

# Sistemas Distribuidos y Paralelos

Ingeniería en Computación



## Modelo de programación sobre memoria compartida

Universidad Nacional de La Plata



Facultad de Informática



# Agenda

2

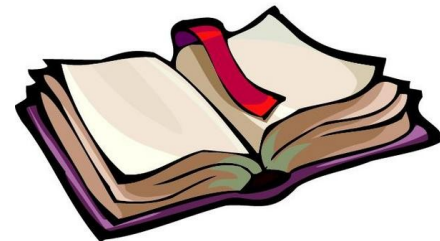
I. Modelo de programación sobre memoria compartida: Introducción

II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



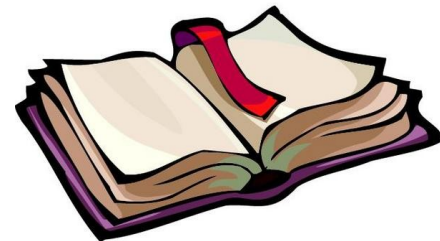
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



# Modelo de programación sobre memoria compartida

4

- En un modelo de programación sobre memoria compartida, los procesos o hilos comparten la memoria y se comunican mediante variables.



- Pueden existir problemas de “interferencia” conocidos como problemas de **condición de carrera**. Para evitar estos problemas, necesitamos sincronizar:
  - Exclusión Mutua
  - Sincronización por condición
- Existen diversas herramientas para el desarrollo de aplicaciones utilizando este modelo:
  - Pthreads
  - OpenMP
  - Otras: Cilk, OpenCL, CUDA, Intel TBB etc.

# Modelo de programación sobre memoria compartida

## Modelos de programación paralela – Modelos de arquitectura paralela

5

- El modelo de programación sobre memoria compartida puede utilizarse en un modelo de arquitectura de memoria compartida con un único espacio de memoria direccionable.
  - Sobre otros modelos de arquitectura, se requiere una capa de abstracción (SSI) que genera overhead impactando en el rendimiento.

Memoria			
UP	UP	...	UP

Software			
Memoria Compartida (Ej: OpenMP)	Memoria Distribuida (Ej: MPI)	Híbrido (EJ: MPI + OpenMP)	
<b>Trivial</b> (Ej: OpenMP sobre Multicore)	<b>Posible</b> (Ej: MPI sobre multicore)	<b>Posible (extraño)</b> (Ej: MPI + OpenMP sobre multicore)	
<b>(NO ADECUADO)</b> Single System Image(SSl) <b>Overhead</b>	<b>Trivial</b> (Ej: MPI sobre cluster)	<b>Posible (muy extraño)</b> (Ej: MPI + OpenMP sobre cluster monocore)	
<b>(NO ADECUADO)</b> Single System Image(SSl) <b>Overhead</b>	<b>Posible</b> (Ej: MPI sobre cluster multicore)	<b>Trivial</b> (Ej: MPI + OpenMP sobre cluster multicore)	

Hardware	<b>Memoria Compartida</b> (Ej: multicore)
	<b>Memoria Distribuida</b> (Ej: <b>cluster monocore</b> <sup>1</sup> )
	<b>Híbrido</b> (Ej: cluster multicore)

<sup>1</sup>Actualmente, los clusters monocore cayeron en desuso pero se mencionan para ejemplificar.

# Modelo de programación sobre memoria compartida

6



**NO** confundir el concepto de **variable global** con el de **variable compartida**

- Los **lenguajes imperativos** permiten definir variables en un área de programa visible por el **programa principal** y los módulos/subrutinas (**procedures o funciones**). El uso de estas **variables globales se desaconseja** en el desarrollo de **aplicaciones secuenciales** (por legibilidad, portabilidad, modularidad, etc.).
- Cuando desarrollamos **aplicaciones concurrentes** siguiendo el **modelo de programación sobre memoria compartida**, es **NECESARIO** definir las **variables compartidas** en un área visible al código de los **procesos o hilos**.
- La confusión surge por lo siguiente:



- Lenguajes como C no tienen un área para definir variables compartidas entre procesos o hilos y deben definirse como variables globales.
- Bibliotecas sobre C definen procesos o hilos como funciones.

# Modelo de programación sobre memoria compartida

7



**Una variable compartida NO es una variable global  
Un proceso o hilo NO es una función o un procedure**

```
Process P[id:0 to P-1]{  
    ...  
}
```

- Conceptualmente:
  - Los **procesos o hilos no son funciones (recordar Concurrencia y Paralelismo) y no reciben** como **parámetros** variables compartidas (porque son compartidas!!!).
    - Recordar **ADA**: los **Task** no son funciones ni procedures.
  - El código de los **procesos o hilos no es portable** sin el contexto del proceso que los contiene (modularidad SO), quien es el que provee las variables compartidas.
- Podemos hacer una analogía con la Programación Orientada a Objetos:
  - Las variables de instancia de un objeto son variables compartidas por todos los métodos de ese objeto. Los métodos no reciben como parámetro las variables de instancia (los lenguajes POO tampoco lo permiten).
  - El código de los métodos no es portable sin el contexto del objeto que los contiene, quien es el que provee de las variables de instancia.

# Modelo de programación sobre memoria compartida

8

## Imperativo (secuencial)

```
program Hello;  
  var v: Integer;
```

```
procedure Proc()  
begin  
  v := v + 1;  
end;
```

Pascal

```
function f():integer  
begin  
  f:= v + 1;  
end;  
...
```

```
#include ...  
int v;
```

```
void f(){  
  v++;  
}  
...
```

C

Imperativo  
Esto está mal

POO  
Esto está bien

## POO (secuencial)

```
public class Clase{  
  private int v;  
  
  public void metodo(){  
    v++;  
  }  
  ...  
}
```

JAVA

```
Object subclass: Clase[  
  |v|  
  
  Clase » metodo()[  
    v:= v + 1;  
  ]  
  ...  
]
```

Smalltalk

Estructura de programa  
variable

subrutina(sin parámetros)  
Modifica variable

## Concurrente

```
int v;  
Process P[id:0 to P-1]{  
  atomic(v++);  
  ...  
}
```

```
#include<pthread.h>  
int v;
```

```
void* f(void* arg){  
  atomic(v++);  
  ...  
}
```

```
main(){  
  ...  
  pthread_create(...,&f,...);  
  ...  
}
```

Concurrente  
Esto está bien

C (Pthreads)

En todos los casos, son estructuras de programa que definen una variable libre que se modifica por una subrutina directamente, sin parámetros. En imperativo está mal, en POO y Concurrente está bien. La **diferencia** está en el **paradigma** (Imperativo ≠ POO ≠ Concurrente)



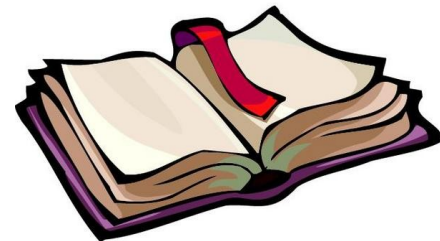
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



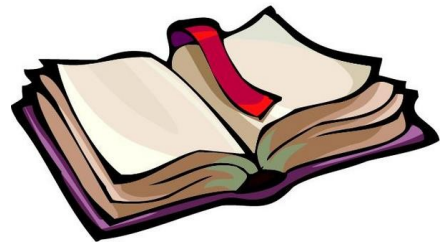
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



# Pthreads

11

- Pthread<sup>1</sup> (Posix threads - estándar Posix 1.0c 1995) es una API para programación multihilo sobre memoria compartida multiplataforma
- Posee un conjunto de funciones que permiten:
  - Gestionar hilos (threads)
  - Sincronizar hilos: Mutexes, Variables condición, Barreras
  - Interactuar con la API Semaphore (no es parte del estándar)
- Pthreads está incluida en el compilador GCC:
  - Cabecera de programa C: `#include<pthread.h>`
  - Compilación: `gcc -pthread fuente.c -o ejecutable`

<sup>1</sup><https://computing.llnl.gov/tutorials/pthreads/>

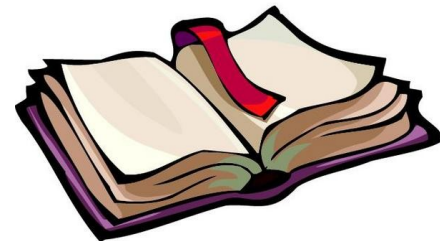
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



- Un programa Pthreads básico requiere de al menos 4 pasos:
  - 1) Definir los Hilos
  - 2) Crearlos
  - 3) Implementar la función de comportamiento del Hilo
  - 4) El programa principal debe esperar que los hilos terminen su ejecución

```
#include<pthread.h>
...
//3) Función de comportamiento del hilo
void* f(void* arg){
    ...
    pthread_exit(NULL);
}
...
int main(int argc, char*argv[]){
    pthread_t miHilo; //1) Definición
    ...
    pthread_create(&miHilo,&attr,&f,&arg); //2) Creación
    ...
    pthread_join(&miHilo,NULL); //4) Espera finalización
    return 0;
}
```

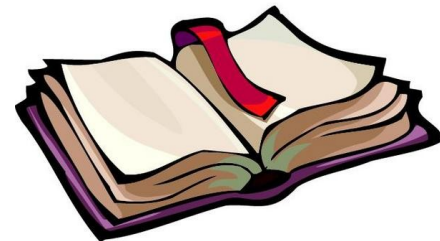
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



- Los hilos son del tipo de datos:

```
pthread_t
```

- Pueden definirse individualmente:

```
pthread_t miHilo1;  
pthread_t miHilo2;
```

- O como arreglos de hilos:

```
pthread_t misHilos[N];
```

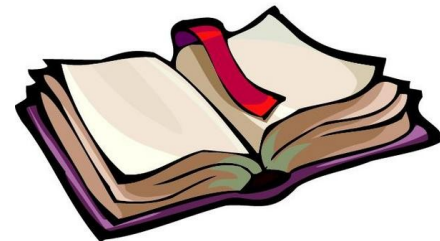
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad





# Pthreads

## Gestión de hilos – Creación de hilos

17

- Una vez definidos, los hilos se crean mediante la función:

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

### Parámetros:

- **thread:** puntero al hilo definido con pthread\_t.
- **attr:** atributos de configuración del hilo a crear.

No vamos a usar los atributos con las aplicaciones paralelas, este parámetro será **NULL**

- **start\_routine:** puntero a la función que implementa el comportamiento del hilo.
- **arg:** argumento pasado como parámetro a la función que implementa el comportamiento del hilo.



### Valor de retorno:

- **pthread\_create** retorna **cero** si tiene **éxito**, sino retorna un código de error.
- Si **pthread\_create** tiene éxito, el hilo creado se **ejecuta inmediatamente**, invocando de forma automática a la función que implementa su comportamiento.

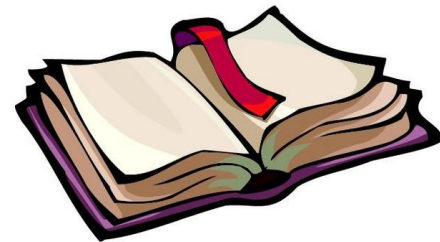
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



- El prototipo para la función que implementa el comportamiento del hilo es:

```
void* funcion(void *arg);
```

Parámetros:

- arg:** un puntero sin tipo (**requiere casting**) que puede apuntar a:
  - Una variable de un tipo simple
  - Una estructura (por ejemplo: un registro): **NO HABITUAL en el modelo de programación sobre memoria compartida!!!**

Sin valor de retorno.

- Los hilos con el mismo comportamiento reciben en la función `pthread_create` el mismo puntero a función.

```
void* funcionA(void *arg){  
...  
}
```

```
pthread_create(&T1, ..., &funcionA,...);  
pthread_create(&T2, ..., &funcionA,...);  
pthread_create(&T3, ..., &funcionB,...);
```

```
void* funcionB(void *arg){  
...  
}
```

- Ejemplo de parámetro **arg** de la función de comportamiento del hilo:

### Una variable de un tipo simple

```
void* funcion(void *arg){
    int x_local=*(int*)arg;
    ...
}
...
int main(int argc, char* argv[]){
    int x=1;
    ...
    pthread_create(&miHilo,&attr,&funcion,(void*)&x);
    ...
}
```

### Una estructura



**NO HABITUAL!!!**  
en el modelo de programación sobre memoria compartida

```
typedef struct str_punto{
    int x;
    int y;
} punto_t;

void* funcion(void *arg){
    punto_t punto_local=*(punto_t*)arg;
    ...
}
...
int main(int argc, char* argv[]){
    punto_t punto;
    ...
    pthread_create(&miHilo,&attr,&funcion,(void*)&punto);
    ...
}
```





En el **modelo de memoria compartida** con **Pthreads**, asumimos que la función del comportamiento de un hilo **recibe un id** o una estructura para ids compuestos. **NO** recibe otros parámetros ni tampoco referencias a variables compartidas.

- Pthreads se utiliza en otros ámbitos diferentes al modelo de programación sobre memoria compartida. Por ejemplo: redes o procesamiento gráfico, donde se pueden crear hilos que no comparten variables.
- En ese contexto, el parámetro de la función de comportamiento del hilo puede referenciar una variable NO compartida (por ejemplo: una variable dentro de función).

**NO USAR en el modelo de programación sobre memoria compartida**



```
<TipoDeRetorno> funcionDeAplicacion(<argumentos>){  
    Tipo miVariableLocal;  
    ...  
    pthread_create(&miHilo,&attr,&funcion,(void*)&miVariableLocal);  
    ...  
}
```

- Pthreads no tiene una forma de identificar los hilos.
- Generalmente, el programador debe pasar un identificador como parámetro a la función que implementa el comportamiento del hilo.

```
...  
void* funcion(void *arg){  
    int id=*(int*)arg;  
    ...  
}  
...  
int main(int argc, char* argv[]){  
    pthread_t misHilos[2];  
    int id1,id2;  
    ...  
        id1=1;  
        pthread_create(&misHilos[id],NULL,&funcion,(void*)&id1);  
        id2=2;  
        pthread_create(&misHilos[id],NULL,&funcion,(void*)&id2);  
    ...  
}
```

- Cuando se definen arreglos de hilos, un **error típico** es asignarles identificadores de la siguiente forma:

```
...  
void* funcion(void *arg){  
    int id=*(int*)arg;  
...  
}  
...  
int main(int argc, char* argv[]){  
    pthread_t misHilos[T];  
...  
    for(int id=0;id<T;id++)  
        pthread_create(&misHilos[id],NULL,&funcion,(void*)&id);  
...  
}
```



El argumento id es un puntero al índice del for. En el for podría modificarse el valor de la variable id antes que el hilo creado la lea. Como consecuencia, puede haber más de un hilo con el mismo id.

- Solución 1:

```
void* funcion(void *arg){  
    int id=*(int*)arg;  
    ...  
}
```



```
int main(int argc, char* argv){  
    pthread_t misHilos[T];  
    int threads_ids[T];  
    ...  
    for(int id=0;id<T;id++){  
        threads_ids[id]=id;  
        pthread_create(&misHilos[id],NULL,&funcion,(void*)&threads_ids[id]);  
    }  
    ...  
}
```

- Solución 2:

```
void* funcion(void *arg){  
    int id=(int*)arg;  
    ...  
}
```



```
int main(int argc, char* argv){  
    pthread_t misHilos[T];  
    ...  
    for(int id=0;id<T;id++){  
        pthread_create(&misHilos[id],NULL,&funcion,(void*)id);  
    }  
    ...  
}
```

**id es un valor, no un puntero!!!**  
**Es pasarle un entero y decirle que es una dirección de memoria.**  
**(El compilador gcc retorna 2 warnings que deben ignorarse)**



- Al finalizar el código de cada hilo, debe invocarse la función que termina la ejecución del hilo, cuyo prototipo es:

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

Parámetros:

- value\_ptr:** es un valor que puede ser NULL o un valor que será enviado a cualquier proceso/hilo que espere por su finalización (en concordancia con la función JOIN).

Sin valor de retorno.

```
...  
void* funcion(void *arg){  
...  
    pthread_exit(NULL);  
}  
...
```

```
...  
void* funcion(void *arg){  
    int ret;  
...  
    pthread_exit(&ret);  
}  
...
```

**Estamos en un modelo de  
programación sobre memoria  
compartida**  
**No usar value\_ptr para retornar  
resultados!!!**  
**Sólo para control!!!**



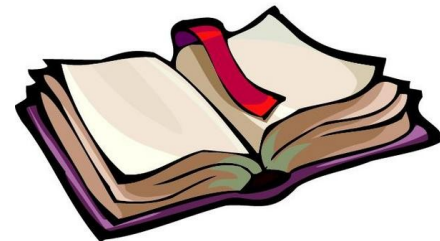
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



- El programa C que se ejecuta y crea los hilos es un proceso que actúa como un hilo más (**main**).
- El main, luego de crear los hilos, debe esperar a que estos finalicen. Para esto utiliza la función **join** cuyo prototipo es:

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

### Parámetros:

- **thread**: hilo por el que debe esperar quien ejecute pthread\_join (main u otro hilo o proceso).
- **value\_ptr**: valor retornado por **pthread\_exit** del **thread** recibido como parámetro. Puede valer NULL si no es necesario recibir ningún dato desde pthread\_exit del hilo del parámetro **thread**.

### Valor de retorno:

- **pthread\_join** retorna **cero** si tiene **éxito**, sino retorna un código de error.

### Espera sobre hilos individuales

```
void* funcionT1(void *arg){
    ...
    pthread_exit(NULL);
}

void* funcionT2(void *arg){
    int ret;
    ...
    pthread_exit(&ret);
}

int main(int argc, char* argv[]){
    pthread_t T1,T2;
    int valorT2;
    pthread_create(&T1,...,&funcionT1,...);
    pthread_create(&T2,...,&funcionT2,...);
    pthread_join(T1,NULL);
    pthread_join(T2, (void**)&valorT2);
    ...
}
```

### Espera sobre un arreglo de hilos

```
void* funcionT2(void *arg){
    int ret;
    ...
    pthread_exit(&ret);
}

int main(int argc, char* argv[]){
    pthread_t misThreads[N];
    for(i=0;i<N;i++){
        ...
        pthread_create(&misThreads[i],...,&funcion,...);
    }
    for(i=0;i<N;i++){
        pthread_join(misThreads[i], (void**)&valorRet);
    }
    ...
}
```

**Solo para Control**  
**NO RETORNAR RESULTADOS!!!**



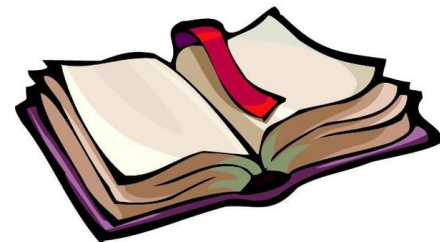
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



# Pthreads

## Sincronización

30

- Pthreads permite la sincronización mediante mecanismos como:
  - Mutexes
  - Variables condición
  - Barreras
- Es posible implementar semáforos mediante la biblioteca *semaphore* pero no es parte del estándar.

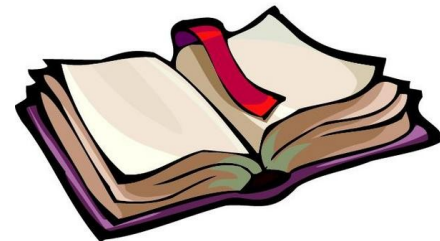
## I. Modelo de programación sobre memoria compartida: Introducción



## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



- Un **mutex**, abreviación de ***mutual exclusion***, se utiliza para sincronización por exclusión mutua.
- Un mutex tienen dos posibles estados:
  - Bloqueado/Locked: apropiado por un hilo. 
  - Desbloqueado/Unlocked: libre. 
- Si un mutex está libre sólo puede ser apropiado por un único hilo.





# Pthreads

## Sincronización - Mutex

33

- El uso de un mutex requiere de al menos 4 pasos:

- 1) Definirlo (compartido)
- 2) Inicializarlo
- 3) Utilizarlo
- 4) Destruirlo

```
#include<pthread.h>
pthread_mutex_t miMutex; //1) Definición compartido
...
void* f(void* arg){
    //3) Utilización
    pthread_mutex_lock(&miMutex);
    //Región crítica
    pthread_mutex_unlock(&miMutex);
}
...
int main(int argc, char*argv[]){
    ...
    pthread_mutex_init(&miMutex, &mutex_attr); //2) Inicialización
    pthread_create(&miHilo, NULL, &f, &arg);
    ...
    pthread_mutex_destroy(&miMutex); //4) Destrucción
}
```

- Un mutex es del tipo de datos:

```
pthread_mutex_t
```

- Pueden definirse individualmente:

```
pthread_mutex_t miMutex1;  
pthread_mutex_t miMutex2;
```

- O como arreglos de mutexes:

```
pthread_mutexes_t misMutexes[N];
```

- Visibilidad:

- Es necesario que los mutexes estén accesibles por todos los hilos
- Lo correcto es definirlos como variables compartidas (fuera del main o en archivos separados)

- Un mutex debe inicializarse antes de ser usado. Existen dos formas de hacerlo:

- Estática: `pthread_mutex_t miMutex = PTHREAD_MUTEX_INITIALIZER;`
- Dinámica, mediante la función:

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

Parámetros:

- **mutex**: puntero al mutex.
- **m\_attr**: parámetro que permite personalizar los atributos del mutex. Una estructura de tipo `pthread_mutexattr_t` que contiene tres atributos:

Protocol: protocolo usado para prevenir inversión de prioridades.

Prioceiling: especifica el límite de prioridad de un mutex.

Process-shared: especifica el uso compartido de un mutex.

**No vamos a usar los atributos con las aplicaciones paralelas, este parámetro será `NULL`**

Valor de retorno:

- `pthread_mutex_init` retorna **cero** si tiene **éxito**, sino retorna un código de error.



- Existen dos funciones básicas para utilizar mutexes:

- Adquirir el mutex:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- Si el mutex está desbloqueado, el propietario será el hilo que invoca a la función. Cualquier otro hilo que invoque a la función se quedará dormido.

- Liberar el mutex:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Parámetros (ambas funciones):

- **mutex**: puntero al mutex.

Valor de retorno (ambas funciones):

- Retornan **cero** si tienen **éxito**, sino retornan un código de error.

- Adicionalmente, pthreads provee la función:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Parámetros:

- **mutex**: puntero al mutex.

Valor de retorno:

- Si el hilo puede adquirir el mutex será el propietario y la función retorna **cero**.
- Si el hilo no puede adquirir el mutex no se demora, continúa su ejecución y la función retorna un valor **distinto de cero**.

- El uso correcto de esta función es:

```
if (pthread_mutex_trylock(&miMutex) == 0){  
    //Región crítica  
    pthread_mutex_unlock(&miMutex);  
}
```

- Una vez que un mutex deja de usarse debe ser destruido con la función:



```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Parámetros:

- **mutex**: puntero al mutex.

Valor de retorno:

- pthread\_mutex\_destroy retorna **cero** si tiene **éxito**, sino retorna un código de error.

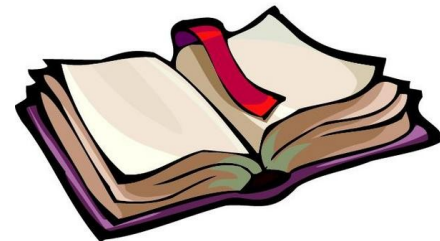
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



- Las variables condición se utilizan para sincronización por condición.
- Permiten detener la ejecución de un hilo a la espera de la ocurrencia de alguna condición.
- Cuando esa condición se cumple, algún otro hilo enviará una señal al hilo dormido para que continúe su ejecución.
- Cada variable condición tiene asociada una cola de espera.
- Cada variable condición debe utilizarse en conjunto con un mutex.





- El uso de variables condición requiere de al menos 4 pasos:

- 1) Definirlas (compartidas)
- 2) Inicializarlas
- 3) Utilizarlas
- 4) Destruirlas

```
#include<pthread.h>
pthread_cond_t c; //1) Definición compartida
```

```
void* f1(void* arg){
    //3) Utilización
    pthread_mutex_lock(&mutex);
    ...
    pthread_cond_wait(&c,&mutex);
    ...
    pthread_mutex_unlock(&mutex);
}
```

```
void* f2(void* arg){
    //3) Utilización
    pthread_mutex_lock(&mutex);
    ...
    pthread_cond_signal(&c);
    ...
    pthread_mutex_unlock(&mutex);
}
```

```
int main(int argc, char*argv[]){
    ...
    pthread_cond_init(&c,&cond_attr); //2) Inicialización
    pthread_create(&miHilo1,NULL,&f1,&arg1);
    pthread_create(&miHilo2,NULL,&f2,&arg2);
    ...
    pthread_cond_destroy(&c); //4) Destrucción
}
```

- Una variable condición es del tipo de datos:

```
pthread_cond_t
```

- Pueden definirse individualmente:

```
pthread_cond_t c1;  
pthread_cond_t c2;
```

- O como arreglos de variables condición:

```
pthread_cond_t cs[N];
```

- Visibilidad:

- Es necesario que las variables condición estén accesibles por todos los hilos
- Lo correcto es definir las como variables compartidas (fuera del main o en archivos separados)

- Una variable condición debe inicializarse antes de ser usada. Existen dos formas de hacerlo:

- Estática: `pthread_cond_t c = PTHREAD_COND_INITIALIZER;`
- Dinámica, mediante la función:

```
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict cond_attr);
```

Parámetros:

- **cond**: puntero a la variable condición.
- **cond\_attr**: parámetro que permite personalizar los atributos de la variable condición. Una estructura de tipo **pthread\_condattr\_t** que permite limitar el alcance de la variable condición y solo puede modificarse mediante funciones:

`pthread_condattr_getpshared`  
`pthread_condattr_setpshared`

No vamos a usar los atributos con las aplicaciones paralelas, este parámetro será **NULL**

Valor de retorno:

- **pthread\_cond\_init** retorna **cero** si tiene éxito, sino retorna un código de error.



- Existen tres funciones básicas para utilizar variables condición:

- Demorar/Dormir un hilo sobre una variable condición:

```
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t * mutex);
```

- Despertar un hilo demorado en una variable condición:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Despertar todos los hilos demorados en una variable condición:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Parámetros (de las tres funciones):

- **cond:** puntero a la variable condición.
- **mutex:** (sólo función pthread\_cond\_wait) puntero a un mutex asociado.

Valor de retorno (de las tres funciones):

- Retornan **cero** si tienen **éxito**, sino retornan un código de error.

- La función *pthread\_cond\_wait* necesita un mutex asociado a la variable condición. El uso habitual es:

```
pthread_mutex_lock(&mutex);  
...  
pthread_cond_wait(&c, &mutex);  
...  
pthread_mutex_unlock(&mutex);
```

- *pthread\_cond\_wait* libera el mutex automáticamente y pone el hilo a dormir. Cuando el hilo despierte, tomará el mutex automáticamente.
- *pthread\_cond\_signal* y *pthread\_cond\_broadcast* no tienen asociado un mutex, pero se suelen usar junto al mutex asociado a la función *pthread\_cond\_wait*:

```
pthread_mutex_lock(&mutex);  
...  
pthread_cond_signal(&c);  
...  
pthread_mutex_unlock(&mutex);
```

```
pthread_mutex_lock(&mutex);  
...  
pthread_cond_broadcast(&c);  
...  
pthread_mutex_unlock(&mutex);
```

- Una vez que una variable condición deja de usarse debe ser destruida con la función:



```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Parámetros:

- **cond:** puntero a la variable condición.

Valor de retorno:

- **pthread\_cond\_destroy** retorna **cero** si tiene **éxito**, sino retorna un código de error.

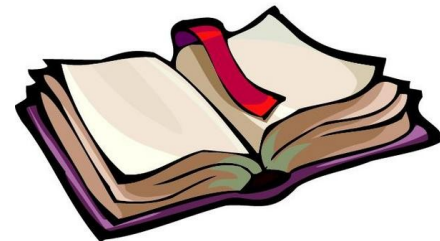
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

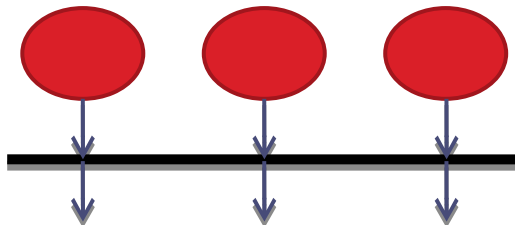
- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



- Las barreras son una herramienta de sincronización muy común.
- Una barrera hace que un conjunto de hilos/procesos se esperen para poder continuar su ejecución.



- Pthreads provee de una implementación sencilla y eficiente de las barreras, evitando que el programador tenga que implementarlas.



- El uso de barreras requiere de al menos 4 pasos:

- 1) Definirlas (compartidas)
- 2) Inicializarlas
- 3) Utilizarlas
- 4) Destruirlas

```
#include<pthread.h>
pthread_barrier_t barrera; //1) Definición compartida
...
void* f(void* arg){
    //3) Utilización
    pthread_barrier_wait(&barrera);
...
}
...
int main(int argc, char*argv[]){
    ...
    pthread_barrier_init(&barrera, &b_attr, 3); //2) Inicialización
    pthread_create(&miHilo1,NULL,&f,&arg);
    pthread_create(&miHilo2,NULL,&f,&arg);
    pthread_create(&miHilo3,NULL,&f,&arg);
    ...
    pthread_barrier_destroy(&barrera); //4) Destrucción
}
```

- Una barrera es del tipo de datos:

```
pthread_barrier_t
```

- Pueden definirse individualmente:

```
pthread_barrier_t barrera1;  
pthread_barrier_t barrera2;
```

- O como arreglos de barreras:

```
pthread_barrier_t barreras[N];
```

- Visibilidad:

- Es necesario que las barreras estén accesibles por todos los hilos
- Lo correcto es definirlas como variables compartidas (fuera del main o en archivos separados)

- Una barrera debe inicializarse antes de ser usada. Se utiliza la función:

```
int pthread_barrier_init(pthread_barrier_t * barrier, const pthread_barrierattr_t *b_attr, unsigned count);
```

Parámetros:

- **barrier:** puntero a la barrera.
- **b\_attr:** parámetro que permite personalizar los atributos de la barrera. Una estructura de tipo **pthread\_barrierattr\_t** que permite definir atributos con las características de la barrera y solo puede modificarse mediante funciones:

pthread\_barrierattr\_getpshared

pthread\_barrierattr\_setpshared

No vamos a usar los atributos con las aplicaciones paralelas, este parámetro será **NULL**



- **count:** número de hilos que deben llegar a la barrera para poder continuar.

Valor de retorno:

- **pthread\_barrier\_init** retorna **cero** si tiene **éxito**, sino retorna un código de error.

- Cuando un hilo llega a la barrera debe ejecutar la función:

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Parámetros:

- **barrier:** puntero a la barrera.

Valor de retorno:

- **pthread\_barrier\_wait** retorna **cero** si tiene **éxito**, sino retorna un código de error.

- Si el número de hilos en la barrera es menor al especificado en el parámetro count de la función de inicialización, el hilo se dormirá.
- En caso contrario, todos los hilos dormidos en la barrera continuarán la ejecución.

- Una vez que una barrera deja de usarse debe ser destruida con la función:



```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```


Parámetros:

- **barrier:** puntero a la barrera.

Valor de retorno:


- pthread\_barrier\_destroy retorna **cero** si tiene **éxito**, sino retorna un código de error.

- En aplicaciones con varias etapas de ejecución, NO crear Hilos por cada etapa. Crear hilos una única vez y separar cada etapa por una barrera.



```
int main(int argc, char* argv[]){  
...  
for(int id=0;id<T;id++){  
    pthread_create(&h[id],NULL,&etapa1,(void*)id);  
}  
...  
for(int id=0;id<T;id++){  
    pthread_create(&h[id],NULL,&etapa2,(void*)id);  
}  
...  
}
```

```
void* etapas(void* arg){  
    ...  
    etapa1();  
    pthread_barrier_wait(&barrera);  
    etapa2();  
    pthread_barrier_wait(&barrera);  
    ...  
}
```



```
int main(int argc, char* argv[]){  
...  
for(int id=0;id<T;id++){  
    pthread_create(&h[id],NULL,&etapas,(void*)id);  
}  
...  
}
```



**Para el sistema operativo el costo de crear hilos una y otra vez es mayor que dormirlos y despertarlos.**

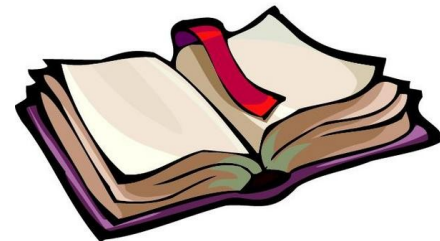
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



# Pthreads

## Afinidad

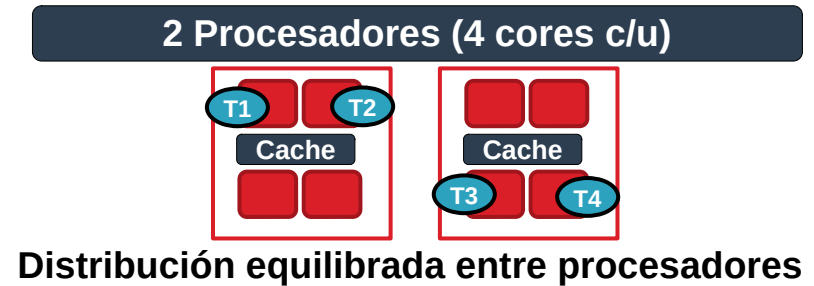
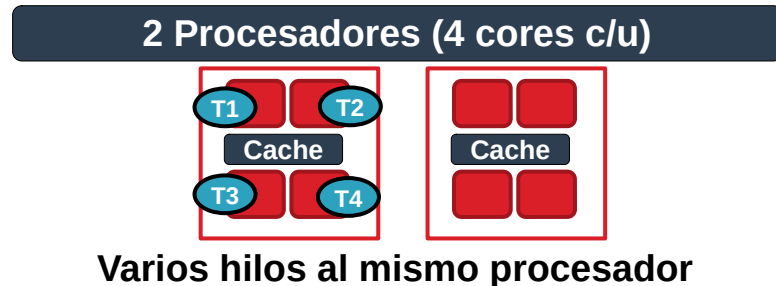
56

**Afinidad:** Elegir la unidad de procesamiento donde debe ejecutar un hilo.

### ¿Por qué usar afinidad?



- Aplicaciones intensivas en memoria pueden obtener mejor rendimiento si se ejecutan con menos hilos que unidades de procesamiento disponibles, esto debido a los fallos de caché.
- Debe asegurarse que dos o más hilos no ejecuten sobre la misma unidad de procesamiento. De esto se encarga el Sistema Operativo, pero el programador podría querer una asignación diferente.





# Pthreads

## Afinidad

57

- Para utilizar las funciones de afinidad en pthreads, es necesario incluir al inicio del archivo las siguientes líneas:

```
#define _GNU_SOURCE  
#include<sched.h>
```

- Para cambiar la afinidad de un hilo utilizamos la función:

```
int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset)
```

Parámetros:

- **thread:** hilo al que debe cambiarse la afinidad.
- **cpusetsize:** tamaño en bytes del CPU Set.
- **cpuset:** conjunto de cores de la arquitectura.

Valor de retorno:

**pthread\_setaffinity\_np** retorna **cero** si tiene **éxito**, sino retorna un código de error.

# Pthreads

## Afinidad – CPU set

58

- El parámetro **cpuset** es de tipo **cpu\_set\_t**, una estructura similar a:

cpuset	0	1	0	1	0	1	0	1
Core ID	0	1	2	3	4	5	6	7

En este ejemplo, el hilo puede ejecutar en los cores impares

- Las variables de tipo **cpu\_set\_t** no pueden modificarse directamente, para esto se utiliza la macro:

```
CPU_SET(int cpu, cpu_set_t *set)
```

# Pthreads

## Afinidad

59

- La función `pthread_setaffinity_np` puede invocarse de dos formas diferentes:

### En el proceso que crea el hilo

```
#define _GNU_SOURCE
#include<pthread.h>
#include<sched.h>

...
int main(int argc, char* argv[]){
    pthread_t h; int idH=1;
    cpu_set_t mask;

    pthread_create(&h, NULL, &funcionHilo, (void*)&idH);

    // Pone en cero la mascara
    CPU_ZERO(&mask);
    // Pone en uno el bit del core 3 en la máscara
    CPU_SET(3, &mask);
    // Cambia la afinidad al hilo
    pthread_setaffinity_np(h, sizeof(cpu_set_t), &mask);

    pthread_join(hilo, NULL);

    ...
}
```

### En el propio hilo

```
#define _GNU_SOURCE
#include<pthread.h>
#include<sched.h>

void* funcionHilo(void *arg){
    // Obtiene el descriptor del hilo
    pthread_t h=pthread_self();
    // Pone en cero la mascara
    CPU_ZERO(&mask);
    // Pone en uno el bit del core 3 en la máscara
    CPU_SET(3, &mask);
    // El hilo se cambia su afinidad
    pthread_setaffinity_np(h, sizeof(cpu_set_t), &mask);
    ...
}

int main(int argc, char* argv[]){
    pthread_t h; int idH=1;

    pthread_create(&h, NULL, &funcionHilo, (void*)&idH);

    ...
}
```

### ¿Cómo sabemos cuál es la numeración de cada core?

- Intuitivamente, podemos suponer la siguiente identificación:



- Sin embargo, cada sistemas operativo (y distribución o versión) identifica los cores de manera diferente.
- En Linux, esta identificación podemos obtenerla mediante el comando:

```
cat /proc/cpuinfo
```

# Pthreads

## Afinidad

61

- El recuadro muestra una salida simplificada de **cpuinfo** en una máquina con dos procesadores quad-core.
- Se pueden observar tres identificadores:
  - **Processor (ID):** identifica la unidad de procesamiento. Numeradas de 0 a 7
  - **Physical id:** identifica el socket del Processor. Numerados 0 y 1
  - **Core id:** identifica el core dentro de ese socket. Numerados 0 a 3 por socket
- La salida nos dice que la topología es la siguiente:



```
processor : 0
physical id : 0
core id : 0
processor : 1
physical id : 1
core id : 0
processor : 2
physical id : 0
core id : 2
processor : 3
physical id : 1
core id : 2
processor : 4
physical id : 0
core id : 1
processor : 5
physical id : 1
core id : 1
processor : 6
physical id : 0
core id : 3
processor : 7
physical id : 1
core id : 3
```

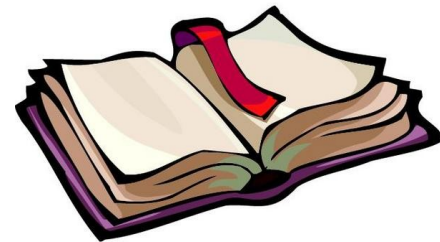
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



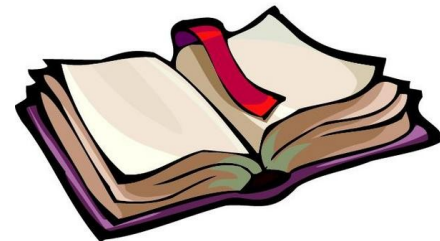
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



- OpenMP<sup>1</sup> es una API para programación paralela sobre memoria compartida multiplataforma
- Se basa en directivas aplicadas a los lenguajes:
  - C
  - C++
  - Fortran
- OpenMP está Incluida en el compilador GCC:
  - Cabecera de programa: `#include<omp.h>`
  - Compilación: `gcc -fopenmp fuente.c -o ejecutable`

<sup>1</sup>[www.openmp.org](http://www.openmp.org)

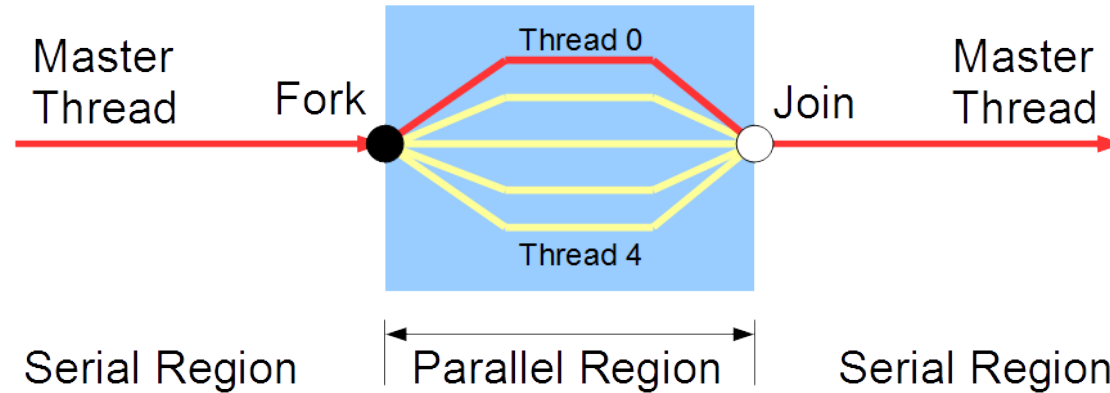


# OpenMP

## Modelo Fork-Join

65

- **OpenMP** sigue el **modelo Fork-Join**: un hilo (**Master thread**) viene ejecutando el código secuencialmente. En algún momento, se divide en T hilos (**Fork**), cada uno ejecuta su parte en forma paralela y luego se esperan (**Join**) en un punto a partir del cual se continua con la ejecución secuencial.



- El **Runtime System** o Runtime Environment es el sistema que proporciona el entorno en el que se ejecutan los programas. El Runtime System de **OpenMP** es el encargado de gestionar todo lo relacionado al modelo Fork-Join.

# OpenMP

## Core Elements

66

- OpenMP se estructura a partir de constructores llamados **Core Elements**:
  - De estructura de control paralela: directivas para la creación de hilos y control de flujo de programa.
  - De trabajo compartido: directivas para la distribución de la carga de trabajo entre hilos.
  - De entorno de datos: directivas para la gestión del entorno de datos y el alcance de las variables.
  - De sincronización: directivas para coordinar la ejecución y sincronización entre hilos.
  - De funciones de usuario y variables de ambiente.

### Core elements

#### Estructura de control paralela

*Directivas:  
parallel*

#### Trabajo compartido

*Directivas:  
do/parallel do  
section  
single  
master  
schedule*

#### Entorno de datos

*Directivas:  
shared  
private*

#### Sincronización

*Directivas:  
critical  
atomic  
barrier*

#### Funciones de usuario y Variables de ambiente

*Funciones de usuario:  
omp\_get\_thread\_num()  
omp\_set\_num\_threads()  
omp\_get\_num\_threads()*

*Variables de ambiente:  
OMP\_NUM\_THREADS  
OMP\_SCHEDULE*

## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



- En el caso específico del **lenguaje C**, OpenMP utiliza la directiva **#pragma**
- Las directivas **#pragma** son propias del compilador del lenguaje C. Su forma de uso es:

**#pragma instrucción**

- Las directivas, permiten proveer de información adicional al compilador, como por ejemplo:
  - Suprimir un mensaje de error específico
  - Chequear dependencias de archivos
  - En el caso de OpenMP, interactuar con el Runtime System para crear hilos
- Si el compilador encuentra una directiva **#pragma** con una instrucción desconocida, la ignora y no retorna ningún error.



- OpenMP combina la directiva **#pragma** con otras directivas para lograr la funcionalidad deseada. La sintaxis básica para el lenguaje C/C++ es:

```
# pragma omp <directiva> [cláusula [ , ...] ...]
```

- La directiva **#pragma omp parallel** se usa para crear hilos y establecer el flujo de ejecución. Por ejemplo:

```
int main(void){  
    #pragma omp parallel  
    {  
        printf("Hello, world.\n");  
    }  
    return 0;  
}
```

- Cada hilo ejecutará el código encerrado entre llaves.

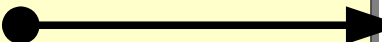
```
int main(void){  
    "Código secuencial"  
    #pragma omp parallel  
    {  
        "Código paralelo"  
    }  
    "Código secuencial"  
    #pragma omp parallel  
    {  
        "Código paralelo"  
    }  
    "Código secuencial"  
    return 0;  
}
```

El programa inicialmente comienza con el **Master thread** (ID=0) ejecutando en forma secuencial.

```
int main(void){  
    "Código secuencial"  
    #pragma omp parallel  
    {  
        "Código paralelo"  
    }  
    "Código secuencial"  
    #pragma omp parallel  
    {  
        "Código paralelo"  
    }  
    "Código secuencial"  
    return 0;  
}
```

Cuando encuentra por primera vez la directiva **#pragma omp parallel**, el Runtime System de OpenMP crea **T hilos**.  
Los hilos se identifican con ID=0..T-1.  
**(FORK)**

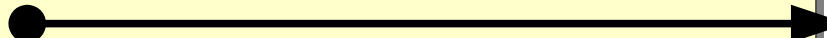
```
int main(void){  
    "Código secuencial"  
    #pragma omp parallel  
    {  
        "Código paralelo"  
    }  
    "Código secuencial"  
    #pragma omp parallel  
    {  
        "Código paralelo"  
    }  
    "Código secuencial"  
    return 0;  
}
```



Cada hilo ejecuta en forma paralela el código entre llaves.



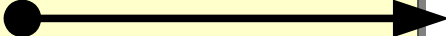
```
int main(void){  
    "Código secuencial"  
    #pragma omp parallel  
    {  
        "Código paralelo"  
    }  
    "Código secuencial"  
    #pragma omp parallel  
    {  
        "Código paralelo"  
    }  
    "Código secuencial"  
    return 0;  
}
```



Cuando un hilo finaliza su ejecución debe esperar a que el resto de los hilos termine de ejecutar.

Se produce una **barrera implícita**.  
**(JOIN)**

```
int main(void){  
    "Código secuencial"  
    #pragma omp parallel  
    {  
        "Código paralelo"  
    }  
    "Código secuencial"  
    #pragma omp parallel  
    {  
        "Código paralelo"  
    }  
    "Código secuencial"  
    return 0;  
}
```



El **Master thread** continua la ejecución de forma secuencial.  
El Runtime System de OpenMP duerme al resto de los hilos.

```
int main(void){  
    "Código secuencial"  
    #pragma omp parallel  
    {  
        "Código paralelo"  
    }  
    "Código secuencial"  
    #pragma omp parallel  
    {  
        "Código paralelo"  
    }  
    "Código secuencial"  
    return 0;  
}
```

Cuando se encuentra la siguiente directiva **#pragma omp parallel**, el Runtime System de OpenMP despierta a los hilos dormidos.  
**(FORK)**



Para el sistema operativo, el costo de crear hilos una y otra vez es mayor que dormirlos y despertarlos.

- Por defecto, OpenMP determina el número de cores de la arquitectura y crea un hilo por cada core.
- El usuario puede indicarle a OpenMP el número de hilos a crear. Existen tres formas de hacerlo:

### Función `omp_set_num_threads`

Mediante una función de usuario en el código.

Ejemplo creando 4 hilos:

```
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    ...  
}
```

### Variables de ambiente

Desde el Sistema Operativo creando la variable de ambiente `OMP_NUM_THREADS`.

Ejemplo Linux creando 4 hilos:

```
export OMP_NUM_THREADS=4
```

```
#pragma omp parallel  
{  
    ...  
}
```

### Directiva `num_threads`

Mediante una directiva en el código.  
Ejemplo creando 4 hilos:

```
#pragma omp parallel num_threads(4)  
{  
    ...  
}
```

- Anidamiento: cuando los hilos crean nuevos hilos.
- En OpenMP se utiliza la función `omp_set_nested`.
- Ejemplo: se crean 2 hilos. Luego, cada hilo crea 4 hilos (8 hilos en total)

```
omp_set_nested(1); // 1:anidamiento habilitado, 0: deshabilitado
#pragma omp parallel num_threads(2)
{
    printf("ID hilo externo:%d\n",omp_get_thread_num());
    #pragma omp parallel num_threads(4)
    {
        printf("ID hilo interno:%d\n",omp_get_thread_num());
    }
}
```



**Si en el código anterior omitimos `omp_set_nested(1)`,  
el número total de hilos será 2!!!  
La creación de los 4 hilos internos se ignora!!!**

Posible salida:

ID hilo externo:0  
ID hilo externo:1  
ID hilo interno:0  
ID hilo interno:3  
ID hilo interno:1  
ID hilo interno:2  
ID hilo interno:0  
ID hilo interno:1  
ID hilo interno:2  
ID hilo interno:3

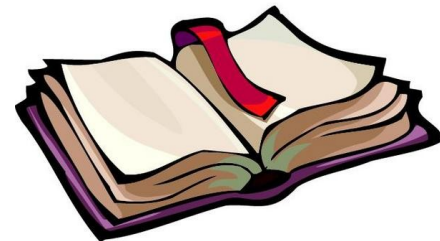
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



- OpenMP posee cuatro directivas para la distribución de trabajo entre hilos:
  - `do/parallel do`, ***omp for***: distribuye entre los hilos las iteraciones de una sentencia `for`
  - ***sections***: asigna a los hilos bloques de código consecutivo e independiente
  - ***single***: especifica un bloque de código que será ejecutado por un único hilo. Existe una barrera implícita al final del bloque
  - ***master***: similar a `single`. El bloque de código lo ejecuta sólo el ***Master thread***. NO existe barrera al final del bloque

- La directiva **omp for**, por defecto, **distribuye** las **iteraciones** de la sentencia **for** proporcionalmente entre los hilos.

```
int main(int argc, char* argv){
    int a[100];

    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < 100; i++)
            a[i] = 2 * i;
    }
    return 0;
}
```

Simplificado

```
int main(int argc, char* argv){
    int a[100];

    #pragma omp parallel for
        for (int i = 0; i < 100; i++)
            a[i] = 2 * i;

    return 0;
}
```



- Un error común al querer paralelizar dos for anidados es el siguiente:

```
#pragma omp parallel for  
for(int y=0; y<Y; ++y) {  
    #pragma omp parallel for  
    for(int x=0; x<X; ++x){  
        f(x,y);  
    }  
}
```



**El for más interno no se paraleliza!!!**

Sólo se paraleliza el for externo.  
El for interno se ejecuta secuencialmente, como si el #pragma interno no existiese.  
El Runtime System de OpenMP detecta que ya existe un grupo de hilos creados en el #pragma externo y no crea (ni despierta) nuevos hilos.

### Solución

```
#pragma omp parallel for collapse(2) // 2 es el número de iteraciones anidadas  
for(int y=0; y<Y; ++y) {  
    for(int x=0; x<X; ++x){  
        f(x,y);  
    }  
}
```



- El programador puede determinar de que forma se distribuyen las iteraciones de una sentencia for entre los hilos.
- Las políticas de distribución de iteraciones pueden ser:
  - **Estática (por defecto):** se distribuyen proporcionalmente entre los hilos.
  - **Dinámica:** se distribuyen por demanda y de a cierta cantidad (chunk).
  - **Guiada:** distribución de iteraciones variable.
  - **Runtime:** distribución que se indica desde el Sistema Operativo por medio de la variable de ambiente `OMP_SCHEDULE`
- Para indicar la distribución, se debe agregar la cláusula ***schedule*** a la directiva.

- **Distribución estática:** es la distribución por defecto

```
#pragma omp parallel for  
for(int y=0; y<8; y++)  
    "Código"
```



```
#pragma omp parallel for schedule(static)  
for(int y=0; y<8; y++)  
    "Código"
```

En este ejemplo, la distribución estática para dos hilos es:

Hilo 0 iteraciones y=0, y=1, y=2, y=3

Hilo 1 iteraciones y=4, y=5, y=6, y=7

- Distribución dinámica:

```
#pragma omp parallel for schedule(dynamic,3)  
for(int y=0; y<12; y++)  
    "Código"
```

Cada hilo solicita iteraciones una y otra vez.

Si hay iteraciones, el Runtime System de OpenMP le asigna una cantidad dada por **chunk** (valor 3 en el ejemplo).

Existe **no determinismo** en la distribución, es decir, dos ejecuciones del mismo programa OpenMP podrían asignar distintas iteraciones a los hilos.

En este ejemplo, una distribución dinámica “**posible**” entre dos hilos sería:

Hilo 0 iteraciones y=0, y=1, y=2

Hilo 1 iteraciones y=3, y=4, y=5

Hilo 0 iteraciones y=6, y=7, y=8

Hilo 0 iteraciones y=9, y=10, y=11

- Distribución guiada:

```
#pragma omp parallel for schedule(guided,2)
for(int y=0; y<32; y++)
    "Código"
```

Cada hilo recibe dinámicamente bloques de iteraciones. El bloque inicialmente es grande y va disminuyendo exponencialmente su tamaño hasta el valor especificado en **chunk** (valor 2 en el ejemplo).

En este ejemplo, una distribución guiada “**posible**” entre dos hilos sería:

Hilo 0 iteraciones y=0 a y=7 (8 iteraciones)

Hilo 1 iteraciones y=8 a y=15 (8 iteraciones)

Hilo 1 iteraciones y=16 a y=19 (4 iteraciones)

Hilo 0 iteraciones y=20 a y=23 (4 iteraciones)

Hilo 0 iteraciones y=24, y=25 (2 iteraciones)

Hilo 0 iteraciones y=26, y=27 (2 iteraciones)

Hilo 1 iteraciones y=28, y=29 (2 iteraciones)

Hilo 0 iteraciones y=30, y=31 (2 iteraciones)

- Distribución runtime:

```
#pragma omp parallel for schedule(runtime)
for(int y=0; y<Y; ++y)
    "Código"
```

La distribución de iteraciones runtime se hace desde el Sistema Operativo de acuerdo al valor de la variable de ambiente **OMP\_SCHEDULE**.

Permite cambiar la política de planificación sin necesidad de compilar el código.

En Linux podemos establecer las políticas de distribución de iteraciones anteriores de la siguiente forma:

<b>Estática:</b>	<b>export OMP_SCHEDULE="static"</b>
<b>Dinámica:</b>	<b>export OMP_SCHEDULE="dynamic,4"</b>
<b>Guiada:</b>	<b>export OMP_SCHEDULE="guided,2"</b>

- La directiva **sections** se utiliza para distribuir secciones de código entre los hilos.

```
#pragma omp parallel
```

```
{  
    #pragma omp sections {  
        { //Código Sección 1 }  
        #pragma omp section  
        { //Código Sección 2 }  
        #pragma omp section  
        { //Código Sección 3 }  
    }  
}
```

Simplificado

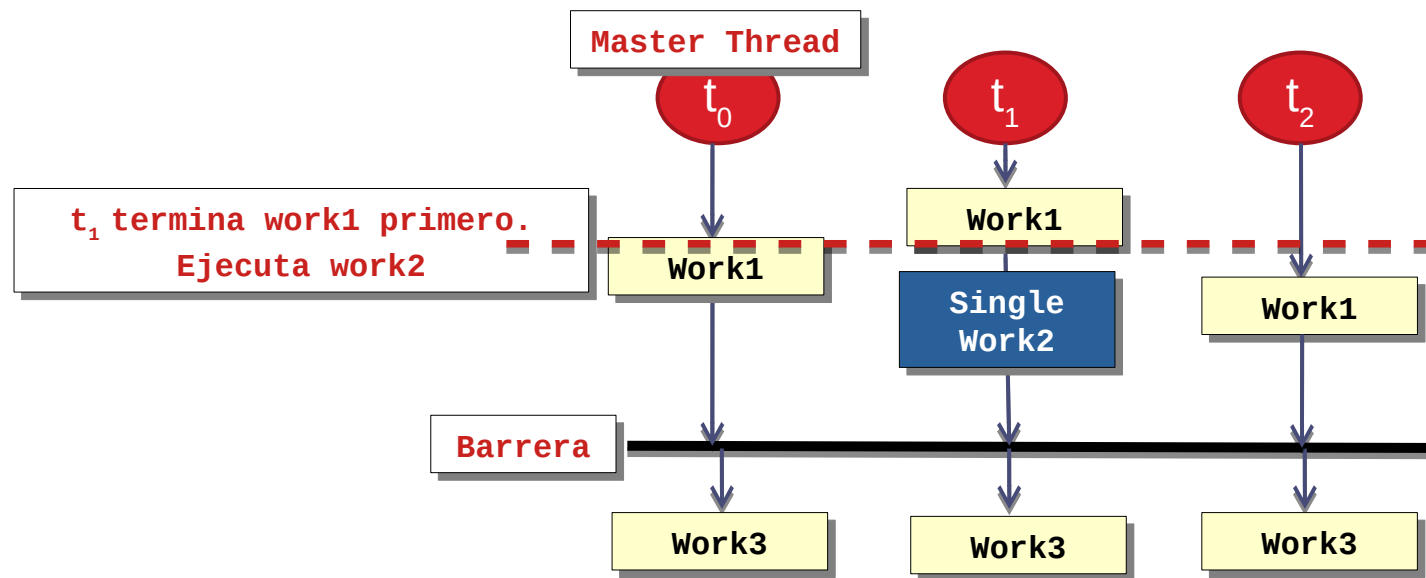
```
#pragma omp parallel sections
```

```
{  
    { //Código Sección 1 }  
    #pragma omp section  
    { //Código Sección 2 }  
    #pragma omp section  
    { //Código Sección 3 }  
}
```

El código de ejemplo tiene tres secciones que se ejecutan en paralelo.  
Cada sección se ejecuta una sola vez por un único hilo.

- La directiva **Single** especifica un bloque de código que será ejecutado por un único hilo. Existe una barrera implícita al final del bloque.

```
#pragma omp parallel
{
    // Work1
    #pragma omp single
    {
        // Work2
    }
    // Work3
}
```



Work1 y Work3 son tareas paralelas, todos los hilos la ejecutan.

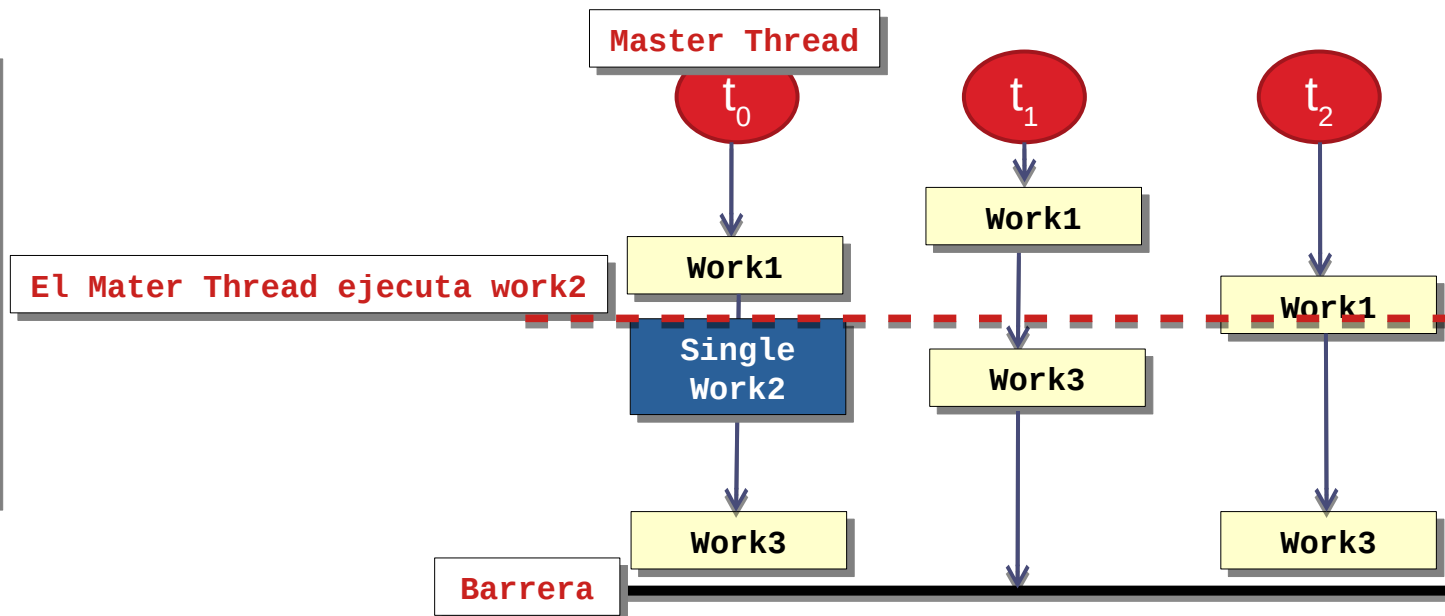
Work2 es una tarea que ejecutará un único hilo, el primero que llegue (en el ejemplo  $t_1$ ).

Los demás hilos no podrán ejecutar Work3 hasta que termine la ejecución del hilo que ejecuta Work2 ( $t_1$ ).



- La directiva **Master** difiere de Single en que es el Master Thread el encargado de ejecutar el bloque de código y no existe barrera al final del bloque.

```
#pragma omp parallel
{
    // Work1
    #pragma omp master
    {
        // Work2
    }
    // Work3
}
```



Work2 será ejecutado por el Master thread.  
El resto de los hilos podrán seguir ejecutando Work3.

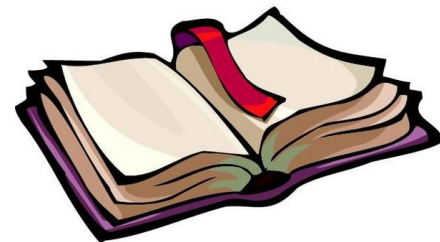
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



# OpenMP

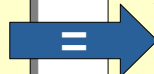
## Entorno de datos

91

- En OpenMP las variables son visibles por todos los hilos.
- A veces es necesario declarar variables privadas para evitar condiciones de carrera.
- Puede ser necesario intercambiar valores entre una región secuencial y una paralela.
- Para gestionar el entorno de los datos, se agregan cláusulas a las directivas:
  - `shared`
  - `private`
  - `firstprivate`
  - `lastprivate`
  - `reduction`

- **shared:** se comparten datos entre la región secuencial y la región paralela.
- Todos los hilos ven y acceden a los datos simultáneamente.
- Por defecto, todas las variables se comparten excepto las declaradas en las regiones paralelas.

```
int b=2;  
#pragma omp parallel for  
    for(int a=0; a<50; ++a)  
        printf("%d", a*b);  
// a privada, b compartida
```



```
int b=2;  
#pragma omp parallel for shared(b)  
    for(int a=0; a<50; ++a)  
        printf("%d", a*b);  
// a privada, b compartida
```

- **private:** la variable afectada por esta cláusula es privada a cada hilo
- Cada hilo tiene una copia local privada, la cual usa como variable temporal, **no se inicializa** y su valor no se mantiene fuera de la región paralela
- Por defecto, el índice de la sentencia for que sigue a un **#pragma omp for** y las variables definidas dentro de la región paralela son privadas.

```
int b=2;
int j,a;
#pragma omp parallel for private(j,a)
for(a=0; a<50; ++a)
    j=a*b;
// j,a privadas, b compartida
```

```
int b=2;
int j;
int a;
#pragma omp parallel for private(j)
for(a=0; a<50; ++a){
    int c=a*b*j;
    j=a*b;
}
// j,a,c privadas, b compartida
```

- Ejemplo de salida utilizando la clausula **private**:

```
int j = 4;  
#pragma omp parallel for private(j)  
    for(a=0; a<3; a++){  
        j+=a;  
        printf("%d\n", j);  
    }  
  
printf("Final %d\n", j);
```

Se crea una copia de **j** para cada hilo.  
No se inicializa.  
El **valor inicial** es “basura”.  
El valor final es el que tenía previo a la región paralela.



### Posible salida de programa

```
4321  
0  
543243  
1234  
Final 4
```



- **firstprivate**: idem **private** pero el valor se inicializa con el valor original de la variable. El valor final es el que tenía previo a la región paralela.

```
int j = 4;
#pragma omp parallel for firstprivate(j)
    for(a=0; a<3; a++){
        j+=a;
        printf("%d\n", j);
    }

printf("Final %d\n", j);
```

Posible salida de programa

```
4
6
7
5
Final 4
```



- **lastprivate:** idem *firstprivate* pero el valor finaliza con el último valor calculado

```
int j;  
#pragma omp parallel for lastprivate(j)  
    for(a=0; a<3; a++){  
        j=a;  
        printf("%d\n", j);  
    }  
  
printf("Final %d\n", j);
```

Posible salida de programa

```
3  
2  
0  
1  
Final 1
```





- **reduction:** reduce el resultado de todos los hilos mediante alguna operación.

```
int sum = 0;
int arreglo[count];
...
#pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < count; ++i)
        sum += arreglo[i];
printf("%d", sum);
```

```
int factorial(int number) {
    int fac = 1;
    #pragma omp parallel for reduction(*:fac)
        for(int n=2; n<=number; ++n)
            fac *= n;
    return fac;
}
```

### Posibles operandos

+

-

\*

&amp;

|

^

&amp;&amp;

||

Max

Min

Se tiene una variable compartida (sum o fac en los ejemplos).

La clausula reduction crea una copia privada (con firstprivate).

Cada hilo trabaja sobre su copia privada.

Al finalizar, se reducen los resultados en la variable compartida a partir del operando pasado a reduction.

- Es posible realizar más de una reducción en una misma iteración:

```
int sum1 = 0;
int sum2 = 0;
int arreglo[count];
...
int valor_maximo=arreglo[0];
...
#pragma omp parallel for reduction(+:sum1,sum2) reduction(max:valor_maximo)
for (int i = 0; i < count; ++i){
    sum1 += funcionA(arreglo[i]);
    sum2 += funcionB(arreglo[i]);
    if( arreglo[i] > valor_máximo){
        valor_maximo=arreglo[i];
    }
}
printf("%d %d %d\n", sum1, sum2, valor_maximo);
```



- Por defecto, todas las variables se comparten excepto las declaradas en las regiones paralelas. Sin embargo, es posible eliminar la opción por defecto para tener mayor control sobre el alcance de las variables.
- **default(none)**: obliga al programador a que haga explícito el alcance de todas las variables.

```
int b=2;
int a;
#pragma omp parallel for default(none) shared(b) private(a)
    for(a=0; a<50; ++a)
        printf("%d", a*b);
// a privada, b compartida
```

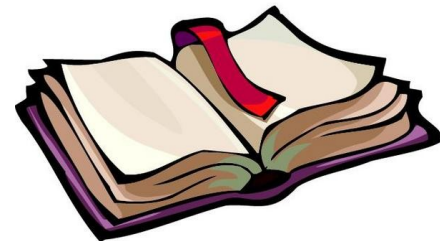
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



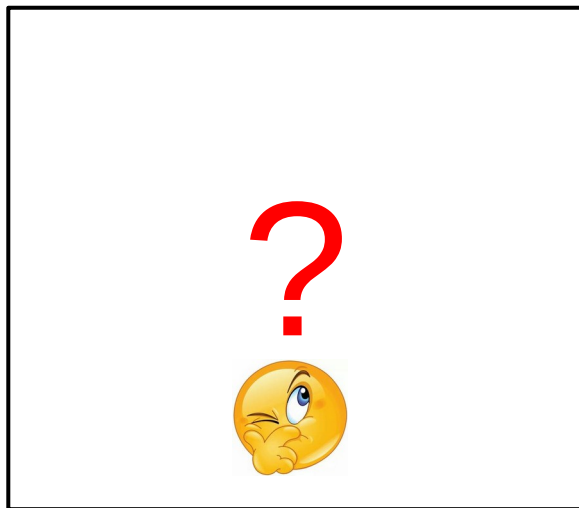
# OpenMP

## Sincronización

101

- OpenMP provee las siguiente clausulas para implementar algunos mecanismos de sincronización:
  - critical
  - atomic
  - ordered
  - barrier
  - nowait
- Estas cláusulas pueden usarse en conjunto con las directivas.

- **critical:** encierra un bloque de código (región crítica) que debe ser ejecutado por un hilo a la vez.
- Ejemplo: calcular la suma y el producto de los elementos de un vector



```
int arreglo[count]
int sum = 0;
int prod = 1;
...
#pragma omp parallel
{
    int sum_local = 0;
    int prod_local = 1;
    #pragma omp for
    for(int i=0;i<count;i++){
        sum_local += arreglo[i];
        prod_local *= arreglo[i];
    }
    sum += sum_local;
    prod *= prod_local;
}
```

```
int arreglo[count]
int sum = 0;
int prod = 1;
...
#pragma omp parallel
{
    int sum_local = 0;
    int prod_local = 1;
    #pragma omp for
    for(int i=0;i<count;i++){
        sum_local += arreglo[i];
        prod_local *= arreglo[i];
    }
    sum += sum_local;
    prod *= prod_local;
}
```

Por defecto las variables **sum** y **prod** son variables compartidas (shared).

Pero “**sum += i**” y “**prod \*= i**” no son instrucciones atómicas.

**Condición de carrera!!!**



**Solución:**

Encerramos las instrucciones conflictivas en una región crítica.

```
#pragma omp critical (lock1){ //lock1:nombre de la región crítica
    sum += sum_local;
    prod *= prod_local;
}
```

- **atomic**: similar a **critical** pero encierra una sola instrucción

```
int arreglo[count]
int sum = 0;
int prod = 1;
...
#pragma omp parallel
{
    int sum_local = 0;
    int prod_local = 1;
    #pragma omp for
    for(int i=0;i<count;i++){
        sum_local += arreglo[i];
        prod_local *= arreglo[i];
    }
    #pragma omp atomic
    sum += sum_local;
    #pragma omp atomic
    prod *= prod_local;
}
```

En este caso, **atomic** maximiza la concurrencia respecto a usar **critical**:

sum += sum\_local; y prod \*= prod\_local;

Se ejecutan de forma concurrente.





## Sincronización – Clausula ordered

- **ordered:** se utiliza cuando es necesario que las instrucciones se ejecuten según el orden de las iteraciones. El orden de la ejecución paralela sería equivalente a una ejecución secuencial.

```
#pragma omp parallel for  
for (int i = 0; i < 3; i++)  
    printf("%d\n", i);
```

Posible salida de programa

0  
3  
2  
1

```
#pragma omp parallel for ordered  
for (int i = 0; i < 3; i++)  
    printf("%d\n", i);
```

Posible salida de programa (ordered)

0  
1  
2  
3



## Sincronización – Clausula ordered

- **Un ejemplo útil de ordered** cuando parte del cuerpo del for puede ejecutarse en paralelo y deben almacenarse los resultados de manera ordenada, según el orden de las iteraciones, hacia un dispositivo de almacenamiento.

```
#pragma omp parallel
{
    #pragma omp for schedule(static,1) ordered
    for (int i = 0; i < N; i++){
        v_out[i] = f(v_in[i])
    }
    #pragma omp ordered
    WriteOutput(v_out[i]);
}
```

Cada hilo accede a posiciones consecutivas para maximizar el paralelismo

Región paralela



Salida secuencial ordenada

- **barrier:** para realizar una **barrera explícita**, los hilos esperan en un punto en el código para continuar la ejecución.

```
#pragma omp parallel {  
    "Codigo"      //Todos los hilos ejecutan esto  
  
    #pragma omp barrier // Se esperan en este punto  
  
    "Otro código" //Cuando todos los hilos alcanzaron la barrera continuan la ejecución  
}
```

- **nowait:** para **evitar la barrera implícita**, especifica que los hilos que terminaron su trabajo puedan continuar sin esperar al resto

```
#pragma omp parallel{
  #pragma omp for nowait
  for(int n=0; n < 10; n++){
    "Código de la iteración"
  }
  // Esta línea puede ser alcanzada mientras otros hilos aún están ejecutando la iteración.
  "Código fuera del loop"
}
```

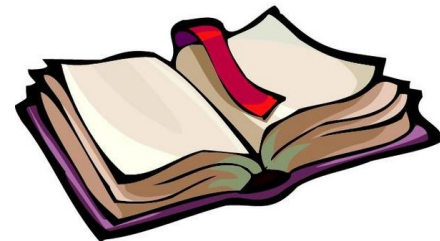
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



## Funciones de usuario y variables de ambiente

- OpenMP posee un conjunto de funciones y variables de ambiente que permiten configurar u obtener valores para la ejecución.
- **Funciones de usuario** (utilizadas desde el programa OpenMP):
  - `omp_get_thread_num()`: retorna el identificador del hilo
  - `omp_set_num_threads(int num_threads)`: determina el número de hilos que se crearán
  - `omp_get_num_threads()`: retorna cuantos hilos que se crearon
  - `omp_set_nested(int value)`: habilita o deshabilita el anidamiento de hilos
- **Variables de ambiente** (utilizadas desde el Sistema Operativo):
  - `OMP_NUM_THREADS`: idem función `omp_set_num_threads`
  - `OMP_SCHEDULE = [static, dynamic]`
  - `OMP_NESTED`: idem función `omp_set_nested`

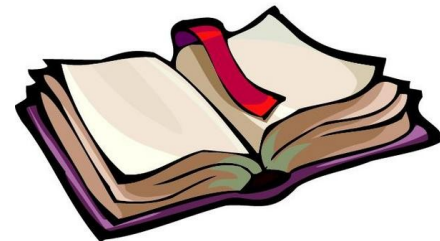
## I. Modelo de programación sobre memoria compartida: Introducción

## II. Posix Threads (Pthreads)

- i. Gestión de hilos (Definición | Creación | Función de comportamiento | Espera por finalización)
- ii. Sincronización (Mutex | Variables Condición | Barreras)
- iii. Afinidad

## III. OpenMP

- i. Estructura de control paralela
- ii. Trabajo compartido
- iii. Entorno de datos
- iv. Sincronización
- v. Funciones de usuario y variables de ambiente
- vi. Afinidad



# OpenMP

## Afinidad

112

- OpenMP permite controlar la afinidad desde el Sistema Operativo mediante las variables de ambiente:



- **OMP\_PLACES**: se indica las ubicaciones posibles

- Lista de processors ID
- En conjunto con **binding**: **cores** (*por defecto*)
- Alternativas con Hyperthreading: **sockets** | **threads**



- **OMP\_PROC\_BIND** o la cláusula **proc\_bind** (**Binding**): cómo se quieren distribuir los hilos en esas ubicaciones.

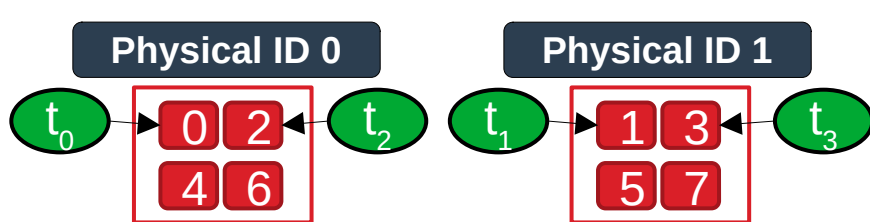
- **close** | **spread** (*por defecto*) | **master**





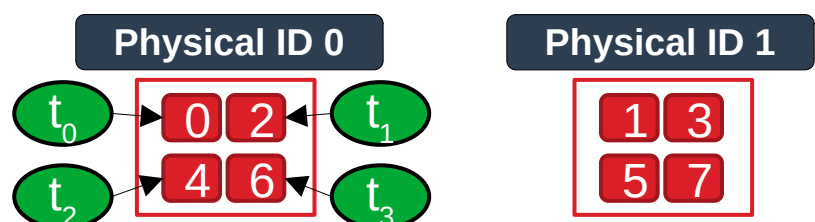
## Afinidad – OMP\_PLACES – Lista de Processors ID

- A partir de la variable de ambiente `OMP_PLACES`, es posible pasar una lista de processors ID donde ubicar los hilos.
- El thread ID coincide con el índice de la lista y el contenido con los Processor ID.
- Suponiendo 4 hilos, dos formas de ubicar los hilos sobre la siguiente arquitectura:



`OMP_PLACES='{0}, {1}, {2}, {3}'`

Equivalente: `OMP_PLACES='{0}:4'`



`OMP_PLACES='{0}, {2}, {4}, {6}'`

Equivalente: `OMP_PLACES='{0}:4:2'`



# OpenMP

## Afinidad – Binding

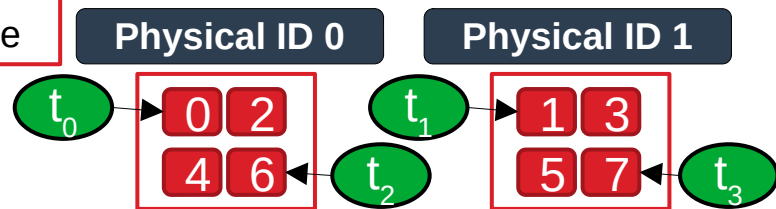
114

- A partir de la variable de ambiente **OMP\_PROC\_BIND** o la cláusula **proc\_bind**, es posible ubicar los hilos automáticamente de acuerdo a ciertas políticas:

**OMP\_PROC\_BIND=spread**

```
#pragma omp parallel proc_bind(spread){  
    ...  
}
```

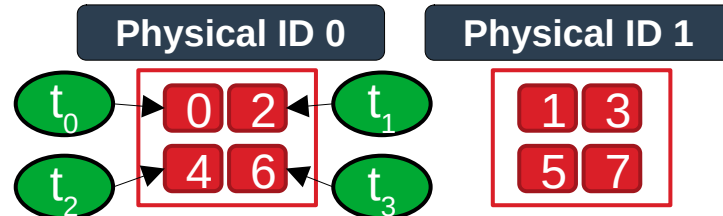
Dispersos  
No comparten cache



**OMP\_PROC\_BIND=close**

```
#pragma omp parallel proc_bind(close){  
    ...  
}
```

Cercanos  
Comparten cache



**OMP\_PROC\_BIND=master**

```
#pragma omp parallel proc_bind(master){  
    ...  
}
```

Cercanos al master thread

