

Threading (ULT y KLT)

Conceptos generales

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

Proceso: Es una instancia independiente de un programa en ejecución. Tiene su propio espacio de memoria.

Thread (hilo): Es una unidad de ejecución dentro de un proceso. Comparte la memoria (heap y segmento de datos) con otros hilos del mismo proceso.

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

ULT: Hilos gestionados totalmente en espacio de usuario por una biblioteca (ej: greenlets, pthreads con implementación en user-space).

KLT: Hilos gestionados por el kernel, el sistema operativo los ve y los planifica individualmente.

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

ULT: Los planifica el scheduler en user-space (la propia app o lib).

KLT: Los planifica el scheduler del kernel.

En sistemas multinúcleo, KLT pueden ejecutarse en paralelo.

ULT no, porque solo hay un hilo de kernel debajo → solo 1 núcleo se usa.

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

El SO maneja los KLT como unidades independientes de planificación.

Cada KLT tiene su propio ID (TID), pila y registros, pero comparte el espacio de memoria con otros hilos del mismo proceso.

A diferencia de los procesos, los KLT comparten memoria.

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

✓ Ventajas de los KLT:

Paralelismo real

Los KLT pueden ser planificados por el sistema operativo en múltiples núcleos, permitiendo ejecución en paralelo de varios hilos.

Bloqueo independiente

Si un KLT se bloquea (por ejemplo, en una llamada a read()), los demás hilos pueden seguir ejecutándose.

En cambio, con ULT, si uno se bloquea, todos los demás se bloquean también.

Mejor integración con el sistema operativo

El SO puede asignar prioridades, manejar señales, hacer balanceo de carga entre núcleos, etc.

✗ Desventajas de los KLT:

Cambio de contexto más costoso

Cambiar de un hilo a otro implica entrar al modo kernel, lo que es más lento que un cambio de contexto entre ULT (que ocurre en user-space).

Mayor sobrecarga de gestión

El sistema operativo necesita mantener estructuras adicionales para cada hilo, lo que consume más recursos.

Menor control desde el usuario

Como el planificador es el kernel, no se puede personalizar fácilmente el scheduling como en los ULT.

Criterio	KLT	ULT
Visibilidad al SO	Sí	No
Ejecución en múltiples núcleos	Sí	No
Coste de cambio de contexto	Alto (modo kernel)	Bajo (solo user-space)
Llamadas bloqueantes	Bloquean solo ese hilo	Bloquean a todos
Control del scheduling	SO (menos control del user)	Usuario (más control)

6. Qué retornan las siguientes funciones:

a. `getpid()`

Retorna el PID (Process ID) del proceso actual.

b. `getppid()`

Retorna el PID del proceso padre del proceso actual.

c. `gettid()`

Retorna el TID (Thread ID) del hilo actual gestionado por el kernel.

d. `pthread_self()`

Retorna un identificador del hilo actual (`pthread_t`) a nivel de POSIX threads.

e. `pth_self()`

Retorna el identificador del ULT actual (solo en bibliotecas como GNU Pth).

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

Mutexes, semáforos, monitores, barreras, spinlocks.

Con ULT también son necesarios, porque aunque no haya paralelismo real, sí hay concurrencia: varios hilos pueden acceder a memoria compartida intercaladamente, lo que puede causar race conditions.

8. Procesos

a. ¿Qué utilidad tiene ejecutar `fork()` sin ejecutar `exec()`?

Permite crear un proceso hijo idéntico al padre, útil para procesamiento paralelo, pipelines, etc., sin cambiar el binario.

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

se crea un nuevo proceso (`fork()`) y se reemplaza su código con otro ejecutable (`exec()`).

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

fork()

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

Que el hijo y el padre comparten las mismas páginas de memoria al principio. Solo si una escribe, se hace una copia real. Ahorra memoria y es eficiente.

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

El hijo se convierte en zombie: termina, pero sus datos quedan en la tabla del sistema hasta que alguien haga wait().

f. ¿Quién tendrá la responsabilidad de hacer el wait() si el proceso padre termina sin hacer wait()?

El proceso hijo es adoptado por init (PID 1), que luego hará el wait() para limpiar su entrada.

9. Kernel Level Threads

a. ¿Qué elementos del espacio de direcciones comparten los threads creados con pthread_create()?

Comparten:

Heap

Segmento de datos y código

Archivos abiertos

Memoria compartida

Señales

No comparten:

Pila (cada hilo tiene su propia pila)

Registros de CPU

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

getpid() → devuelve el PID del proceso principal (igual para todos los hilos).

gettid() → devuelve el Thread ID (TID) único de cada hilo, visible por el kernel (solo en Linux).

c. ¿Por qué pthread_join() es importante en programas que usan múltiples hilos?

¿Cuándo se liberan los recursos de un hilo zombie?

pthread_join() permite:

Esperar a que un hilo termine.

Recuperar su valor de retorno.

Liberar automáticamente sus recursos.

Si no se llama, el hilo se vuelve un zombie y sus recursos quedan reservados hasta que se hace pthread_join() (o pthread_detach()).

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

No, en KLT solo el hilo bloqueado queda esperando.

Los demás hilos pueden seguir ejecutándose normalmente (a diferencia de los ULT).

e. Describí qué ocurre a nivel de sistema operativo cuando se invoca pthread_create()

(¿es syscall? ¿usa clone?).

Sí, implica una llamada al sistema (syscall).

Internamente en Linux, usa clone(), que permite crear un nuevo hilo compartiendo parte del espacio de direcciones del proceso padre.

10. User Level Threads

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

Porque el sistema operativo no los ve.

Solo ve un hilo de kernel → los ULT se ejecutan dentro de ese único hilo.

Por eso, solo uno puede correr a la vez, incluso en máquinas multinúcleo.

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

Cambio de contexto más rápido (todo en user space, sin pasar por el kernel).

Mayor control del scheduling por parte del desarrollador.

Menor sobrecarga del sistema operativo (no hay syscall para crear/cambiar de hilo).

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

getpid() → PID del proceso (igual para todos los ULT).

gettid() → ID del hilo de kernel que ejecuta todos los ULT (el mismo para todos).

pthread_self() → ID del hilo lógico en el espacio de usuario (único por ULT).

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

Se bloquea todo el proceso, ya que el hilo de kernel que ejecuta los ULT queda detenido.

Todos los ULTs quedan congelados hasta que read() termine.

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

Cooperativo (non-preemptive): cada hilo cede voluntariamente el control.

Preemptivo (preemptive): hay interrupciones periódicas (menos común en ULTs).

El más común: cooperativo (por simplicidad y menor complejidad técnica).

11. Global Interpreter Lock

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

El GIL es un candado global que permite que solo un hilo de Python/Ruby se ejecute a la vez, incluso en CPUs multinúcleo.

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

Porque cada proceso tiene su propio GIL, lo que permite usar múltiples núcleos en paralelo.

Práctica guiada

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

```
apt update
```

```
apt install build-essential libpthreads-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.

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:

```
capi@capi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P
3/practica3/01-strace$ ./01-subprocess
Parent process: PID = 5189, THREAD_ID = 5189
Child process: PID = 5190, THREAD_ID = 5190
capi@capi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/01-strace$ ./02-kl-thread
Parent process: PID = 5393, THREAD_ID = 5393
Child thread: PID = 5393, THREAD_ID = 5394
capi@capi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/01-strace$ ./03-ul-thread
Parent process: PID = 5433, THREAD_ID = 5433, PTH_ID = 104578014808304
Child thread: PID = 5433, THREAD_ID = 5433, PTH_ID = 104578014810848
```

i. Comparten el mismo PID y THREAD_ID

ii. Comparten el mismo PID pero con diferente THREAD_ID

iii. Tienen distinto PID

01: usa fork. crea un proceso completamente separado. Tienen distinto pid. el Tid también es distinto.iii

02: los pthreads son hilos que gestiona el kernel. comparten el mismo pid (porque están en el mismo proceso) pero tienen diferente thread_id pq cada hilo tiene su propio hilo. ii.

03: usa gnu pthread que hace hilos a nivel de usuario (o sea que el kernel no los ve, el usuario solo ve un solo hilo). Entonces el pid entre padre e hijo son iguales. El pthread_id es distinto pero esto es de la librería, es medio una mentirita y para el usuario comparten el mismo thread_id. i.

c. Ejecute cada programa usando strace (strace ./nombre_programa > /dev/null) y responda:

i. ¿En qué casos se invoca a la syscall clone o clone3 y en cuál no? ¿Por qué?

```
/usr/practica3/01-strings$ strace ./01-subprocess> /dev/null
execve("./01-subprocess", [".01-subprocess"], 0x7fff90517390 /* 55 vars */) = 0
brk(NULL)                               = 0x63fd92c94000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x78363cab4000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No existe el archivo o el directorio)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=86795, ...}) = 0
mmap(NULL, 86795, PROT_READ, MAP_PRIVATE, 3, 0) = 0x78363ca9e000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\13\0\0\0\0\0\0\0\0\0\3\0-\0\1\0\0\0\220\243\2\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\0\0\0\0..." , 784, 64) = 784
fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 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\0\0\0\0..." , 784, 64) = 784
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x78363c800000
mmap(0x78363c828000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x78363c828000
mmap(0x78363c9b0000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x78363c9b0000
mmap(0x78363c9ff000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x78363c9ff000
mmap(0x78363ca05000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x78363ca05000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x78363ca9b000
arch_prctl(ARCH_SET_FS, 0x78363ca9b740) = 0
set_tid_address(0x78363ca9ba10)        = 6038
set_robust_list(0x78363ca9ba20, 24)    = 0
rseq(0x78363ca9c060, 0x20, 0, 0x53053053) = 0
mprotect(0x78363c9ff000, 16384, PROT_READ) = 0
mprotect(0x63fd7d9fa000, 4096, PROT_READ) = 0
mprotect(0x78363cae000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x78363ca9e000, 86795)          = 0
gettid()                                = 6038
getpid()                                = 6038
fstat(1, {st_mode=S_IFCHR|0666, st_rdev=makedev(0x1, 0x3), ...}) = 0
ioctl(1, TCGETS, 0x7ffe6dd740f0)       = -1 ENOTTY (Función ioctl no apropiada para el dispositivo)
getrandom("\xdcc\xce\x61\x37\xb6\x17\x07\xb9", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x63fd92c94000
brk(0x63fd92cb5000)                    = 0x63fd92cb5000
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x78363ca9ba10) = 6039
wait4(-1, NULL, 0, NULL)                = 6039
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=6039, si_uid=1000, si_status=0, si_utime=0, si_stime=0} ---
write(1, "Parent process: PID = 6038, THRE...", 45) = 45
exit_group(0)                           = ?
+++ exited with 0 +++
```

No hay ni clone ni clone 3 porque fork() es una syscall separada que crea un proceso nuevo (NO un hilo).

[illegible]

En este `pthread_create` llama internamente a `clone()` y lo usa para crear el hilo de kernel.

En el 3 no se usa ni clone ni clone 3 porque pth hace hilos a nivel de usuario, no llama al kernel para crear hilos posta. Todo es dentro del mismo proceso, sin invocar syscalls de creación de hilos.

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.

```
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREA
D|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, child_tid=0x74442d800990, parent_tid=0x74442d800990, exit_signal=0,
stack=0x74442d000000, stack_size=0x7fff80, tls=0x74442d8006c0} => {parent_tid=[0]}, 88)
= 6166
```

CLONE_VM: El hijo comparte el mismo espacio de memoria que el padre (no hay copia). Es decir, mismo heap, stack, etc.

CLONE_THREAD: El hijo es tratado como un hilo del proceso padre. No un proceso separado.

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.

Programa	¿CLONE_VM?	¿CLONE_THREAD?	¿Por qué?
01-subprocess	✗ No	✗ No	Usa <code>fork()</code> , crea proceso separado.
02-kl-thread	✓ Sí	✓ Sí	Crea hilos de kernel, necesitan compartir memoria y PID.
03-ul-thread	✗ No	✗ No	Hilos de usuario, no llaman al kernel.

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.

1. Para 01-subprocess.c (con `fork()`):

El programa crea un proceso hijo con `fork()`. Ambos, el padre y el hijo, ejecutan `printf()` y, por lo tanto, ambos invocan `write(1, ...)` para escribir en la salida estándar.

2. Para 02-kl-thread.c (con hilos de kernel usando `pthread_create()`):

Este programa crea un hilo usando `pthread_create()`. Cuando el hilo hijo ejecuta `printf()`, invoca `write(1, ...)` también. Como los hilos de kernel son procesos ligeros dentro del mismo proceso principal, verás invocaciones a `write()` tanto para el padre como para el hilo hijo.

3. Para 03-ul-thread.c (hilos de usuario con `pth`):

Este programa crea hilos de usuario con la biblioteca `pth`. Los hilos de usuario son gestionados por el programa, no por el sistema operativo directamente. Como resultado, en este caso no se generan procesos nuevos, por lo que no es necesario utilizar `-f` con `strace`. Solo el proceso principal y los hilos que él crea ejecutan dentro del mismo PID.

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`?

`Strace -f` se usa para seguir los procesos creados por el proceso principal, es decir, si el proceso principal crea subprocesos (usando `fork()`, `clone()`, `pthread_create()`, etc.), `strace -f` seguirá también a esos procesos secundarios (subprocesos o hilos).

02:

Cuando usas -f, strace verá tanto las invocaciones de write() del hilo principal como de los hilos hijos.

Los hilos del kernel creados con pthread_create() usan clone(), que genera un nuevo subproceso que es tratado como un hilo del mismo proceso.

03:

Los hilos de usuario (pth en este caso) no crean procesos adicionales, ya que no usan clone() o pthread_create() para crear un nuevo proceso.

```
write(1, "Parent process: PID = 12345, THREAD_ID = 12345\n", 47) = 47
```

```
write(1, "Child thread: PID = 12345, THREAD_ID = 12346\n", 47) = 47
```

Todos los hilos (principal e hijo) están ejecutándose dentro del mismo proceso, por lo que no necesitas -f para rastrear las invocaciones de write(), ya que no hay procesos hijos que seguir.

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@so:/home/so/codigo-para-practicas/practica3/02-memoria# ./01-subprocess
Parent process: PID = 3438, THREAD_ID = 3438
Parent process: number = 42
Child process: PID = 3439, THREAD_ID = 3439
Child process: number = 84
Parent process: number = 42
```

1. Caso fork() (01-subprocess.c):

Se crea un nuevo proceso hijo que es una copia exacta del proceso padre (memoria, variables, etc.).

PERO:

→ El hijo tiene su propia copia de las variables globales.

→ Lo que modifique el hijo NO afecta al padre

Concretamente en tu programa:

number empieza en 42.

El hijo duplica su propia number a 84.

El padre sigue viendo number = 42.

```
root@so:/home/so/codigo-para-practicas/practica3/02-memoria# ./02-kl-thread
Parent process: PID = 3440, THREAD_ID = 3440
Parent process: number = 42
Child thread: PID = 3440, THREAD_ID = 3441
Child process: number = 84
Parent process: number = 84
```

Caso pthread_create() (02-kl-thread.c):

Se crea un nuevo hilo de nivel kernel (KLT, Kernel Level Thread).

Comparte el mismo espacio de memoria que el padre.

PERO tienen diferentes IDs de hilo (gettid()).

Concretamente en tu programa:

number empieza en 42.

El hijo (hilo) duplica number a 84.

El padre ahora ve también number = 84 porque ambos comparten la misma variable.

```
root@so:/home/so/codigo-para-practicas/practica3/02-memoria# ./03-ul-thread
Parent process: PID = 3442, THREAD_ID = 3442, PTH_ID = 94331816294640
Parent process: number = 42
Child thread: PID = 3442, THREAD_ID = 3442, PTH_ID = 94331816297184
Child thread: number = 84
Parent process: number = 84
```

3. Caso pth_spawn() (03-ul-thread.c):

Se crea un hilo de usuario (User-Level Thread, ULT).

También comparte el mismo espacio de memoria que el padre.

A nivel del kernel, sólo hay un proceso visible.

La diferencia con pthreads es que aquí la planificación la hace la propia librería Pth (no el kernel).

En tu programa:

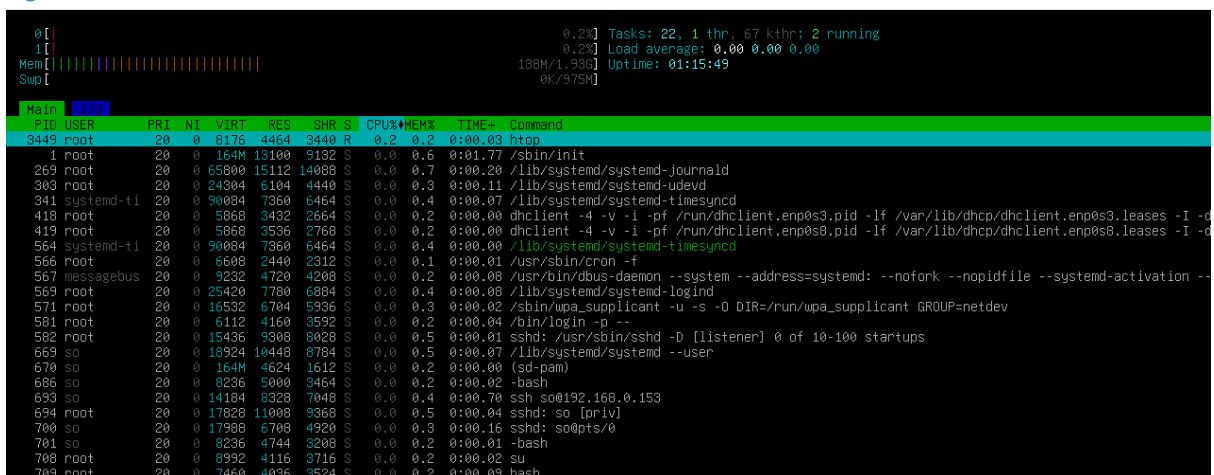
number empieza en 42.

El hilo duplica number a 84.

El padre ve number = 84 al final.

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.



htop output showing system statistics and a list of running processes.

System Statistics:

- Tasks: 22, 1 thr, 67 kthr, 2 running
- Load average: 0.00 0.00 0.00
- Uptime: 01:15:49
- Mem: 138M/1.93G
- 0K/975M

Process List:

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
3449	root	20	0	8176	4464	3440	R	0.2	0.2	0:00.03	htop
1	root	20	0	164M	13100	9132	S	0.0	0.6	0:01.77	/sbin/init
269	root	20	0	65800	15112	14088	S	0.0	0.7	0:00.20	/lib/systemd/systemd-journald
303	root	20	0	24304	6104	4440	S	0.0	0.3	0:00.11	/lib/systemd/systemd-udev
341	systemd-ti	20	0	90084	7360	6464	S	0.0	0.4	0:00.07	/lib/systemd/systemd-timesyncd
418	root	20	0	5868	3432	2664	S	0.0	0.2	0:00.00	dhclient -4 -v -i -pf /run/dhclient.enp0s3.pid -lf /var/lib/dhcp/dhclient.enp0s3.leases -I -d
419	root	20	0	5868	3536	2768	S	0.0	0.2	0:00.00	dhclient -4 -v -i -pf /run/dhclient.enp0s8.pid -lf /var/lib/dhcp/dhclient.enp0s8.leases -I -d
564	systemd-ti	20	0	90084	7360	6464	S	0.0	0.4	0:00.00	/lib/systemd/systemd-timesyncd
566	root	20	0	6608	2440	2312	S	0.0	0.1	0:00.01	/usr/sbin/cron -f
567	messagebus	20	0	9232	4720	4208	S	0.0	0.2	0:00.08	/usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation --
569	root	20	0	25420	7780	6804	S	0.0	0.4	0:00.08	/lib/systemd/systemd-logind
571	root	20	0	16532	6704	5936	S	0.0	0.3	0:00.02	/sbin/wpa_supplicant -u -s -O DIR=/run/wpa_supplicant GROUP=netdev
581	root	20	0	6112	4160	3592	S	0.0	0.2	0:00.04	/bin/login -p --
582	root	20	0	15436	9308	8028	S	0.0	0.5	0:00.01	sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
669	so	20	0	18924	10448	8764	S	0.0	0.5	0:00.07	/lib/systemd/systemd --user
670	so	20	0	164M	4624	1512	S	0.0	0.2	0:00.00	(sd-pam)
686	so	20	0	8236	5000	3464	S	0.0	0.2	0:00.02	-bash
693	so	20	0	14184	8328	7048	S	0.0	0.4	0:00.70	ssh so@192.168.0.153
694	root	20	0	17828	11008	9368	S	0.0	0.5	0:00.04	sshd: so [priv]
700	so	20	0	17988	6708	4920	S	0.0	0.3	0:00.16	sshd: so@pts/0
701	so	20	0	8236	4744	3208	S	0.0	0.2	0:00.01	-bash
708	root	20	0	8992	4116	3716	S	0.0	0.2	0:00.02	su
709	root	20	0	7460	4036	3524	S	0.0	0.2	0:00.09	bash

b. Ejecute los distintos ejemplos con make (usar make help para ver cómo) y observe cómo aparecen los resultados, cuánto tarda cada thread y cuánto tarda el programa completo en finalizar.

se

c. ¿Cuántos threads se crean en cada caso?

KLT:

C:

```
capicapi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/03-cpu-bound$ make run_klt
./klt
Starting the program.
[Thread 134124067096256] Doing some work...
[Thread 134124077582016] Doing some work...
[Thread 134124056610496] Doing some work...
[Thread 134124046124736] Doing some work...
[Thread 134124035638976] Doing some work...
2500000th prime is 41161739
[Thread 134124077582016] Done with work in 84.022794 seconds.
2500000th prime is 41161739
[Thread 134124056610496] Done with work in 84.959104 seconds.
2500000th prime is 41161739
[Thread 134124035638976] Done with work in 86.998932 seconds.
2500000th prime is 41161739
[Thread 134124046124736] Done with work in 87.041026 seconds.
2500000th prime is 41161739
[Thread 134124067096256] Done with work in 87.827785 seconds.
All threads are done in 87.828712 seconds
```

Python:

```
capicapi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3$ python3 -u "/home/capi/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/03-cpu-bound/klt.py"
Starting the program.
[thread_id=137215181063872] Doing some work...
[thread_id=137215170578112] Doing some work...
[thread_id=137215160092352] Doing some work...
[thread_id=137215076206272] Doing some work...
[thread_id=137215065720512] Doing some work...
500000th prime is 7368787
[thread_id=137215160092352] Done with work in 339.57690691947937 seconds.
500000th prime is 7368787
[thread_id=137215076206272] Done with work in 341.61664032936096 seconds.
500000th prime is 7368787
[thread_id=137215181063872] Done with work in 348.88388562202454 seconds.
500000th prime is 7368787
[thread_id=137215170578112] Done with work in 354.21234941482544 seconds.
500000th prime is 7368787
[thread_id=137215065720512] Done with work in 354.5322301387787 seconds.
All threads are done in 354.62728667259216 seconds
```

ULT:

C:

```

capi@capi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/03-cpu-bound$ make run_ult
./ult
Starting the program.
[Thread 95471796286176] Doing some work...
2500000th prime is 41161739
[Thread 95471796286176] Done with work in 31.192584 seconds.
[Thread 95471796353232] Doing some work...
2500000th prime is 41161739
[Thread 95471796353232] Done with work in 32.129400 seconds.
[Thread 95471796420288] Doing some work...
2500000th prime is 41161739
[Thread 95471796420288] Done with work in 31.425511 seconds.
[Thread 95471796487344] Doing some work...
2500000th prime is 41161739
[Thread 95471796487344] Done with work in 31.834025 seconds.
[Thread 95471796554400] Doing some work...
2500000th prime is 41161739
[Thread 95471796554400] Done with work in 32.135367 seconds.
All threads are done in 159.000000 seconds

```

python:

```

(venv) capi@capi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3$ python3 -u "/home/capi/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/03-cpu-bound/ult.py"
Starting the program.
[greenlet_id=123592495599744] Doing some work...
5000000th prime is 7368787
[greenlet_id=123592495599744] Done with work in 68.7728819847107 seconds.
[greenlet_id=123592481652256] Doing some work...
5000000th prime is 7368787
[greenlet_id=123592481652256] Done with work in 69.69430303573608 seconds.
[greenlet_id=123592481273184] Doing some work...
5000000th prime is 7368787
[greenlet_id=123592481273184] Done with work in 72.56086611747742 seconds.
[greenlet_id=123592479911872] Doing some work...
5000000th prime is 7368787
[greenlet_id=123592479911872] Done with work in 71.1054835319519 seconds.
[greenlet_id=123592479912032] Doing some work...
5000000th prime is 7368787
[greenlet_id=123592479912032] Done with work in 72.93916749954224 seconds.
All greenlets are done in 355.0893816947937 seconds

```

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

Los de python tardaron una booooocha más

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

Tardan prácticamente lo mismo (en python es una mentirita)

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?

KLT:

```

• (venv) capi@capi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3$ python -u "/home/capi/Escritorio/Facul
tad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/03-cpu-bound/klt.py"
Starting the program.
[thread_id=130157929039552] Doing some work...
[thread_id=130157840959168] Doing some work...
500000th prime is 7368787
[thread_id=130157840959168] Done with work in 144.42213487625122 seconds.
500000th prime is 7368787
[thread_id=130157929039552] Done with work in 144.64695048332214 seconds.
All threads are done in 144.64776182174683 seconds
• (venv) capi@capi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3$

```

ULT:

```

• (venv) capi@capi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3$ python3 -u "/home/capi/Escritorio/Facu
ltad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/03-cpu-bound/ult.py"
Starting the program.
[greenlet_id=129980286750848] Doing some work...
500000th prime is 7368787
[greenlet_id=129980286750848] Done with work in 69.61041045188904 seconds.
[greenlet_id=129980284976672] Doing some work...
500000th prime is 7368787
[greenlet_id=129980284976672] Done with work in 73.1119909286499 seconds.
All greenlets are done in 142.73326706886292 seconds
• (venv) capi@capi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3$

```

Probablemente tarda menos con menos hilos pq hay overhead de cambio de contextos y no es una tarea que se pueda paralelizar mucho.

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

No tiene mucho sentido usar varios hilos de usuario para tareas cpu bound porque aunque usemos varios hilos en realidad no se está usando más de un núcleo del proce por lo que no se ejecutan realmente en paralelo. Entonces al hacer más hilos medio que salis perdiendo porque agregas todo lo de la gestión de hilos.

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:

```

• (venv) capi@capi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/04-io-bound$ ./klt
Starting the program.
[Thread 124388158473920] Doing some work...
[Thread 124388147988160] Doing some work...
[Thread 124388137502400] Doing some work...
[Thread 124388127016640] Doing some work...
[Thread 124388116530880] Doing some work...
[Thread 124388158473920] Done with work in 10.000153 seconds.
[Thread 124388147988160] Done with work in 10.000077 seconds.
[Thread 124388137502400] Done with work in 10.000060 seconds.
[Thread 124388127016640] Done with work in 10.000060 seconds.
[Thread 124388116530880] Done with work in 10.000061 seconds.
All threads are done in 10.002214 seconds
• (venv) capi@capi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/04-io-bound$

```

ult:

```

• (venv) capi@capi-Aspire-S5-371T:~/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/04-io-bound$ ./ult
Starting the program.
[Thread 107078042108640] Doing some work...
[Thread 107078042175696] Doing some work...
[Thread 107078042242752] Doing some work...
[Thread 107078042309808] Doing some work...
[Thread 107078042376864] Doing some work...
[Thread 107078042108640] Done with work in 10.009874 seconds.
[Thread 107078042175696] Done with work in 10.009799 seconds.
[Thread 107078042242752] Done with work in 10.009708 seconds.
[Thread 107078042309808] Done with work in 10.009625 seconds.
[Thread 107078042376864] Done with work in 10.009494 seconds.
All threads are done in 10.010551 seconds

```

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

Tardaron prácticamente lo mismo

En los sistemas que manejan hilos del usuario (ULT), la gestión de hilos la realiza la propia aplicación (no el kernel), y cuando un hilo está en espera, otro hilo puede ser ejecutado.

Con hilos del kernel (KLT), el sistema operativo puede también intercalar la ejecución de hilos cuando estos están en espera de E/S, aprovechando el tiempo de inactividad.

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

ULT:

```

/capi/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/04-io-bound/ult.py"
Starting the program.
[greenlet_id=129946132549760] Doing some work...
[greenlet_id=129946117946272] Doing some work...
[greenlet_id=129946118182432] Doing some work...
[greenlet_id=129946116959712] Doing some work...
[greenlet_id=129946116959872] Doing some work...
[greenlet_id=129946132549760] Done with work.
[greenlet_id=129946117946272] Done with work.
[greenlet_id=129946118182432] Done with work.
[greenlet_id=129946116959712] Done with work.
[greenlet_id=129946116959872] Done with work.
All greenlets are done in 10.016585111618042 seconds

```

KLT:

```

/capi/Escritorio/Facultad/Septimo_Cuatrimestre/S0/Prácticas/P3/practica3/04-io-bound/kl.py"
Starting the program.
[thread_id=124162641233600] Doing some work...
[thread_id=124162630747840] Doing some work...
[thread_id=124162620262080] Doing some work...
[thread_id=124162536376000] Doing some work...
[thread_id=124162525890240] Doing some work...
[thread_id=124162641233600] Done with work.
[thread_id=124162630747840] Done with work.
[thread_id=124162620262080] Done with work.
[thread_id=124162536376000] Done with work.
[thread_id=124162525890240] Done with work.
All threads are done in 10.002647876739502 seconds

```

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

tardan casi lo mismo también

El GIL (Global Interpreter Lock): En Python, el GIL asegura que solo un hilo de ejecución esté ejecutando código de Python en un momento dado, incluso cuando hay múltiples hilos. Sin embargo, esto no afecta las operaciones de E/S (como sleep), ya que el GIL no bloquea otros hilos cuando están esperando E/S. En este tipo de operaciones, tanto los ULTs como los KLTs son bastante eficientes.

Operaciones IO-bound: En este tipo de tareas, la mayor parte del tiempo los hilos están esperando E/S. Dado que tanto los ULTs como los KLTs pueden aprovechar este tiempo de espera sin usar activamente la CPU, la diferencia en los tiempos de ejecución es mínima.

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.

ok

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).

```
500000th prime is 7368787
[thread_id=124687508047552] Done with work in 331.395388841629 seconds.
500000th prime is 7368787
[thread_id=124687610808000] Done with work in 335.5415132045746 seconds.
500000th prime is 7368787
[thread_id=124687631779520] Done with work in 339.60392904281616 seconds.
500000th prime is 7368787
[thread_id=124687621293760] Done with work in 339.7001214027405 seconds.
500000th prime is 7368787
[thread_id=124687529019072] Done with work in 340.75376176834106 seconds.
500000th prime is 7368787
[thread_id=124687497561792] Done with work in 341.2253770828247 seconds.
500000th prime is 7368787
[thread_id=124687518533312] Done with work in 343.3446891307831 seconds.
All threads are done in 343.3491325378418 seconds
```

c. ¿Cómo se comparan los tiempos de ejecución de klt.py usando la versión normal de Python en contraste con la versión sin GIL?

El GIL tiene más impacto en operaciones IO-bound, donde los hilos pueden estar esperando en operaciones de entrada/salida (como leer o escribir archivos), y en esos casos, deshabilitar el GIL permite que los hilos se ejecuten simultáneamente mientras esperan. Pero en el caso de operaciones que requieren mucho cálculo de CPU, el rendimiento generalmente está limitado por la capacidad de los núcleos de la CPU, no por la gestión del GIL.

d. ¿Qué conclusión puede sacar respecto a los KLT con el GIL de Python en tareas CPU-Bound?

Contestado arriba :p