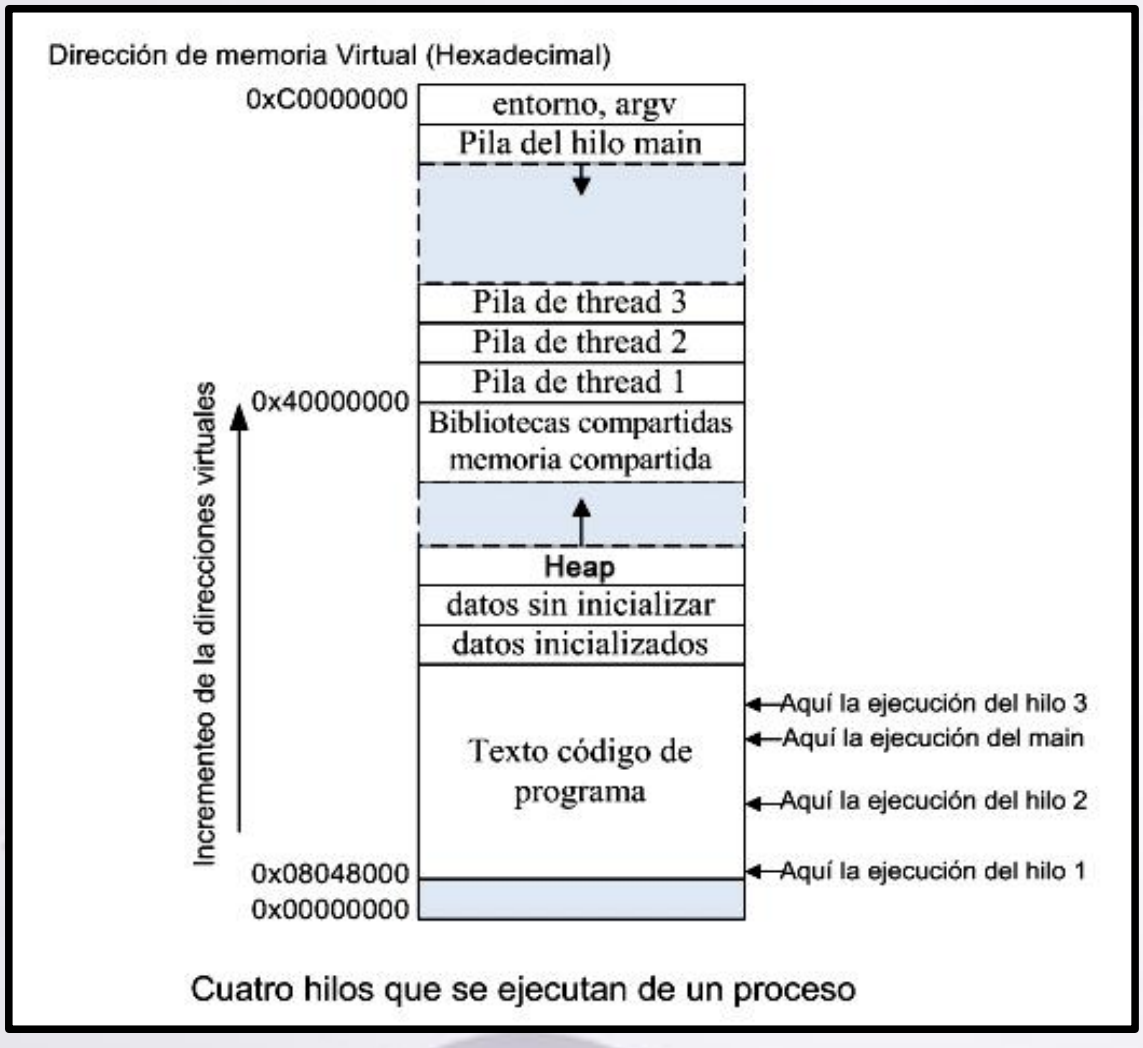
HILOS POSIX (THREADS)

HILOS

- Los hilos son un mecanismo que permiten a una aplicación realizar múltiples tareas al mismo tiempo.
- Un proceso puede tener múltiples hilos.
- Los hilos se ejecutan de forma independiente en el mismo programa.



Cuatro hilos que se ejecutan de un proceso

Todos los hilos de un proceso, comparten la misma memoria global, incluyendo los datos inicializados, datos sin inicializar, y pila, pero cada hilo tiene una pila privada para las variables locales.

Hilos

- Como los procesos, permiten ejecutar funciones en forma concurrente.
- Se ejecutan dentro de un mismo proceso. Comparten el mismo espacio de memoria.
- Útiles cuando en un proceso es necesario que varias tareas se realicen en simultáneo y trabajen en forma colaborativa.
- Todos los hilos comparten las mismas variables globales y la pila, pero cada hilo tiene una pila privada para las variables locales.
- El `main()` de un proceso es un hilo más y el primero que se crea.

Además de la memoria global, los hilos también comparten otros atributos.

Algunos de los atributos que comparten los hilos son:

- ID del proceso.
- ID del proceso padre.
- Descriptores de archivos abiertos.
- Credenciales de proceso (usuario y ID de grupo).

Algunos de los atributos que **no** comparten los hilos son:

- ID de hilos (TID).
- Datos específicos del hilo.
- Pila del hilo.

Ventajas de hilos frente a los procesos

- El intercambio de información entre los hilos es fácil y rápido. En cambio, entre procesos compartir información es más complejo y lento (IPC).

- La creación del hilo es más rápida que la creación de un proceso (típicamente entre x10 y x100 más rápido).

Desventajas de hilos

- Un error en un hilo (por ejemplo, la modificación de la memoria) puede afectar la ejecución de los restantes hilos de un proceso, por compartir el mismo espacio de direcciones y otros atributos.

- Cada hilo compite por el uso de un espacio de memoria virtual finito con otros hilos. Si bien este rango puede ser grande (4 GB) distintos procesos disponen de todo este rango de memoria.

- Generalmente es necesario implementar mecanismos de sincronización (mutex, semáforos) para que los hilos trabajen correctamente en forma colaborativa.

Creación de Hilos

La función `pthread_create()` crea un nuevo hilo.

```
#include <pthread.h>

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

rc = pthread_create(puntero_hilo, atributo, funcion, argumento);
```

Devuelve 0 si tuvo éxito o un número positivo en caso de error.

El nuevo hilo de ejecución se inicia llamando a la función identificada por `start` con el argumento `arg`.

Si los atributos son definidos como `NULL`, tomará los atributos por defecto.

Terminación de hilos:

La ejecución de un hilo puede terminar por alguno de los siguientes motivos:

- La función `start` del hilo retorna con un valor.
- Con la llamada a la función `pthread_exit()`.
- El hilo es cancelado con la función `pthread_cancel()`.
- Cuando el hilo principal (`main`) o cualquier otro hilo del proceso termina con la llamada `exit()`, causa que todos los hilos del proceso terminen. Una llamada a `return()` del hilo `main()`, hace que todos los hilos en el proceso terminen inmediatamente.

Terminación de un hilo:

La función `pthread_exit()` finaliza la ejecución de un hilo en particular.

```
include <pthread.h>

void pthread_exit(void *retval);
```

El argumento `retval` especifica el valor de retorno para el hilo.

El valor al que apunta `retval` no debe colocarse en la pila del hilo. ¿Por qué?

Si el hilo principal (`main`) o cualquier hilo llama a `pthread_exit()` en lugar de llamar a `exit()`, los otros hilos se siguen ejecutando.

ID de hilos o TID

Cada hilo dentro de un proceso se identifica mediante un ID de hilo (TID). Un hilo puede obtener su propia identificación con `pthread_self`().

```
include <pthread.h>

pthread_t pthread_self(void);
```

Devuelve el TID del hilo que hace la llamada.

Unión con un hilo

La función `pthread_join()` espera a que el hilo identificado por `thread` termine. Si ese hilo ya ha terminado, `pthread_join()` vuelve inmediatamente. Esta operación se llama unión.

```
include <pthread.h>

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

Devuelve 0 si tiene éxito, o un número positivo en caso de error.

Si `retval` es un puntero no nulo, entonces recibe una copia del valor de retorno del hilo terminado, es decir, el valor que se especificó cuando el hilo realizó un retorno o llamado `pthread_exit()`.

Unión con un hilo

La tarea que realiza `pthread_join()` para hilos es similar a la realizada por `waitpid()` para los procesos. Con las siguientes diferencias:

- Los hilos son iguales, no tienen jerarquías. Cualquier hilo de un proceso puede usar `pthread_join()` para unirse con cualquier otro hilo en el proceso.

- Los procesos tienen una relación jerárquica. Cuando un proceso padre crea un hijo usando un `fork()`, es el único proceso que puede hacer un `wait()` para esperar la finalización de ese hijo.

Atributos de los hilos

El argumento de `attr` de la función `pthread_create()`, es de tipo `pthread_attr_t`, se puede utilizar para especificar los atributos utilizados en la creación de un nuevo hilo.

Los atributos incluyen la siguiente información:

- Localización y tamaño de la pila del hilo.
- Prioridad de los hilos.
- Si el hilo es unible (`joinable`) o no.

Atributos para unir o liberar hilos:

`PTHREAD_CREATE_JOINABLE` : el hilo es unible.
`PTHREAD_CREATE_DETACHED`: el hilo no es unible.

Liberación de hilos

Por defecto un hilo es unible, es decir que cuando termina otro hilo puede obtener su estado de retorno (y ser esperado) usando la función `phtread_join()`.

Podemos liberar al hilo con la función `pthread_detach()`, es decir el sistema limpia y remueve el hilo automáticamente cuando este termina.

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Devuelve 0 si tiene éxito, o un número positivo en caso de error.

Datos tipo const

El prototipo de la función `pthread_create()` espera un argumento del tipo `const`,

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

Se define un argumento tipo `const` cuando se desea proteger a esa variable de posibles modificaciones. Por tanto, que una variable sea `const` significa que es de **solo lectura**.

Vale aclarar que la variable `attr` no es definida como `const` cuando se la declara,

```
pthread_attr_t attr;
```

pero al definirla como `const` entre los argumentos de `pthread_create()`, se asegura que dicha función no podrá modificar la variable `attr`.

Puntero a función

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

El tercer argumento de `pthread_create()` indica que:

1. Es un puntero a una función, `void * (*start)(void *).`
2. Su argumento de entrada es un puntero a `void`, `void * (*start)(void *).`
3. Su argumento de salida es un puntero a `void`, `void * (*start)(void *).`

Que un argumento de entrada sea definido como `void` implica que en realidad podrá ser cualquier tipo de dato cuando se cree la función que luego será asignada al hilo. Sí será necesario castear a puntero `void` esta variable. Por ejemplo,

```
void *hola(void * n) { ... };

int rc, t = 0;
pthread_t hilo[1];

rc = pthread_create(&hilo[0], NULL, hola, (void *)(&t) );
```

Bibliografía

Kerrisk, Michael. *The linux programming Interface.* 2011. **Capítulo 29.**