Sistemas Concurrentes y Distribuidos.
Seminario 1. Programación multihebra y sincronización con semáforos.

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Curso 13-14

Índice

Sistemas Concurrentes y Distribuidos.
Seminario 1. Programación multihebra y sincronización con semáforos.

Introducción

Este seminario tiene cuatro partes, inicialmente se repasa el concepto de hebra, a continuación se da una breve introducción a la interfaz (posix) de las librerías de hebras disponibles en linux. A continuación, se estudia el mecanismo de los semáforos como herramienta para solucionar problemas de sincronización y, por último, se hace una introducción a una librería para utilizar semáforos con hebras posix.

- El objetivo es conocer algunas llamadas básicas de dicho interfaz para el desarrollo de ejemplos sencillos de sincronización con hebras usando semáforos (práctica 1)
- Las partes relacionadas con hebras posix están basadas en el texto disponible en esta web: https://computing.llnl.gov/tutorials/pthreads/

Indice de la sección

Sección 1
Concepto e Implementaciones de Hebras

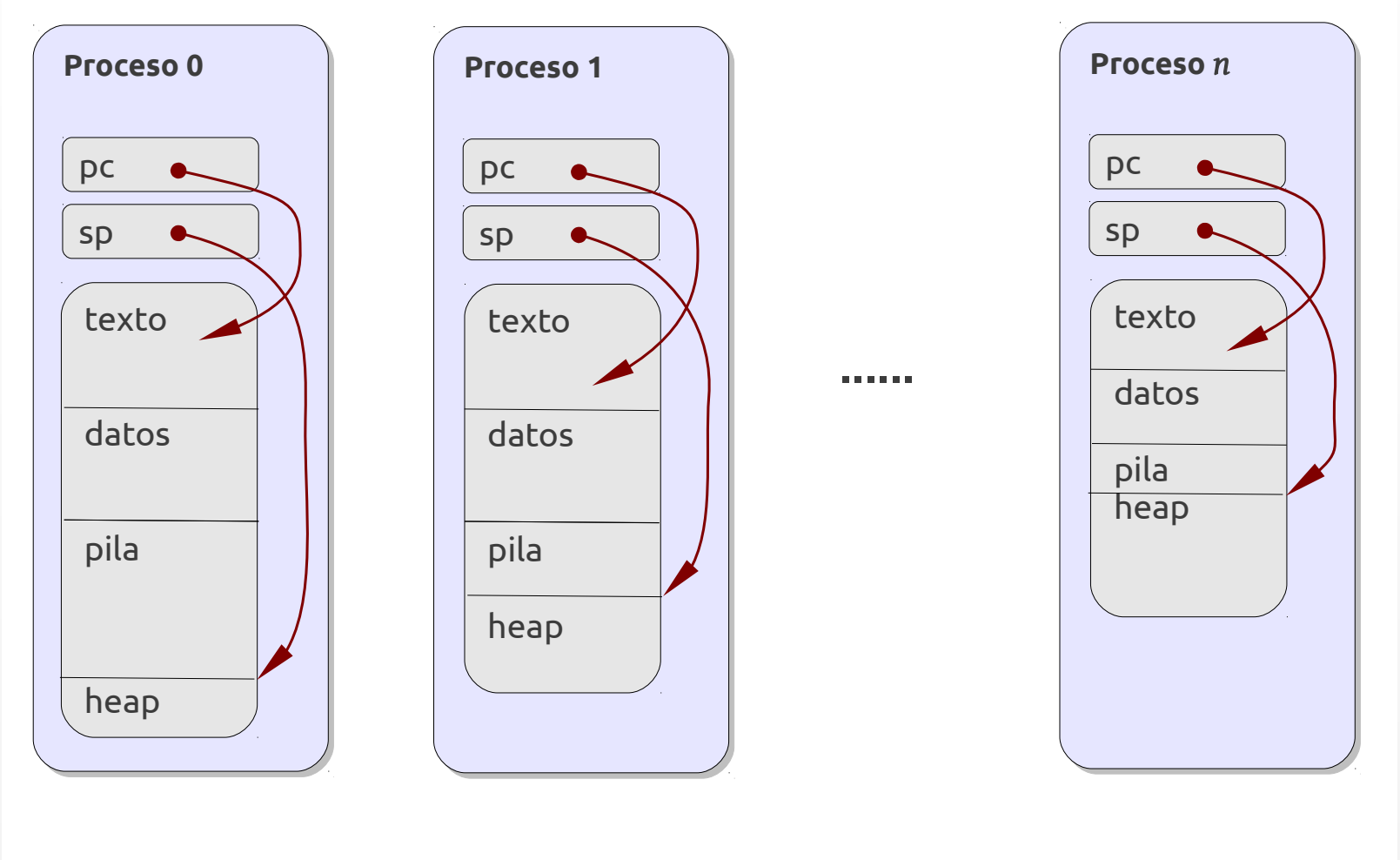
Procesos: estructura

En un sistema operativo pueden existir muchos procesos ejecutándose concurrentemente, cada proceso corresponde a un programa en ejecución y tiene su propio flujo de control.

- Cada proceso ocupa una zona de memoria con (al menos) estas partes:
  - **texto**: zona con la secuencia de instrucciones que se están ejecutando.
  - **datos**: espacio (de tamaño fijo) ocupado por variables globales.
  - **pila**: espacio (de tamaño cambiante) ocupado por variables locales.
  - **mem. dinámica** (***heap***): espacio ocupado por variables dinámicas.
- Cada proceso tiene asociados (entre otros) estos datos:
  - **contador de programa** (**pc**): dir. en memoria (zona de texto) de la siguiente instrucción a ejecutar.
  - **puntero de pila** (**sp**): dir. en memoria (zona de pila) de la última posición ocupada por la pila.

Diagrama de la estructura de los procesos

Podemos visualizarla (simplificadamente) como sigue:

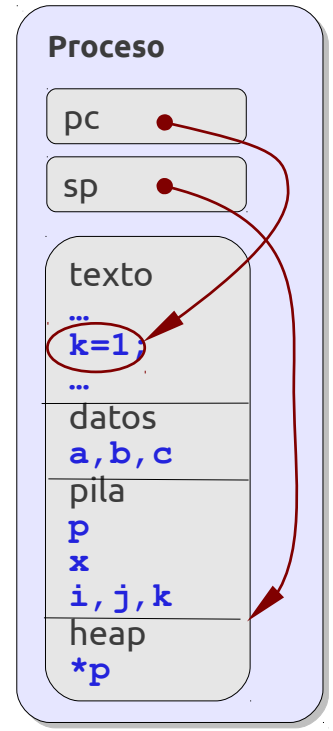


Ejemplo de un proceso

En el siguiente programa escrito en C/C++, el estado del proceso (durante la ejecución de **k=1;**) es el que se ve a la derecha:

```
int a,b,c ; // variables globales

void subprograma1()
{
   int i,j,k ; // vars. locales (1)
   k = 1 ;
}
void subprograma2()
{
   float x ; // vars. locales (2)
   subprograma1() ;
}
int main()
{
   char * p = new char ; // "p" local
   *p = 'a'; // "*p" en el heap
   subprograma2() ;
}
```



Procesos y hebras

La gestión de varios procesos no independientes (cooperantes) es muy útil pero consume una cantidad apreciable de recursos del SO:

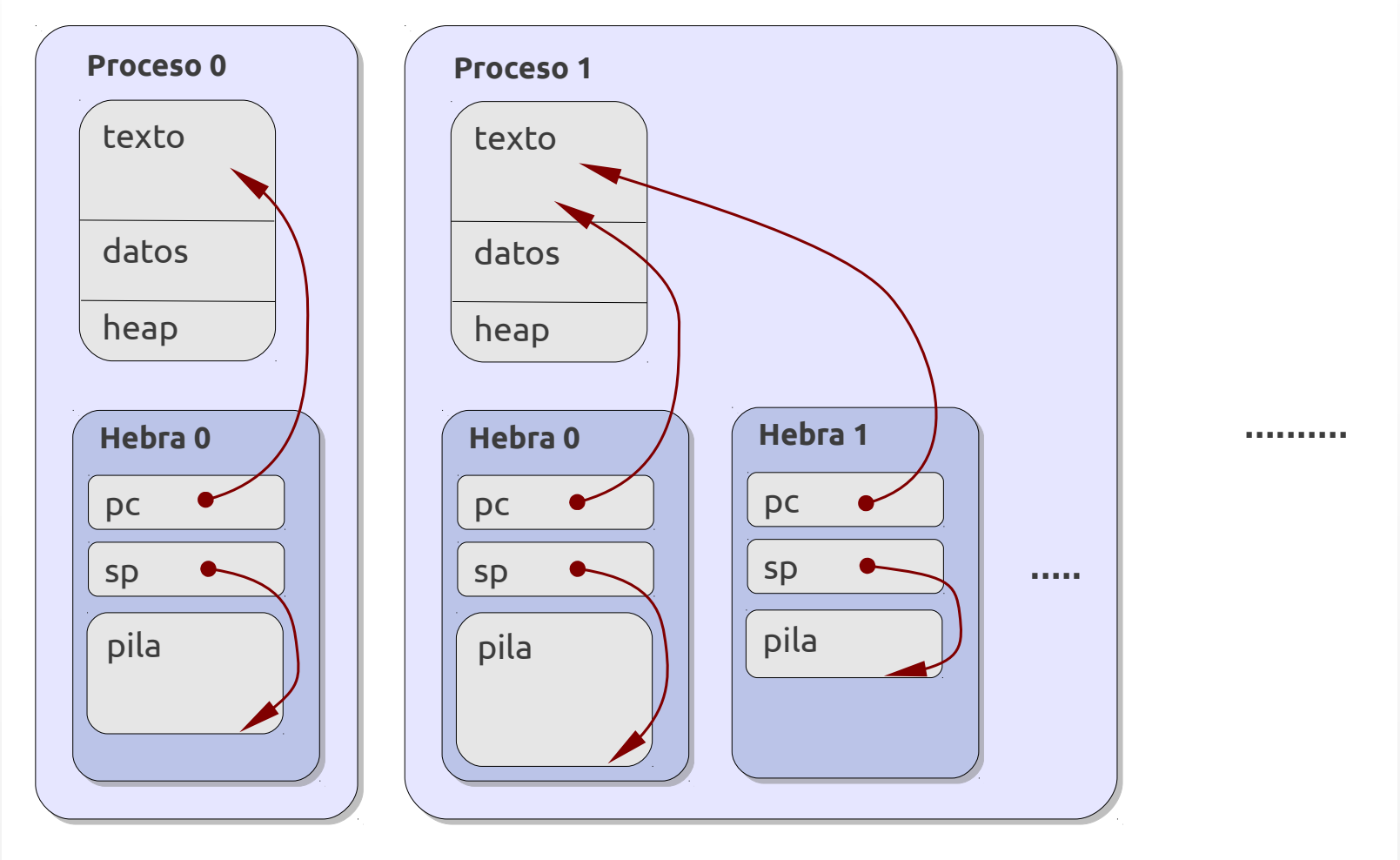
- Tiempo de procesamiento para repartir la CPU entre ellos
- Memoria con datos del SO relativos a cada proceso
- Tiempo y memoria para comunicaciones entre esos procesos

para mayor eficiencia en esta situación se diseñó el concepto de **hebra**:

- Un proceso puede contener una o varias hebras.
- Una hebra es un flujo de control en el texto (común) del proceso al que pertencen.
- Cada hebra tiene su propia pila (vars. locales), vacía al inicio.
- Las hebras de un proceso comparten la zona de datos (vars. globales), y el *heap*.

Diagrama de la estructura de procesos y hebras

Podriamos visualizarlos (simplificadamente) como sigue:



Inicio y finalización de hebras

Al inicio de un programa, existe una única hebra (que ejecuta la función **main** en C/C++). Durante la ejecución del programa:

- Una hebra A en ejecución puede crear otra hebra B en el mismo proceso de A.
- Para ello, A designa un subprograma f (una función C/C++) del texto del proceso, y después continúa su ejecución. La hebra B:
  - ejecuta el subprograma f concurrentemente con el resto de hebras.
  - termina normalmente cuando finaliza de ejecutar dicho subprograma
- Una hebra puede finalizar en cualquier momento (antes de terminar el subprograma con que se inició).
- Una hebra puede finalizar otra hebra en ejecución B, sin esperar que termine
- Una hebra puede esperar a que cualquier otra hebra en ejecución finalice.

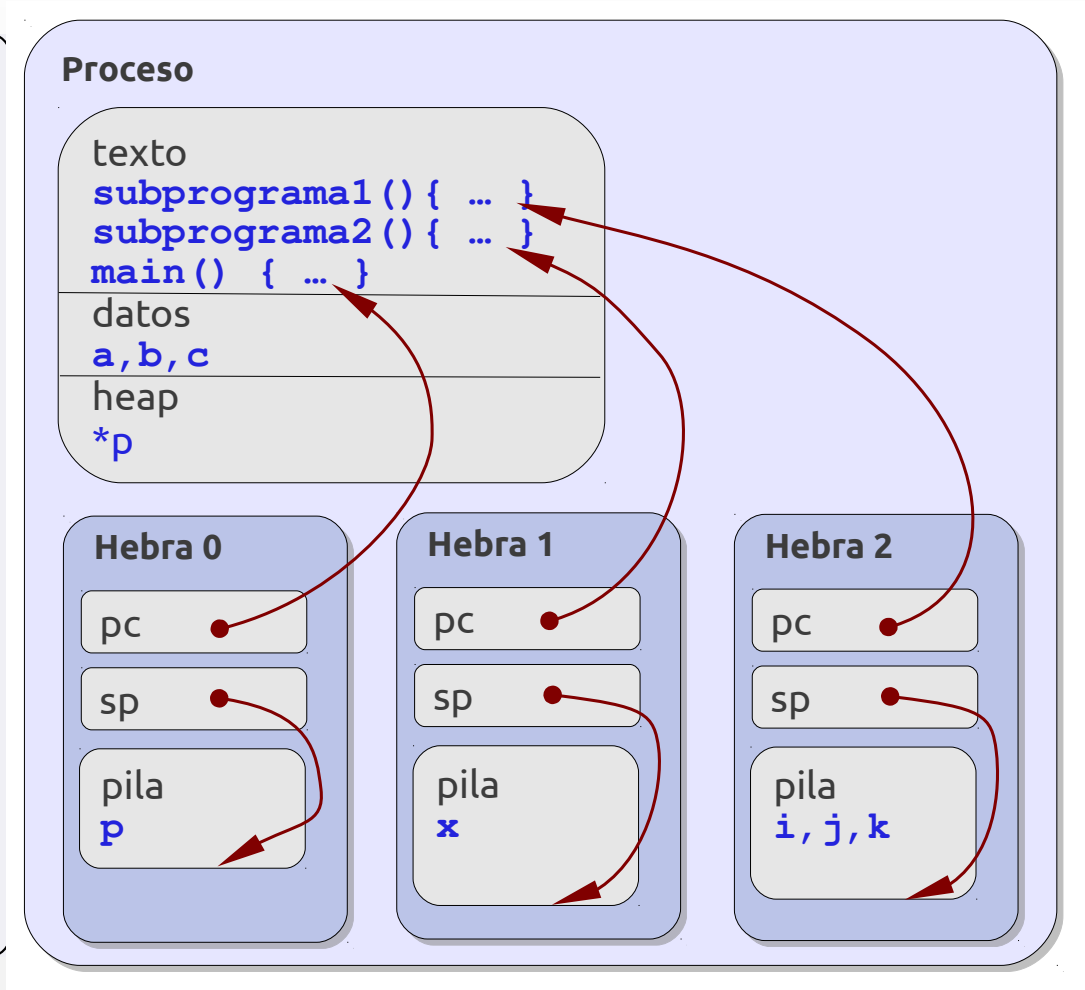
Ejemplo de estado de un proceso con tres hebras

En **main** se crean dos hebras, después se llega al estado que vemos:

```
int a,b,c ;

void subprograma1()
{
    int i,j,k ;
    // ...
}
void subprograma2()
{
    float x ;
    // ...
}

int main()
{
    char * p = new char ;
    // cr.he.subp.2
    // cr.he.subp.1
    // ...
}
```



Indice de la sección

Sección 2
Hebras POSIX

Introducción

En esta sección veremos algunas llamadas básicas de la parte de hebras de POSIX (IEEE std 1003.1), en la versión de 2004:

- El estándar POSIX define los parámetros y la semántica de un amplio conjunto de funciones para diversos servicios del SO a los programas de usuario.
  - http://pubs.opengroup.org/onlinepubs/009695399/

- Una parte de las llamadas de POSIX están dedicadas a gestión (creación, finalización, sincronización, etc...) de hebras, son las que aparecen aquí:
  - http://pubs.opengroup.org/onlinepubs/009695399/idx/threads.html

La librería NPTL

Los ejemplos se han probado usando la *Native POSIX Thread Library* (NTPL) en Linux (ubuntu), que implementa la parte de hebras de la versión de 2004 de POSIX

- Está disponible para Linux desde el kernel 2.6 (2004).
- En esta implementación una hebra POSIX se implementa usando una *hebra del kernel* de Linux (se dice que las hebras POSIX son 1-1).
- Como consecuencia de lo anterior, hebras distintas de un mismo proceso pueden ejecutarse en procesadores distintos,
- Por tanto, este tipo de hebras constituyen una herramienta ideal para el desarrollo de aplicaciones que pueden aprovechar el potencial de rendimiento que ofrecen los sistemas multiprocesador (o multinúcleo) con memoria compartida.

Creación de hebras con *pthread_create*

Esta función sirve para crear una nueva hebra, su declaración es como sigue:

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

Parámetros (más información aquí):

nombre	**tipo**	**descripción**
thread	**pthread_t** *	para referencias posteriores en las operaciones sobre la hebra
attr	**attr_t** *	atributos de la hebra (puede ser NULL para valores por def.)
start_routine	**void** **nombre*(**void** *)	función a ejecutar por la hebra
arg	**void** *	puntero que se pasa como parámetro para **start_routine** (puede ser NULL).

Ejemplo de creación de hebras.

En el siguiente ejemplo se crean dos hebras:

```
#include <iostream>
#include <pthread.h>
using namespace std ;

void* proc1( void* arg )
{  for( unsigned long i = 0 ; i < 5000 ; i++ )
      cout << "hebra 1, i == " << i << endl ;
   return NULL ;
}
void* proc2( void* arg )
{  for( unsigned long i = 0 ; i < 5000 ; i++ )
      cout << "hebra 2, i == " << i << endl ;
   return NULL ;
}
int main()
{
   pthread_t hebra1, hebra2 ;
   pthread_create(&hebra1,NULL,proc1,NULL);
   pthread_create(&hebra2,NULL,proc2,NULL);
   // ... finalizacion ....
}
```

Finalización de hebras

Una hebra finaliza cuando:

- termina la función designada en **pthread_create** (solo para hebras distintas de la inicial, en estos casos es equivalente a llamar a **pthread_exit**).
- llama a **pthread_exit** explícitamente
- es terminada por otra hebra con una llamada a **pthread_cancel**
- el proceso (de esta hebra) termina debido a una llamada a **exit** desde cualquier hebra de dicho proceso.
- la hebra inicial termina **main** sin haber llamado a **pthread_exit** (es decir, para permitir que las hebras creadas continúen al acabar la hebra inicial, es necesario que dicha hebra principal llame a **pthread_exit** antes del fin de **main**)

La función *pthread_exit*

La función **pthread_exit** causa la finalización de la hebra que la llama:

```
void pthread_exit( void* value_ptr );
```

Parámetros:

nombre	tipo	descripción
value_ptr	void *	puntero que recibirá la hebra que espere (vía *join*) la finalización de esta hebra (si hay alguna) (puede ser NULL)

más información:
http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_exit.html

Ejemplo de *pthread_exit*

El ejemplo anterior se puede completar así:

```
#include <iostream>
#include <pthread.h>
using namespace std ;

void* proc1( void* arg )
{  for( unsigned long i = 0 ; i < 5000 ; i++ )
      cout << "hebra 1, i == " << i << endl ;
   return NULL ;
}
void* proc2( void* arg )
{  for( unsigned long i = 0 ; i < 5000 ; i++ )
      cout << "hebra 2, i == " << i << endl ;
   return NULL ;
}
int main()
{
   pthread_t hebra1, hebra2 ;
   pthread_create(&hebra1,NULL,proc1,NULL);
   pthread_create(&hebra2,NULL,proc2,NULL);
   pthread_exit(NULL); // permite continuar a hebra1 y hebra2
}
```

La operación de unión.

POSIX provee diversos mecanismos para sincronizar hebras, veremos dos de ellos:

- Usando la operación de *unión* (*join*).
- Usando semáforos.

La operación de unión permite que (mediante una llamada a **pthread_join**) una hebra A espere a que otra hebra B termine:

- A es la hebra que invoca la unión, y B la hebra *objetivo*.
- Al finalizar la llamada, la hebra objetivo ha terminado con seguridad.
- Si B ya ha terminado, no se hace nada.
- Si la espera es necesaria, se produce sin que la hebra que llama (A) consuma CPU durante dicha espera (A queda suspendida).

La función *pthread_join*

La función **pthread_join** está declarada como sigue:

```
int pthread_join( pthread_t thread, void **value_ptr );
```

Parámetros y resultado:

nombre	**tipo**	**descripción**
thread	**thread_t**	identificador de la hebra objetivo
value_ptr	**void** **	puntero a la variable que recibirá el dato (de tipo **void** *) enviado por la hebra objetivo al finalizar (vía **return** o **pthread_exit**)
resultado	**int**	0 si no hay error, en caso contrario se devuelve un código de error.

más información aquí:
http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_join.html

Ejemplo de *pthread_join*

Puede ser útil, por ejemplo, para que la hebra principal realice algún procesamiento posterior a la finalización de las hebras:

```
#include <pthread.h>

void* proc1( void* arg ) { /* .... */ }
void* proc2( void* arg ) { /* .... */ }

int main()
{
   pthread_t hebra1, hebra2 ;

   pthread_create(&hebra1,NULL,proc1,NULL);
   pthread_create(&hebra2,NULL,proc2,NULL);

   pthread_join(hebra1,NULL);
   pthread_join(hebra2,NULL);

   // calculo posterior a la finalizacion de las hebras.....
}
```

Hebras idénticas

En muchos casos, un problema se puede resolver con un proceso en el que varias hebras distintas ejecutan el mismo algoritmo con distintos datos de entrada. En estos casos

- Es necesario que cada hebra reciba parámetros distintos
- Esto se puede hacer a traves del parámetro de tipo **void** * que recibe el subprograma que ejecuta una hebra.
- Dicho parámetro suele ser un índice o un puntero que permita a la hebra recuperar el conjunto de parámetros de entrada de una estructura de datos en memoria compartida, inicializada por la hebra principal (si es un índice entero, es necesario convertirlo hacia/desde el tipo **`void *`**)
- La estructura puede usarse para guardar también resultados de la hebra, o en general datos de la misma que deban ser leídos por otras.

Ejemplo básico de paso de parámetros a hebras

10 hebras ejecutan concurrentemente **func(0)**, **func(1)**, ..., **func(9)**

```
#include <iostream>
#include <pthread.h>
using namespace std ;

const unsigned num_hebras = 10 ;
unsigned long func( unsigned long n ) { /* ..calculo arbitrario.. */ }
void* proc( void* i ) { return (void *) func( (unsigned long)(i) ) ; }

int main()
{
   pthread_t id_hebra[num_hebras] ;
   for( unsigned i = 0 ; i < num_hebras ; i++ )
       pthread_create( &(id_hebra[i]), NULL, proc, (void *)i );

   for( unsigned i = 0 ; i < num_hebras ; i++ )
   {   unsigned long resultado ;
       pthread_join( id_hebra[i], (void **) (&resultado) );
       cout << "func(" <<  i << ") == " << resultado << endl  ;
   }
}
```

(nota: solo funciona si **sizeof(unsigned long)==sizeof(void *)**)

Cálculo de numérico de integrales

La programación concurrente puede ser usada para resolver más rápidamente multitud de problemas, entre ellos los que conllevan muchas operaciones con números flotantes

- Un ejemplo típico es el cálculo del valor I de la integral de una función f de variable real (entre 0 y 1, por ejemplo) y valores reales positivos:

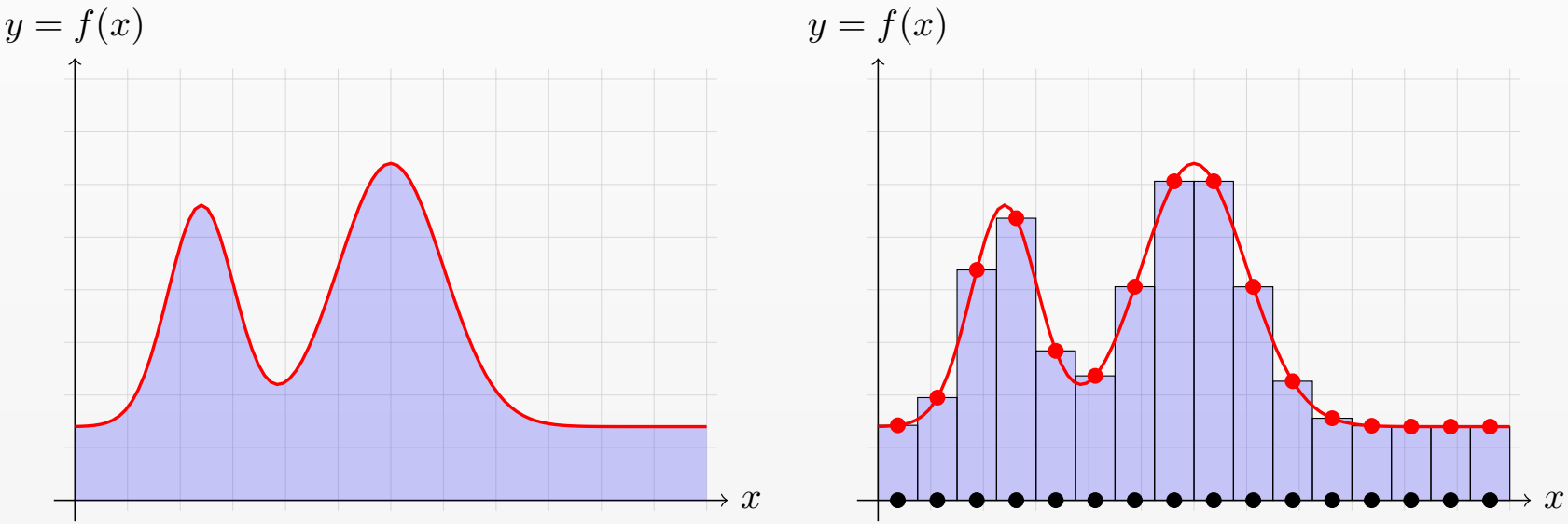
$$I = \int_0^1 f(x)\,dx$$

- El cálculo se puede hacer evaluando la función f en un conjunto de m puntos uniformemente espaciados en el intervalo $[0,1]$, y aproximando I como la media de todos esos valores:

$$I \approx \frac{1}{m}\sum_{i=0}^{m-1} f(x_i) \qquad \text{donde:} \qquad x_i = \frac{i+1/2}{m}$$

Interpretación geométrica

Aproximamos el área azul (es I) (izquierda), usando la suma de las áreas de las m barras (derecha):



- Cada punto de muestra es el valor x_i (puntos negros)
- Cada barra tiene el mismo ancho $1/m$, y su altura es $f(x_i)$.

Cálculo secuencial del número π

Para verificar la corrección del método, se puede usar una integral I con valor conocido. A modo de ejemplo, usaremos una función f cuya integral entre 0 y 1 es el número π:

$$I = \pi = \int_0^1 \frac{4}{1+x^2}\,dx \qquad \text{aquí} \quad f(x) = \frac{4}{1+x^2}$$

una implementación secuencial sencilla sería mediante esta función:

```
unsigned long m = ..... ;   // número de muestras

double f( double x )        // implementa función f
{ return 4.0/(1+x*x) ;      //    f(x) = 4/(1+x^2)
}
double calcular_integral_secuencial( )
{  double suma = 0.0 ;                          // inicializar suma
   for( unsigned long i = 0 ; i < m ; i++ )     // para cada i entre 0 y m-1
      suma += f( (i+0.5)/m );                   //      añadir f(x_i) a la suma actual
   return suma/m ;                              // devolver valor promedio de f
}
```

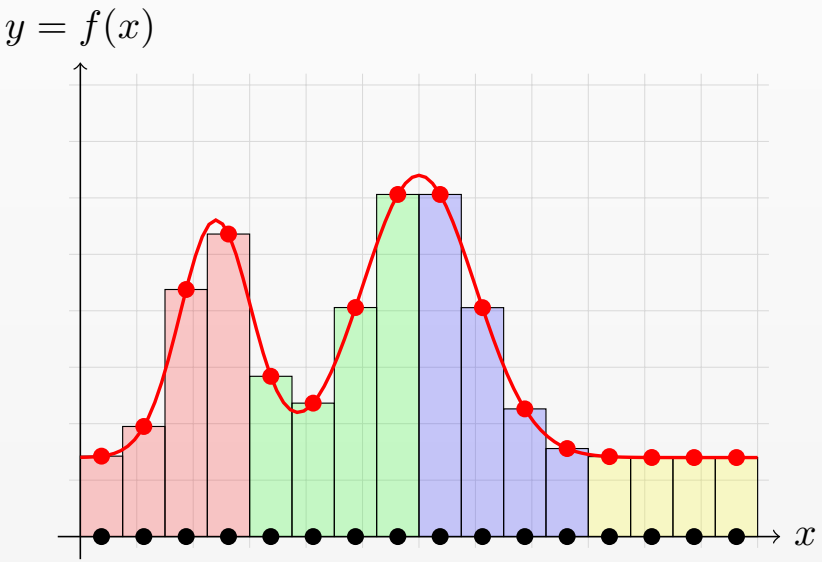
Versión concurrente de la integración

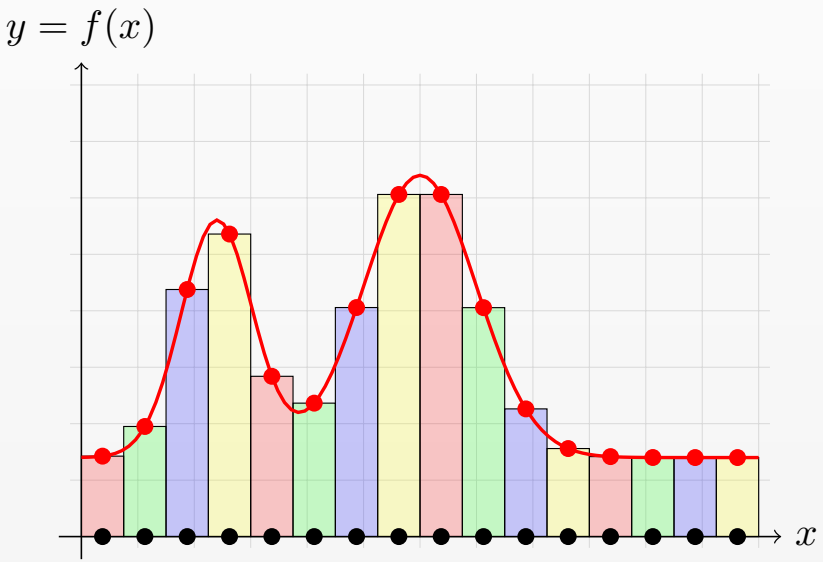
El cálculo citado anteriormente se puede hacer mediante un total de n hebras idénticas (asumimos que m es múltiplo de n)

- Cada una de las hebras evalua f en m/n puntos del dominio
- La cantidad de trabajo es similar para todas, y los cálculos son independientes.
- Cada hebra calcula la suma parcial de los valores de f
- La hebra principal recoge las sumas parciales y calcula la suma total.
- En un entorno con k procesadores o núcleos, el cálculo puede hacerse hasta k veces más rápido. Esta mejora ocurre solo para valores de m varios órdenes de magnitud más grandes que n.

Distribución de cálculos

Para distribuir los cálculos entre hebras, hay dos opciones simples, hacerlo de forma **contigua** (izquierda) o de forma **entrelazada** (derecha)





Cada valor $f(x_i)$ es calculado por:

- **Contigua:** la hebra número i/n.
- **Entrelazada:** la hebra número $i \mod n$.

Esquema de la implementación concurrente

```
const unsigned long m = .... ; // número de muestras
const unsigned long n = .... ; // número de hebras
double resultado_parcial[n] ;  // vector de resultados parciales

double f( double x )           // implementa función f:
{ return 4.0/(1+x*x) ;         //    f(x) = 4/(1+x^2)
}
void * funcion_hebra( void * ih_void ) // función que ejecuta cada hebra
{
   unsigned long ih = (unsigned long) ih_void ; // número o índice de esta hebra
   double sumap = 0.0 ;
   // calcular suma parcial en "sumap"
   .....
   resultado_parcial[ih] = sumap ; // guardar suma parcial en vector.
}
double calcular_integral_concurrente( )
{
   // crear y lanzar n hebras, cada una ejecuta "funcion_concurrente"
   .....
   // esperar (join) a que termine cada hebra, sumar su resultado
   .....
   // devolver resultado completo
   .....
}
```

Medición de tiempos

Para apreciar la reducción de tiempos que se consigue con la programación concurrente, se pueden medir los tiempos que tardan la versión secuencial y la concurrente. Para hacer esto con la máxima precisión, se puede usar la función `clock_gettime` en linux:

- Forma parte de POSIX, en concreto de la extensiones de tiempo real.
- Sirve para medir tiempo real transcurrido entre instantes de la ejecución de un programa con muy alta precisión (nanosegundos).
- Para facilitar su uso, en lugar de usarla directamente se pueden usar indirectamente a través de otras funciones (que se proporcionan) que permiten calcular ese tiempo como un valor real en segundos, haciendo abstracción de los detalles no relevantes.

Uso de las funciones de medición de tiempos

Para medir el tiempo que tarda un trozo de programa, se puede usar este esquema en C++

```
#include "fun_tiempos.h"
// ....
void funcion_cualquiera( /*.....*/ )
{  .....
   struct timespec inicio = ahora() ;  // inicio = inicio del tiempo a medir
   ....                                // actividad cuya duracion se quiere medir
   struct timespec fin = ahora() ;     // fin = fin del tiempo a medir
   ....
   cout << "tiempo transcurrido == "   // escribe resultados:
        << duracion( &inicio, &fin )   // tiempo en segundos entre "inicio" y "fin"
        << " seg." << endl ;
}
```

los archivos `fun_tiempo.h` (cabeceras) y `fun_tiempo.c` (implementación) se encuentran disponibles para los alumnos.

Compilando programas con hebras POSIX

Para compilar un archivo `ejemplo.cpp`, que use las funciones definidas en `fun_tiempos.c` (y use hebras posix), y obtener el archivo ejecutable `ejemplo`, podemos dar estos pasos:

```
gcc -g -c fun_tiempos.c    # compila "fun_tiempos.c" y genera "fun_tiempos.o"
g++ -g -c ejemplo.cpp      # compila "ejemplo.cpp" y genera "ejemplo.o"
g++ -o ejemplo ejemplo.o fun_tiempos.o -lrt -lpthread # enlaza, genera "ejemplo"
```

- el *switch* `-lrt` sirve para enlazar las librerías correspondientes a la extensión de tiempo real de POSIX (incluye `clock_gettime`)
- el *switch* `-lpthreads` sirve para incluir las funciones de hebras POSIX
- también es posible integrarlo todo en un `makefile` y usar `make`

Actividad: medición de tiempos de cálculo concurrente.

Como actividad para los alumnos en este seminario se propone realizar y ejecutar una implementación sencilla del cálculo concurrente del número π, tal y como hemos visto aquí:

- El programa aceptará como parámetros en la línea de comandos el número de hebras a lanzar y el número de muestras
- En la salida se presenta el valor exacto de π y el calculado (sirve para verificar si el programa es correcto, ya que deben diferir por muy poco para un número de muestras del orden de cientos o miles)
- Asimismo , el programa imprimirá la duración del cálculo concurrente y el secuencial.

Se debe razonar acerca de como el número de procesadores disponibles y el número de hebras afecta al tiempo del cálculo concurrente en relación al secuencial (en Linux, para conocer el número de CPUs disponibles y sus características, se puede ver el archivo `/proc/cpuinfo`)

Indice de la sección

Semáforos

Los **semáforos** constituyen un mecanismo de nivel medio que permite solucionar los problemas derivados de la ejecución concurrente de procesos no independientes. Sus características principales son:

- permite bloquear los procesos sin mantener ocupada la CPU
- resuelven fácilmente el problema de exclusión mutua con esquemas de uso sencillos
- se pueden usar para resolver problemas de sincronización (aunque en ocasiones los esquemas de uso son complejos)
- el mecanismo se implementa mediante instancias de una estructura de datos a las que se accede únicamente mediante subprogramas específicos.

Estructura de un semáforo

Un semáforo es un instancia de una estructura de datos (un registro) que contiene los siguientes elementos:

- Un conjunto de procesos bloqueados (se dice que están esperando en el semáforo).
- Un valor natural (entero no negativo), al que llamaremos *valor del semáforo*

Estas estructuras de datos residen en memoria compartida. Al principio de un programa que use semáforos, debe poder inicializarse cada uno de ellos:

- el conjunto de procesos asociados (bloqueados) estará vacío
- se deberá indicar un valor inicial del semáforo

Operaciones sobre los semáforos

Además de la inicialización, solo hay dos operaciones básicas que se pueden realizar sobre una variable de tipo semáforo (que llamamos s) :

- **sem_wait**(s)
  - Si el valor de s es mayor que cero, decrementar en una unidad dicho valor
  - Si el valor de s es cero, bloquear el proceso que la invoca en el conjunto de procesos bloqueados asociado a s
- **sem_signal**(s)
  - Si el conjunto de procesos bloqueados asociado a s no está vacío, desbloquear uno de dichos procesos.
  - Si el conjunto de procesos bloqueados asociado a s está vacío, incrementar en una unidad el valor de s.

En un semáforo cualquiera, estas operaciones se ejecutan de forma atómica, es decir, no puede haber dos procesos distintos ejecutando estas operaciones a la vez sobre un mismo semáforo (excluyendo el período de bloqueo que potencialmente conlleva la llamada a **sem_wait**).

Uso de semáforos para exclusión mutua

Los semáforos se pueden usar para EM usando un semáforo inicializado a 1, y haciendo **sem_wait** antes de la sección crítica y **sem_signal** después de la sección crítica:

```
{ variables compartidas y valores iniciales }
var sc_libre : semaphore := 1 ; { 1 si SC esta libre, 0 si SC esta ocupada }
```

```
while true do begin
  sem_wait( sc_libre    ); { esperar bloqueado hasta que 'sc_libre' sea 1 }
  { seccion critica :   ..... }
  sem_signal( sc_libre ); { desbl. proc. en espera o poner 'sc_libre' a 1 }
  { resto seccion:   ..... }
end
```

En cualquier instante de tiempo, la suma del valor del semáforo más el número de procesos en la SC es la unidad. Por tanto, solo puede haber 0 o 1 procesos en SC, y se cumple la exclusión mutua.

Uso de semáforos para sincronización

El problema del Productor-Consumidor se puede resolver fácilmente con semáforos:

```
{ variables compartidas }
var
 x              : integer ;           { contiene cada valor producido      }
 producidos     : integer   := 0 ;    { numero de valores producidos       }
 consumidos     : integer   := 0 ;    { numero de valores consumidos       }
 puede_leer     : semaphore := 0 ;    { 1 si se puede leer "x", 0 si no    }
 puede_escribir : semaphore := 1 ;    { 1 si se puede escribir "x", 0 si no }
```

```
{ Proceso Productor (calcula "x") }
var a : integer ;
begin
   while true begin
      a := ProducirValor() ;
      sem_wait( puede_escribir  );
      x := a ; { sentencia E }
      producidos := producidos + 1 ;
      sem_signal( puede_leer ) ;
   end
end
```

```
{ Proceso Consumidor (lee "x") }
var b : integer ;
begin
   while true do begin
      sem_wait( puede_leer ) ;
      b := x ; { sentencia L }
      consumidos := consumidos + 1 ;
      sem_signal( puede_escribir ) ;
      UsarValor(b) ;
   end
end
```

Indice de la sección

Sección 4
Sincronización de hebras con semáforos POSIX

Introducción

Una parte de las llamadas del estándard POSIX son útiles para gestionar semáforos para sincronización de hebras o de procesos. Veremos las llamadas básicas que permiten sincronizar hebras en un proceso. Son las siguientes:

- **sem_init**: inicializa un semáforo (dando el valor inicial)
- **sem_wait**: realiza la operación **wait** sobre un semáforo
- **sem_post**: realiza la operación **signal** sobre un semáforo
- **sem_getvalue**: devuelve el valor actual de un semáforo
- **sem_destroy**: destruye un semáforo y libera la memoria ocupada.

La función *sem_init*

La función **sem_init** está declarada como sigue:

```
int sem_init( sem_t* sem, int pshared, unsigned value );
```

Parámetros y resultado:

nombre	**tipo**	**descripción**
sem	**sem_t** *	puntero al identificador del semáforo
pshared	**int**	distinto de cero solo si el semáforo será compartido con otros procesos.
value	**unsigned**	valor inicial
resultado	**int**	0 si se ha inicializado el semáforo, en caso contrario es un código de error.

más información aquí:
http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_init.html

La función *sem_wait*

La función **sem_wait** está declarada como sigue:

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

Parámetros y resultado:

nombre	**tipo**	**descripción**
sem	**sem_t** *	puntero al identificador del semáforo
resultado	**int**	0 solo si no ha habido error

más información aquí:
http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_wait.html

La función *sem_post*

La función **sem_post** está declarada como sigue:

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

Parámetros y resultado:

nombre	tipo	descripción
sem	**sem_t** *	puntero al identificador del semáforo
resultado	**int**	0 solo si no ha habido error

Nota: la selección de la hebra a desbloquear (si hay alguna) depende de los parametros de *scheduling* asignados a la hebra (normalmente es FIFO).
Más información aquí:
http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_post.html

La función *sem_getvalue*

La función **sem_getvalue** está declarada como sigue:

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

Parámetros y resultado:

nombre	tipo	descripción
sem	**sem_t** *	puntero al identificador del semáforo
sval	**int** *	puntero a la variable donde se almacena el valor
resultado	**int**	0 solo si no ha habido error

Nota: al finalizar la llamada, el valor del semáforo podría ya ser distinto del valor leído (es importante tenerlo en cuenta).
Más información aquí:
http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_getvalue.html

La función *sem_destroy*

La función **sem_destroy** está declarada como sigue:

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

Parámetros y resultado:

nombre	tipo	descripción
sem	**sem_t** *	puntero al identificador del semáforo
resultado	**int**	0 solo si no ha habido error

Solo se aplica a semáforos inicializados con **sem_init** en los cuales no hay hebras esperando. Después de destruir un semáforo, se puede reusar haciendo **sem_init**.
Más información aquí:
http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_destroy.html

Ejemplo de exclusión mutua con semáforos POSIX

Los semáforos se pueden usar para exclusión mutua, por ejemplo,
En este ejemplo, se usa un semáforo (de nombre **mutex**) inicializado a 1, para escribir en la salida estándar una línea completa sin interrupciones.

```
#include <iostream>
#include <pthread.h>
#include <semaphore.h>
using namespace std ;

sem_t mutex ; // semaforo en memoria compartida

void* proc( void* p )
{
   sem_wait( &mutex );
      cout << "hebra numero: " << ((unsigned long) p) << ". " << endl ;
   sem_post( &mutex );
   return NULL ;
}

....
```

Ejemplo de exclusión mutua con semáforos POSIX (2)

El procedimiento principal debe inicializar el semáforo y crear las hebras, como se incluye aquí:

```
...

int main()
{
   const unsigned num_hebras = 50 ;
   pthread_t id_hebra[num_hebras] ;

   sem_init( &mutex, 0, 1 );

   for( unsigned i = 0 ; i < num_hebras ; i++ )
       pthread_create( &(id_hebra[i]), NULL, proc, (void *)i );

   for( unsigned i = 0 ; i < num_hebras ; i++ )
       pthread_join( id_hebra[i], NULL );

   sem_destroy( &mutex );

}
```

Ejemplo de sincronización con semáforos POSIX

En este otro ejemplo, hay una hebra que escribe una variable global y otra que la lee (cada una en un bucle). Se usan dos semáforos para evitar dos lecturas o dos escrituras seguidas, y además otro para exclusión mútua en las escrituras al terminal:

```
sem_t
   puede_escribir,  // inicializado a 1
   puede_leer,      // inicializado a 0
   mutex ;          // inicializado a 1

unsigned long
   valor_compartido ; // valor para escribir o leer

const unsigned long
   num_iter = 10000 ; // numero de iteraciones

......
```

Ejemplo de sincronización con semáforos POSIX (2)

La hebra escritora espera que se pueda escribir, entonces escribe y señala que se puede leer:

```
...

void* escribir( void* p )
{
   unsigned long contador = 0 ;

   for( unsigned long i = 0 ; i < num_iter ; i++ )
   {
      contador = contador + 1 ; // genera un nuevo valor
      sem_wait( &puede_escribir ) ;
      valor_compartido = contador ; // escribe el valor
      sem_post( &puede_leer ) ;

      sem_wait( &mutex ) ;
      cout << "valor escrito == " << contador << endl << flush ;
      sem_post( &mutex ) ;
   }
   return NULL ;
}

...
```

Ejemplo de sincronización con semáforos POSIX (3)

La hebra lectora espera a que se pueda leer, entonces lee y señala que se puede escribir:

```
....

void* leer( void* p )
{
   unsigned long  valor_leido ;

   for( unsigned long i = 0 ; i < num_iter ; i++ )
   {
      sem_wait( &puede_leer ) ;
      valor_leido = valor_compartido ; // lee el valor generado
      sem_post( &puede_escribir ) ;

      sem_wait( &mutex ) ;
      cout << "valor leido   == " << valor_leido << endl << flush ;
      sem_post( &mutex ) ;
   }
   return NULL ;
}

...
```

Ejemplo de sincronización con semáforos POSIX (4)

El procedimiento **main** crea los semáforos y hebras y espera que terminen:

```
....

int main()
{
   pthread_t hebra_escritora, hebra_lectora ;

   sem_init( &mutex, 0, 1 ) ; // semaforo para EM: inicializado a 1

   sem_init( &puede_escribir, 0, 1); // inicialmente se puede escribir
   sem_init( &puede_leer,     0, 0); // inicialmente no se puede leer

   pthread_create( &hebra_escritora, NULL, escribir, NULL );
   pthread_create( &hebra_lectora,   NULL, leer,     NULL );

   pthread_join( hebra_escritora, NULL ) ;
   pthread_join( hebra_lectora,   NULL ) ;

   sem_destroy( &puede_escribir );
   sem_destroy( &puede_leer );
}
```