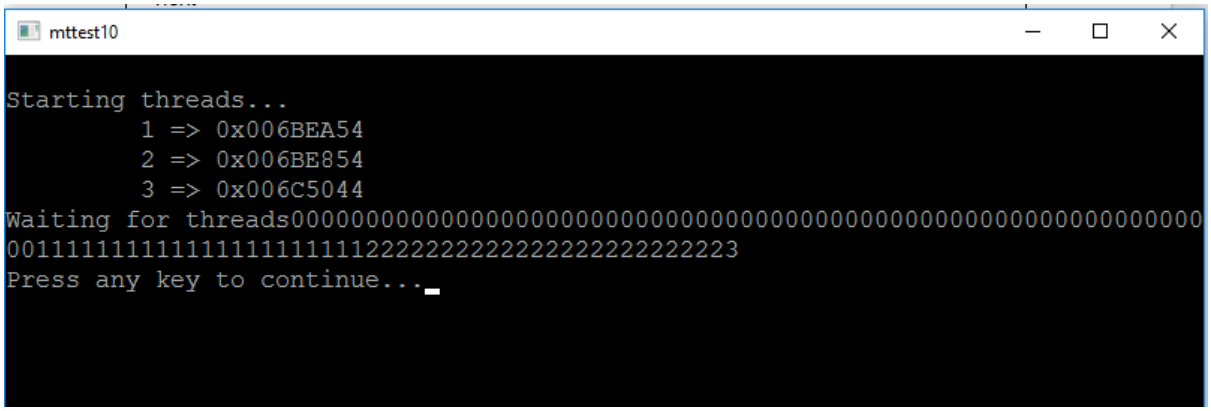
```
   next

  // RETOCADO, para que muestre el valor de n,
   ? "Waiting for threads"
   while inkey() != K_ESC
      n := hb_threadWait( aThreads, 0.1, .T. )
      if n == len( aThreads )
         ?? str(n,1)
         wait
         exit
      endif
      ?? str(n,1)
   enddo


proc thFunc( cGT )
   /* allocate own GT driver */
   hb_gtReload( cGT )
   if ! dbExists( "test" ) .and. dbExists( "../test" )
      use ../test shared
   else
      use test shared
   endif
   browse()
return
```

Este ejemplo es interesante, nos crea 3 hilos con 3 browses sobre la misma tabla, con el mismo alias, y hacer un bucle esperando la finalización de los 3 browses.

Si lo ejecutas, y te pones en la consola que muestra lo siguiente;



Como vimos, cuando devuelve cero, es que no ha finalizado ningún hilo. Si pulsamos ESC sobre la ventana de un browse, este muere, y nos pinta un 1, indicando, de qué de los 3 threads que estamos controlando, uno ha terminado.

A continuación, volvemos a matar otro browse, vemos que muestra un 2, indicando que ya son 2 los threads terminados y si volvemos a matar el browse que queda, veremos que ha pintado 3 y sale del bucle.

Id de Threads

Varias funciones para Threads.

hb_threadSelf() -> <pThID> | NIL
Devuelve Id del hilo donde estoy, incluido el principal.

hb_threadID([<pThID>]) -> <nThNo>
Devuelve el número de thread según pThID. Si no se pasa, devuelve el actual.

En la librería **xhb**, la que hace compatibles varias funciones de xHarbour para Harbour, en el fichero **contrib/xhb/xhbmt.prg,** tenéis disponibles los wrappers para la gente de xHarbour.

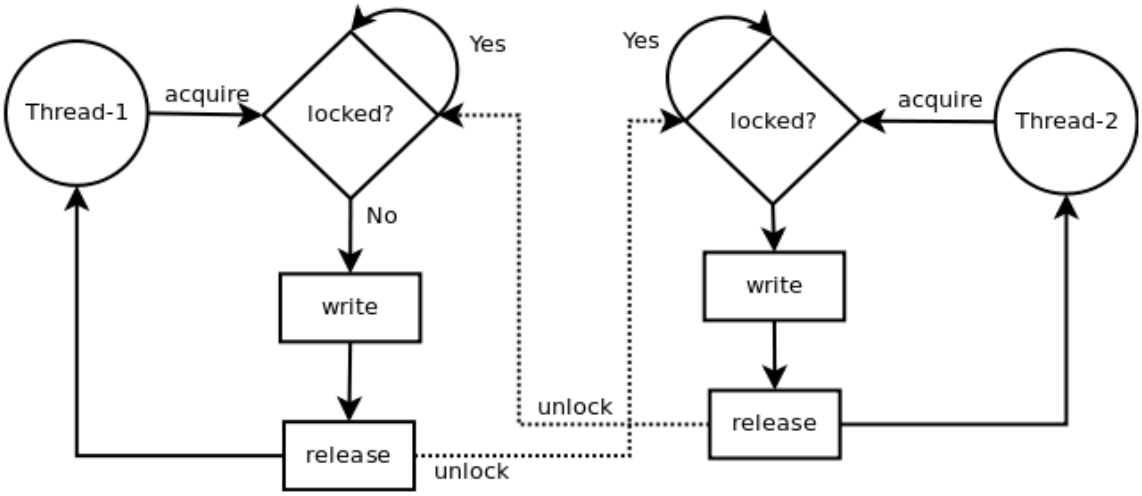
HARBOUR	XHARBOUR
hb_threadStart	StartThread
hb_mutexSubscribe	Subscribe
hb_mutexSubscribeNow	SubscribeNow
Comprobar hb_threadID()	IsSameThread
Compobar hb_threadID()	IsValidThread
hb_threadQuitRequest	KillThread
hb_threadQuitRequest(pThID) hb_threadJoin(pThID)	StopThread
hb_idleSleep(nTimeOut / 1000	ThreadSleep
hb_mutexLock	hb_MutexTryLock
hb_mutexLock	hb_MutexTimeOutLock
iif(PCount() < 1, hb_threadID(), hb_threadID(pThID))	GetSystemThreadId

Terminar Threads

hb_threadQuitRequest(<pThID>) -> <lOK>
Intenta ‘matar’ el hilo pasado pThID. Devuelve .T. si lo consigue.
Aunque se aconseja que los hilos terminen limpiamente, es decir, una vez lanzado, debe de finalizar por sí mismo.

hb_threadTerminateAll() -> NIL
Manda un mensaje de salida a todos los threads en ejecución y espera que terminen.
Solo se debe de llamar desde el thread principal, aunque, esto no debería usarse.

MUTEX



¿Qué es un mutex?
En un proceso concurrente, el acceso a compartir datos puede llegar a crear inconsistencia de datos. Para evitarlo usaremos unos mecanismos que nos va a permitir modificar datos de manera consistente. Ver Exclusión Mutua

hb_mutexCreate() -> <pMtx>
Crea un mutex , que podemos usarlo como semáforo o para suscripción de mensajes.
De momento, vamos a usarlo como semáforo, que por cierto, desde aquí un tributo a quién lo inventó, Edsger Dijkstra en 1965!!! Semáforo (Informática)
Lo que nos devuelve es un puntero que usaremos posteriormente.

hb_mutexLock(<pMtx> [, <nTimeOut>]) -> <lLocked>
Simplemente lo que haces al llamar a **hb_mutexLock()** es establecer un bloqueo en el Thread.

Si el mutex se encuentra bloqueado por otro thread, el thread actual queda a la espera de que se termine el bloqueo, así, cuando el Mutex vuelva a estar disponible, solo unos de los threads obtendrá acceso, mientras que los demás threads seguirán esperando.
nTimeOut, indica la cantidad de segundos que estaremos esperando para conseguir el bloqueo, por defecto , lo intentará indefinidamente.

hb_mutexUnlock(<pMtx>) -> <lOK>
Desbloquea Mutex y lo deja para que otro thread lo pueda bloquear.

Ejemplo de mutex funcionando como semáforo;

```
#include "hbthread.ch"
STATIC s_hMutex
proc main()
   cls
   s_hMutex   := hb_mutexCreate()

   hb_threadStart( @EscribeLog(), "Thread A" )
   hb_threadStart( @EscribeLog(), "Thread B" )
   hb_threadStart( @EscribeLog(), "Thread C" )
   hb_threadStart( @EscribeLog(), "Thread D" )
   ? "Salimos"
   hb_threadWaitForAll()

return

proc EscribeLog( cText )
   static n := 0
  /* El que bloquea, escribe e incrementa n */
  if hb_mutexLock( s_hMutex )
     ? cText , ++n
    hb_mutexUnLock( s_hMutex )
  endif

return
```

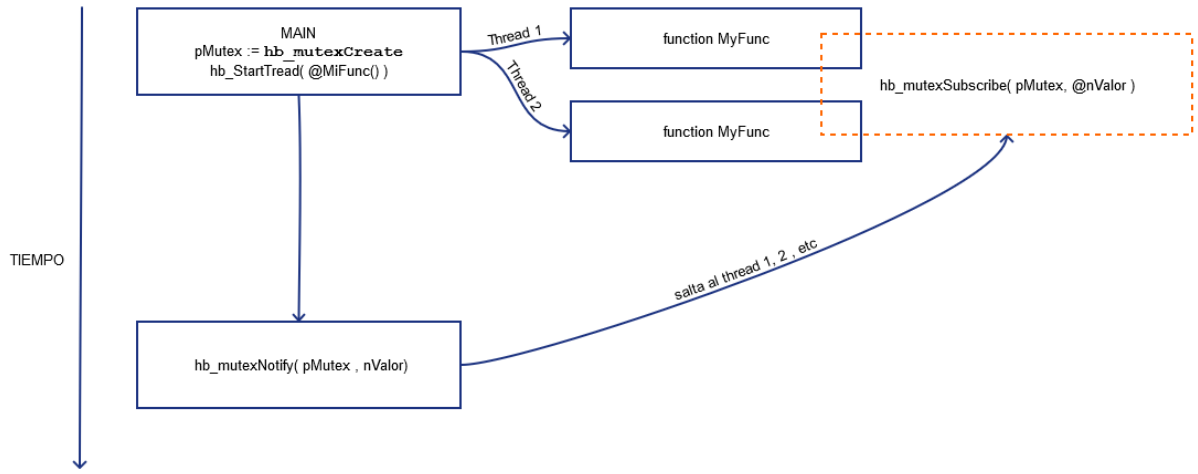
Hay que tener especial cuidado que es lo que bloqueamos. Por ejemplo, realizar esto;;

```
if bb_mutexLock( s_hMutex )
   Funcion_que_tarda_un_monton()
   hb_mutexUnLock( s_hMutex )
endif
```

Toda la magia de los threads se pierde. Es como si al querer modificar un registro de una tabla, bloqueamos TODA LA TABLA, ¿Verdad que no tiene sentido?

Hay que ser cuidadoso, pero como podéis ver, tampoco es tan tan complicado.

Subscribe & Notify



Anteriormente vimos un mutex trabajando como un semáforo, ahora veremos qué es eso de la suscripción de mensajes.

hb_mutexSubscribe(<pMtx>, [<nTimeOut>] [, @<xSubscribed>]) -> <lSubscribed>

Dentro de un thread, suscribimos el thread para ser avisado por una notificación, que lógicamente, vendrá desde otro hilo distinto, por lo tanto , **hb_mutexSubscribe** y **hb_mutexNotify** corren en hilos distintos, y lo que comparten en común es un mutex., <pMtx>.

Si hay notificaciones pendientes, continúa la ejecución del thread, si no, estará suspendido a la espera de una notificación, que puede especificar cuantos segundos esperaremos a ser notificados, <nTimeOut>, por defecto, se queda indefinidamente.

En la variable pasada por referencia, @<xSubscribed>, vamos a obtener el valor que nos envía desde la notificación. Retorna si tuvo éxito la subscripción al mutex o no, <lSubscribed>

hb_mutexSubscribeNow(<pMtx>, [<nTimeOut>] [, @<xSubscribed>]) -> <lSubscribed>
El funcionamiento es similar **hb_mutexSubscribe,** la única diferencia es que antes de comenzar, borra todas las notificaciones pendientes.

Antes de ver un par de ejemplos, vamos a seguir explicando con **hb_mutexNotify.**

hb_mutexNotify(<pMtx> [, <xVal>]) -> NIL
Emite una notificación a todos los threads esperando que están suscritos al mutex, <pMtx>.

Pero, por cada notificación sólo uno de los threads responderá, por lo tanto, tendremos que ir lanzando notificaciones para que diferentes threads vayan respondiendo.

Si no hay threads a la espera, se van quedando en una cola, que serán enviados conforme se vayan suscribiendo los threads que queramos.

Podemos enviar un parámetro, <xVal>, que como dijimos posteriormente, será recibido por la variable pasada por referencia en **hb_mutexSubscribe,** @<xSubscribed>
A tener en cuenta que no existe ningún tipo de relación entre el orden de suscripción y el orden de notificación.

Hay un ejemplo de esto en **harbour/contrib/mt/mttest07.prg** donde podéis observar lo aquí explicado, y aunque es 'duro' de entender si no sabemos nada, os pongo una versión 'modificada' con anotaciones y con pausas, para poder observar que está realizando y observar el comportamiento para intentar asimilarlo con tranquilidad ;-)

En el ejemplo de Harbour, podéis observar cómo se autoalimentan el hijo del padre y el padre del hijo a través de dos mutex**, s_ mtxJobs y s_mtxResults.**

```
#define N_THREADS 2
#define N_JOBS    5

static s_mtxJobs, s_Result := 0

proc main()
   local aThreads, i, nDigit
   cls

   ? "Main start"

   aThreads := {}
   s_mtxJobs := hb_mutexCreate()

   ? "Arrancamos Thread. "
   ? "Cuando se arranquen los hilos, estos estaran suspendidos a
la espera"
   ? "de la notificacion"
   ? "Pulsa una tecla para continuar"
   inkey( 0)

   for i := 1 to N_THREADS
      ? "Thread <" + hb_ntos( i ) + ">"
      aadd( aThreads, hb_threadStart( @thFunc() ) )
   next

   ? "Pulsa tecla para empezar a enviar Notificaciones"
   ? "En cada Notificacion, veras que se ejecuta <Run Thread by
Nofify> que"
   ? "parte debajo de hb_mutexSubscribe en la funcion thFunc()"
   ? ""
    inkey(0)

   nDigit := 1
   for i := 1 to N_JOBS
      ? "Notifica Job: <" + hb_ntos( i ) + "> Pulsa una tecla para
```

```
continuar"
      hb_mutexNotify( s_mtxJobs, nDigit )
     inkey( 0 )
      nDigit++
   next


   ? "Ahora vamos a enviar NIL, para salir del bucle while de la
funcion thFunc()."
   for i := 1 to N_THREADS
      hb_mutexNotify( s_mtxJobs, NIL )
      //?? "<" + hb_ntos( i ) + ">"
   next

   ? "Esperando a los threads..."
   aEval( aThreads, {| x | hb_threadJoin( x ) } )
   ? "Threads se unieron al principal"

   ? "Value Total:", s_Result
   ? "End of main"
return

proc thFunc()
   local xJob

   while .T.
      ? "Thread Subscribe: " + "0x" + hb_NumToHex( hb_threadSelf()
)
      hb_mutexSubscribe( s_mtxJobs,, @xJob )
      ? "Run Thread by Nofify"
      if xJob == NIL
          ?? "..... exit thread......."
         exit
      endif

      // Si no protejo la variable, el valor final puede tener un
valor inesperado
      hb_mutexLock( s_mtxJobs )
      s_Result += xJob
      hb_mutexUnLock( s_mtxJobs )

   enddo

return
```

hb_mutexNotifyAll(<pMtx> [, <xVal>]) -> NIL
Emite una notificación a todos los threads que estén a la espera.
Si no hay threads, no se ejecuta funcion alguna, no se agregan ni se quitan notificaciones que estuviesen en la cola.

```
#define N_THREADS 5

static s_mtxJobs

proc main()
   local aThreads, i
   cls

   ? "Main start"

   aThreads := {}
   s_mtxJobs := hb_mutexCreate()

   for i := 1 to N_THREADS
      aadd( aThreads, hb_threadStart( @thFunc() ) )
   next

   ? "Ahora vamos a enviar notificacion a todos los hilos."
   inkey( 1 )
   hb_mutexNotifyAll( s_mtxJobs )

   ? "Esperando a los threads..."
   aEval( aThreads, {| x | hb_threadJoin( x ) } )
   ? "Threads se unieron al principal"

   ? "End of main"
return

proc thFunc()

   ? "Thread Subscribe: " + "0x" + hb_NumToHex( hb_threadSelf() )
   hb_mutexSubscribe( s_mtxJobs )
   ? "Run Thread by Nofify & dead " + "0x" + hb_NumToHex(
hb_threadSelf() )

return
```

Y por último, vemos esta función;
hb_mutexEval(<pMtx>, <bCode> | <@sFunc()> [, <params,...>]) -> <xCodeResult>
Aparentemente, lo que hace es ejecutar un codeblock , protegiendo el contenido entre hilos.

NOTAS FINALES de Threads en Harbour.

En una clase, podemos hacer un method “synchronized”, simplemente poniendo SYNC en la declaración. Esto lo hace de semáforo por nosotros;

METHOD Inc() INLINE ::Sum++ **SYNC**

Recordad, es importante que los bloqueos sean lo más cortos en el tiempo, para evitar que los threads queden suspendidos esperando el desbloqueo.

Es importante ver los ejemplos de Harbour, son muy ilustrativos, aunque para los que no estamos acostumbrados, pueden ser un poco lioso al inicio

Listado de ejemplos de Harbour

harbour/contrib/mt/mttest1.prg

- Muestra el uso como detached local y valores complejos usando un thread

harbour/contrib/mt/mttest2.prg

- Muestra el uso del comando QUIT y el estamento ALWAYS en ejecución. El thread hijo usa QUITt antes que el principal

harbour/contrib/mt/mttest3.prg

- Muestra el uso del comando QUIT y el estamento ALWAYS en ejecución. El thread main usa QUITt antes que el thread hijo

harbour/contrib/mt/mttest4.prg

- Muestra el uso de variables protegidas y sin proteger

harbour/contrib/mt/mttest5.prg

- Muestra el uso thread static variables

harbour/contrib/mt/mttest6.prg

- Usando variables tipo memvar con hilos

harbour/contrib/mt/mttest7.prg

- Ejemplo de uso de Mutex para enviar/recibir mensajes entre hilos.

harbour/contrib/mt/mttest8.prg

- Usando el compartir variables de memoria entre hilos

harbour/contrib/mt/mttest9.prg

- Muestra como usar el mismo alias entre distintos hilos, usando estas 2 funciones **hb_dbRequest y hb_dbDetach.**

- **Nota:** Un regalo de despedida lo tenéis abajo ;-)

harbour/contrib/mt/mttest10.prg

- Muestra una consola por thread mostrando un browse

harbour/contrib/mt/mttest11.prg

- Muestra un ejemplo de un thread asincrono mostrando un reloj

harbour/contrib/mt/mttest12.prg

- Variables a nivel de hilo, pueden ser declaradas como;
  THREAD STATICt_Var1
  Esta variable es una variable static por cada hilo creado.

Mención especial para el código fuente **/contrib/httpd/core.prg,** donde podemos ver muchas de estas técnicas explicadas.

De regalo ;-)

Esto fue escrito hace ya 6 años para un sistema para hacer una indexación con threads, aunque en la mayoría de los casos existe penalización por tema de velocidad de disco, es bueno saber COMO se podría hacer.

En más de 500 servidores diferentes hemos encontrado que un máximo de 5 hilos por tabla y un índice a la vez, la ganancia es sustancial a un sistema monolítico.

```
/*
   Example multiThreads index.
   One thread by table , and one thread by index.
   2010 Rafa Carmona

   Thread Main
       |--------->  Thhread child table for test.dbf
       |                      |----> Thread child index fname
       |                      |
       |                      |----->Thread child index fcode
       |
       |--------->  Thhread child table for test2.dbf
                              |----->Thread child index fname2


 */
#include "hbthread.ch"

proc Main( uCreate )
    Local nSeconds
    Local aFiles  := { "test", "test2" }        // Arrays files dbf
    Local aNtx    := { { "fname", "fcode" },;   // files index for test
                     { "fName2" } }           // files index for test2
```

```
   Local aExpr   := { { "name", "code" },;
                    { "dtos(fecha)+str(code)" }  }     // Expresions
   Local cDbf

   if empty( lCreate )
     lCreate := "0"
   endif

   setmode( 25,80 )
   cls

   if uCreate = "1"
     ? "Create test.dbf and test2.dbf"
     dbCreate("test",{ {"name","C",1,0 },{"code","N",7,0 } } )
     use test
     while lastRec() < 1000000
          dbAppend()
          field->name := chr( recno() )
          field->code := recno()
     enddo
     close
     dbCreate("test2",{ {"fecha","D",8,0 },{"code","N",7,0 } } )
     use test2
     while lastRec() < 1000000
          dbAppend()
          field->fecha := date() + recno()
          field->code := recno()
     enddo
     close
   endif

   cls
   // Threads
   nSeconds := Seconds()
   for each cDbf in aFiles
     ? "Process.: " + cDbf
     hb_threadStart( @aCreateIndexe(), cDbf, aNtx[ cDbf:__enumindex ],
aExpr[ cDbf:__enumindex ], cDbf:__enumindex  )
   next

   ? "Wait for threads ...."
   hb_threadWaitForAll()

   ? hb_valTostr( Seconds() - nSeconds )

   ? "finish"

return

function aCreateIndexe( cFile, aNtx, aExpr, nPosDbf )
      Local nContador := 1
      Local cFileNtx, cExpr
      Local nLong := Len( aNtx )
      Local aThreads := {}
```

```
      Local cAlias

      use ( cFile )
      cAlias := alias()
      hb_dbDetach( cAlias )  // Libero el alias

      for each cFileNtx in aNtx
         cExpr  := aExpr[ cFileNtx:__enumindex ]
         nContador := 1
         nPos := cFileNtx:__enumindex
         aadd( aThreads, hb_threadStart( @crea(), cAlias,cExpr,
cFileNtx, nPos, nPosDbf ) )
      next

      aEval( aThreads, { |x| hb_threadJoin( x ) } )  // wait threads childs
      hb_dbRequest( cAlias, , , .T.)  // Restaura el alias
      close

RETURN NIL

proc crea( cAlias, cExpr, cFileNtx, nPos, nPosDbf )
     Local nContador := 1

      hb_dbRequest( cAlias, , , .T.)  // Restaura el alias

      INDEX ON &(cExpr) TO &(cFileNtx) ;
           EVAL {|| hb_dispOutAt( nPosDbf, iif( nPos = 1, 20, 40 ),
alltrim( hb_valtostr( nContador) ), "GR+/N" ), nContador += INT(
LASTREC() / 100 ) , .T. } ;
           EVERY INT( LASTREC() / 100 )

      hb_dbDetach( cAlias )         // Libera el alias


return
```