Práctica 3 - Hilos

Conceptos generales

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

La diferencia fundamental es que un hilo se refiere a un flujo de ejecución, al cual se le asigna tiempo de computo, mientras que un proceso puede tener multiples hilos y es al que se le asignan los recursos como archivos, memoria, etc.

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

Los user level threads son hilos lógicos, transparentes al sistema operativos, que se usan para simular concurrencia más no se ejecutan en paralelo necesariamente, los kernel level threads se refieren a como el sistema operativo ve los hilos de un proceso, estos se ejecutan de forma paralela y concurrente pero a diferencia de los ULT requieren cambio de contexto para poder ejecutarse. Normalmente los ULT se mapean a KLT de alguna forma.

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

La planificación de hilos ULT se hace a nivel del runtime de la librería de hilos provista por el lenguaje, mientras que la planificación de los KLT lo hace el SO. Solo los KLT se ejecutan de forma paralela.

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

Les asigna tiempo de compute en los diferentes núcleos disponibles, son más ligeros que los procesos puesto que comparten recursos y son más fáciles de crear pero todos poseen program counter, stack y estado.

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

La ventaje es que los KLT se ejecutan en concurrente, la desventaja es que hay overhead en cuanto a crearlos, sacarlos y ponerlos en un núcleo ya que interviene el SO y eso siempre ralentiza.

6. Qué retornan las siguientes funciones:

a. `getpid()` → Retorna el **PID (ID del proceso actual)**.
b. `getppid()` → Retorna el **PID del proceso padre (PPID)**.
c. `gettid()` → Retorna el **TID (ID del hilo a nivel de kernel)**. *(Linux específico)*
d. `pthread_self()` → Retorna el **ID del hilo a nivel de usuario (pthread_t)**. *(POSIX threads)*
e. `pth_self()` → Retorna el **ID del hilo en la biblioteca GNU Pth** *(menos común, alternativa a pthreads)*.

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

- **Mutex** (*Mutual Exclusion*): Bloquea recursos para un solo hilo/proceso a la vez.
- **Semáforos**: Controlan el acceso a recursos compartidos con un contador.
- **Variables de condición** (*Condition Variables*): Permiten a los hilos esperar por señales.
- **Barreras**: Sincronizan hilos hasta que todos alcancen un punto.
- **Spinlocks**: Bucles de espera activa (útil en multicore con bloqueos cortos).

Si son necesario los mecanismo en ULT si comparten datos, simplemente por el hecho de que la ejecución de un hilo puede interrumpir a otro y se pueden romper las suposiciones hechas por el otro hilo en ese caso.

8. Procesos

a. ¿Qué utilidad tiene ejecutar fork() sin ejecutar exec()? -> Util para crear procesos que tengan mismo código, por ejemplo un servidor web que ejecute multiples clientes haría fork().

b. ¿Qué utilidad tiene ejecutar `fork()` + `exec()`? -> Crea procesos y le cambiamos el código con exec(), para crear procesos distintos al padre.

c. ¿Cuál de las 2 asigna un nuevo PID: `fork()` o `exec()`? -> fork() asigna nuevo PID

d. ¿Qué implica el uso de Copy-On-Write (COW) cuando se hace `fork()`? -> Es una optimización para que el fork() no copie toda la memoria del padre al hijo al inicio, sino recién cuando uno de ellos las modifique.

e. ¿Qué consecuencias tiene no hacer `wait()` sobre un proceso hijo? -> Sino hay wait() sobre los hijos y estos terminan quedan como procesos zombies.

f. ¿Quién tendrá la responsabilidad de hacer el `wait()` si el padre termina sin hacerlo? -> El proceso init (PID 1) debe limpiarlo.

9. Kernel Level Threads

a. ¿Qué elementos del espacio de direcciones comparten los threads creados con `pthread_create()`?
Los hilos (KLT) comparten **todo el espacio de direcciones del proceso padre**, incluyendo:

- **Código ejecutable** (segmento *text*).
- **Datos globales y heap** (variables globales, memoria dinámica con `malloc`).

- **Archivos abiertos** (file descriptors).
- **Señales y handlers de señales**.

No comparten:

- **Stack** (cada hilo tiene su propio stack).
- **Registros de CPU y contador de programa** (contexto de ejecución).

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

- `getpid()`: Retorna el **PID del proceso** (compartido por todos sus hilos).
- `gettid()`: Retorna el **TID (Thread ID) único del hilo** (asignado por el kernel).
- **Relación**: Todos los hilos de un mismo proceso tendrán el mismo `pid` (vía `getpid()`), pero cada uno tiene un `tid` distinto (vía `gettid()`).

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

- Recolecta estado, evitando fugas de memoria y sincroniza hilos.
- Libera recursos y evita hilos zombies.

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

- El hilo se bloquea pero los demás no.

e. ¿Qué ocurre a nivel del SO al invocar `pthread_create()`? ¿Es syscall? ¿Usa `clone()`?
En Linux, `pthread_create()` usa la syscall `clone()` con flags específicos:
`clone(..., CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, ...);`

10. User Level Threads

a. ¿Por qué los ULTs no se pueden ejecutar en paralelo sobre múltiples núcleos?

- **El kernel no los ve**: Los ULTs son manejados **enteramente en espacio de usuario** (por una biblioteca, no por el SO).
- **El proceso es la unidad de planificación**: El kernel asigna CPU al **proceso completo**.

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

1. **Menor overhead**: No hay syscalls para creación de y conmutación.
2. **Flexibilidad**: La biblioteca de hilos los planifica.
3. **Portabilidad**: Funcionan en sistemas sin hilos.

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

- `getpid()`: Retorna el **PID del proceso** (igual para todos los ULTs, ya que el kernel solo ve el proceso).
- `gettid()`: **No aplica** (en ULTs, el kernel no asigna TIDs; retorna el PID del proceso).

- `pth_self()`: Retorna el **ID del hilo a nivel de usuario** (asignado por la biblioteca GNU Pth, no por el kernel).

d. ¿Qué pasaría si un ULT realiza una syscall bloqueante como `read()`?
Bloquea todos los otros hilos.

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

1. **Tipos de scheduling**:
   - **Cooperativo (no-preemptivo)**: Cada hilo cede el control explícitamente (ej: con `pth_yield()`).
   - **Por prioridades**: La biblioteca asigna CPU según prioridades definidas.
   - **Round-robin**: Los hilos se turnan en intervalos fijos (raro en ULTs).
2. **Más común**:
   - **Scheduling cooperativo** (ej: GNU Pth). Es simple y evita problemas de concurrencia, pero depende de que los hilos cedan el control voluntariamente.

11. Global Interpreter Lock

a. ¿Qué es el GIL? ¿Qué impacto tiene sobre programas multi-thread en Python y Ruby?

- El **GIL** es un *mutex* (cerrojo global) que **evita que múltiples hilos ejecuten código interpretado al mismo tiempo** en implementaciones como **CPython (Python)** y **MRI (Ruby)**.
- - **I/O-bound**: Funcionan bien (el GIL se libera durante operaciones bloqueantes como `read()` o `sleep()`).
- **CPU-bound**: **No hay paralelismo real** (aunque haya múltiples hilos, solo uno usa la CPU a la vez).
- **Consecuencia**: Los programas intensivos en CPU **no se aceleran con hilos**, sino que incluso pueden ser más lentos por el overhead del GIL.

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

- **Hilos (Threads)**:
  - Comparten el mismo intérprete (y el GIL) → **no paralelizan carga CPU**.
  - Útiles solo para **I/O concurrente** (ej: servidores web).
- **Procesos (Multiprocessing)**:
  - Cada proceso tiene su propio intérprete (y su propio GIL) → **paralelismo real en múltiples núcleos**.
  - Evitan el cuello de botella del GIL.

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:

a. Compile los 3 programas C usando el comando make.

```
root@vbox:/home/so/codigo-para-practicas/practica3/01-strace# make
cc -Wall -Werror    01-subprocess.c  -lpth -o 01-subprocess
cc -Wall -Werror    02-kl-thread.c  -lpth -o 02-kl-thread
cc -Wall -Werror    03-ul-thread.c  -lpth -o 03-ul-thread
```

b. 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:
i. Comparten el mismo PID y THREAD_ID
ii. Comparten el mismo PID pero con diferente THREAD_ID
iii. Tienen distinto PID

```
root@vbox:/home/so/codigo-para-practicas/practica3/01-strace# ./01-subprocess Parent process: PID = 4560, THREAD_ID = 4560 Child process: PID = 4561, THREAD_ID = 4561
```

```
root@vbox:/home/so/codigo-para-practicas/practica3/01-strace# ./02-kl-thread Parent process: PID = 4592, THREAD_ID = 4592 Child thread: PID = 4592, THREAD_ID = 4593
```

```
root@vbox:/home/so/codigo-para-practicas/practica3/01-strace# ./03-ul-thread Parent process: PID = 4601, THREAD_ID = 4601, PTH_ID = 94790874634480 Child thread: PID = 4601, THREAD_ID = 4601, PTH_ID = 94790874637024
```

c. Ejecute cada programa usando strace (strace ./nombre*programa > /dev/null) y responda:*
i. ¿En qué casos se invoca a la systemcall clone o clone3 y en cuál no? ¿Por qué?
1. En el caso de subprocess se clona.
2. En el caso de klthread usa clone3
3. Creería que no se usa ninguno.
`ii. 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. - CLONE_VM se usa en klthread, no vi ningún otro.`
iii. 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.
***1.** `CLONE_VM` **(Compartir espacio de memoria)** - Indica que el nuevo "hilo/proceso" **compartirá el mismo espacio de direcciones de memoria** que el proceso padre.*
*- **Relación con hilos vs procesos**:*

- ***Hilos (KLT)**: Usan `CLONE_VM` → Todos comparten memoria (heap, datos globales).*
- ***Procesos**: No usan `CLONE_VM` → Cada uno tiene su propio espacio de memoria aislado.*
*1. `CLONE_THREAD` **(Hilo dentro del mismo grupo de hilos)** - El nuevo hilo se agrega al **mismo grupo de hilos** que el padre (mismo TGID _Thread Group ID*, que es el PID visto con `getpid()`).
iv. 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.
- Solo se ven los del proceso padre PERO EN EL CASO DEL UL SE DEBERÍAN VER TODOS DADO QUE HAY UN SOLO HILO A NIVEL DE KERNEL.

v. 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?

- -f hace el seguimiento de los hilos y llamadas creadas por fork(), vfork() y clone2().
- Klthread solo imprime el padre (aunque supuestamente se debería seguir tmb al hijo)
- Subprocesos imprime hijo y padre
- En UL Solo se ven los del proceso padre PERO EN EL CASO DEL HIJO SE DEBERÍAN VER TODOS DADO QUE HAY UN SOLO HILO A NIVEL DE KERNEL.

4. Resuelva y responda utilizando el contenido del directorio practica3/02-memory:

a. Compile los 3 programas C usando el comando make.
b. Ejecute los 3 programas.
c. Observe qué pasa con la modificación a la variable number en cada caso. ¿Por qué suceden cosas distintas en cada caso?

```
root@vbox:/home/so/codigo-para-practicas/practica3/02-memoria# ./01-subprocess Parent process: PID = 8112, THREAD_ID = 8112 Parent process: number = 42 Child process: PID = 8113, THREAD_ID = 8113 Child process: number = 84 Parent process: number = 42
root@vbox:/home/so/codigo-para-practicas/practica3/02-memoria# ./02-kl-thread Parent process: PID = 8171, THREAD_ID = 8171 Parent process: number = 42 Child thread: PID = 8171, THREAD_ID = 8172 Child process: number = 84 Parent process: number = 84
root@vbox:/home/so/codigo-para-practicas/practica3/02-memoria# ./03-ul-thread Parent process: PID = 8191, THREAD_ID = 8191, PTH_ID = 94445797058800 Parent process: number = 42 Child thread: PID = 8191, THREAD_ID = 8191, PTH_ID = 94445797061344 Child thread: number = 84 Parent process: number = 84
```

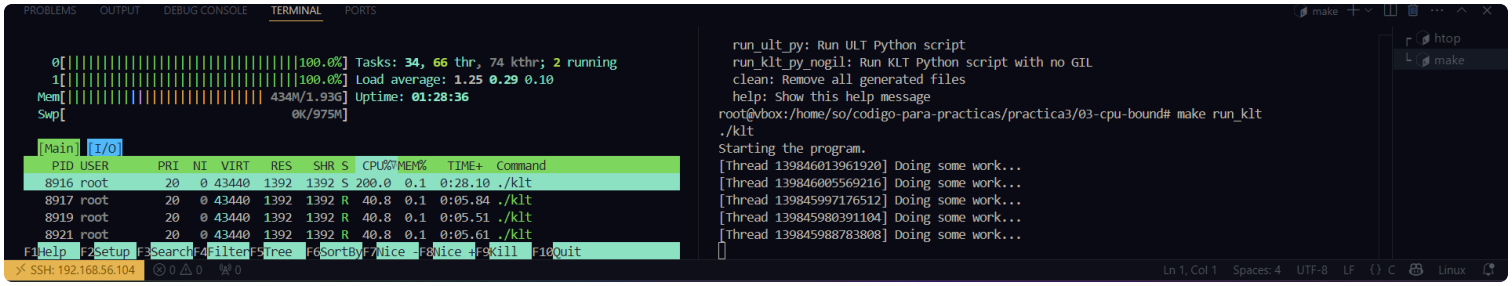
1. **Proceso hijo (`fork()`)**
   - **Padre**: `number = 42` (inicial) → **sigue siendo 42** después del hijo.
   - **Hijo**: Modifica `number = 84` (**solo afecta al hijo**).
   - **Causa**: Memoria **no compartida** (copia independiente por `fork()`).
2. **Hilo KLT (`pthread_create()`)**

- **Padre/hilo**: Comparten `number`.
- Cambio a `84` en el hijo **afecta al padre** (valor final: `84`).
- **Causa**: Memoria **compartida** (mismo espacio de direcciones).

3. **Hilo ULT (`pth_create()`)**
   - Mismo comportamiento que KLT: `number = 84` al final (**compartido**).
   - **Diferencia clave**: El kernel **no lo ve** como hilo (solo 1 PID).

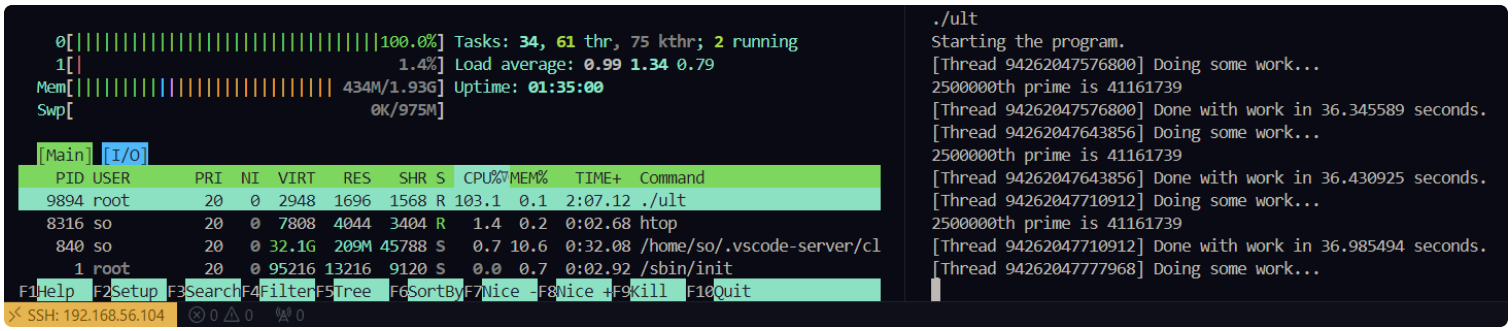
5. El directorio practica3/03-cpu-bound contiene programas en C y en Python que ejecutan una tarea CPU-Bound (calcular el enésimo número primo).

a. Ejecute htop en una terminal separada para monitorear el uso de CPU en los siguientes incisos.
b. 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.
c. ¿Cuántos threads se crean en cada caso?
KLT:




```
2500000th prime is 41161739 [Thread 139846013961920] Done with work in 159.257282 seconds.
2500000th prime is 41161739 [Thread 139845988783808] Done with work in 159.427690 seconds.
2500000th prime is 41161739 [Thread 139845980391104] Done with work in 159.546438 seconds.
2500000th prime is 41161739 [Thread 139846005569216] Done with work in 159.864912 seconds.
2500000th prime is 41161739 [Thread 139845997176512] Done with work in 160.276795 seconds.
All threads are done in 160.279294 seconds
```

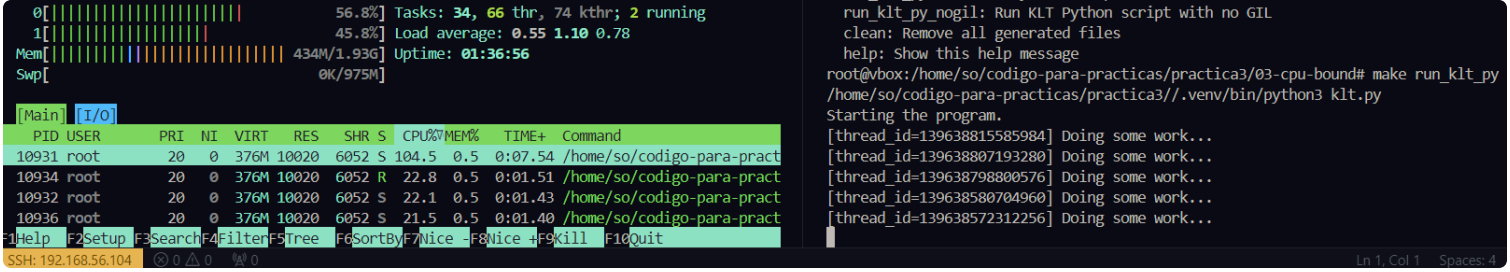
ULT:



1 hilo de ejecución real.

```
All threads are done in 186.000000 seconds
```

PY KLT:



5 hilos reales, 1 solo se debería ejecutar a la vez

PY 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`

Hilos secuenciales.

PY KLT NO GIL:

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

Este se rompia con el comando original, tuve que tirar

`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`d. ¿Cómo se comparan los tiempos de ejecución de los programas escritos en C (ult y klt)? KLT es más rápido, aun a pesar de tener más hilos q procesadores. e. ¿Cómo se comparan los tiempos de ejecución de`

`los programas escritos en Python (ult.py y klt.py)? No cambian realmente porque el GIL limita la ejecución paralela a un solo hilo por interprete, de hecho klt debería ser más lento. f. 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?` 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`KLT sin gil:` 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`

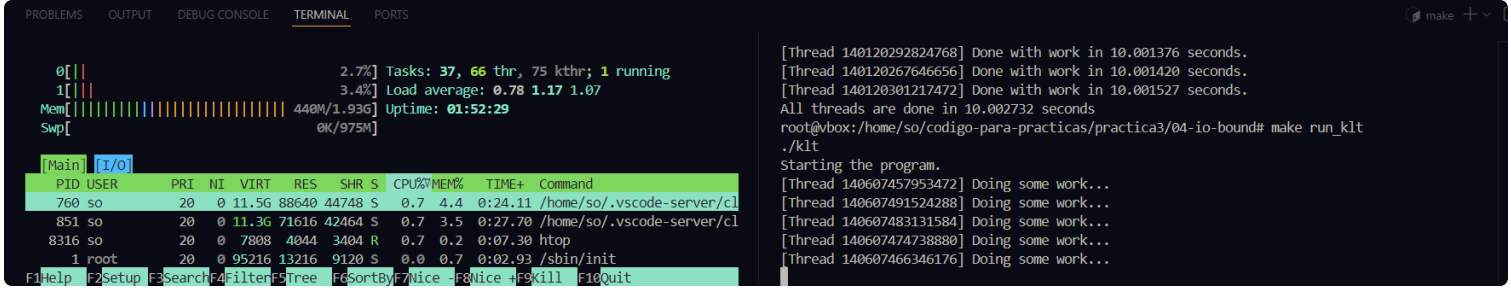
g. ¿Qué conclusión puede sacar respecto a los ULT en tareas CPU-Bound?
Más hilos que se ejecuten efectivamente en paralelo gana más velocidad, ult no sería el caso de éxito de velocidad.
Adoso las preguntas del 7 (por que ejecute el sin GIL acá también)

7 Diríjase nuevamente en la terminal a practica3/03-cpu-bound y modifique klt.py de forma que vuelva a crear 5 threads.

a. Ejecute htop en una terminal separada para monitorear el uso de CPU en los siguientes incisos.
b. 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).
c. ¿Cómo se comparan los tiempos de ejecución de klt.py usando la versión normal de Python en contraste con la versión sin GIL?
d. ¿Qué conclusión puede sacar respecto a los KLT con el GIL de Python en tareas CPU-Bound?
Python sin GIL y KLT es mucho más rápido porque hay paralelismo real.

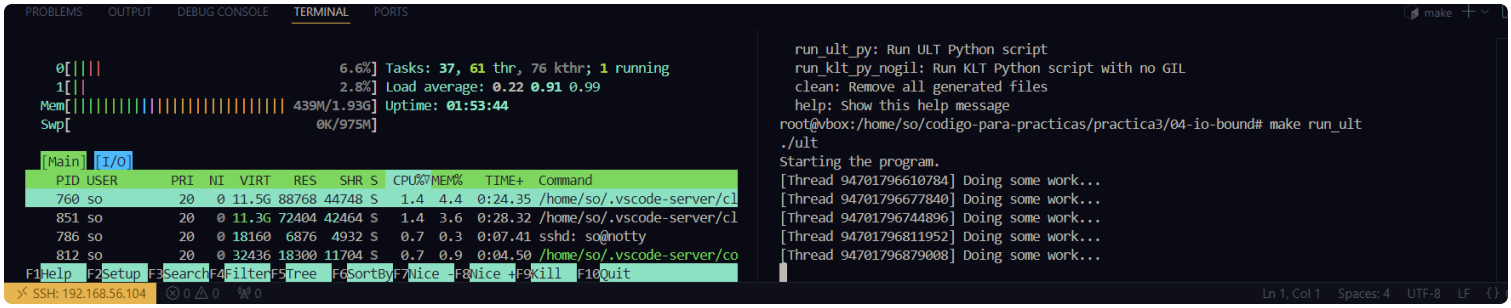
6. El directorio practica3/04-io-bound contiene programas en C y en Python que ejecutan una tarea que simula ser IO-Bound (tiene una llamada a sleep lo que permite interleaving de forma similar al uso de IO).

a. Ejecute htop en una terminal separada para monitorear el uso de CPU en los siguientes incisos.
b. 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.
KLT:



Tarda 10 segundos cada hilo y en total 10 segundos.

ULT:



Tardo 10 segundos y cada hilo 10 segundos (rari).

KLT PY:

Mismo que lo anterior, 10 segundos.

ULT PY:

10 segundos.

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

Casi ni hay diferencia.

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

Casi ni hay diferencia.

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

ULT vs KLT parece no haber diferencia en tarea de IO-Bound