

Concurrencia y Paralelismo

Clase 5



Facultad de Informática
UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- ◆ Librería Pthreads:

https://drive.google.com/uc?id=1T2TBPMgrc1ax0z_vrDWMiSaSXJOTJ9GR&export=download

- ◆ Semáforos y monitores en Pthreads:

<https://drive.google.com/uc?id=1gLhJodkLt8ukBGLPNgauoHRX2HGX0sN9&export=download>

- ◆ Librería OpenMP:

<https://drive.google.com/uc?id=1hbMW8CiW9yYtmdkrPyDPhP74l-jKagQd&export=download>

Librerías para Cómputo Paralelo

➤ Para memoria compartida.

- Basadas en threads: *Pthreads*
- Basadas en Directivas: *OpenMP*



Pthreads

Pthreads

Thread: proceso “liviano” que tiene su propio contador de programa y su pila de ejecución, pero no controla el “contexto pesado” (por ejemplo, las tablas de página).

- Algunos sistemas operativos y lenguajes proveen mecanismos para permitir la programación de aplicaciones “multithreading”.
- En principio estos mecanismos fueron heterogéneos y poco portables \Rightarrow a mediados de los 90 la organización POSIX auspició el desarrollo de una biblioteca en C para multithreading (*Pthreads*).
- Con esta biblioteca se pueden crear threads, asignarles atributos, darlos por terminados, identificarlos, etc.

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);
```

```
int pthread_exit (void *res);
```

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

```
int pthread_cancel (pthread_t thread);
```

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

Pthreads – Primitivas de Sincronización

Exclusión mutua

- Las secciones críticas se implementan en Pthreads utilizando *mutex locks* (bloqueo por exclusión mutua) por medio de variables *mutex*.
- Una variable *mutex* tienen dos estados: locked (bloqueado) and unlocked (desbloqueado). En cualquier instante, sólo UN thread puede bloquear un *mutex*. *Lock* es una operación atómica.
- Para entrar en la sección crítica un Thread debe lograr tener control del *mutex* (bloquearlo).
- Cuando un Thread sale de la SC debe desbloquear el *mutex*.
- Todos los *mutex* deben inicializarse como desbloqueados.

Pthreads – Primitivas de Sincronización

Exclusión mutua

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

```
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

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  mutex;  
int hayElemento;  
tipo_elemento Buffer;  
...  
main()  
{ hayElemento= 0;  
  pthread_init ();  
  pthread_mutex_init(&mutex, NULL);  
  
  /* Create y join de threads productores y consumidores*/  
}
```

Pthreads - Primitivas de Sincronización

Productores/Consumidores con Exclusión Mutua

Código para los *productores*.

```
void *productor (void *datos)
{
    tipo_elemento elem;
    int ok;
    ...
    while (true)
    {
        ok = 0;
        generar_elemento(&elem);
        while (ok == 0)
        {
            pthread_mutex_lock(&mutex);
            if (hayElemento == 0)
            {
                Buffer = elem;
                hayElemento = 1;
                ok = 1;
            }
            pthread_mutex_unlock(&mutex);
        }
    }
}
```

Pthreads - Primitivas de Sincronización

Productores/Consumidores con Exclusión Mutua

Código para los *consumidores*.

```
void *consumidor(void *datos)
{
    int ok;
    tipo_elemento elem;
    ....
    while (true)
    {
        ok = 0;
        while (ok == 0)
        {
            pthread_mutex_lock(&mutex);
            if (hayElemento == 1)
            {
                elem = Buffer;
                hayElemento = 0;
                ok = 1;
            }
            pthread_mutex_unlock(&mutex);
        }
        procesar_elemento(elem);
    }
}
```

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 performance.
- 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

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 vacio, lleno;
pthread_mutex_t mutex;
int hayElemento;
tipo_elemento Buffer;
...
main()
{ ...
  hayElemento= 0;
  pthread_init();
  pthread_cond_init(&vacio, NULL);
  pthread_cond_init(&lleno, NULL);
  pthread_mutex_init(&mutex, NULL);
  ...
}
```

Pthreads - Primitivas de Sincronización

Productores/Consumidores con Variables Condición

Código para los *productores*.

```
void *productor(void *datos)
{
    tipo_element elem;

    while (true)
    {
        generar_elemento(elem);
        pthread_mutex_lock (&mutex);
        while (hayElemento == 1)
            pthread_cond_wait (&vacio, &mutex);
        Buffer = elem;
        hayElemento = 1;
        pthread_cond_signal (&lleno);
        pthread_mutex_unlock (&mutex);
    }
}
```

Pthreads - Primitivas de Sincronización

Productores/Consumidores con Variables Condición

Código para los *consumidores*.

```
void *consumidor(void *datos)
{
    tipo_element elem;

    while (true)
    {
        pthread_mutex_lock (&mutex);
        while (hayElemento == 0)
            pthread_cond_wait (&lleno, &mutex);
        elem= Buffer;
        hayElemento = 0;
        pthread_cond_signal (&vacio);
        pthread_mutex_unlock (&mutex);
        procesar_elemento(elem);
    }
}
```

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



Semáforos en Pthreads

Semáforos con Pthreads

- Los threads pueden sincronizar por semáforos (*librería semaphore.h*).
- Declaración y operaciones con semáforos en Pthreads:
 - ✓ `sem_t semaforo` → se declaran globales a los threads.
 - ✓ `sem_init (&semaforo, alcance, inicial)` → en esta operación se inicializa el semáforo `semaforo`. Inicial es el valor con que se inicializa el semáforo. Alcance indica si es compartido por los hilos de un único proceso (0) o por los de todos los procesos ($\neq 0$).
 - ✓ `sem_wait(&semaforo)` → equivale al P.
 - ✓ `sem_post(&semaforo)` → equivale al V.
 - ✓ Existen funciones extras para: wait condicional, obtener el valor de un semáforo y destruir un semáforo (ESTE TIPO DE FUNCIONES EXTRAS NO SE PUEDEN USAR EN LA PRÁCTICA DE LA MATERIA).

Semáforos con Pthreads

Productor / consumidor

- Las funciones de **Productor** y **Consumidor** serán ejecutadas por threads independientes.
- Acceden a un buffer compartido (**datos**).
- El productor deposita una secuencia de enteros de **1** a **numItems** en el buffer.
- El consumidor busca estos valores y los suma.
- Los semáforos **vacio** y **lleno** garantizan el acceso alternativo de productor y consumidor sobre el buffer.

```
#include <pthread.h>
#include <semaphore.h>
#define SHARED 1

void *Productor(void *);
void *Consumidor(void *);

sem_t vacio, lleno;
int dato, numItems;
```

```
int main(int argc, char * argv[ ])
{
    .....
    sem_init (&vacio, SHARED, 1);
    sem_init (&lleno, SHARED, 0);
    .....
    pthread_create (&pid, &attr, Productor, NULL);
    pthread_create (&cid, &attr, Consumidor, NULL);
    pthread_join (pid, NULL);
    pthread_join (cid, NULL);
}
```

Semáforos con Pthreads

Productor / consumidor

```
void *Productor (void *arg)
{ int item;
  for (item = 1; item <= numItems; item++)
    { sem_wait(&vacio);
      dato = item;
      sem_post(&lleno);
    }
  pthreads_exit();
}

void *Consumidor (void *arg)
{ int total = 0, item, aux;
  for (item = 1; item <= numItems; item++)
    { sem_wait(&lleno);
      aux = dato;
      sem_post(&vacio);
      total = total + aux;
    }
  printf("TOTAL: %d\n", total);
  pthreads_exit();
}
```



Monitores en Pthreads

Monitores con Pthreads

- Pthreads no permite manejar la Exclusión Mutua por medio de las variables *mutex*.
- Pthreads nos permite manejar la Sincronización por Condición utilizando *variables condición* para que un *thread* se auto bloquee hasta que se alcance un estado determinado del programa. Una variable de condición siempre tiene un *mutex* asociada a ella.
- Pthreads no posee “*Monitores*”, pero con las dos herramientas que mencionamos se puede simular el uso de monitores: con *mutex* se hace la exclusión mutua que nos brindaba implícitamente el monitor, y con las variables condición la sincronización.
 - ✓ El acceso exclusivo al monitor se simula usando una variable *mutex* la cual se bloquea antes del llamada al *procedure* y se desbloquea al terminar el mismo (una variable *mutex* diferente para cada monitor).
 - ✓ Cada llamado de un proceso a un *procedure* de un monitor debe ser reemplazado por el código de ese *procedure*.

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

- Esta es la solución que vimos en la teoría de monitores para el problema de lectores/escritores. Ahora veremos como simularla en *Pthreads*.

monitor Controlador

```
{ int nr = 0, nw = 0, dr = 0, dw = 0;
  cond ok_leer, ok_escribir;

  procedure pedido_leer( )
  { if (nw > 0)
    { dr = dr + 1;
      wait (ok_leer);
    }
    else nr = nr + 1;
  }

  procedure libera_leer( )
  { nr = nr - 1;
    if (nr == 0 and dw > 0)
    { dw = dw - 1;
      signal (ok_escribir);
      nw = nw + 1;
    }
  }

  procedure pedido_escribir( )
  { if (nr > 0 OR nw > 0)
    { dw = dw + 1;
      wait (ok_escribir);
    }
    else nw = nw + 1;
  }

  procedure libera_escribir( )
  { if (dw > 0)
    { dw = dw - 1;
      signal (ok_escribir);
    }
    else { nw = nw - 1;
          if (dr > 0)
          { nr = dr;
            dr = 0;
            signal_all (ok_leer);
          }
        }
  }
}
```

Process lector[id: 0..L-1]

```
{ while (true)
  { Controlador.pedido_leer();
    //Leer sobre la BD
    Controlador.libera_leer();
  };
}
```

Process escritor[id: 0..E-1]

```
{ while (true)
  { Controlador.pedido_escribir();
    //Leer sobre la BD
    Controlador.libera_escribir();
  };
}
```

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

```
#include <pthread.h>

void *Escritor(void *);
void *Lector(void *);

int main(int argc, char * argv[ ])
{ int nr = 0, nw = 0, dr = 0, dw = 0, i;
  pthread_cond_t  ok_leer, ok_escribir;
  pthread_t  lectores[L], escritores[E];
  .....
  pthread_init();
  pthread_cond_init(&ok_leer, NULL);
  pthread_cond_init(&ok_escribir, NULL);
  .....
  for (i=0; i<E;i++) pthread_create (&escritores[i], &attr, Escritor, NULL);
  for (i=0; i<L;i++) pthread_create (&lectores[i], &attr, Lector, NULL);
}
```

Por cada monitor se requiere un *mutex* para
simular la EM implícita de los mismos.

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

- Como hay solo un monitor se pone solo una variable *mutex* que además de declarar se inicializa en el *main* del programa.

```
#include <pthread.h>

void *Escritor(void *);
void *Lector(void *);

int main(int argc, char * argv[ ])
{ int nr = 0, nw = 0, dr = 0, dw = 0, i;
  pthread_cond_t  ok_leer, ok_escribir;
  pthread_t  lectores[L], escritores[E];
  pthreads_mutex_t mutex;
  .....
  pthread_init();
  pthread_cond_init(&ok_leer, NULL);
  pthread_cond_init(&ok_escribir, NULL);
  pthread_mutex_init(&mutex, NULL);
  .....
  for (i=0; i<E;i++) pthread_create (&escritores[i], &attr, Escritor, NULL);
  for (i=0; i<L;i++) pthread_create (&lectores[i], &attr, Lector, NULL);
}
```


Monitores con Pthreads

Ejemplo: *Lectores y escritores*

- Se agrega en los procesos el bloque y desbloqueo de *mutex* en los llamados a los procedure del monitor.

```
void *lector (void*)  
{ while (true)  
  { pthread_mutex_lock (&mutex);  
    Controlador.pedido_leer();  
    pthread_mutex_unlock (&mutex);  
    //Leer sobre la BD  
    pthread_mutex_lock (&mutex);  
    Controlador.libera_leer();  
    pthread_mutex_unlock (&mutex);  
  }  
}
```

```
void *escritor (void*)  
{ while (true)  
  { pthread_mutex_lock (&mutex);  
    Controlador.pedido_escribir();  
    pthread_mutex_unlock (&mutex);  
    //Leer sobre la BD  
    pthread_mutex_lock (&mutex);  
    Controlador.libera_escribir ();  
    pthread_mutex_unlock (&mutex);  
  }  
}
```

El próximo paso es reemplazar los llamados de los procedimientos por el código de los mismos

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

```
void *escriptor (void*)
{ while (true)
  { pthread_mutex_lock (&mutex);
    if (nr>0 OR nw>0)
      { dw = dw + 1;
        pthread_cond_wait (& ok_escribir, &mutex);
      }
    else nw = nw + 1;
    pthread_mutex_unlock (&mutex);
    //Escribe sobre la BD
    pthread_mutex_lock (&mutex);
    if (dw > 0)
      { dw = dw - 1;
        pthread_cond_signal(&ok_escribir);
      }
    else
      { nw = nw - 1;
        if (dr > 0)
          { nr = dr;
            dr = 0;
            pthread_cond_broadcast(&ok_leer);
          };
      };
    pthread_mutex_unlock (&mutex);
  };
  pthreads_exit();
};
```

```
void *lector(void*)
{ while (true)
  { pthread_mutex_lock (&mutex);
    if (nw>0)
      { dr = dr + 1;
        pthread_cond_wait (& ok_leer, &mutex);
      }
    else nr = nr + 1;
    pthread_mutex_unlock (&mutex);
    //Leer sobre la BD
    pthread_mutex_lock (&mutex);
    nr = nr - 1;
    if (nr == 0 and dw > 0)
      { dw = dw - 1;
        pthread_cond_signal(&ok_escribir);
        nw = nw + 1;
      };
    pthread_mutex_unlock (&mutex);
  };
  pthreads_exit();
};
```



OpenMP

Modelo de Programación Basado en Directivas: *OpenMP*

- Modelos basados en threads → primitivas de bajo nivel.
- Modelos basados en directivas → constructores de alto nivel.
 - Liberan al programador del manejo de Threads.
 - OpenMP es un estándar para programación paralela basada en directivas.
 - OpenMP es una API que puede ser usada con C, C++ y FORTRAN.
 - Las directivas de OpenMP proveen soporte para concurrencia, sincronización y manejo de datos obviando el uso explícito de mutex, variables condición, alcance de los datos e inicialización de threads.

OpenMP: *Características Básicas*

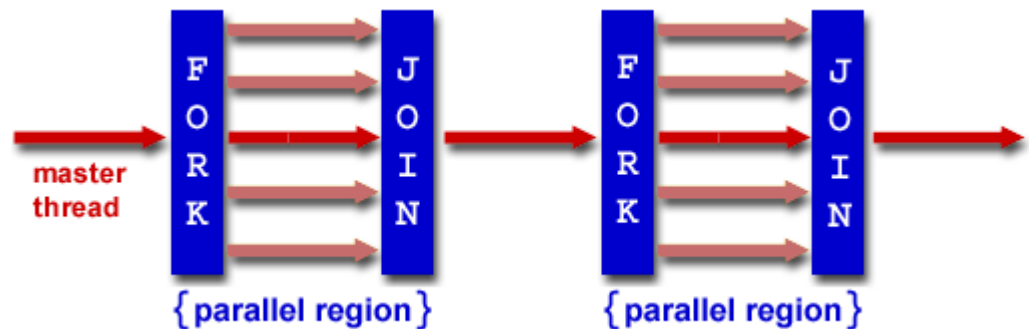
- Paralelismo basado en Threads. Memoria compartida.
- Paralelismo explícito.
- Permite alterar dinámicamente la cantidad de Threads usados.
- Basado en directivas del compilador.

#pragma omp nombre_directiva [lista de cláusulas]

- Usa el modelo Fork-Join

OpenMP: *Modelo Fork-Join*

- Comienza con un único proceso (thread master).
- FORK: Al encontrar un constructor paralelo (o directiva paralela), el thread master crea un grupo de threads (quedando él con id 0 del grupo).
- El bloque encerrada por el constructor de la región paralela es ejecutada en paralelo entre todos los threads.
- JOIN: cuando el conjunto de threads finaliza el bloque paralelo sincronizan y terminan, continuando únicamente el thread master.



OpenMP: *Constructor de región paralela*

➤ Sintaxis de la directiva `parallel`

`#pragma omp parallel [Lista de cláusulas] { Bloque paralelo }`

➤ Cuantos threads son creados?

- Por medio de la función `omp_set_num_threads()`.
- Variable de entorno `OMP_NUM_THREAD`.

➤ Lista de cláusulas:

- `if (expresión_escalar)`: Paralelización condicional. (Default $\neq 0$).
- `private (lista_variables)`: Variables locales a cada threads.
- `shared (lista_variables)`: Variables compartidas por todos los threads.
- `firstprivate (lista_variables)`: Variables locales inicializadas.
- `default (private | shared | none)`: Tipo por default de las variables usadas en el bloque paralelo. None obliga a especificar todas las variables.
- `reduction (operator: list)`: Variables locales a cada threads que al finalizar sus copias se reducen a un único valor para usar en el master por medio de alguna operación (+, *, -, &, |, ^, &&, ||).

OpenMP: *Ejemplos de Directiva Parallel*

```
#include <omp.h>
```

```
...
```

```
#pragma omp parallel if (is_parallel== 1) private(a) shared(b) firstprivate(c)
{ Bloque paralelo }
```

```
...
```

```
#include <omp.h>
```

```
....
```

```
/* Suponiendo que se crean 3 threads (además del master) */
```

```
#pragma omp parallel reduction(+: sum)
```

```
{ sum = omp_get_thread_num();
}
```

```
/* En este punto sum es igual a 6 */
```

```
...
```


OpenMP: *Ejemplos de Directiva Parallel*

Ejemplo donde cada Thread imprime su identificador y el master además imprime el total.

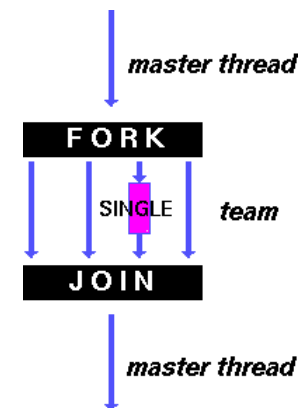
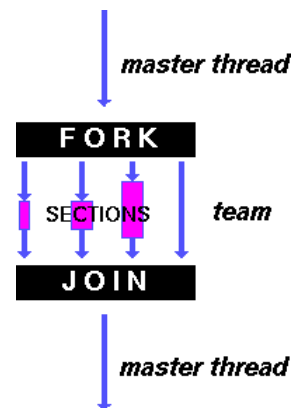
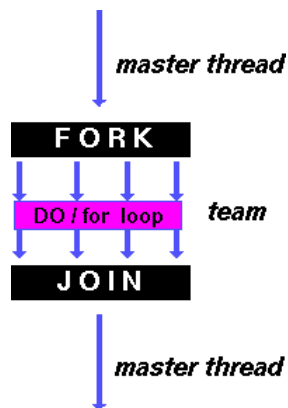
```
#include <omp.h>

main ()
{ int nthreads, tid;

    #pragma omp parallel private(tid)
    { tid = omp_get_thread_num();
      printf("Hola mundo del Thread = %d\n", tid);
      if (tid == 0)
      { nthreads = omp_get_num_threads();
        printf("Número de Threads = %d\n", nthreads);
      }
    }
}
```

OpenMP: *Constructor de trabajo compartido*

- La directiva parallel puede ser utilizada en conjunto con otras directivas para especificar concurrencia entre iteraciones y tareas (constructores de trabajo compartido) → No crea nuevos threads.
- Diferentes tipos de constructores.
 - Directiva for → Divide las iteraciones de un loop entre los threads (paralelismo de datos).
 - Directiva sections → Trabajo dividido en secciones separadas (paralelismo funcional).
 - Directiva single → Serializa una sección del código.



OpenMP: Constructores de trabajo compartido

Directiva For

- La forma general de esta directiva es:

```
#pragma omp for [lista de cláusulas]  
for_loop
```

- Lista de cláusulas:

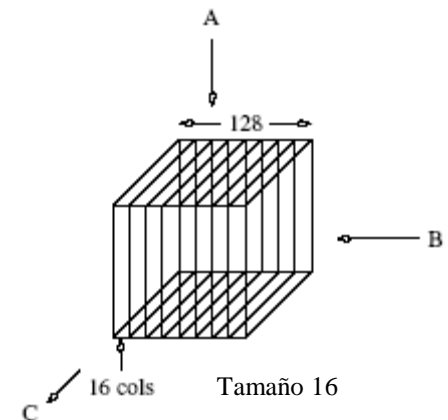
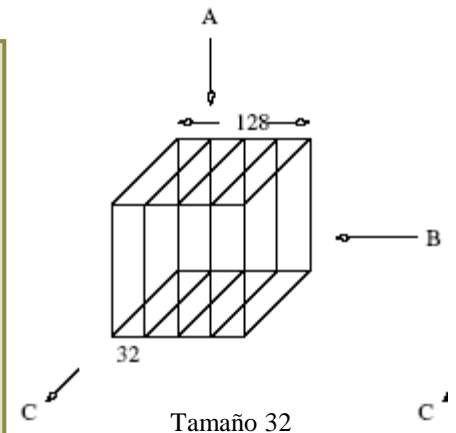
- ✓ ordered: Indica que dentro tiene un constructor de sincronización ordered.
- ✓ nowait: Evita la sincronización implícita.
- ✓ lastprivate (lista_variables): Variables locales donde el thread que realiza la última iteración deja el valor en el master al finalizar el bloque.
- ✓ private, shared, firstprivate, reduction.
- ✓ schedule (tipo [,tamaño]): Indica como se reparten las iteraciones.
 - static: divide en bloques de tamaño elementos y reparte en forma Round Robin.
 - dynamic: divide en bloques de tamaño elementos y reparte bajo demanda.
 - guided: se va achicando el tamaño del bloque.
 - runtime: determinado por la variable de entorno OMP_SCHEDULE.

OpenMP: *Ejemplo de directiva For* *Con cláusula schedule*

Ejemplo para multiplicar matrices con 4 threads.

```
#include <omp.h>

main ()
{ int *a, *b, *c, dim, i, j, k;
  ....
  #pragma omp parallel default(private) shared (a, b, c, dim)
  { #pragma omp for schedule(static,32)
    for (i = 0; i < dim ; i++)
      for (j = 0; j < dim ; j++)
        { c[i, j] = 0;
          for (k = 0; k < dim ; k++) c[i, j] += a[i,k]*b[k,j];
        }
  }
  ....
}
```



OpenMP: *Constructores de trabajo compartido*

Directiva Sections

- Esta directiva permite la asignación de tareas paralelas no iterativas.

- La forma general de esta directiva es:

```
#pragma omp sections [lista de cláusulas ]  
{ #pragma omp section  
  { bloque }  
  
  #pragma omp section  
  { bloque }  
  
  ....  
}
```

- Lista de cláusulas: private, firstprivate, lastprivate, reduction, nowait.

OpenMP: *Ejemplo de directiva Sections*

```
#include <omp.h>
#define N    1000

main ()
{ int i;
  float a[N], b[N], c[N];
  for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;

  #pragma omp parallel shared(a,b,c) private(i)
  { #pragma omp sections nowait
    { #pragma omp section
      for (i=0; i < N/2; i++) c[i] = a[i] / b[i];
      #pragma omp section
      for (i=N/2; i < N; i++) c[i] = b[i] / a[i];
    }
    ....
  }
  ....
}
```

OpenMP: *Constructores de trabajo compartido*

Directiva Single

- Esta directiva permite que un único thread (aleatorio) ejecute un bloque de código. El resto espera.
- La forma general de esta directiva es:

```
#pragma omp single [lista de cláusulas ]  
    {bloque }
```
- Lista de cláusulas: private, firstprivate, nowait.

OpenMP: *Ejemplo de directiva Single*

Ejemplo donde cada Thread imprime su identificador y uno imprime el total.

```
#include <omp.h>

main ()
{ int nthreads, tid;

    #pragma omp parallel private(tid)
    { tid = omp_get_thread_num();
      printf("Hola mundo del Thread = %d\n", tid);

      #pragma omp single private(nthreads) nowait
      { nthreads = omp_get_num_threads();
        printf("Número de Threads = %d\n", nthreads);
      }
    }
}
```


OpenMP: *Constructores combinados*

Directiva Parallel → crea los threads.
For / Sections → da trabajo a los threads. } **Parallel For
Parallel Sections**

➤ Combina las cláusulas de ambas directivas.

```
#include <omp.h>

main ()
{ int *a, *b, *c, dim, i, j, k;
  ....
  #pragma omp parallel for default(private) shared (a, b, c, dim) schedule(static,32)
    for (i = 0; i < dim ; i++)
      for (j = 0; j < dim ; j++)
        { c[i, j] = 0;
          for (k = 0; k < dim ; k++) c[i, j] += a[i,k]*b[k,j];
        }
  ....
}
```

OpenMP: *Directivas parallel anidadas*

- La variable de entorno OMP_NESTED indica si está habilitado o no el anidamiento de bloques paralelos.
- Si no está activo ejecuta los bloques internos como secuenciales.

```
#pragma omp parallel for default(private) shared (a, b, c, dim) schedule(static,32)
for (i = 0; i < dim ; i++)
{ #pragma omp parallel for default(private) shared (a, b, c, dim) schedule(static,32)
  for (j = 0; j < dim ; j++)
  { c[i, j] = 0;
    for (k = 0; k < dim ; k++) c[i, j] += a[i,k]*b[k,j];
  }
}
```

OpenMP: *Constructores de Sincronización*

- Hay casos en que explícitamente se requiere coordinar la ejecución de múltiples threads:
 - Ejecución en cierto orden (sincronización).
 - Atomicidad (exclusión mutua).
- OpenMP provee una serie de constructores de sincronización.
 - `#pragma omp barrier`
 - `#pragma omp master`
structured block
 - `#pragma omp critical [(name)]`
structured block
 - `#pragma omp ordered`
structured block

OpenMP: *Ejemplo de Constructores de Sincronización*

Productores – consumidores con la directiva critical.

```
#pragma omp parallel sections
{ #pragma omp section
  { tarea = Producir_Tarea ();
    #pragma omp critical (buffer)
    { insert_en_buffer (tarea);
    }
  }

  #pragma omp section
  { #pragma omp critical (buffer)
    { tarea = extraer_de_buffer ();
    }
    Consumir_Tarea (tarea);
  }
}
```

OpenMP: *Ejemplo de Constructores de Sincronización*

Calcular la suma acumulativa de una lista usando directiva *ordered*.

```
suma_acumulativa[0] = lista[0];
#pragma omp parallel for private(i) shared(suma_acumulativa, lista, n) ordered
for (i = 1; i < n; i++)
{ /* Procesamiento del elemento lista[i] */
  #pragma omp ordered
  { suma_acumulativa[i] = suma_acumulativa[i-1] + lista[i];
  }
}
```

OpenMP: *Funciones de la librería OpenMP*

Además de las directivas OpenMP soporta una serie de funciones que permite al programador controlar la ejecución del programa con mayor nivel de abstracción que Pthreads.

OpenMP: *Funciones de la librería OpenMP*

➤ Funciones básica

- `void omp_set_num_threads (int num_threads);`
- `int omp_get_num_threads ();`
- `int omp_get_thread_num ();`
- `int omp_get_num_procs ();`
- `int omp_in_parallel();`

➤ Funciones relacionadas con la creación de threads

- `void omp_set_dynamic (int dynamic_threads);`
- `int omp_get_dynamic ();`
- `void omp_set_nested (int nested);`
- `int omp_get_nested ();`

OpenMP: *Funciones de la librería OpenMP*

➤ Funciones para la exclusión mutua simple

- `void omp_init_lock (omp_lock_t *lock);`
- `void omp_destroy_lock (omp_lock_t *lock);`
- `void omp_set_lock (omp_lock_t *lock);`
- `void omp_unset_lock (omp_lock_t *lock);`
- `int omp_test_lock (omp_lock_t *lock);`

➤ Funciones para la exclusión mutua recursiva

- `void omp_init_nest_lock (omp_nest_lock_t *lock);`
- `void omp_destroy_nest_lock (omp_nest_lock_t *lock);`
- `void omp_set_nest_lock (omp_nest_lock_t *lock);`
- `void omp_unset_nest_lock (omp_nest_lock_t *lock);`
- `int omp_test_nest_lock (omp_nest_lock_t *lock);`