Sistemas Paralelos
Sistemas Distribuidos y Paralelos

Clase 3

Facultad de Informática

UNLP

Programación en Memoria Compartida

Modelo de Memoria Compartida

- En el modelo de memoria compartida, múltiples tareas se ejecutan en paralelo.
- Cada tarea puede tener memoria local “exclusiva” y todas (o por grupos) pueden acceder a un mapa de memoria común.
- Programación Paralela: sincronización y comunicación de datos.
- La comunicación y sincronización de estas tareas se realiza escribiendo y leyendo áreas de memoria compartida.
- Al usar un Espacio de Direcciones Compartido
  - Comunicación implícita (acceso a todas las memorias).
  - Concurrencia y sincronización explícitas. Minimizar overhead.

Modelo de Memoria Compartida

- El problema de la sincronización en memoria compartida (con sus efectos indeseables como los deadlocks) es responsabilidad del programador, utilizando las herramientas que provea el lenguaje. Toda sincronización disminuye la eficiencia.
- El programador en general NO maneja la distribución de los datos ni lo relacionado a la comunicación de los mismos.
- La localidad de los datos será muy importante en la perfomance (en particular en arquitecturas NUMA con memoria compartida). En algunos lenguajes el programador podrá actuar sobre la localidad de los datos, en otros tendrá que re-estructurar el código.
- El programador logra muy buena transparencia en un modelo de memoria compartida. La ubicación de los datos, su replicación y su migración son transparentes. Esto hace muy difícil la predicción de perfomance a partir de la lectura del algoritmo.

Programación en Plataformas de Espacio de Direcciones Compartido

- Los modelos de programación proveen un soporte para expresar la concurrencia y sincronización:
  - Los modelos basados en procesos suponen datos locales (privados) de cada proceso.
  - Los modelos basados en threads o procesos "livianos" suponen que toda la memoria es global (*Pthreads*).
  - Los modelos basados en directivas extienden el modelo basado en threads para facilitar su creación y sincronización (*OpenMP*).

Pthreads

Fundamentos de Threads

- Un *thread* es un único hilo de control en el flujo de un programa. Por ejemplo (multiplicación de matrices):

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] = dot_product( get_row(a, row), get_col(b, col));
```

- Puede transformarse en:

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] = create_thread (dot_product(get_row(a,row),get_col(b,col)));
```

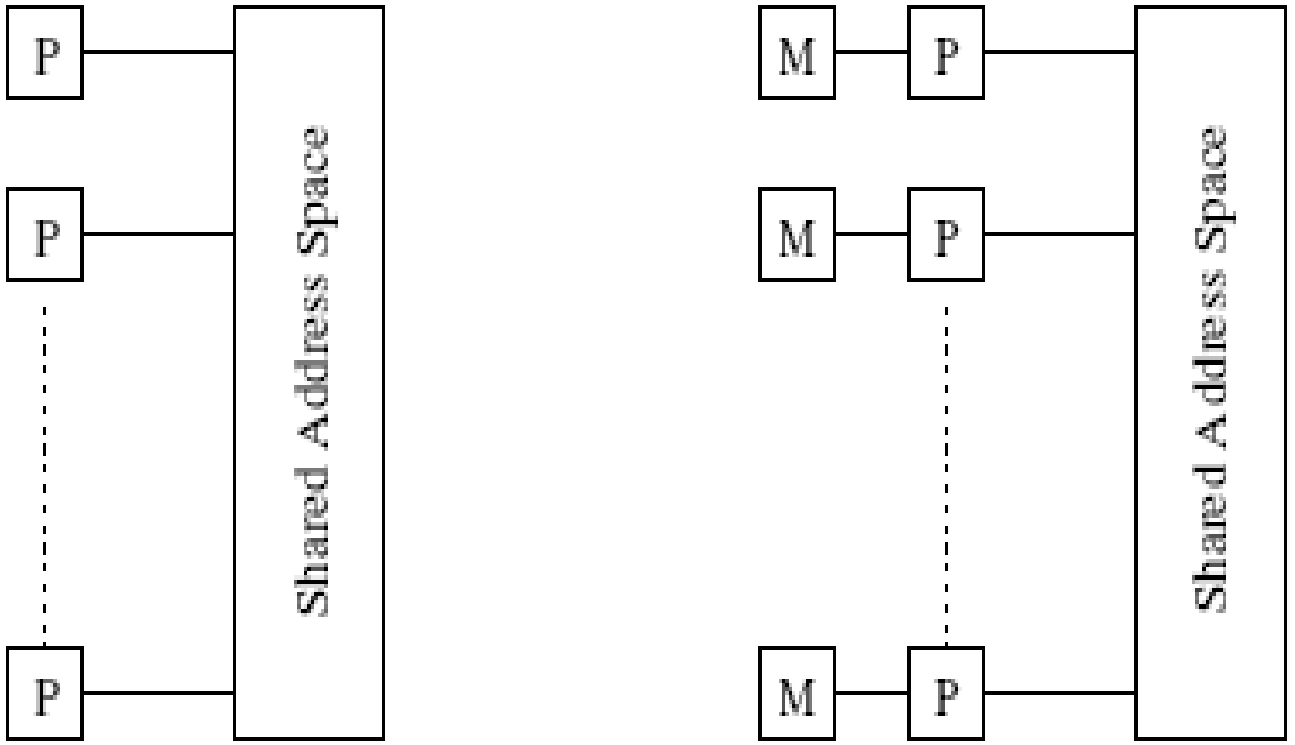
- En este caso el *Thread* funciona como una instancia de una función que retorna antes que la función se haya terminado de ejecutar.

- Diferencia entre *Threads* y *Procesos*. Importancia de la reducción de tiempo en el context-switching entre threads.

Modelo lógico de memoria de Threads

- Cualquier thread puede acceder a toda la memoria disponible en la máquina.
- La pila (*Stack*) correspondiente al CALL de la función es generalmente tratado como local al thread por razones de "vida".
- Lo anterior implica que tenemos un modelo lógico de máquina con memoria global accesible por todos los threads y memoria local para los *stacks*. Da idea de arquitectura UMA para la memoria global.
- El modelo resulta poco rendidor si la memoria está físicamente distribuida. Arquitectura NUMA para la memoria global.

Modelo lógico de memoria de Threads



Modelo de Threads

- La aplicación más común del modelo de threads aparece en los servidores. Múltiples servicios se instancian como threads (dando lugar a un programa concurrente en el servidor). Estos threads compiten por los recursos que administra el servidor.

- La implementación de threads se hace a menudo por bibliotecas, no por directivas al compilador. Estas bibliotecas están bastante optimizadas y permiten buena portabilidad.

Utilidad de los Threads

- ***Portabilidad***: permite migrar aplicaciones entre arquitecturas.
- ***Scheduling y balance de carga***: soporta mapping dinámico a nivel de sistema que minimiza el overhead por ociosidad (lo hacen las APIs de threads).
- ***Soporte para el manejo de la latencia de memoria***: oculta la latencia por accesos a memoria al permitir multithread.
- ***Facilidad de uso y programación de aplicaciones***: más fácil de programar que Pasaje de Mensajes (no requiere el manejo de la comunicación de datos). ¿Performance?

POSIX – API de Threads

- Numerosas APIs para el manejo de Threads.
- Normalmente llamada Pthreads, POSIX a emergido como un API estandard para manejo de Threads, provista por la mayoría de los vendedores.
- Los conceptos que se discutirán son independientes de la API y pueden ser igualmente válidos para utilizar JAVA Threads, NT Threads, Solaris Threads, etc.
- Funciones reentrantes.

Pthreads - Creación y terminación

- Pthreads provee funciones básicas para especificar concurrencia:

```
#include <pthread.h>

int pthread_create ( pthread_t   *thread_handle,
                     const pthread_attr_t *attribute,
                     void * (*thread_function)(void *),
                     void *arg);
```

La función pthread_create invoca a la función thread_function como un thread

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

int pthread_cancel (pthread_t thread);

int pthread_exit (void *res);
```

- El “main” debe esperar a que todos los threads terminen.

Pthreads - Creación y terminación

Ejemplo: *cálculo de PI*

```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);
....
main()
  {    ...
       pthread_t   p_threads[MAX_THREADS];
       pthread_attr_t    attr;
       pthread_attr_init (&attr);
       for (i=0; i< num_threads; i++)
          { hits[i] = i;
            pthread_create(&p_threads[i], &attr, compute_pi, (void *) &hits[i]);
           }
       for (i=0; i< num_threads; i++)
          { pthread_join(p_threads[i], NULL);
            total_hits += hits[i];
           }
       ....
    }
```

Pthreads - Creación y terminación

```
void *compute_pi (void *s)
  {  int seed, i, *hit_pointer;
     double rand_no_x, rand_no_y;
     int local_hits;
     hit_pointer = (int *) s;
     seed = *hit_pointer;
     local_hits = 0;

     for (i = 0; i < sample_points_per_thread; i++)
        { rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
          rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
          if (((rand_no_x - 0.5)*(rand_no_x - 0.5)+(rand_no_y - 0.5)*(rand_no_y - 0.5)) < 0.25)
                        local_hits ++;
          seed *= i;
        }

     *hit_pointer = local_hits;
     pthread_exit(0);
   }
```

- ¿Si acumulo directamente sobre ***total_hits***?

Pthreads – Primitivas de Sincronización
Exclusión mutua

- Comunicación implícita → se pone el esfuerzo en sincronizar tareas concurrentes.
- Cuando múltiples threads tratan de manejar los mismos datos, el resultado puede ser incoherente si no se sincroniza adecuadamente:

```
if (mi_costo < mejor_costo) mejor_costo = mi_costo;
```

- Si tenemos 2 threads y el valor inicial de mejor_costo (memoria compartida) es 100, y cada thread tiene su mi_costo en 50 y 75, el valor a guardar en memoria global podría ser cualquiera de los dos.
- Esto depende del scheduling de los threads → Race condition

Pthreads – Primitivas de Sincronización
Exclusión mutua

- El código anterior funciona bien si fuese una sentencia atómica → corresponde a una sección crítica.
- Las secciones críticas se implementan en Pthreads utilizando mutex_locks (bloqueo por exclusión mutua).
- Mutex_locks tienen dos estados: locked (bloqueado) and unlocked (desbloqueado). En cualquier instante, sólo UN thread puede bloquear un mutex_lock. Lock es una operación atómica .
- Para entrar en la sección crítica un Thread debe lograr tener control del mutex_lock (bloquearlo).
- Cuando un Thread sale de la SC debe desbloquear el mutex_lock.
- Todos los mutex_lock deben inicializarse como desbloqueados.

Pthreads – Primitivas de Sincronización
Exclusión mutua

- La API Pthreads provee las siguientes funciones para manejar los mutex-locks:

```
int pthread_mutex_lock ( pthread_mutex_t *mutex);

int pthread_mutex_unlock (pthread_mutex_t *mutex);

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

Pthreads – Primitivas de Sincronización
Exclusión mutua

Ahora se pude escribir el código para calcular el mínimo de una lista de números:

```
pthread_mutex_t     minimum_value_lock;
        ...
        main()
         {  ....
            pthread_mutex_init(&minimum_value_lock, NULL);
            ....
          }

        void *find_min(void *list_ptr)
         { ....
           pthread_mutex_lock(&minimum_value_lock);
           if (my_min < minimum_value) minimum_value = my_min;
           pthread_mutex_unlock(&minimum_value_lock);
           ...
          }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Exclusión Mutua

El escenario de productores-consumidores impone las siguientes restricciones:

- Un thread productor no debe sobrescribir el buffer compartido cuando el elemento anterior no ha sido tomado por un thread consumidor.
- Un thread consumidor no puede tomar nada de la estructura compartida hasta no estar seguro de que se ha producido algo anteriormente.
- Los consumidores deben excluirse entre sí.
- Los productores deben excluirse entre sí.
- En este ejemplo el buffer es de tamaño 1.

Pthreads - Primitivas de Sincronización Productores/Consumidores con Exclusión Mutua

Main de la solución al problema de productores-consumidores.

```
pthread_mutex_t    task_queue_lock;
int task_available;
...
main()
   { task_available = 0;
     pthread_init ();
     pthread_mutex_init(&task_queue_lock, NULL);

       /* Create y join de threads productores y consumidores*/
    }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Exclusión Mutua

Código para los *productores*.

```
void *producer(void *producer_thread_data)
  {  ....
     while (!done())
           { inserted = 0;
             create_task(&my_task);
             while (inserted = = 0)
               {  pthread_mutex_lock(&task_queue_lock);
                  if (task_available == 0)
                    {  insert_into_queue(my_task);
                       task_available = 1;
                       inserted = 1;
                     }
                  pthread_mutex_unlock(&task_queue_lock);
                }
           }
  }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Exclusión Mutua

Código para los *consumidores*.

```
void *consumer(void *consumer_thread_data)
  { int extracted;
    struct task my_task;
     /* local data structure declarations */
     while (!done())
        { extracted = 0;
          while (extracted == 0)
              {  pthread_mutex_lock(&task_queue_lock);
                 if (task_available == 1)
                    { extract_from_queue(&my_task);
                      task_available = 0;
                      extracted = 1;
                     }
                 pthread_mutex_unlock(&task_queue_lock);
              }
           process_task(my_task);
        }
   }
```

Pthreads - Primitivas de Sincronización
Tipos de Exclusión Mutua (*Mutex*)

- Pthreads soporta tres tipos de Mutexs (Locks): Normal, Recursive y Error Check
  - ✓ Un Mutex con el atributo Normal NO permite que un thread que lo tienen bloqueado vuelva a hacer un lock sobre él (deadlock).
  - ✓ Un Mutex con el atributo Recursive SI permite que un thread que lo tienen bloqueado vuelva a hacer un lock sobre él. Simplemente incrementa una cuenta de control.
  - ✓ Un Mutex con el atributo ErrorCheck responde con un reporte de error al intento de un segundo bloqueo por el mismo thread.
- El tipo de Mutex puede setearse entre los atributos antes de su inicialización.

Pthreads - Primitivas de Sincronización
Overhead de Bloqueos por Exclusión Mutua

- Los locks representan puntos de serialización → si dentro de las secciones críticas ponemos segmentos largos de programa tendremos una degradación importante de perfomance.

- A menudo se puede reducir el overhead por espera ociosa, utilizando la función pthread_mutex_trylock. Retorna el control informando si pudo hacer o no el lock.

  int pthread_mutex_trylock (pthread_mutex_t *mutex_lock).

  - ✓ Evita tiempos ociosos.
  - ✓ Menos costoso por no tener que manejar las colas de espera.

Pthreads - Primitivas de Sincronización Reduciendo el Overhead de Exclusión Mutua

Encontrar *k* ocurrencias de un determinado valor.

```
void *find_entries (void *start_pointer)    /* Es la función del Thread */
     { struct database_record *next_record;
       int count;
       current_pointer = start_pointer;
       while (count < requested_number_of_records)
          { next_record = find_next_entry(current_pointer);
            count = output_record(next_record);
          }
     }

int output_record(struct database_record *record_ptr)
    { int count;
      pthread_mutex_lock(&output_count_lock);
      output_count ++;
      count = output_count;
      pthread_mutex_unlock(&output_count_lock);
      if (count <= requested_number_of_records) print_record(record_ptr);
      return (count);
    }
```

Pthreads - Primitivas de Sincronización Reduciendo el Overhead de Exclusión Mutua

Reducir el overhead para el mismo problema.

```
int output_record(struct database_record *record_ptr)
    { int count;
      int lock_status;
      lock_status=pthread_mutex_trylock(&output_count_lock);
      if (lock_status == EBUSY)
        { insert_into_local_list(record_ptr);
          return (0);
        }
      else
       { count = output_count;
         output_count += number_on_local_list + 1;
         pthread_mutex_unlock(&output_count_lock);
         print_records(record_ptr, local_list, requested_number_of_records - count);
         return (count + number_on_local_list + 1);
       }
    }
```

Pthreads - Primitivas de Sincronización
Variables Condición

- Podemos utilizar variables de condición para que un thread se autobloquee hasta que se alcance un estado determinado del programa.
- Cada variable de condición estará asociada con un predicado. Cuando el predicado se convierte en verdadero (TRUE) la variable de condición da una señal para el/los threads que están esperando por el cambio de estado de la condición.
- Una única variable de condición puede asociarse a varios predicados (difícil el debug).
- Una variable de condición siempre tiene un mutex asociada a ella. Cada thread bloquea este mutex y testea el predicado definido sobre la variable compartida.
- Si el predicado es falso, el thread espera en la variable condición utilizando la función pthread_cond_wait (NO USA CPU).

Pthreads - Primitivas de Sincronización
Variables Condición

La API Pthreads provee las siguientes funciones para manejar las variables condición:

```
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)

int pthread_cond_signal (pthread_cond_t *cond)

int pthread_cond_broadcast (pthread_cond_t *cond)

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

int pthread_cond_destroy (pthread_cond_t *cond)
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Variables Condición

Main de la solución al problema de productores-consumidores.

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
...
main()
   { ...
     task_available = 0;
     pthread_init();
     pthread_cond_init(&cond_queue_empty, NULL);
     pthread_cond_init(&cond_queue_full, NULL);
     pthread_mutex_init(&task_queue_cond_lock, NULL);
     ...
   }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Variables Condición

Código para los *productores*.

```
void *producer(void *producer_thread_data)
   { int inserted;

     while (!done())
       { create_task ();
         pthread_mutex_lock (&task_queue_cond_lock);
         while (task_available = = 1)
                pthread_cond_wait (&cond_queue_empty, &task_queue_cond_lock);
          insert_into_queue ();
          task_available = 1;
          pthread_cond_signal (&cond_queue_full);
          pthread_mutex_unlock (&task_queue_cond_lock);
       }
  }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Variables Condición

Código para los *consumidores*.

```
void *consumer(void *consumer_thread_data)
   { while (!done())
        { pthread_mutex_lock (&task_queue_cond_lock);
          while (task_available = = 0)
                pthread_cond_wait (&cond_queue_full, &task_queue_cond_lock);
          my_task = extract_from_queue ();
          task_available = 0;
          pthread_cond_signal (&cond_queue_empty);
          pthread_mutex_unlock (&task_queue_cond_lock);
          process_task (my_task);
        }
   }
```

Pthreads – Atributos y sincronización

- La API Pthreads permite que se pueda cambiar los atributos por defecto de las entidades, utilizando attributes objects.
- Un attribute object es una estructura de datos que describe las propiedades de la entidad en cuestión (thread, mutex, variable de condición).
- Una vez que estas propiedades están establecidas, el attribute object es pasado al método que inicializa la entidad.
- Ventajas
  - ✓ Esta posibilidad mejora la modularidad.
  - ✓ Facilidad de modificación del código.

Pthreads – Atributos para Threads

- La API Pthreads provee las siguientes funciones para manejar los atributos para Threads:

```
int pthread_attr_init (pthread_attr_t  *attr);

int pthread_attr_destroy (pthread_attr_t  *attr);
```

- Las propiedades asociadas con el *attribute object* pueden ser cambiadas con las siguientes funciones:

pthread_attr_setdetachstate

pthread_attr_setguardsize_np

pthread_attr_setstacksize

pthread_attr_setinheritsched

pthread_attr_setschedpolicy

pthread_attr_setschedparam

Pthreads – Atributos para *Mutex*

- La API Pthreads provee las siguientes funciones para manejar los atributos para Mutex:

```
int pthread_mutexattr_init (pthread_mutexattr_t  *attr);

int pthread_mutexattr_settype_np ( pthread_mutexattr_t *attr,   int type);
```

- Aquí *type* especifica el tipo de *mutex* y puede tomar los valores:
  - PTHREAD_MUTEX_NORMAL_NP
  - PTHREAD_MUTEX_RECURSIVE_NP
  - PTHREAD_MUTEX_ERRORCHECK_NP

Pthreads – Constructores de Sincronización Compuestos

- Pthreads provee soporte para un conjunto básico de operaciones de sincronización.
- Se pueden construir operaciones de más alto nivel, a partir de los constructores básicos de sincronización.
- Bloqueos para Lecturas-Escrituras y Barreras.

Pthreads – Bloqueos para Lecturas-Escrituras

- En muchas aplicaciones una estructura de datos es leída (consultada) con mayor frecuencia que escrita (modificada). En tales casos puede convenir diferenciar los bloqueos de lectura y de escritura.
- Un bloqueo de lectura es habilitado aún cuando haya otros threads realizando un bloqueo de lectura.
- Si hubiera un bloqueo de escritura sobre el dato o en cola, el thread deberá realizar un condition wait.
- Múltiples threads pidiendo un bloqueo de escritura se encolarán con alguna política (normalmente FIFO)
- Con esta descripción, se han definido funciones para:
  - ✓ Bloqueos de lectura (mylib_rwlock_rlock)
  - ✓ Bloqueos de escritura (mylib_rwlock_wlock)
  - ✓ Desbloqueo (mylib_rwlock_unlock)

Pthreads – Bloqueos para Lecturas-Escrituras

- El bloqueos de lectura-escritura se basan en la estructura de datos *mylib_rwlock_t* con los siguientes componentes:
  - ✓ Cantidad de lectores.
  - ✓ Un entero (0/1) indicando si hay un escritor presente.
  - ✓ Una variable de condición *readers_proceed* que es TRUE cuando los lectores pueden avanzar.
  - ✓ Una variable de condición *writer_proceed* que es TRUE cuando uno de los escritores puede avanzar.
  - ✓ Cantidad de escritores en espera *pending_writers*.
  - ✓ Una variable mutex *read_write_lock* asociada con la estructura de datos compartida.

Pthreads – Bloqueos para Lecturas-Escrituras

Tipo de datos e inicialización

```
typedef struct   { int readers;
                   int writer;
                   pthread_cond_t readers_proceed;
                   pthread_cond_t writer_proceed;
                   int pending_writers;
                   pthread_mutex_t read_write_lock;
                 }   mylib_rwlock_t;

void mylib_rwlock_init (mylib_rwlock_t *L)
          {  L-> readers = L-> writer = L-> pending_writers = 0;
             pthread_mutex_init(&(L-> read_write_lock), NULL);
             pthread_cond_init(&(L-> readers_proceed), NULL);
             pthread_cond_init(&(L-> writer_proceed), NULL);
          }
```

Pthreads – Bloqueos para Lecturas-Escrituras

Bloqueo de lectura

```
void mylib_rwlock_rlock(mylib_rwlock_t *L)
 {  /* Si hay un escritos escribiendo o esperando se duerme en la variable condición,
       sino incrementa la cantidad de lectores */

    pthread_mutex_lock(&(L-> read_write_lock));
    while ((L-> pending_writers > 0) || (L-> writer > 0))
            pthread_cond_wait(&(L-> readers_proceed),  &(L-> read_write_lock));
    L-> readers ++;
    pthread_mutex_unlock(&(L-> read_write_lock));
}
```

Pthreads – Bloqueos para Lecturas-Escrituras

Bloqueo de escritura

```
void mylib_rwlock_wlock (mylib_rwlock_t *L)
  { /* Mientras haya un escritor o un lector incrementa los escritores pendientes y
       se duerme, sino incrementa la cantidad de escritores */

       pthread_mutex_lock(&(L-> read_write_lock));

       while ((L-> writer > 0) || (L-> readers > 0))
           { L-> pending_writers ++;
             pthread_cond_wait(&(L-> writer_proceed), &(L-> read_write_lock));
             L-> pending_writers --;
           }
        L-> writer ++;

       pthread_mutex_unlock(&(L-> read_write_lock));
}
```

Pthreads – Bloqueos para Lecturas-Escrituras

Desbloqueo

```
void mylib_rwlock_unlock(mylib_rwlock_t *L)
{     pthread_mutex_lock(&(L-> read_write_lock));

      if (L-> writer > 0)    L-> writer = 0;
      else if (L-> readers > 0)    L-> readers --;

      if ((L-> readers == 0) && (L-> pending_writers > 0))
                              pthread_cond_signal(&(L-> writer_proceed))
      else  if (L-> pending_writers == 0)
                              pthread_cond_broadcast(&(L-> readers_proceed));

      pthread_mutex_unlock(&(L-> read_write_lock));
}
```

Pthreads – Barreras

- Podemos construir una barrera para Threads.
- Para implementar las barreras, necesitaremos un contador, un *mutex* y una *variable de condición*.
- Un entero se utilizará para saber cuantos threads han alcanzado la barrera.
- Si la cuenta es menor que el total de threads involucrados en la barrera, el thread debe ejecutar un *wait condicional*.
- El último thread que alcanza la barrera (y setea el contador al valor del número de threads) despierta a todos los demás utilizando un *broadcast* de la variable de condición.

Pthreads – Barreras

Tipo de datos e inicialización

```
typedef struct {  pthread_mutex_t   count_lock;
                  pthread_cond_t   ok_to_proceed;
                  int count;
               }  mylib_barrier_t;


void mylib_init_barrier (mylib_barrier_t *b)
          {  b -> count = 0;
             pthread_mutex_init(&(b -> count_lock), NULL);
             pthread_cond_init(&(b -> ok_to_proceed), NULL);
           }
```

Pthreads – Barreras

Código de la barrera

```
void mylib_barrier (mylib_barrier_t *b, int num_threads)
    {   pthread_mutex_lock(&(b -> count_lock));
        b -> count ++;
        if (b -> count == num_threads)
              {  b -> count = 0;
                 pthread_cond_broadcast(&(b -> ok_to_proceed));
               }
        else
               pthread_cond_wait(&(b -> ok_to_proceed),&(b -> count_lock));

        pthread_mutex_unlock(&(b -> count_lock));
    }
```

Pthreads – Barreras

- La barrera mostrada anteriormente es llamada “lineal”.
- El tiempo de ejecución de esta función es O(n) para n threads.
- Esta implementación podría mejorarse en tiempo utilizando múltiples variables para la barrera, organizadas en un árbol. Barrera logarítmica.