Universidad Nacional Autónoma de México

Facultad de Ingeniería

Sistemas operativos

OpenMP desde la terminal: cómo el kernel ejecuta el paralelismo.

Profesor: Dr. Gunnar Eyal Wolf Iszaevich

Grupo 08

Integrantes:

- Sierra García Mariana

- Tapia García Andres

Semestre 2026-1

Índice

1. Introducción a OpenMP

La necesidad de ejecutar tareas en paralelo surge con los primeros sistemas multiusuario y multiprocesador de los años 60 y 70. Unix introdujo el concepto de *procesos* y el kernel Linux heredó esta arquitectura ampliándola con soporte de hilos ligeros (*lightweight processes*).

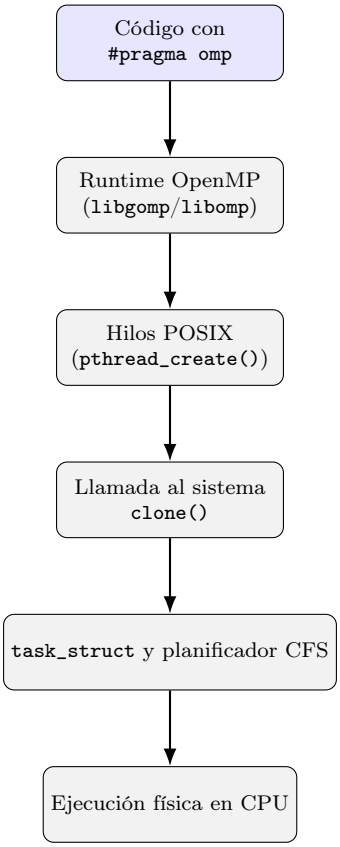
En los años 90 surgió OpenMP, una API estandar para paralelismo en memoria compartida en C, C++ y Fortran (Informatec Digital, 2025). OpenMP permite crear y sincronizar hilos usando directivas de compilador (#pragma), en lugar de invocar manualmente a la biblioteca POSIX Threads (pthreads) (Swahn, 2016).

2. Cómo el kernel Linux ejecuta hilos

En Linux, tanto los procesos como los hilos son instancias de una misma estructura de datos: `task_struct`. Esta contiene el contexto de ejecución (registros, estado, apuntadores, etc.) y es gestionada por el planificador del kernel (LinuxVox, 2025). La llamada clave es `clone()`, que permite crear tareas con distintos grados de independencia. Los hilos comparten el mismo espacio de direcciones, lo que constituye la base de los hilos POSIX (Whileint, 2021).

```
# Observar tareas asociadas a un proceso
ls /proc/<PID>/task/

# Ver cantidad de hilos
cat /proc/<PID>/status | grep Threads
```



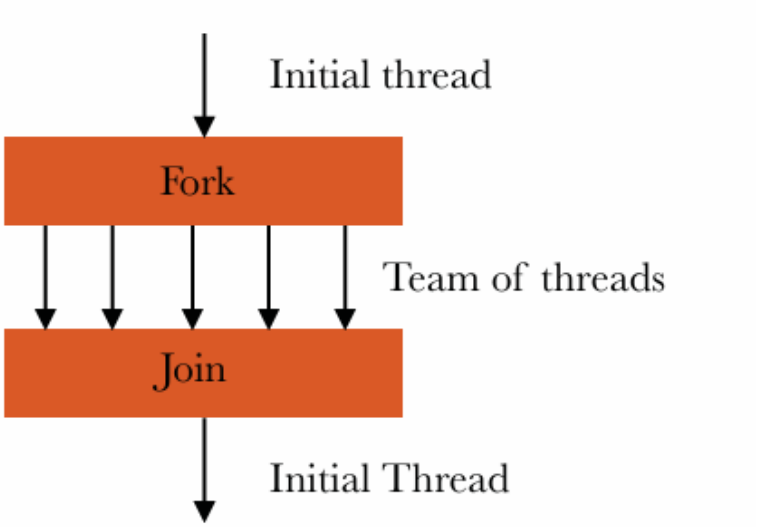
El **Completely Fair Scheduler (CFS)** distribuye los hilos entre los núcleos del procesador, asegurando la igualdad en el uso de CPU (Linux Kernel Development Community, 2025). Así, cada hilo OpenMP se traduce finalmente en una tarea que el kernel programa.

No confundir el planificador CFS con el algoritmo FCFS que vimos en clase. Aunque suenan parecidos, según (Wolf, 2025a) el FCFS solo ejecuta los procesos en el orden que llegan, sin importar si uno tarda mucho o poco. En cambio el CFS reparte el tiempo de CPU de forma mas igualada, dejando que todos los hilos tengan oportunidad de correr.

2.1. Modelo Fork-Join de OpenMP

OpenMP utiliza un modelo de paralelismo llamado **Fork-Join** para ejecutar tareas en paralelo sobre hilos del sistema. Cuando el hilo principal (master thread) encuentra una directiva paralela, como un `#pragma omp parallel for`, ocurre un fork: el runtime de OpenMP crea un equipo de hilos que ejecutan tareas en la región paralela. Cada hilo ejecuta una parte del trabajo.

Una vez que todos los hilos terminan con sus tareas, ocurre un **join**: los hilos extras terminan y el hilo master continúa ejecutando el código que sigue después de la región paralela. Este comportamiento asegura que el programa mantenga la coherencia y que las tareas paralelas terminen antes de continuar con la ejecución secuencial.



Recuperado de (Swahn, 2016)

- **Fork:** La creación de hilos por parte del runtime OpenMP se traduce en llamadas a `pthread_create()`, que internamente usan `clone()` para generar tareas (`task_struct`) en el kernel.
- **Join:** El master thread espera a que todos los hilos terminen, lo que corresponde a la sincronización que realiza el kernel antes de que el hilo principal continúe. Cada hilo finaliza su ejecución y el CFS se encarga de que los hilos tengan oportunidad de correr en los núcleos de la CPU.

El modelo Fork-Join no utliza la llamada al sistemma Fork() vista en clase(Wolf, 2025b), dado que esta crea **procesos** independientes con espacios de memoria separados. En su lugar el modelo activa un conjunto de hilos que comparten el **mismo espacio de direcciones**, algo similiar a la creación de múltiples instancias de **threading.Thread() en Python.** Este enfoque contrasta con `fork()` y permite que la estructura paralela concluya coherentemente mediante la operación de join, que sincroniza la finalización de todas las hebras antes de continuar con la ejecución secuencial.

3. OpenMP: del codigo al kernel

OpenMP realiza el paralelismo con sus directivas, El compilador traduce estas mismas en llamadas al *runtime* de OpenMP (`libgomp`), y tambien usa pthreads para crear los hilos reales.

Ejemplo de manejo de Hilos en OpenMP con clausulas

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

int main() {
  #pragma omp parallel num_threads(4)
  {
    printf("Hilo %d de %d\n", omp_get_thread_num(),
    omp_get_num_threads());
  }
}
```

Cada hilo ejecutará la sección paralela de manera concurrente, y de esta manera el kernel lo tratará como una `task_struct` independiente.

3.1. Ejecución de Código en C con biblioteca omp.

Ejecución En Bash

```
gcc -fopenmp NombreArchivo.c -o NombreEjecutable
./NombreEjecutable
```

4. Cláusulas de OpenMP y sincronización

OpenMP ofrece diferentes formas de trabajar con los hilos, mediante sus clausulas, mencionaremos las principales para entender la idea de como funciona OpenMP.

4.1. Section

Distribuye a los hilos generados por parallel en las secciones definidas dentro de la clausula sections

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            printf("Sección 1 ejecutada por el hilo %d\n",
                omp_get_thread_num());
        }

        #pragma omp section
        {
            printf("Sección 2 ejecutada por el hilo %d\n",
                omp_get_thread_num());
        }
    }
}
```

4.2. For

Distribuye iteraciones de un bucle entre los hilos del equipo:

```
#pragma omp parallel for
for (int i=0; i<8; i++)
  printf("i=%d ejecutado por hilo %d\n", i, omp_get_thread_num());
```

4.3. Critical

Permite exclusión mutua es decir, solo un hilo accede a la sección a la vez.

```
int contador = 0;
omp_set_num_threads(4); // Se puede fijar el número de hilos
#pragma omp parallel for
    for (int i = 0; i < 20; i++) {
        #pragma omp critical
        {
          contador++;
          printf("Hilo %d incrementó el contador a %d (iteración %d)\n",
          omp_get_thread_num(), contador, i);
        }
    }
printf("Valor final del contador = %d\n", contador);
```

4.4. Barrier

La directiva `barrier` fuerza a que **todos los hilos del equipo** que participan en la región paralela se detengan y esperen hasta que todos hayan alcanzado ese punto. Es una herramienta de sincronización que asegura que ningún hilo continúe la ejecución más allá de la barrera antes que los demás.

```
#pragma omp parallel num_threads(4)
{
  printf("Primera parte\n");
  #pragma omp barrier
  printf("Segunda parte\n");
}
```

4.5. Reduction

Combina resultados parciales en una operación final sin interferencia:

```
int suma=0;
#pragma omp parallel for reduction(+:suma)
for (int i=0;i<100000;i++)
  suma++;
printf("Suma total: %d\n",suma);
```

4.6. Atomic

Garantiza que una operación simple sobre una variable compartida se ejecute de forma atómica (que solo un hilo modifique a la vez una variable), evitando condiciones de carrera con menor sobrecarga que `critical`.

```
int contador = 0;

#pragma omp parallel for
for (int i = 0; i < 1000; i++) {
  #pragma omp atomic
  contador++;
}
printf("Valor final: %d\n", contador);
```

La información de esta sección fue recuperada de (Microsoft, 2025)

5. Temas Extra para la exposición

5.1. Comparación con semáforos

Cláusula OpenMP	Equivalente en semaforos POSIX
`critical`	`sem_wait / sem_post` (mutex binario)
`barrier`	Semáforo de conteo o barrera sincronizada
`reduction`	Operaciones atómicas con semáforos múltiples

Recomendamos fuertemente leer la lectura de (Swahn, 2016) , ya que abarca esta relación de openMP con pthreads que presentamos aquí.

5.2. Codigo Paralelizado contra Codigo Secuencial

Comparativa: Secuencial vs Paralelo

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

int main() {
    long N = 100000000; // tamaño del vector
    double *vector = malloc(N * sizeof(double));

    if (vector == NULL) {
        printf("Error al asignar memoria\n");
        return 1;
    }

    // Inicializar el vector
    for (long i = 0; i < N; i++) {
        vector[i] = 1.0;
    }

    /* ================================
     *   SUMA SECUENCIAL
     * ================================ */
```

```c
    double t0 = omp_get_wtime();
    double suma_secuencial = 0.0;

    for (long i = 0; i < N; i++) {
        suma_secuencial += vector[i];
    }

    double t1 = omp_get_wtime();
    double tiempo_secuencial = t1 - t0;

    /* ================================
     *    SUMA PARALELIZADA
     * ================================ */
    int num_hilos = 4;
    double suma_paralela = 0.0;

    double t2 = omp_get_wtime();

    #pragma omp parallel num_threads(num_hilos)
    {
        double suma_local = 0.0;

        #pragma omp for
        for (long i = 0; i < N; i++) {
            suma_local += vector[i];
        }

        #pragma omp atomic
        suma_paralela += suma_local;
    }

    double t3 = omp_get_wtime();
    double tiempo_paralelo = t3 - t2;

    /* ================================
     *    RESULTADOS
     * ================================ */
    printf("Suma secuencial: %f\n", suma_secuencial);
    printf("Tiempo secuencial: %f segundos\n\n", tiempo_secuencial);

    printf("Suma paralela:   %f\n", suma_paralela);
    printf("Tiempo paralelo: %f segundos\n", tiempo_paralelo);

    free(vector);
    return 0;
}
```

5.2.1. Ejecución

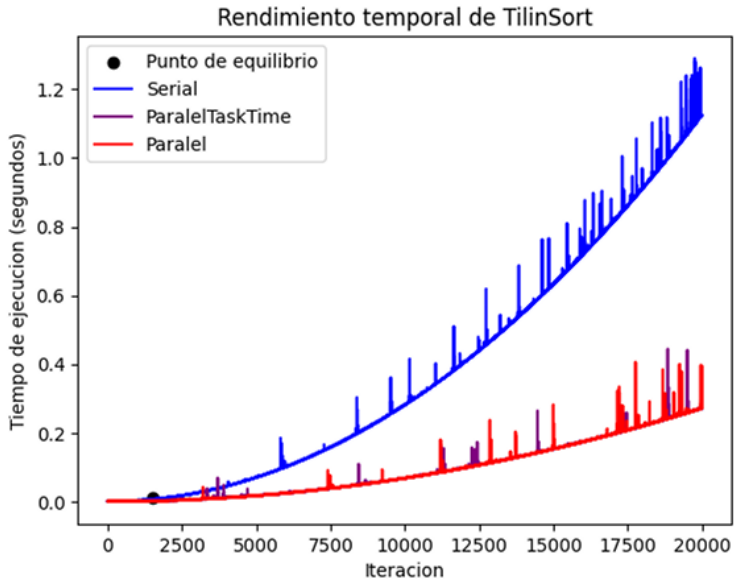
```
andrestg@192:~/Respaldo_Repositorios/programasC$ ls
vs.c
andrestg@192:~/Respaldo_Repositorios/programasC$ gcc -fopenmp vs.c -o comparativa
andrestg@192:~/Respaldo_Repositorios/programasC$ ./comparativa
Suma secuencial: 100000000.000000
Tiempo secuencial: 0.415696 segundos

Suma paralela:   100000000.000000
Tiempo paralelo: 0.142633 segundos
andrestg@192:~/Respaldo_Repositorios/programasC$
```

El código paralelizado fue casi 3 veces más rápido que el código secuencial.

5.3. Consideraciones

Para pequeñas instancias la paralelización no es una buena opcion, ya que tarda mas y gasta muchisimos mas recursos, por eso es importante conocer el punto de inflexion de cada algoritmo para saber si conviene o no la paralelización. Mostramos una gráfica obtenida de un proyecto, donde se paralelizo un algoritmo de ordenamiento recuperado de (Tapia, 2023):



Proyecto que pondremos a su disposición en el repositorio

hasta en el código en el que hacemos un versus entre el paralelizado y el secuencial tuvimos que poner una iteración increiblemente alta:

```
long N = 100000000; // tamaño del vector
    double *vector = malloc(N * sizeof(double));
```

para que se pudiera observar como es que el codigo paralelizado le gana por mucho al secuencial para iteraciones grandes.

Referencias

Informatec Digital. (2025). *Openmp: Qué es, cómo funciona y todo lo que necesitas saber.* Descargado de https://informatecdigital.com/que-es-openmp/

Linux Kernel Development Community. (2025). *Gestor de tareas cfs: Documentación oficial.* Descargado de https://www.kernel.org/doc/html/latest/translations/sp_SP/scheduler/sched-design-CFS.html

LinuxVox. (2025). *task_struct en el kernel de linux.* Descargado de https://linuxvox.com/blog/task_struct-linux/

Microsoft. (2025). *Openmp directives.* Descargado de https://learn.microsoft.com/es-es/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170 (Consultado el 11 de noviembre de 2025)

Swahn, H. (2016). *Pthreads and openmp: A study on the relationship between posix threads and openmp.* Descargado de https://www.diva-portal.org/smash/get/diva2%3A944063/FULLTEXT02

Tapia, A. (2023). *Proyecto02- versiones paralelas de un algoritmo serial.* (Curso Estructura de Datos II [Profesor: Solano Galvez Jorge])

Whileint. (2021). *¿qué es la llamada al sistema de clonación en linux?* Descargado de https://whileint.com/tech/pablo/que-es-la-llamada-al-sistema-de-clonacion-en-linux/

Wolf, G. (2025a). *Planificación de procesos (sistemas operativos 2025.10.21).* Video en YouTube. Descargado de https://www.youtube.com/watch?v=NUCwBGS0aj8

Wolf, G. (2025b, sep). *Sistemas operativos 2025.09.04: Administración de procesos.* Video en YouTube. Descargado de https://youtu.be/34QOTfafCI4 (Consultado el 21 de noviembre de 2025)