Práctica 3 - Threads

> Requisitos >
>
> Para realizar esta práctica se puede usar la misma máquina virtual de la práctica 1 o una de su elección si resulta más cómodo (por ejemplo una VM con interfaz gráfica y un IDE).

Threading (ULT y KLT)

Conceptos Generales

1. ¿Cuál es la diferencia fundamental entre un proceso y un thread?

La diferencia fundamental entre un proceso y un thread radica principalmente en la forma en que gestionan la memoria y los recursos del sistema.

- Un proceso es una instancia de un programa en ejecución. Cada proceso tiene su propio espacio de direcciones de memoria, lo que significa que cada proceso tiene su propia copia del código del programa, sus datos y su pila. Por defecto, los procesos no comparten memoria de lectura/escritura con otros procesos. Los procesos son administrados por el scheduler del sistema operativo y, si hay múltiples procesadores (ya sean chips físicos, cores o mediante hyperthreading), pueden ejecutarse en paralelo.
- Por otro lado, un thread (o hilo) es una unidad de ejecución dentro de un proceso. Múltiples threads pueden existir dentro del mismo proceso y comparten el mismo espacio de direcciones. Esto significa que los threads dentro del mismo proceso pueden acceder y modificar las mismas variables globales, el heap y el código del programa. Sin embargo, cada thread tiene su propia pila de ejecución, registros y contador de programa.
  Otra diferencia relevante es que los procesos son más pesados de crear y cambiar de contexto, ya que requieren duplicación o aislamiento completo del espacio de direcciones y otros recursos. En cambio, los threads son más ligeros, ya que compartir memoria entre ellos reduce el costo de creación, conmutación de contexto y comunicación.

2. ¿Qué son los User-Level Threads (ULT) y cómo se diferencian de los Kernel-Level Threads (KLT)?

Los User-Level Threads (ULT) son threads que se gestionan por un scheduler en el espacio de usuario, es decir, por una biblioteca dentro del proceso de la aplicación.

- Una característica fundamental de los ULTs es que no se ejecutan en paralelo sobre múltiples núcleos de procesador.
- Suelen utilizar un modelo de scheduling no preemptivo o cooperativo, donde cada hilo debe ceder voluntariamente el control para que otro hilo pueda ejecutarse.
- Los ULTs dentro del mismo proceso comparten el mismo espacio de memoria de lectura/escritura.
- Sin embargo, una desventaja importante es que si un ULT realiza una llamada al sistema bloqueante, todo el proceso se bloqueará, ya que el kernel solo es consciente de la existencia del proceso, no de sus threads internos.

Por otro lado, los Kernel-Level Threads (KLT) son threads que se gestionan directamente por el scheduler del sistema operativo.

- Los KLTs pueden ejecutarse en paralelo en múltiples procesadores o núcleos, si están disponibles.
- Aunque comparten el mismo espacio de direcciones (código, datos, heap) dentro de un proceso, cada KLT tiene su propia pila de ejecución.
- A diferencia de los ULTs, si un KLT realiza una operación bloqueante, solo ese thread se bloqueará, permitiendo que otros threads del mismo proceso (u otros procesos) continúen su ejecución.

3. ¿Quién es responsable de la planificación de los ULT? ¿y los KLT? ¿Cómo afecta esto al rendimiento en sistemas con múltiples núcleos?

Como describí antes, los ULTs son planificados por un scheduler en el espacio de usuario normalmente proporcionado por una biblioteca dentro del proceso de la aplicación, utilizando un modelo de scheduling no preemptivo o cooperativo mientras que, los KLTs son planificados directamente por el scheduler del sistema operativo que reside en el espacio del kernel.

En sistemas con múltiples núcleos, esta diferencia en la planificación tiene un impacto significativo en el rendimiento:

- *User-Level Threads (ULT):* Dado que la planificación de los ULT se realiza en el espacio de usuario y el kernel solo es consciente del proceso en sí, los ULT dentro del mismo proceso generalmente no pueden ejecutarse en paralelo en múltiples núcleos. Aunque puede haber múltiples ULTs listos para ejecutarse, el kernel asignará tiempo de CPU al proceso como una sola entidad. Si el scheduler de ULT decide cambiar entre sus threads, esto ocurrirá dentro del tiempo de CPU asignado al proceso en un único núcleo. Por lo tanto, *los ULT no aprovechan inherentemente la capacidad de procesamiento paralelo de los sistemas multi-core.* Además, si un ULT realiza una llamada al sistema bloqueante, todo el proceso se bloqueará, impidiendo que otros ULTs del mismo proceso se ejecuten.

- *Kernel-Level Threads (KLT):* Como los KLT son gestionados por el scheduler del sistema operativo, pueden ejecutarse realmente en paralelo en múltiples procesadores o núcleos si están disponibles. El kernel puede asignar diferentes KLTs del mismo proceso (o de diferentes procesos) a diferentes núcleos, *permitiendo una utilización eficiente del hardware multi-core y un aumento significativo en el rendimiento para tareas que pueden ser paralelizadas.* Además, si un KLT realiza una operación bloqueante, solo ese thread se bloqueará, mientras que otros KLTs del mismo proceso pueden continuar su ejecución.

4. ¿Cómo maneja el sistema operativo los KLT y en qué se diferencian de los procesos?

Arriba ya hablamos de como maneja el sistema operativo los KLT, las diferencias con los procesos serían:

Característica	Proceso	KLT
Identidad	PID (Process ID)	TID (Thread ID)
Espacio de memoria	Propio e independiete	Compartido con otros threads del proceso
Pila de ejecución	Tiene su propia Pila	Tiene su propia Pila
Planificación	Planificado individualmente por el scheduler del SO	Planificado individualmente por el scheduler del SO
Ejecución en múltiples procesadores	Pueden ejecutarse paralelamente	Pueden ejecutarse paralelamente
Cambio de contexto	Más costoso	Más liviano
Comunicación	Mas compleja, necesita mecanismos de Inter-Process Communication (IPC)	Más simple, uso de memoria compartida
Creación	Más lenta y consumidora de recursos, se hace mediante `fork()`	Más rápida y barata de realizar, se hace mediante `pthread_create()`
Bloqueo	Si un proceso realiza una llamada al sistema bloqueante, todo el proceso se bloqueará	Si un KLT realiza una llamada al sistema bloqueante, solo ese thread se bloqueará

5. ¿Qué ventajas tienen los KLT sobre los ULT? ¿Cuáles son sus desventajas?

Ventajas de los KLT sobre los ULT:

- *Paralelismo real:* Los KLT pueden ejecutarse en paralelo, aprovechando la capacidad de procesamiento concurrente del hardware en sistemas con múltiples procesadores.
- *Manejo de llamadas al sistema bloqueantes:* Si un KLT realiza una llamada al sistema que se bloquea, solo ese thread se bloqueará, permitiendo que otros threads del mismo proceso continúen su ejecución.
- *Mejor aprovechamiento en escenarios con bloqueo:* Debido a que el bloqueo de un KLT no afecta a otros, son más adecuados para aplicaciones que realizan muchas operaciones de entrada/salida donde un thread podría quedar bloqueado esperando datos, mientras otros threads pueden seguir procesando.
- *Integración con el sistema operativo:* Al ser gestionados por el kernel, los KLT pueden tener una integración más profunda con otras funcionalidades del sistema operativo, como la planificación y la gestión de recursos.

Desventajas de los KLT sobre los ULT:

- *Mayor costo de creación y cambio de contexto:* La creación y la conmutación de contexto entre KLT generalmente implican una intervención del kernel, lo que conlleva una mayor sobrecarga en comparación con los ULT. Los ULT, al ser gestionados en espacio de usuario, tienen un costo de creación y cambio de contexto mucho menor.
- *Mayor dependencia del sistema operativo:* La implementación de KLT es específica del sistema operativo. Los ULT, por otro lado, son independientes del sistema operativo, ya que su gestión se realiza en espacio de usuario por una biblioteca.
- *Potencialmente mayor uso de recursos del kernel:* Cada KLT requiere estructuras de datos en el kernel para su gestión, lo que puede llevar a un mayor consumo de recursos del kernel en comparación con los ULT, donde muchos ULT pueden ser gestionados por un único thread a nivel de kernel.

6. Qué retornan las siguientes funciones:
    1. `getpid()`
    2. `getppid()`
    3. `gettid()`
    4. `pthread_self()`
    5. `pth_self()`

Función	Nivel	Qué identifica	Tipo de retorno
`getpid()`	Kernel	PID del proceso actual	`` `pid_t ``
`getppid()`	Kernel	PID del proceso padre del actual	`pid_t`
`gettid()`	Kernel	TID del thread actual	`pid_t`

Función	Nivel	Qué identifica	Tipo de retorno
`pthread_self()`	Espacio de Usuario	TID del thread actual en pthreads	`pthread_t`
`pth_self()`	Espacio de Usuario	TID del thread actual en GNU Pth (GNU Portable Threads)	`pth_t`

7. ¿Qué mecanismos de sincronización se pueden usar? ¿Es necesario usar mecanismos de sincronización si se usan ULT?

Para coordinar la ejecución y el acceso a recursos compartidos entre múltiples threads o procesos, se pueden utilizar diversos mecanismos de sincronización. La elección del mecanismo adecuado depende del tipo de entidad de ejecución (procesos, KLTs o ULTs), el entorno de ejecución (kernel o espacio de usuario) y si comparten o no memoria.

- *Para procesos que comparten memoria:*
  - Semáforos POSIX (`sem_open`).
  - Mutexes de Pthreads ubicados en regiones de memoria compartida.
  - Memoria compartida y sincronización explícita con semáforos o mutexes compartidos.
- *Para KLTs:*
  - Mutexes de Pthreads.
  - Variables condición.
  - Barreras.
  - Semáforos.
- *Para la sincronización de E/S en procesos:*
  - E/S sincrónica (bloqueo en lectura/escritura).
  - `flock()` → Bloqueo asesor (advisory) sobre archivos.
  - `lockf()` → Bloqueo POSIX sobre regiones de archivos.
  - `fcntl()` → Permite tanto bloqueos advisory como mandatory en archivos.
- *Para ULTs:*
  - Mutexes provistos por la biblioteca de ULT.
  - Semáforos provistos por la biblioteca de ULT.
  - Otros mecanismos de sincronización que ofrezca la biblioteca de ULT.

Es necesario usar mecanismos de sincronización incluso si se utilizan User-Level Threads (ULTs). **Razón:**

- Debido a que múltiples ULTs pueden acceder y modificar las mismas áreas de memoria compartida, *sin una sincronización adecuada, pueden ocurrir condiciones de carrera (race*

conditions), donde el resultado de la ejecución depende del orden no determinístico en que los threads acceden a los datos compartidos. Esto puede llevar a *inconsistencia de datos y comportamientos inesperados en la aplicación.* Por lo tanto, para *garantizar la integridad de los datos y la correcta coordinación* entre los ULTs que acceden a recursos compartidos, es imprescindible utilizar mecanismos de sincronización proporcionados por la biblioteca de ULT que se esté utilizando.

8. Procesos:
    1. ¿Qué utilidad tiene ejecutar `fork()` sin ejecutar `exec()`?

- Ejecutar `fork()` sin `exec()` permite que el proceso padre y el hijo **continúen ejecutando el mismo programa**, pero **puedan divergir en comportamiento** ejecutando distintas secciones del código. *La utilidad principal radica en que ambos procesos pueden colaborar en paralelo:* por ejemplo, el padre puede seguir ejecutando su tarea principal, y el hijo puede realizar tareas auxiliares como manejar conexiones, atender peticiones o generar logs.

    2. ¿Qué utilidad tiene ejecutar `fork()` + `exec()`?

- Esta combinación es la **forma estándar en sistemas Unix/Linux** para iniciar nuevos programas desde otro proceso. La utilidad principal es permitir al proceso padre lanzar otro programa y continuar ejecutándose en paralelo o bien esperar su finalización mediante `wait()` o `waitpid()`.

    3. ¿Cuál de las 2 asigna un nuevo PID `fork()` o `exec()`?

- `fork()` es la que asigna un **nuevo PID** al proceso hijo.

    4. ¿Qué implica el uso de Copy-On-Write (COW) cuando se hace `fork()`?

- Cuando se llama a `fork()`, el sistema operativo no copia inmediatamente toda la memoria del proceso padre al hijo. En cambio, usa la técnica de **Copy-On-Write (COW)**:
    - Padre e hijo comparten inicialmente las mismas páginas de memoria, marcadas como de solo lectura.
    - Si uno de los dos intenta escribir en alguna página, se crea *una copia privada solo para ese proceso*, y el otro sigue utilizando la versión original.
    - *Ahorra memoria y mejora el rendimiento*, ya que evita copias innecesarias.
    - Es especialmente útil si el hijo ejecutará enseguida un `exec()`, ya que no se necesita copiar memoria que pronto será descartada.

    5. ¿Qué consecuencias tiene no hacer `wait()` sobre un proceso hijo?

- Si el proceso padre no llama a `wait()` (o `waitpid()`), cuando el hijo termina su ejecución este queda en estado **zombie**.
    - Un proceso *zombie* es un proceso que ha finalizado pero cuya entrada en la *tabla de procesos del kernel* aún no ha sido eliminada. Permanece en espera de que su padre recoja su estado de salida mediante `wait()`.

    - Los zombies *consumen entradas en la tabla de procesos*. Si se acumulan muchos, se puede llegar al límite de procesos que el sistema permite, provocando fallos al intentar crear nuevos procesos.
6. ¿Quién tendrá la responsabilidad de hacer el `wait()` si el proceso padre termina sin hacer `wait()`?

- Si un proceso padre **muere sin haber hecho** `wait()`, el sistema operativo re-asigna automáticamente sus procesos hijos a un nuevo padre: el proceso `init` (tradicionalmente PID 1).
    - Este proceso se encarga de *adoptar los procesos huérfanos*. Además, ejecuta periódicamente llamadas a `wait()` para recoger el estado de salida de estos procesos y *limpiar zombies*.

9. Kernel Level Threads:
    1. ¿Qué elementos del espacio de direcciones comparten los threads creados con `pthread_create()`?

- Los threads creados con `pthread_create()` comparten el mismo **espacio de direcciones de memoria** dentro de un proceso. Esto incluye: código, datos globales y estáticos, heap y archivos abiertos.

    2. ¿Qué relaciones hay entre `getpid()` y `gettid()` en los KLT?

- `getpid()` nos va a retornar el PID del proceso al que pertenece el thread, todos los threads dentro del mismo proceso tendrán el mismo PID. `gettid()` nos retorna el TID del thread actual a nivel de kernel, cada KLT tiene su TID único asignado por el kernel. Por lo tanto, `getpid()` identifica al proceso, mientras que `gettid()` identifica a un thread específico dentro de ese proceso a nivel del sistema operativo.

    3. ¿Por qué `pthread_join()` es importante en programas que usan múltiples hilos? ¿Cuándo se liberan los recursos de un hilo zombie?

- `pthread_join()` es importante en programas que usan múltiples hilos porque **permite que el hilo principal (o cualquier otro hilo) espere a que un hilo específico termine su ejecución:**
    - Al llamar a `pthread_join(t, NULL)`, el hilo que realiza la llamada se bloquea hasta que el hilo `t` finaliza.
    - Esto es crucial para asegurar que todas las tareas realizadas por los hilos se completen antes de que el proceso principal termine o continúe con otras operaciones que dependan de los resultados de esos hilos.
    - Si no se llama a `pthread_join()` para un hilo, ese hilo puede convertirse en un hilo "zombie" al terminar. Los recursos de un hilo zombie se liberan cuando el proceso que lo creó termina.

    4. ¿Qué pasaría si un hilo del proceso bloquea en `read()`? ¿Afecta a los demás hilos?

- Si hablamos de **KLTs** solo ese thread se va a bloquear pero si hablamos de **ULTs** se loquean todos los demás **ULTs** dentro del mismo proceso.

5. Describí qué ocurre a nivel de sistema operativo cuando se invoca `pthread_create()` (¿es syscall? ¿usa clone?).

- `pthread_create()` **no es una syscall directa**, sino que se trata de una función proporcionada por la biblioteca que **utiliza una syscall interna** para crear un hilo. La función realiza los siguientes pasos:
  - *Reserva espacio para el descriptor de hilo:* Esto incluye información como su estado, identificador, y pila.
  - *Crea un hilo (subproceso):* Internamente, si el sistema operativo lo permite, `pthread_create()` invoca una *syscall de creación de un hilo*, que en Linux típicamente es *`clone()`*. Esta syscall es muy flexible y se puede usar para crear un nuevo hilo en el espacio de memoria del proceso actual, lo que permite compartir ciertas características entre el hilo padre y el hilo hijo.
  - *Configuración del hilo:* En el caso de Linux, `clone()` puede ser llamado con ciertos parámetros para especificar qué recursos se comparten entre el hilo padre y el hijo.
  - *Vinculación con la biblioteca de hilos:* Después de que el hilo hijo es creado, la biblioteca de hilos registra el hilo recién creado en sus estructuras internas y lo coloca en la cola de ejecución.

10. User Level Threads:
    1. ¿Por qué los ULTs no se pueden ejecutar en paralelo sobre múltiples núcleos?

- Los **ULTs no pueden ejecutarse en paralelo sobre múltiples núcleos** debido a que el **sistema operativo no los conoce como hilos independientes**, sino que los gestiona a través de un solo hilo de ejecución del proceso, que es el hilo del **proceso padre**.

    2. ¿Qué ventajas tiene el uso de ULTs respecto de los KLTs?

- Los **ULTs** tienen varias ventajas sobre los **KLTs**, entre las que destacan:
  - *Menor sorecarga de gestión:* Los ULTs son gestionados completamente en el espacio de usuario, por lo que *el sistema operativo no necesita involucrarse* en la creación, destrucción o planificación de estos hilos, lo que reduce la sobrecarga asociada con las llamadas al sistema para manejar hilos.
  - *Cambios de contexto más rápidos.*
  - *No requieren privilegios del sistema operativo en su uso.*
  - *Tienen un menor consumo de recursos.*

    3. ¿Qué relaciones hay entre `getpid()`, `gettid()` y `pth_self()` (en GNU Pth)?

- `getpid()`: Esta función devuelve el **PID de proceso** del proceso en ejecución.
- `gettid()`: Esta función devuelve el **TID de hilo** de un hilo, y a diferencia de `getpid()`, cada hilo dentro de un proceso tiene un **TID de hilo único**. Para los hilos gestionados por el kernel (KLTs), esta ID es proporcionada por el kernel. Para los hilos gestionados a nivel de usuario (ULTs), esta ID está gestionada por la biblioteca de hilos.
- `pth_self()`: Específica de la biblioteca **GNU Pth**, esta función devuelve el **TID del hilo gestionado por la biblioteca de hilos**. En sistemas con ULTs, este identificador es

gestionado por la biblioteca de hilos y puede ser diferente del TID del hilo a nivel del sistema operativo, ya que el kernel no está al tanto de estos hilos.

4. ¿Qué pasaría si un ULT realiza una syscall bloqueante como `read()`?

- Si un **ULT** realiza una llamada al sistema bloqueante, todo el proceso se bloqueará, ya que el kernel solo es consciente de la existencia del proceso, no de sus threads internos.

5. ¿Qué tipos de scheduling pueden tener los ULTs? ¿Cuál es el más común?

- Los **ULTs** pueden tener 2 tipos de scheduling (**la primera es la más común**):
  - *Planificación cooperativa:*
    - En este tipo de planificación, los hilos se ejecutan hasta que decidan ceder el control explícitamente. Esto significa que un hilo puede ejecutarse durante un período prolongado si no hace una llamada explícita para ceder el control, lo que puede llevar a que un hilo monopolice la CPU.
    - *Ventaja:* Menos sobrecarga porque no hay interrupciones frecuentes.
    - *Desventaja:* Si un hilo no cede el control, puede bloquear a los demás.
  - *Planificación Preemptiva:*
    - En este caso, la biblioteca de hilos interrumpe a los hilos en ejecución después de un cierto tiempo para darle la oportunidad a otros hilos de ejecutarse.
    - *Ventaja:* Mejor uso de la CPU, ya que los hilos no pueden bloquearse entre sí.
    - *Desventaja:* Requiere más complejidad para manejar los cambios de contexto.

11. Global Interpreter Lock:
    1. ¿Qué es el GIL (Global Interpreter Lock)? ¿Qué impacto tiene sobre programas multi-thread en Python y Ruby?

- El **GIL (Global Interpreter Lock)** es un mecanismo presente en las implementaciones **oficiales** de Python (**CPython**) y Ruby (**MRI**) que **permite que solo un hilo ejecute bytecode del lenguaje a la vez dentro de un proceso**.
  - El *GIL* existe porque simplifica la implementación del *intérprete*, sobre todo en lo que respecta a la *gestión de memoria compartida* y el *garbage collector*, y evita la necesidad de locks más finos en estructuras internas del intérprete.
  - EL impacto que tiene en programas multithread es que aunque uses *hilos reales del sistema (KLTs)*, el GIL *impide que se ejecuten en paralelo* si realizan código Python. *Solo un hilo puede tener el GIL a la vez*, así que otros deben esperar aunque estén marcados como "runnable". Esto *limita seriamente el paralelismo real* en *tareas CPU-bound*, como cálculos numéricos o procesamiento de datos pesados.
  - En programas que son *I/O Bound*, el GIL suele liberarse mientras se hace la operación de entrada y salida, en esos casos, los otros hilos *puede aprovechar ese tiempo*, presentando una mejora en la eficiencia.

    2. ¿Por qué en CPython o MRI se recomienda usar procesos en vez de hilos para tareas intensivas en CPU?

- Dado que el GIL limita la ejecución concurrente de hilos en tareas CPU-bound, la solución en Python y Ruby es usar procesos:
  - *Ventajas:*
    - Cada proceso tiene *su propia instancia del intérprete* y su *propio GIL*, lo que permite que *se ejecuten en paralelo real en múltiples núcleos*.
    - Se evita la contención del GIL, lo cual es clave para *tareas intensivas en CPU*.

Práctica Guiada

1. Instale las dependencias necesarias para la práctica (strace, git, gcc, make, libc6-dev, libpth-dev, python3, htop y podman):

```
apt update
apt install build-essential libpth-dev python3 python3-venv strace git htop
podman
```

2. Clone el repositorio con el código a usar en la práctica: `git clone https://gitlab.com/unlp-so/codigo-para-practicas.git`
3. Resuelva y responda utilizando el contenido del directorio `practica3/01-strace`:

    1. Compile los 3 programas C usando el comando make.
    2. Ejecute cada programa individualmente, observe las diferencias y similitudes del PID y THREAD_ID en cada caso. Conteste en qué mecanismo de concurrencia las distintas tareas:
        1. Comparten el mismo PID y THREAD_ID.
    - En el programa `03-ul-thread` se ve que desde el punto de vista del sistema operativo, el PID y TID son los mismos (todo corre en el mismo hilo del kernel) pero el PTH_ID (identificador de GNU Pth) es diferente, el usuario ve múltiples "hilos" pero el kernel no.
        2. Comparten el mismo PID pero con diferente THREAD_ID.
    - En el programa `02-kl-thread` el PID es el mismo (mismo proceso), pero los TID son distintos porque se está usando `pthread_create()` para crear hilos a nivel de kernel.
        3. Tienen distinto PID.
    - En el programa `01-subprocess` cada proceso tiene su propio PID y TID, ya que un proceso creado con `fork()` es completamente independiente.

Salida de los programas:

```
so@so:~/codigo-para-practicas/practica3/01-strace$ ./01-subprocess
Parent process: PID = 5094, THREAD_ID = 5094
Child process: PID = 5095, THREAD_ID = 5095
so@so:~/codigo-para-practicas/practica3/01-strace$ ./02-kl-thread
Parent process: PID = 5146, THREAD_ID = 5146
Child thread: PID = 5146, THREAD_ID = 5147
so@so:~/codigo-para-practicas/practica3/01-strace$ ./03-ul-thread
Parent process: PID = 5188, THREAD_ID = 5188, PTH_ID = 94511353649392
Child thread: PID = 5188, THREAD_ID = 5188, PTH_ID = 94511353651936
```

3. Ejecute cada programa usando strace `strace ./nombre_programa > /dev/null` y responda:
    1. ¿En qué casos se invoca a la systemcall `clone` o `clone3` y en cuál no? ¿Por qué?

- En el programa `01` si se usa `clone`, lo vemos en esta línea: `clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7f7c6cc04a10) = 5418`. Esto ocurre porque el programa hace uso de `fork()` que internamente usa `clone()` para crear un nuevo proceso.
- En el programa `02` si se usa `clone3`, lo vemos en esta línea: `clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, child_tid=0x7f6042bd6990, parent_tid=0x7f6042bd6990, exit_signal=0, stack=0x7f60423d6000, stack_size=0x7fff80, tls=0x7f6042bd66c0} ⇒ {parent_tid=[0]}, 88) = 5476`. Esto ocurre porque el programa hace uso de `pthread_create` que internamente hace uso de `clone3()` con los flags que se le envían por parámetro para crear un hilo a nivel de kernel.
- En el porgrama `03` o se hace uso ni de `clone` ni de `clone3` ya que el programa no está creando hilos a nivel de kernel sino que a nivel de usuario.

    2. Observe los flags que se pasan al invocar a `clone` o `clone3` y verifique en qué caso se usan los flags CLONE_THREAD y CLONE_VM.

- En el caso del programa `01` no se usan las flags CLONE_THREAD y CLOSE_VM, lo que sugiere que **no se está compartiendo el espacio de memoria (como sería el caso de los hilos de kernel), ni se está creando un hilo como parte de un grupo de hilos existente**. Pero se usan estas:
    - *CLONE_CHILD_CLEARTID:* Este flag se usa para limpiar el identificador del hilo (TID) del hijo cuando el hilo termina.
    - *CLONE_CHILD_SETTID:* Permite establecer el TID del hijo al valor especificado en `child_tidptr` (en este caso `0x7f7c6cc04a10`).
    - *SIGCHLD:* No es un flag de `clone()`, pero se refiere a la señal que el padre recibe cuando el hijo termina.

- En el caso del programa `02` si se están usando las CLONE_THREAD y CLOSE_VM **ya que se están creando hilos**, exactamente se usa todas estas:
  - *CLONE_VM:* Permite que el proceso hijo comparta el mismo espacio de memoria (por lo que la escritura en la memoria afectará a ambos procesos).
  - *CLONE_THREAD:* Este flag indica que el proceso hijo es un hilo, *es parte del mismo grupo de hilos que el padre*, compartiendo recursos como la memoria, los descriptores de archivo, etc.
  - *CLONE_VM + CLONE_THREAD:* Juntos, estos flags indican que se está creando un *hilo* dentro del mismo espacio de direcciones de memoria, compartiendo recursos con el proceso padre.
  - *CLONE_FS, CLONE_FILES, CLONE_SIGHAND:* Permiten compartir el sistema de archivos, los descriptores de archivo y las señales, respectivamente. Esto es característico de la creación de hilos que comparten más que solo la memoria.
  - *CLONE_SETTLS:* Se utiliza para compartir el área de almacenamiento de TLS (Thread Local Storage) entre el padre y el hijo.
  - *CLONE_PARENT_SETTID:* Establece el TID del padre (en el hijo).
  - *CLONE_CHILD_CLEARTID:* Similar al caso anterior, limpia el TID del hijo cuando termina.

3. Investigue qué significan los flags CLONE_THREAD y CLONE_VM usando la manpage de clone y explique cómo se relacionan con las diferencias entre procesos e hilos.

- Según la **manpage** de `clone` "*`CLONE_VM`: If CLONE_VM is set, the calling process and the child process run in the same memory space.*". Es decir, **ambos comparten el mismo espacio de direcciones de memoria**. Cambios realizados por un proceso (o hilo) en la memoria serán visibles para el otro. Si este flag **no** se establece, el hijo obtiene **una copia privada de la memoria**, como en una llamada `fork()`.
- Según la **manpage** de `clone` "*`CLONE_THREAD`: If CLONE_THREAD is set, the child is placed in the same thread group as the calling process.*". Esto hace que el nuevo proceso se comporte como un **hilo** del proceso padre. Comparten no solo la memoria (lo que requiere `CLONE_VM`) sino también: mismo PID de grupo de hilos (thread group ID), descriptores de archivos, handlers de señales, etc. Además, la señal `exit_signal` debe ser 0 (sin señales al finalizar el hijo), como es típico en hilos. Se deja como nota: "*`CLONE_THREAD` implica que `CLONE_VM` esté activado, ya que los hilos deben compartir memoria.*"

4. `printf()` eventualmente invoca la syscall write (con primer argumento 1, indicando que el file descriptor donde se escribirá el texto es STDOUT). Vea la salida de strace y verifique qué invocaciones a write(1, ...) ocurren en cada caso.

- En el programa `01` se hace esta invocación: `write(1, "Parent process: PID = 5417, THRE"..., 45) = 45`.

- En el programa `02` se hace esta invocación: `write(1, "Parent process: PID = 5474, THRE"..., 88) = 88`
- En el programa `03` se hace esta invocación: `write(1, "Parent process: PID = 5518, THRE"..., 138) = 138`

5. Pruebe invocar de nuevo strace con la opción -f y vea qué sucede respecto a las invocaciones a write(1, ...). Investigue qué es esa opción en la manpage de strace. ¿Por qué en el caso del ULT se puede ver la invocación a write(1, ...) por parte del thread hijo aún sin usar -f?

- Según la **manpage** de `strace` "*`-f`: Trace child processes as they are created by currently traced processes as a result of the `fork(2)`, `vfork(2)` or `clone(2)` system calls.*". Es decir, **también traza los procesos hijos** que se crean durante la ejecución.
- En el caso del programa `01` ahora podemos ver 2 writes, el proceso hijo con `pid 5818` hace `write(1, "Parent process: PID = 5817, THRE"..., 89)`, y luego de un wait, el proceso padre con `pid 5817` hace `write(1, "Parent process: PID = 5817, THRE"..., 45)`.
- En el caso del programa `02` se crea un hilo con el uso de `clone3` pero solo se vio este write hecho por el proceso padre: `[pid 5885] write(1, "Parent process: PID = 5885, THRE"..., 88) = 88`. Se debe a que `-f` solo tracea a los procesos hijos, acá creamos un KLT.
- En el caso del programa `03` se puede ver la invocación a write (del padre, no del hijo) sin usar `-f` ya que los ULT **no usan** `clone` **ni** `fork`, sino que son gestionados completamente en espacio de usuario, es decir, el hilo hijo *no es un nuevo PID ni TID*, simplemente es código ejecutado en el mismo contexto de proceso del padre, entonces, Como no hay `clone`, no se crea un hilo "real" desde el punto de vista del kernel. Por eso `strace` **ve todas las llamadas sin necesidad del** `-f`, ya que no hay procesos/hilos nuevos que tracear.

Salida del programa 01 sin usar -f:

```
so@so:~/codigo-para-practicas/practica3/01-strace$ strace ./01-subprocess > /dev/null
execve("./01-subprocess", ["./01-subprocess"], 0x7ffee2c88d40 /* 28 vars */) = 0
brk(NULL)                               = 0x55ec9582c000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7c6cdee000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No existe el fichero o el directorio)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=23038, ...}, AT_EMPTY_PATH)
```

```
= 0
mmap(NULL, 23038, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f7c6cde8000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0\0"...,
832) = 832
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1922136, ...},
AT_EMPTY_PATH) = 0
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
mmap(NULL, 1970000, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f7c6cc07000
mmap(0x7f7c6cc2d000, 1396736, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7f7c6cc2d000
mmap(0x7f7c6cd82000, 339968, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x17b000) = 0x7f7c6cd82000
mmap(0x7f7c6cdd5000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ce000) = 0x7f7c6cdd5000
mmap(0x7f7c6cddb000, 53072, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f7c6cddb000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f7c6cc04000
arch_prctl(ARCH_SET_FS, 0x7f7c6cc04740) = 0
set_tid_address(0x7f7c6cc04a10)         = 5417
set_robust_list(0x7f7c6cc04a20, 24)     = 0
rseq(0x7f7c6cc05060, 0x20, 0, 0x53053053) = 0
mprotect(0x7f7c6cdd5000, 16384, PROT_READ) = 0
mprotect(0x55ec7396c000, 4096, PROT_READ) = 0
mprotect(0x7f7c6ce26000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f7c6cde8000, 23038)           = 0
gettid()                                = 5417
getpid()                                = 5417
newfstatat(1, "", {st_mode=S_IFCHR|0666, st_rdev=makedev(0x1, 0x3), ...},
AT_EMPTY_PATH) = 0
ioctl(1, TCGETS, 0x7ffd73270d00)        = -1 ENOTTY (Función ioctl no
apropiada para el dispositivo)
getrandom("\xb9\x23\xc4\x6d\xfd\xa8\x04\xdb", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x55ec9582c000
brk(0x55ec9584d000)                     = 0x55ec9584d000
```

```
clone(child_stack=NULL,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x7f7c6cc04a10) = 5418
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=5418, si_uid=1000,
si_status=0, si_utime=0, si_stime=0} ---
wait4(-1, NULL, 0, NULL)                = 5418
write(1, "Parent process: PID = 5417, THRE"..., 45) = 45
exit_group(0)                           = ?
+++ exited with 0 +++
```

Salida del programa 01 con -f:

```
so@so:~/codigo-para-practicas/practica3/01-strace$ strace -f ./01-subprocess
> /dev/null
execve("./01-subprocess", ["./01-subprocess"], 0x7ffe75ac4f88 /* 28 vars */)
= 0
brk(NULL)                               = 0x560ea8cc9000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7ff968103000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No existe el fichero o
el directorio)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=23038, ...}, AT_EMPTY_PATH)
= 0
mmap(NULL, 23038, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff9680fd000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0\0"...,
832) = 832
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1922136, ...},
AT_EMPTY_PATH) = 0
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
mmap(NULL, 1970000, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7ff967f1c000
mmap(0x7ff967f42000, 1396736, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7ff967f42000
mmap(0x7ff968097000, 339968, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x17b000) = 0x7ff968097000
mmap(0x7ff9680ea000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ce000) = 0x7ff9680ea000
```

```
mmap(0x7ff9680f0000, 53072, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7ff9680f0000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7ff967f19000
arch_prctl(ARCH_SET_FS, 0x7ff967f19740) = 0
set_tid_address(0x7ff967f19a10)         = 5817
set_robust_list(0x7ff967f19a20, 24)     = 0
rseq(0x7ff967f1a060, 0x20, 0, 0x53053053) = 0
mprotect(0x7ff9680ea000, 16384, PROT_READ) = 0
mprotect(0x560e8b9bc000, 4096, PROT_READ) = 0
mprotect(0x7ff96813b000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7ff9680fd000, 23038)           = 0
gettid()                                = 5817
getpid()                                = 5817
newfstatat(1, "", {st_mode=S_IFCHR|0666, st_rdev=makedev(0x1, 0x3), ...},
AT_EMPTY_PATH) = 0
ioctl(1, TCGETS, 0x7ffd4b6dbc80)        = -1 ENOTTY (Función ioctl no
apropiada para el dispositivo)
getrandom("\x8e\x90\x70\x06\x9e\xca\xd3\x5b", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x560ea8cc9000
brk(0x560ea8cea000)                     = 0x560ea8cea000
clone(child_stack=NULL,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLDstrace: Process 5818
attached
, child_tidptr=0x7ff967f19a10) = 5818
[pid  5817] wait4(-1,  <unfinished ...>
[pid  5818] set_robust_list(0x7ff967f19a20, 24) = 0
[pid  5818] gettid()                    = 5818
[pid  5818] getpid()                    = 5818
[pid  5818] write(1, "Parent process: PID = 5817, THRE"..., 89) = 89
[pid  5818] exit_group(0)               = ?
[pid  5818] +++ exited with 0 +++
<... wait4 resumed>NULL, 0, NULL)       = 5818
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=5818, si_uid=1000,
si_status=0, si_utime=0, si_stime=0} ---
write(1, "Parent process: PID = 5817, THRE"..., 45) = 45
exit_group(0)                           = ?
+++ exited with 0 +++
```

Salida del programa 02 sin -f:

```
so@so:~/codigo-para-practicas/practica3/01-strace$ strace ./02-kl-thread >
/dev/null
execve("./02-kl-thread", ["./02-kl-thread"], 0x7fff4eb57080 /* 28 vars */) =
0
brk(NULL)                               = 0x55c6071ab000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f6042dc1000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No existe el fichero o
el directorio)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=23038, ...}, AT_EMPTY_PATH)
= 0
mmap(NULL, 23038, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f6042dbb000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0\0"...,
832) = 832
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1922136, ...},
AT_EMPTY_PATH) = 0
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
mmap(NULL, 1970000, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f6042bda000
mmap(0x7f6042c00000, 1396736, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7f6042c00000
mmap(0x7f6042d55000, 339968, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x17b000) = 0x7f6042d55000
mmap(0x7f6042da8000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ce000) = 0x7f6042da8000
mmap(0x7f6042dae000, 53072, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f6042dae000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f6042bd7000
arch_prctl(ARCH_SET_FS, 0x7f6042bd7740) = 0
set_tid_address(0x7f6042bd7a10)         = 5474
set_robust_list(0x7f6042bd7a20, 24)     = 0
rseq(0x7f6042bd8060, 0x20, 0, 0x53053053) = 0
mprotect(0x7f6042da8000, 16384, PROT_READ) = 0
mprotect(0x55c5cc883000, 4096, PROT_READ) = 0
mprotect(0x7f6042df9000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
```

```
rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f6042dbb000, 23038)           = 0
gettid()                                = 5474
getpid()                                = 5474
newfstatat(1, "", {st_mode=S_IFCHR|0666, st_rdev=makedev(0x1, 0x3), ...},
AT_EMPTY_PATH) = 0
ioctl(1, TCGETS, 0x7ffe221daf70)        = -1 ENOTTY (Función ioctl no
apropiada para el dispositivo)
getrandom("\x76\xcd\x71\xd7\xb6\x7b\x60\xcf", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x55c6071ab000
brk(0x55c6071cc000)                     = 0x55c6071cc000
rt_sigaction(SIGRT_1, {sa_handler=0x7f6042c60720, sa_mask=[],
sa_flags=SA_RESTORER|SA_ONSTACK|SA_RESTART|SA_SIGINFO,
sa_restorer=0x7f6042c16050}, NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) =
0x7f60423d6000
mprotect(0x7f60423d7000, 8388608, PROT_READ|PROT_WRITE) = 0
rt_sigprocmask(SIG_BLOCK, ~[], [], 8)   = 0
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE
_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
child_tid=0x7f6042bd6990, parent_tid=0x7f6042bd6990, exit_signal=0,
stack=0x7f60423d6000, stack_size=0x7fff80, tls=0x7f6042bd66c0} ⇒
{parent_tid=[0]}, 88) = 5476
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
write(1, "Parent process: PID = 5474, THRE"..., 88) = 88
exit_group(0)                           = ?
+++ exited with 0 +++
```

Salida del programa 02 con -f:

```
so@so:~/codigo-para-practicas/practica3/01-strace$ strace -f ./02-kl-thread
> /dev/null
execve("./02-kl-thread", ["./02-kl-thread"], 0x7ffd2c0e9f98 /* 28 vars */) =
0
brk(NULL)                               = 0x560c69739000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7fbba1ff1000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No existe el fichero o
el directorio)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=23038, ...}, AT_EMPTY_PATH)
= 0
mmap(NULL, 23038, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fbba1feb000
close(3)                                = 0
```

```
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0\0"..., 
832) = 832
pread64(3, 
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1922136, ...}, 
AT_EMPTY_PATH) = 0
pread64(3, 
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
mmap(NULL, 1970000, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 
0x7fbba1e0a000
mmap(0x7fbba1e30000, 1396736, PROT_READ|PROT_EXEC, 
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7fbba1e30000
mmap(0x7fbba1f85000, 339968, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x17b000) = 0x7fbba1f85000
mmap(0x7fbba1fd8000, 24576, PROT_READ|PROT_WRITE, 
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ce000) = 0x7fbba1fd8000
mmap(0x7fbba1fde000, 53072, PROT_READ|PROT_WRITE, 
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fbba1fde000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 
0x7fbba1e07000
arch_prctl(ARCH_SET_FS, 0x7fbba1e07740) = 0
set_tid_address(0x7fbba1e07a10)         = 5885
set_robust_list(0x7fbba1e07a20, 24)     = 0
rseq(0x7fbba1e08060, 0x20, 0, 0x53053053) = 0
mprotect(0x7fbba1fd8000, 16384, PROT_READ) = 0
mprotect(0x560c6601d000, 4096, PROT_READ) = 0
mprotect(0x7fbba2029000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, 
rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7fbba1feb000, 23038)           = 0
gettid()                                = 5885
getpid()                                = 5885
newfstatat(1, "", {st_mode=S_IFCHR|0666, st_rdev=makedev(0x1, 0x3), ...}, 
AT_EMPTY_PATH) = 0
ioctl(1, TCGETS, 0x7ffd56fb50a0)        = -1 ENOTTY (Función ioctl no 
apropiada para el dispositivo)
getrandom("\x42\x45\x1c\x9a\x55\xeb\x5e\x6f", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x560c69739000
brk(0x560c6975a000)                     = 0x560c6975a000
rt_sigaction(SIGRT_1, {sa_handler=0x7fbba1e90720, sa_mask=[], 
sa_flags=SA_RESTORER|SA_ONSTACK|SA_RESTART|SA_SIGINFO, 
sa_restorer=0x7fbba1e46050}, NULL, 8) = 0
```

```
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) =
0x7fbba1606000
mprotect(0x7fbba1607000, 8388608, PROT_READ|PROT_WRITE) = 0
rt_sigprocmask(SIG_BLOCK, ~[], [], 8)    = 0
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE
_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
child_tid=0x7fbba1e06990, parent_tid=0x7fbba1e06990, exit_signal=0,
stack=0x7fbba1606000, stack_size=0x7fff80, tls=0x7fbba1e066c0}strace:
Process 5886 attached
 ⇒ {parent_tid=[5886]}, 88) = 5886
[pid  5885] rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
[pid  5885] futex(0x7fbba1e06990, FUTEX_WAIT_BITSET|FUTEX_CLOCK_REALTIME,
5886, NULL, FUTEX_BITSET_MATCH_ANY <unfinished ...>
[pid  5886] rseq(0x7fbba1e06fe0, 0x20, 0, 0x53053053) = 0
[pid  5886] set_robust_list(0x7fbba1e069a0, 24) = 0
[pid  5886] rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
[pid  5886] gettid()                     = 5886
[pid  5886] getpid()                     = 5885
[pid  5886] rt_sigprocmask(SIG_BLOCK, ~[RT_1], NULL, 8) = 0
[pid  5886] madvise(0x7fbba1606000, 8368128, MADV_DONTNEED) = 0
[pid  5886] exit(0)                      = ?
[pid  5885] <... futex resumed>)         = 0
[pid  5885] write(1, "Parent process: PID = 5885, THRE"..., 88) = 88
[pid  5885] exit_group(0)                = ?
[pid  5886] +++ exited with 0 +++
+++ exited with 0 +++
```

Salida del programa 03 sin -f:

```
so@so:~/codigo-para-practicas/practica3/01-strace$ strace ./03-ul-thread >
/dev/null
execve("./03-ul-thread", ["./03-ul-thread"], 0x7ffda4794340 /* 28 vars */) =
0
brk(NULL)                                = 0x55606f150000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f06c6fba000
access("/etc/ld.so.preload", R_OK)       = -1 ENOENT (No existe el fichero o
el directorio)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=23038, ...}, AT_EMPTY_PATH)
= 0
mmap(NULL, 23038, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f06c6fb4000
close(3)                                 = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpth.so.20", O_RDONLY|O_CLOEXEC) =
```

```
3
read(3,
"\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\200I\0\0\0\0\0\0"..., 832) =
832
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=81272, ...}, AT_EMPTY_PATH)
= 0
mmap(NULL, 93552, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f06c6f9d000
mmap(0x7f06c6fa1000, 49152, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x4000) = 0x7f06c6fa1000
mmap(0x7f06c6fad000, 12288, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x10000) = 0x7f06c6fad000
mmap(0x7f06c6fb0000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x12000) = 0x7f06c6fb0000
mmap(0x7f06c6fb2000, 7536, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f06c6fb2000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0\0"...,
832) = 832
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1922136, ...},
AT_EMPTY_PATH) = 0
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
mmap(NULL, 1970000, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f06c6dbc000
mmap(0x7f06c6de2000, 1396736, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7f06c6de2000
mmap(0x7f06c6f37000, 339968, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x17b000) = 0x7f06c6f37000
mmap(0x7f06c6f8a000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ce000) = 0x7f06c6f8a000
mmap(0x7f06c6f90000, 53072, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f06c6f90000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f06c6db9000
arch_prctl(ARCH_SET_FS, 0x7f06c6db9740) = 0
set_tid_address(0x7f06c6db9a10)         = 5518
set_robust_list(0x7f06c6db9a20, 24)     = 0
rseq(0x7f06c6dba060, 0x20, 0, 0x53053053) = 0
mprotect(0x7f06c6f8a000, 16384, PROT_READ) = 0
```

```
mprotect(0x7f06c6fb0000, 4096, PROT_READ) = 0
mprotect(0x55605cbc1000, 4096, PROT_READ) = 0
mprotect(0x7f06c6ff2000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f06c6fb4000, 23038)           = 0
pipe2([3, 4], 0)                        = 0
fcntl(3, F_GETFL)                       = 0 (flags O_RDONLY)
fcntl(3, F_SETFL, O_RDONLY|O_NONBLOCK)  = 0
fcntl(4, F_GETFL)                       = 0x1 (flags O_WRONLY)
fcntl(4, F_SETFL, O_WRONLY|O_NONBLOCK)  = 0
getrandom("\xcf\x1e\xf7\x21\x24\x68\x2b\x21", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x55606f150000
brk(0x55606f171000)                     = 0x55606f171000
rt_sigprocmask(SIG_BLOCK, NULL, [], 8)  = 0
rt_sigprocmask(SIG_SETMASK, [], [], 8)  = 0
rt_sigprocmask(SIG_SETMASK, ~[RTMIN RT_1], NULL, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], ~[KILL STOP RTMIN RT_1], 8) = 0
gettid()                                = 5518
getpid()                                = 5518
newfstatat(1, "", {st_mode=S_IFCHR|0666, st_rdev=makedev(0x1, 0x3), ...},
AT_EMPTY_PATH) = 0
ioctl(1, TCGETS, 0x7ffffca2db10)        = -1 ENOTTY (Función ioctl no
apropiada para el dispositivo)
brk(0x55606f193000)                     = 0x55606f193000
rt_sigprocmask(SIG_BLOCK, NULL, [], 8)  = 0
rt_sigprocmask(SIG_SETMASK, ~[KILL STOP RTMIN RT_1], [], 8) = 0
rt_sigpending([], 8)                    = 0
read(3, 0x55606f1605d0, 128)            = -1 EAGAIN (Recurso no disponible
temporalmente)
rt_sigprocmask(SIG_SETMASK, [], ~[KILL STOP RTMIN RT_1], 8) = 0
pselect6(4, [3], [], [], {tv_sec=0, tv_nsec=0}, NULL) = 0 (Timeout)
rt_sigprocmask(SIG_SETMASK, ~[KILL STOP RTMIN RT_1], NULL, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], ~[KILL STOP RTMIN RT_1], 8) = 0
gettid()                                = 5518
getpid()                                = 5518
rt_sigprocmask(SIG_SETMASK, ~[KILL STOP RTMIN RT_1], [], 8) = 0
rt_sigpending([], 8)                    = 0
read(3, 0x55606f1605d0, 128)            = -1 EAGAIN (Recurso no disponible
temporalmente)
rt_sigprocmask(SIG_SETMASK, [], ~[KILL STOP RTMIN RT_1], 8) = 0
pselect6(4, [3], [], [], {tv_sec=0, tv_nsec=0}, NULL) = 0 (Timeout)
rt_sigprocmask(SIG_SETMASK, ~[KILL STOP RTMIN RT_1], NULL, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], ~[KILL STOP RTMIN RT_1], 8) = 0
close(3)                                = 0
close(4)                                = 0
```

```
write(1, "Parent process: PID = 5518, THRE"..., 138) = 138
exit_group(0)                           = ?
+++ exited with 0 +++
```

Salida del programa 03 con -f:

```
so@so:~/codigo-para-practicas/practica3/01-strace$ strace -f ./03-ul-thread
> /dev/null
execve("./03-ul-thread", ["./03-ul-thread"], 0x7ffc77f33ce8 /* 28 vars */) =
0
brk(NULL)                               = 0x55c2eef15000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7fe2a3a4f000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No existe el fichero o
el directorio)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=23038, ...}, AT_EMPTY_PATH)
= 0
mmap(NULL, 23038, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fe2a3a49000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpth.so.20", O_RDONLY|O_CLOEXEC) =
3
read(3,
"\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\200I\0\0\0\0\0\0"..., 832) =
832
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=81272, ...}, AT_EMPTY_PATH)
= 0
mmap(NULL, 93552, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7fe2a3a32000
mmap(0x7fe2a3a36000, 49152, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x4000) = 0x7fe2a3a36000
mmap(0x7fe2a3a42000, 12288, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x10000) = 0x7fe2a3a42000
mmap(0x7fe2a3a45000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x12000) = 0x7fe2a3a45000
mmap(0x7fe2a3a47000, 7536, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fe2a3a47000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0\0"...,
832) = 832
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1922136, ...},
```

```
AT_EMPTY_PATH) = 0
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
mmap(NULL, 1970000, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7fe2a3851000
mmap(0x7fe2a3877000, 1396736, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7fe2a3877000
mmap(0x7fe2a39cc000, 339968, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x17b000) = 0x7fe2a39cc000
mmap(0x7fe2a3a1f000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ce000) = 0x7fe2a3a1f000
mmap(0x7fe2a3a25000, 53072, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fe2a3a25000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7fe2a384e000
arch_prctl(ARCH_SET_FS, 0x7fe2a384e740) = 0
set_tid_address(0x7fe2a384ea10)         = 5917
set_robust_list(0x7fe2a384ea20, 24)     = 0
rseq(0x7fe2a384f060, 0x20, 0, 0x53053053) = 0
mprotect(0x7fe2a3a1f000, 16384, PROT_READ) = 0
mprotect(0x7fe2a3a45000, 4096, PROT_READ) = 0
mprotect(0x55c2d7d7a000, 4096, PROT_READ) = 0
mprotect(0x7fe2a3a87000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7fe2a3a49000, 23038)           = 0
pipe2([3, 4], 0)                        = 0
fcntl(3, F_GETFL)                       = 0 (flags O_RDONLY)
fcntl(3, F_SETFL, O_RDONLY|O_NONBLOCK)  = 0
fcntl(4, F_GETFL)                       = 0x1 (flags O_WRONLY)
fcntl(4, F_SETFL, O_WRONLY|O_NONBLOCK)  = 0
getrandom("\x3d\x14\xeb\x01\xc7\x9d\x5b\xeb", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x55c2eef15000
brk(0x55c2eef36000)                     = 0x55c2eef36000
rt_sigprocmask(SIG_BLOCK, NULL, [], 8)  = 0
rt_sigprocmask(SIG_SETMASK, [], [], 8)  = 0
rt_sigprocmask(SIG_SETMASK, ~[RTMIN RT_1], NULL, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], ~[KILL STOP RTMIN RT_1], 8) = 0
gettid()                                = 5917
getpid()                                = 5917
newfstatat(1, "", {st_mode=S_IFCHR|0666, st_rdev=makedev(0x1, 0x3), ...},
AT_EMPTY_PATH) = 0
ioctl(1, TCGETS, 0x7ffd74fe6360)        = -1 ENOTTY (Función ioctl no
apropiada para el dispositivo)
```

```
brk(0x55c2eef58000)                     = 0x55c2eef58000
rt_sigprocmask(SIG_BLOCK, NULL, [], 8)  = 0
rt_sigprocmask(SIG_SETMASK, ~[KILL STOP RTMIN RT_1], [], 8) = 0
rt_sigpending([], 8)                    = 0
read(3, 0x55c2eef255d0, 128)            = -1 EAGAIN (Recurso no disponible
temporalmente)
rt_sigprocmask(SIG_SETMASK, [], ~[KILL STOP RTMIN RT_1], 8) = 0
pselect6(4, [3], [], [], {tv_sec=0, tv_nsec=0}, NULL) = 0 (Timeout)
rt_sigprocmask(SIG_SETMASK, ~[KILL STOP RTMIN RT_1], NULL, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], ~[KILL STOP RTMIN RT_1], 8) = 0
gettid()                                = 5917
getpid()                                = 5917
rt_sigprocmask(SIG_SETMASK, ~[KILL STOP RTMIN RT_1], [], 8) = 0
rt_sigpending([], 8)                    = 0
read(3, 0x55c2eef255d0, 128)            = -1 EAGAIN (Recurso no disponible
temporalmente)
rt_sigprocmask(SIG_SETMASK, [], ~[KILL STOP RTMIN RT_1], 8) = 0
pselect6(4, [3], [], [], {tv_sec=0, tv_nsec=0}, NULL) = 0 (Timeout)
rt_sigprocmask(SIG_SETMASK, ~[KILL STOP RTMIN RT_1], NULL, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], ~[KILL STOP RTMIN RT_1], 8) = 0
close(3)                                = 0
close(4)                                = 0
write(1, "Parent process: PID = 5917, THRE"..., 138) = 138
exit_group(0)                           = ?
+++ exited with 0 +++
```

4. Resuelva y responda utilizando el contenido del directorio `practica3/02-memory`:
    1. Compile los 3 programas C usando el comando make.
    2. Ejecute los 3 programas.
    3. Observe qué pasa con la modificación a la variable number en cada caso. ¿Por qué suceden cosas distintas en cada caso?

- En el programa `01` se crea un nuevo proceso hijo con `fork()`, que tiene su propia copia del espacio de memoria. Cuando el hijo cambia `number = 84`, lo hace en su propia copia, por lo tanto no afecta al padre.
- En el programa `02` se crea un hilo del kernel con `pthread_create()`, que comparte el espacio de memoria con el hilo principal. Ambos hilos acceden a la misma variable `number`. Cuando el hilo hijo la modifica, el cambio se refleja en el hilo padre.
- En el programa `03` se usan hilos de usuario (user-level threads). El código ejecuta las "tareas concurrentes" dentro del mismo proceso y memoria compartida. Por eso, la variable `number` es la misma para ambos contextos (padre e hijo).

Salida de los programas:

```
so@so:~/codigo-para-practicas/practica3/02-memoria$ ./01-subprocess
Parent process: PID = 6135, THREAD_ID = 6135
Parent process: number = 42
Child process: PID = 6136, THREAD_ID = 6136
Child process: number = 84
Parent process: number = 42
so@so:~/codigo-para-practicas/practica3/02-memoria$ ./02-kl-thread
Parent process: PID = 6175, THREAD_ID = 6175
Parent process: number = 42
Child thread: PID = 6175, THREAD_ID = 6176
Child process: number = 84
Parent process: number = 84
so@so:~/codigo-para-practicas/practica3/02-memoria$ ./03-ul-thread
Parent process: PID = 6191, THREAD_ID = 6191, PTH_ID = 93950489557232
Parent process: number = 42
Child thread: PID = 6191, THREAD_ID = 6191, PTH_ID = 93950489559776
Child thread: number = 84
Parent process: number = 84
```

5. El directorio `practica3/03-cpu-bound` contiene programas en C y en Python que ejecutan una tarea CPU-Bound (calcular el enésimo número primo).
    1. Ejecute htop en una terminal separada para monitorear el uso de CPU en los siguientes incisos.
    2. Ejecute los distintos ejemplos con make (usar `make help` para ver cómo) y observe cómo aparecen los resultados, cuánto tarda cada thread y cuanto tarda el programa completo en finalizar.
    3. ¿Cuántos threads se crean en cada caso?

    - Esto se ve en los resultados de ir ejecutando los programas más abajo.

    4. ¿Cómo se comparan los tiempos de ejecución de los programas escritos en C (ult y klt)?

    - En C cuando usamos KLTs el tiempo de ejecución es menor al de los ULTs, esto se debe al hecho de que los KLTs pueden aprovechar el paralelismo de las CPUs mientras que los ULTs no.

    5. ¿Cómo se comparan los tiempos de ejecución de los programas escritos en Python (ult.py y klt.py)?

    - Comparación:
        - `klt.py` con GIL:
            - Acá usamos *KLTs*, es el que más tarda de los 3, esto se debe a que el GIL impide que múltiples hilos de Python ejecuten bytecode al mismo tiempo en CPU-bound tasks.
        - `ult.py` con GIL:

        - Acá usamos *ULTs* (en Python implementados con `greenlet`), es el segundo que más tarda de los 3, esto se debe a que los *ULTs* cooperan y cambian de contexto voluntariamente. Aunque siguen siendo secuenciales (por el GIL), su *scheduler cooperativo* reduce el overhead de concurrencia en comparación con hilos reales.
    - `klt.py` sin GIL:
        - Acá usamos *KLTs*, es el que menos tarda de los 3, esto se debe a que al eliminar el GIL, los hilos *pueden ejecutar código Python en paralelo real*, lo cual *aprovecha múltiples núcleos* del procesador.

6. Modifique la cantidad de threads en los scripts Python con la variable NUM_THREADS para que en ambos casos se creen solamente 2 threads, vuelva a ejecutar y comparar los tiempos. ¿Nota algún cambio? ¿A qué se debe?

- Si, hay cambios, los **tiempos de ejecución de los programas se reducen bastente**, esto se debe a que:
    - En el caso de `klt.py` con GIL ahora al tener 2 hilos, estos se van a turnar por el GIL, lo que hace que el tiempo total es casi igual al trabajo de uno solo (420s).
    - En el caso de `ult.py` con GIL, los **ULTs** corren uno tras otro, y terminan un poco antes que los KLT porque pueden hacer un cambio de contexto más eficiente (menos overhead), pero *no hay paralelismo real*.
    - En el caso de `klt.py` sin GIL, los 2 threads trabajan en paralelo real, aprovechando dos cores, esto hace que el tiempo se reduzca aproximadamente a la *mitad*, porque *las tareas sí se ejecutan simultáneamente*.

7. ¿Qué conclusión puede sacar respecto a los ULT en tareas CPU-Bound?

- En conclusión, los **ULT** no son adecuados para tareas CPU-Bound en Python, ya que no aprovechan el paralelismo real. Como no hay múltiples threads del sistema operativo, **todo el trabajo se hace en un solo núcleo**.

Ejecución de programa C KLT:

```
so@so:~/codigo-para-practicas/practica3/03-cpu-bound$ make run_klt
./klt
Starting the program.
[Thread 140602803467968] Doing some work...
[Thread 140602795075264] Doing some work...
[Thread 140602786682560] Doing some work...
[Thread 140602778289856] Doing some work...
[Thread 140602769897152] Doing some work...
2500000th prime is 41161739
[Thread 140602778289856] Done with work in 219.315730 seconds.
2500000th prime is 41161739
[Thread 140602803467968] Done with work in 220.596754 seconds.
```

```
2500000th prime is 41161739
[Thread 140602786682560] Done with work in 226.152454 seconds.
2500000th prime is 41161739
[Thread 140602795075264] Done with work in 229.008468 seconds.
2500000th prime is 41161739
[Thread 140602769897152] Done with work in 232.218011 seconds.
All threads are done in 232.232920 seconds
```

Ejecución de programa C ULT:

```
so@so:~/codigo-para-practicas/practica3/03-cpu-bound$ make run_ult
cc -Wall -Werror    ult.c common.o  -lpth -o ult
./ult
Starting the program.
[Thread 93956085564128] Doing some work...
2500000th prime is 41161739
[Thread 93956085564128] Done with work in 139.043977 seconds.
[Thread 93956085631184] Doing some work...
2500000th prime is 41161739
[Thread 93956085631184] Done with work in 142.285714 seconds.
[Thread 93956085698240] Doing some work...
2500000th prime is 41161739
[Thread 93956085698240] Done with work in 141.414405 seconds.
[Thread 93956085765296] Doing some work...
2500000th prime is 41161739
[Thread 93956085765296] Done with work in 138.973491 seconds.
[Thread 93956085832352] Doing some work...
2500000th prime is 41161739
[Thread 93956085832352] Done with work in 142.217776 seconds.
All threads are done in 704.000000 seconds
```

Ejecución de programa PYTHON KLT:

```
so@so:~/codigo-para-practicas/practica3/03-cpu-bound$ make run_klt_py
python3 -m venv /home/so/codigo-para-practicas/practica3//.venv
/home/so/codigo-para-practicas/practica3//.venv/bin/pip install --upgrade
pip
Requirement already satisfied: pip in /home/so/codigo-para-
practicas/practica3/.venv/lib/python3.11/site-packages (23.0.1)
Collecting pip
  Using cached pip-25.0.1-py3-none-any.whl (1.8 MB)
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 23.0.1
    Uninstalling pip-23.0.1:
```

```
      Successfully uninstalled pip-23.0.1
Successfully installed pip-25.0.1
/home/so/codigo-para-practicas/practica3//.venv/bin/pip install -r
/home/so/codigo-para-practicas/practica3//requirements.txt
Collecting greenlet (from -r /home/so/codigo-para-
practicas/practica3//requirements.txt (line 1))
  Using cached greenlet-3.2.0-cp311-cp311-
manylinux_2_24_x86_64.manylinux_2_28_x86_64.whl.metadata (4.1 kB)
Collecting gevent (from -r /home/so/codigo-para-
practicas/practica3//requirements.txt (line 2))
  Using cached gevent-25.4.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (13 kB)
Collecting zope.event (from gevent→-r /home/so/codigo-para-
practicas/practica3//requirements.txt (line 2))
  Using cached zope.event-5.0-py3-none-any.whl.metadata (4.4 kB)
Collecting zope.interface (from gevent→-r /home/so/codigo-para-
practicas/practica3//requirements.txt (line 2))
  Using cached zope.interface-7.2-cp311-cp311-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x
86_64.whl.metadata (44 kB)
Requirement already satisfied: setuptools in /home/so/codigo-para-
practicas/practica3/.venv/lib/python3.11/site-packages (from zope.event
>gevent→-r /home/so/codigo-para-practicas/practica3//requirements.txt (line
2)) (66.1.1)
Using cached greenlet-3.2.0-cp311-cp311-
manylinux_2_24_x86_64.manylinux_2_28_x86_64.whl (583 kB)
Using cached gevent-25.4.1-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.1 MB)
Using cached zope.event-5.0-py3-none-any.whl (6.8 kB)
Using cached zope.interface-7.2-cp311-cp311-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x
86_64.whl (259 kB)
Installing collected packages: zope.interface, zope.event, greenlet, gevent
Successfully installed gevent-25.4.1 greenlet-3.2.0 zope.event-5.0
zope.interface-7.2
/home/so/codigo-para-practicas/practica3//.venv/bin/python3 klt.py
Starting the program.
[thread_id=140336272365248] Doing some work...
[thread_id=140336263972544] Doing some work...
[thread_id=140336043128512] Doing some work...
[thread_id=140336255579840] Doing some work...
[thread_id=140336034735808] Doing some work...
500000th prime is 7368787
[thread_id=140336034735808] Done with work in 1308.1192812919617 seconds.
500000th prime is 7368787
[thread_id=140336272365248] Done with work in 1330.0280947685242 seconds.
```

```
500000th prime is 7368787
[thread_id=140336255579840] Done with work in 1330.1990897655487 seconds.
500000th prime is 7368787
[thread_id=140336263972544] Done with work in 1332.9473986625671 seconds.
500000th prime is 7368787
[thread_id=140336043128512] Done with work in 1333.5137746334076 seconds.
All threads are done in 1333.5368020534515 seconds
```

Ejecución de programa PYTHON ULT:

```
so@so:~/codigo-para-practicas/practica3/03-cpu-bound$ make run_ult_py
/home/so/codigo-para-practicas/practica3//.venv/bin/python3 ult.py
Starting the program.
[greenlet_id=140649693252320] Doing some work...
500000th prime is 7368787
[greenlet_id=140649693252320] Done with work in 244.9661693572998 seconds.
[greenlet_id=140649688350400] Doing some work...
500000th prime is 7368787
[greenlet_id=140649688350400] Done with work in 265.60367012023926 seconds.
[greenlet_id=140649688184800] Doing some work...
500000th prime is 7368787
[greenlet_id=140649688184800] Done with work in 221.21369433403015 seconds.
[greenlet_id=140649686410368] Doing some work...
500000th prime is 7368787
[greenlet_id=140649686410368] Done with work in 203.3567361831665 seconds.
[greenlet_id=140649686410528] Doing some work...
500000th prime is 7368787
[greenlet_id=140649686410528] Done with work in 204.1185998916626 seconds.
All greenlets are done in 1139.3216335773468 seconds
```

Ejecución de programa PYTHON KLT sin GIL:

Error >

cuando quise tirar el comando para correr el código me tiro este error:

```
so@so:~/codigo-para-practicas/practica3/03-cpu-bound$ make
run_klt_py_nogil
podman run -it --rm -v .:/mnt docker.io/felopez/python-nogil:latest
python3 -X gil=0 /mnt/klt.py
Trying to pull docker.io/felopez/python-nogil:latest...
Getting image source signatures
Copying blob 10af00adc39b done
Copying blob 3ca3e5140333 done
```

```
Copying config a41db0f0b9 done
Writing manifest to image destination
Storing signatures
Error: /usr/bin/slirp4netns failed: "open(\"/dev/net/tun\"): No such
file or directory\nWARNING: Support for seccomp is
experimental\nWARNING: Support for IPv6 is experimental\nchild
failed(1)\nWARNING: Support for seccomp is experimental\nWARNING:
Support for IPv6 is experimental\n"
make: *** [../common.mk:38: run_klt_py_nogil] Error 127
```

La solución fue correr el código con:

```
su -c 'podman run -it --rm --network=none -v .:/mnt
docker.io/felopez/python-nogil:latest python3 -X gil=0 /mnt/klt.py'
```

en vez de con `make run_run_klt_py_nogil`

```
so@so:~/codigo-para-practicas/practica3/03-cpu-bound$ su -c 'podman run -it
--rm --network=none -v .:/mnt docker.io/felopez/python-nogil:latest python3
-X gil=0 /mnt/klt.py'
Contraseña:
Trying to pull docker.io/felopez/python-nogil:latest...
Getting image source signatures
Copying blob 3ca3e5140333 done
Copying blob 10af00adc39b done
Copying config a41db0f0b9 done
Writing manifest to image destination
Storing signatures
Starting the program.
[thread_id=140691490371264] Doing some work...
[thread_id=140691481962176] Doing some work...
[thread_id=140691473553088] Doing some work...
[thread_id=140691465144000] Doing some work...
[thread_id=140691250345664] Doing some work...
500000th prime is 7368787
[thread_id=140691465144000] Done with work in 409.9800899028778 seconds.
500000th prime is 7368787
[thread_id=140691473553088] Done with work in 412.3069341182709 seconds.
500000th prime is 7368787
[thread_id=140691490371264] Done with work in 415.7076003551483 seconds.
500000th prime is 7368787
[thread_id=140691250345664] Done with work in 415.6333644390106 seconds.
500000th prime is 7368787
```

```
[thread_id=140691481962176] Done with work in 422.2555377483368 seconds.
All threads are done in 422.2680640220642 seconds
```

Ejecución de programa PYTHON KLT con 2 Threads:

```
so@so:~/codigo-para-practicas/practica3/03-cpu-bound$ make run_klt_py
/home/so/codigo-para-practicas/practica3//.venv/bin/python3 klt.py
Starting the program.
[thread_id=139928883164864] Doing some work...
[thread_id=139928874772160] Doing some work...
500000th prime is 7368787
[thread_id=139928883164864] Done with work in 420.18873739242554 seconds.
500000th prime is 7368787
[thread_id=139928874772160] Done with work in 420.2887396812439 seconds.
All threads are done in 420.3469572067261 seconds
```

Ejecución de programa PYTHON ULT con 2 Threads:

```
so@so:~/codigo-para-practicas/practica3/03-cpu-bound$ make run_ult_py
/home/so/codigo-para-practicas/practica3//.venv/bin/python3 ult.py
Starting the program.
[greenlet_id=140422582979296] Doing some work...
500000th prime is 7368787
[greenlet_id=140422582979296] Done with work in 237.2826886177063 seconds.
[greenlet_id=140422577307328] Doing some work...
500000th prime is 7368787
[greenlet_id=140422577307328] Done with work in 174.65504837036133 seconds.
All greenlets are done in 411.9894025325775 seconds
```

Ejecución de programa PYTHON KLT sin GIL y 2 Threads:

```
so@so:~/codigo-para-practicas/practica3/03-cpu-bound$ su -c 'podman run -it
--rm --network=none -v .:/mnt docker.io/felopez/python-nogil:latest python3
-X gil=0 /mnt/klt.py'
Contraseña:
Starting the program.
[thread_id=140472969586368] Doing some work...
[thread_id=140472961193664] Doing some work...
500000th prime is 7368787
[thread_id=140472961193664] Done with work in 138.21149921417236 seconds.
500000th prime is 7368787
[thread_id=140472969586368] Done with work in 138.68540263175964 seconds.
All threads are done in 138.69032835960388 seconds
```

6. El directorio `practica3/04-io-bound` contiene programas en C y en Python que ejecutan una tarea que simula ser IO-Bound (tiene una llamada a sleep lo que permite interleaving de forma similar al uso de IO).
    1. Ejecute `htop` en una terminal separada para monitorear el uso de CPU en los siguientes incisos.
    2. Ejecute los distintos ejemplos con make (usar `make help` para ver cómo) y observe cómo aparecen los resultados, cuánto tarda cada thread y cuanto tarda el programa completo en finalizar.
    3. ¿Cómo se comparan los tiempos de ejecución de los programas escritos en C (ult y klt)?

- Para los programas en **C** los tiempos son muy parecidos esta vez, esto se debe a que en **tareas I/O-bound**, los hilos pasan la mayor parte del tiempo **esperando** (por ejemplo, `sleep()`, operaciones de red, disco, etc.), y no compitiendo por la CPU.
    - En el caso de los *KLTs*, al ser gestionados por el sistema operativo, cuando un hilo está esperando I/O, el *kernel puede planificar otro* para que use la CPU, aprovechando mejor el paralelismo.
    - En el caso de los *ULTs*, si estos están bien diseñados, los *ULT* pueden hacer yield manualmente durate operaciones de I/O, permitiendo que otros hilos se ejecuten más eficientemente.

    4. ¿Cómo se comparan los tiempos de ejecución de los programas escritos en Python (ult.py y klt.py)?

- Acá los tiempos son bastante rápidos ya que el **GIL** no afecta negativamente en tareas I/O-Bound:
    - En `klt.py` con GIL aunque el GIL impide la ejecución paralela real en CPU-bound, *no bloquea operaciones de I/O*, porque Python *libera el GIL* cuando una función hace `sleep()`, espera un socket, etc.
    - En `ult.py` con GIL se usan librerías como `greenlet` o `gvent` que permiten la concurrencia cooperativa, estos funcionan muy bien con I/O, pero requieren que el código sea explícitamente no bloqueante.
    - En `klt.py` sin GIL no hay muchos cambios al aso que usa GIL ya que los bloqueos de I/O ya permiten a otros hilos avanzar sin problemas.

    5. ¿Qué conclusión puede sacar respecto a los ULT en tareas IO-Bound?

- En conclusión, los **ULT** se desempeñan correctamente en tareas I/O-bound, pero suelen ser ligeramente menos eficientes que los KLT debido al modelo de planificación cooperativa y al overhead adicional.

Ejecución de programa C KLT:

```
so@so:~/codigo-para-practicas/practica3/04-io-bound$ make run_klt
cc -Wall -Werror   -c -o common.o common.c
cc -Wall -Werror    klt.c common.o  -lpth -o klt
./klt
Starting the program.
[Thread 140189649520320] Doing some work...
[Thread 140189641127616] Doing some work...
[Thread 140189632734912] Doing some work...
[Thread 140189624342208] Doing some work...
[Thread 140189615949504] Doing some work...
[Thread 140189641127616] Done with work in 10.000223 seconds.
[Thread 140189649520320] Done with work in 10.000720 seconds.
[Thread 140189615949504] Done with work in 10.000276 seconds.
[Thread 140189624342208] Done with work in 10.000657 seconds.
[Thread 140189632734912] Done with work in 10.001152 seconds.
All threads are done in 10.002021 seconds
```

Ejecución de programa C ULT:

```
so@so:~/codigo-para-practicas/practica3/04-io-bound$ make run_ult
cc -Wall -Werror    ult.c common.o  -lpth -o ult
./ult
Starting the program.
[Thread 94628825678560] Doing some work...
[Thread 94628825745616] Doing some work...
[Thread 94628825812672] Doing some work...
[Thread 94628825879728] Doing some work...
[Thread 94628825946784] Doing some work...
[Thread 94628825678560] Done with work in 10.106368 seconds.
[Thread 94628825745616] Done with work in 10.106560 seconds.
[Thread 94628825812672] Done with work in 10.106568 seconds.
[Thread 94628825879728] Done with work in 10.106569 seconds.
[Thread 94628825946784] Done with work in 10.106570 seconds.
All threads are done in 10.106652 seconds
```

Ejecución de programa PYTHON KLT:

```
so@so:~/codigo-para-practicas/practica3/04-io-bound$ make run_klt_py
/home/so/codigo-para-practicas/practica3//.venv/bin/python3 klt.py
Starting the program.
[thread_id=140051762374336] Doing some work...
[thread_id=140051753981632] Doing some work...
[thread_id=140051745588928] Doing some work...
[thread_id=140051737196224] Doing some work...
```

```
[thread_id=140051728787136] Doing some work...
[thread_id=140051762374336] Done with work.
[thread_id=140051745588928] Done with work.
[thread_id=140051728787136] Done with work.
[thread_id=140051753981632] Done with work.
[thread_id=140051737196224] Done with work.
All threads are done in 10.050863265991211 seconds
```

Ejecución de programa PYTHON ULT:

```
so@so:~/codigo-para-practicas/practica3/04-io-bound$ make run_ult_py
/home/so/codigo-para-practicas/practica3//.venv/bin/python3 ult.py
Starting the program.
[greenlet_id=140538327696544] Doing some work...
[greenlet_id=140538326987488] Doing some work...
[greenlet_id=140538320812160] Doing some work...
[greenlet_id=140538319119328] Doing some work...
[greenlet_id=140538319119488] Doing some work...
[greenlet_id=140538327696544] Done with work.
[greenlet_id=140538326987488] Done with work.
[greenlet_id=140538320812160] Done with work.
[greenlet_id=140538319119328] Done with work.
[greenlet_id=140538319119488] Done with work.
All greenlets are done in 10.29958724975586 seconds
```

Ejecución de programa PYTHON KLT sin GIL:

```
so@so:~/codigo-para-practicas/practica3/04-io-bound$ su -c 'podman run -it --rm --network=none -v .:/mnt docker.io/felopez/python-nogil:latest python3 -X gil=0 /mnt/klt.py'
Contraseña:
Starting the program.
[thread_id=140400935429824] Doing some work...
[thread_id=140400927037120] Doing some work...
[thread_id=140400918644416] Doing some work...
[thread_id=140400910251712] Doing some work...
[thread_id=140400901859008] Doing some work...
[thread_id=140400935429824] Done with work.
[thread_id=140400927037120] Done with work.
[thread_id=140400918644416] Done with work.
[thread_id=140400910251712] Done with work.
[thread_id=140400901859008] Done with work.
All threads are done in 10.012255668640137 second
```

7. Diríjase nuevamente en la terminal a `practica3/03-cpu-bound` y modifique klt.py de forma que vuelva a crear 5 threads.
    1. Ejecute htop en una terminal separada para monitorear el uso de CPU en los siguientes incisos.
    2. Ejecute una versión de Python que tenga el GIL deshabilitado usando: `make run_klt_py_nogil` (esta operación tarda la primera vez ya que necesita descargar un container con una versión de Python compilada explícitamente con el GIL deshabilitado).
    3. ¿Cómo se comparan los tiempos de ejecución de klt.py usando la versión normal de Python en contraste con la versión sin GIL?
    - Este análisis está hecho en el punto de CPU-Bound.
    4. ¿Qué conclusión puede sacar respecto a los KLT con el GIL de Python en tareas CPU-Bound?
    - En conclusión, en tareas CPU-Bound, los KLT (Kernel-Level Threads) en Python se ven fuertemente limitados por el GIL (Global Interpreter Lock), lo que impide aprovechar el paralelismo real y reduce la eficiencia de la concurrencia.