



**INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO**

Aplicaciones para Comunicaciones en Red



Tarea P1-1 : Hilos

Profesor: Rangel Gonzales Josue

Equipo:

- Sánchez Sosa Edith Kathya
- Romero Lucero Alan

GRUPO: 3CV15

Introducción	3
Ejecución de un programa	3
Flujo de ejecución de un programa	4
Librerías de flujos	4
¿Qué sabe el SO de un programa en ejecución?	5
El proceso	5
Multiprogramación	7
Desarrollo	7
Concurrencia y Paralelismo	7
Que son los hilos	9
Diferencias entre hilos y procesos	10
Características de los hilos	11
Ciclo de vida de los hilos	13
Funciones para el manejo de hilos POSIX	14
Fuentes	15

Introduccion

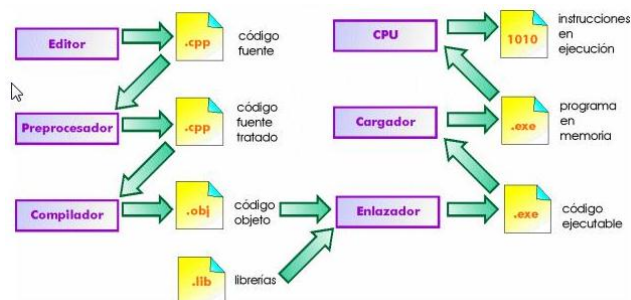
Para comprender los temas que a continuación se desglosan es indispensable aclarar ciertos conceptos primero.

Ejecución de un programa

A un programa en ejecución se le suele llamar también proceso.

El proceso de ejecución de un programa escrito en un lenguaje de programación y mediante un compilador tiene los siguientes pasos:

1. Escritura del programa fuente con un editor (programa que permite a una computadora actuar de modo similar a una máquina de escribir electrónica) y guardarlo en un dispositivo de almacenamiento.
2. Introducir el programa fuente en memoria.
3. Compilar el programa con el compilador.
4. Verificar y corregir errores de compilación.
5. Obtención del programa objeto
6. El enlazador (linker) obtiene el programa ejecutable.
7. Se ejecuta el programa y, si no existen errores, se tendrá la salida del programa.



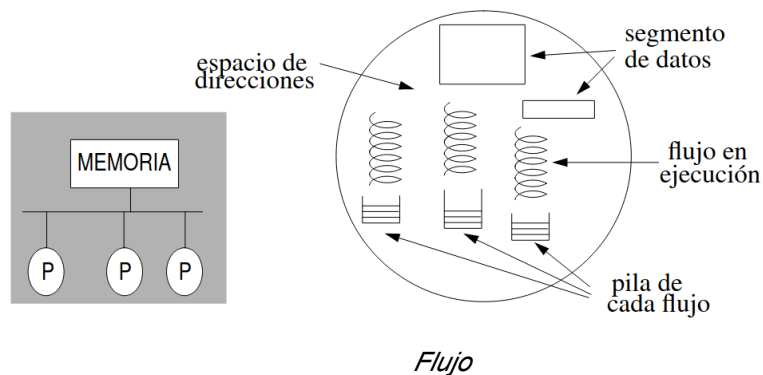
Flujo de ejecución de un programa

“Una task (tarea) es un entorno de ejecución en el cual pueden correr threads (flujos).

Como unidad básica de asignación de recursos, una task incluye un espacio de direcciones y acceso protegido a los recursos del sistema (como procesadores y memoria). La noción de proceso en UNIX se representa en Mach por una task que tiene un único thread de control.

Un thread es una unidad básica de utilización de la CPU. Es totalmente equivalente a un contador de programa independiente operando en una task. Todos los threads de una task comparten el acceso a todos los recursos de la task”.

RASHID et al., 1988, diseñadores de Mach



Librerías de flujos

- Se ofrece la posibilidad de cambio de contexto, entidades de flujo y primitivas de sincronización
- Se ofrece una interfaz de trabajo, pero se puede acceder a cualquier rutina pública.
- La planificación de flujos es totalmente transparente al SO

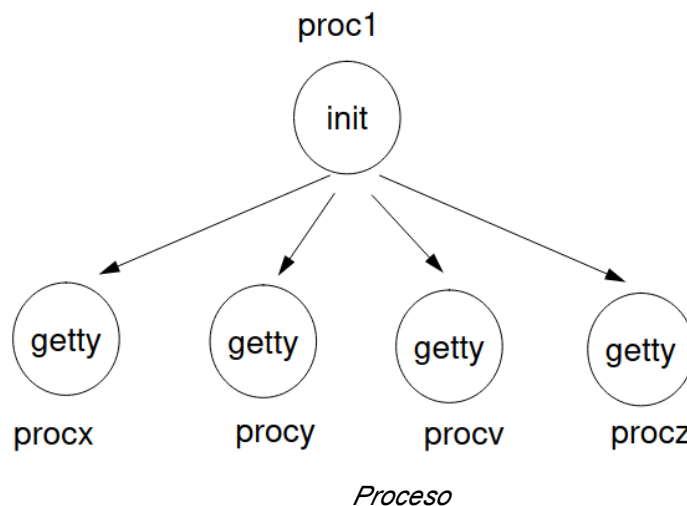
- No sabe la concurrencia que hay en la librería.
- Sólo podrá haber tanto paralelismo como “procesadores virtuales” ofrezca.
- Un bloqueo de un servicio del SO, impide la planificación de cualquier flujo.

¿Qué sabe el SO de un programa en ejecución?

- Usuario que lo ejecuta (asignación justa de recursos)
- Momento en que ha empezado la ejecución
- Tiempo que lleva ejecutándose (contabilidad de recursos)
- Lugar de memoria que ocupa
- Cantidad de memoria que ocupa
- Si hay más de un programa en ejecución
- Identificador único
- Prioridad

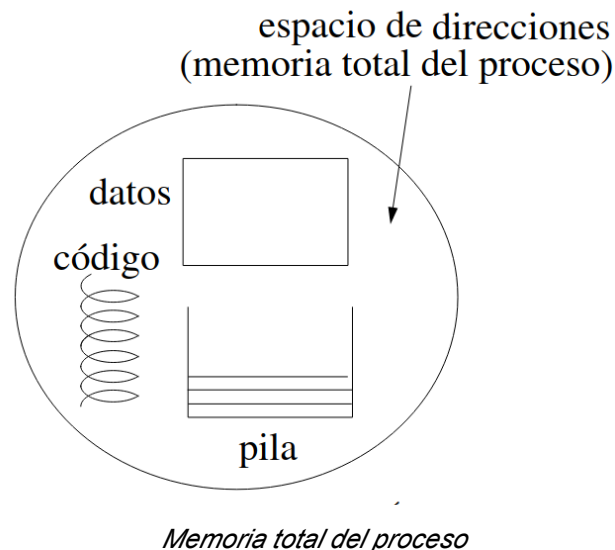
El proceso

El proceso es un objeto de sistema, a cada instancia de programa en ejecución corresponde un proceso



En el objeto proceso el SO tiene toda la información sobre el entorno de ejecución de una instancia de programa para asignarle el recurso al procesador.

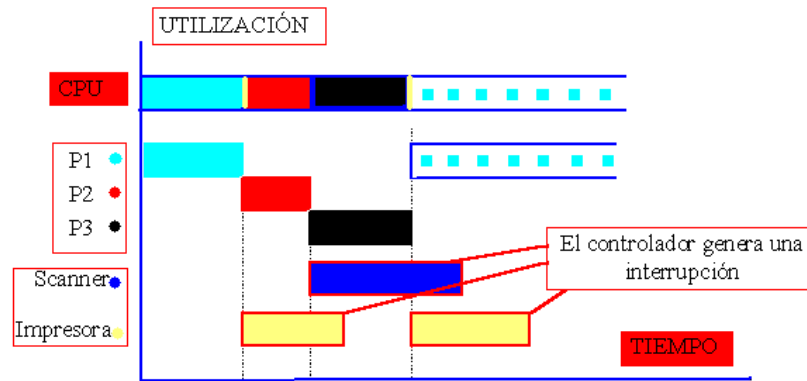
Podemos representar cada objeto proceso por el código que ejecuta, los datos que manipula, los recursos que utiliza y su pila:



Es lo que se conoce también como MÁQUINA VIRTUAL: cada programa tiene la “ilusión” de estar trabajando con una máquina “dedicada” a él: procesador, memoria, recursos, etc.

Multiprogramación

La multiprogramación es uno de los tipos más básicos de procesamiento paralelo que se puede emplear en muchos entornos diferentes. Esencialmente, hace posible que varios programas estén activos al mismo tiempo, mientras aún se ejecutan a través de un solo procesador. Su funcionalidad en este entorno implica un proceso continuo de realizar secuencialmente tareas asociadas con la función de un programa, y luego pasar a ejecutar una tarea asociada con el siguiente programa.

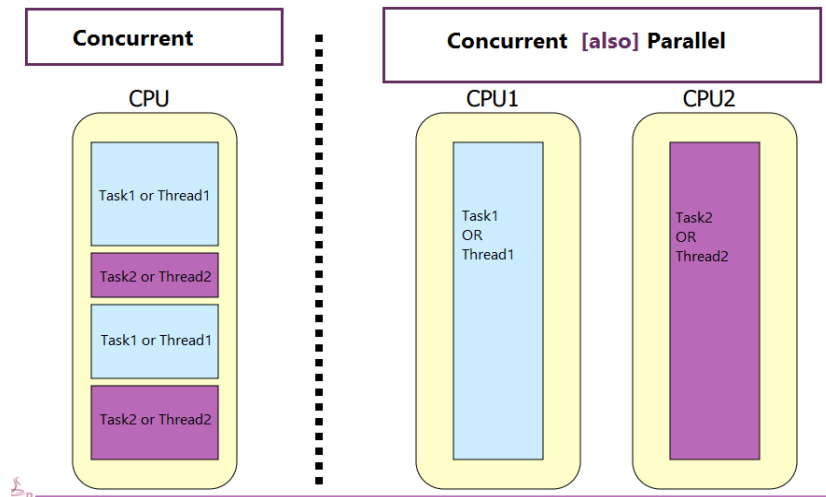


Utilización de los recursos con multiprogramación

Desarrollo

Concurrencia y Paralelismo

La concurrencia y el paralelismo conceptualmente se pueden considerar lo mismo, ya que ambas buscan el mismo objetivo, ejecutar simultáneamente dos procesos. El paralelismo nace con la necesidad de ejecutar más de un proceso a la vez, dado que antes para ejecutar procesos se tenía que hacer uno, esperar a que termine y después iniciar el siguiente proceso. La idea es simple, para ejecutar dos procesos de manera paralela se necesitan dos procesadores, y cada uno ejecuta un proceso por separado, de tal forma que cada uno terminará sin haber interrumpido al otro e incluso pueden entregar su resultado en el mismo instante en el tiempo. Esto tampoco requiere que ambos procesos sean independientes entre sí, pueden estar relacionados.



Ejemplo de concurrencia en un procesador (izquierda) y paralelismo en múltiples procesadores (derecha)

La concurrencia resuelve el principal problema del paralelismo, donde si se busca ejecutar 100 procesos, para que sean paralelos se requiere de 100 procesadores ejecutando esos procesos. Para lograr la concurrencia se carga un primer proceso a memoria y se comienza a ejecutar, el CPU lo ejecutará durante un tiempo determinado por el planificador de proceso, terminado el tiempo asignado de ejecución se toma “captura” del estado en que el proceso se encuentra en ese momento, entonces el CPU pasa a ejecutar otro proceso, acaba su tiempo, se pasa al siguiente y así se repite el procedimiento hasta que el proceso termina su ejecución. Esto, además de evitar que un proceso evite la ejecución de otro, previene que un proceso esté ocioso esperando que una operación de entrada se realice, por ejemplo, ya que puede cambiar ese proceso en el tiempo que no hará nada. Se debe aclarar que, aunque los procesos son interrumpidos, para una persona que está usando la computadora le parecerá que se ejecutan “al mismo tiempo” ya que el tiempo en que se ejecutan y el cambio entre procesos se hace muy rápido, de manera que es imperceptible para el humano.

En la actualidad, la concurrencia y el paralelismo funcionan al mismo tiempo. Se tienen procesadores con múltiples núcleos que ejecutan más de un proceso a la vez, pero en

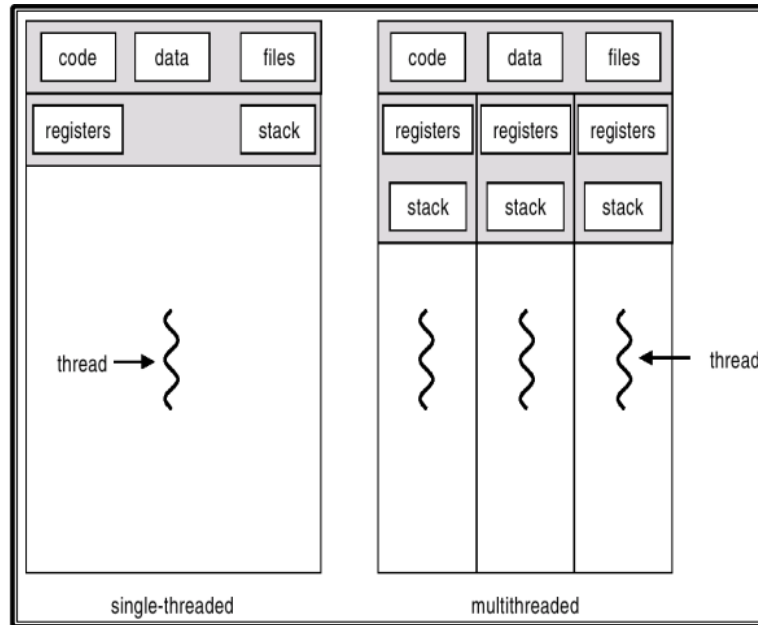
cada núcleo se ejecutan los procesos de manera concurrente, logrando así altas velocidades de procesamiento.

Que son los hilos

Un hilo, también llamado *proceso ligero*, es una unidad básica de utilización de CPU.

Un hilo nace a partir de un proceso y es llamado ligero debido a que comparten toda la memoria y el espacio de almacenamiento permanente. Es decir, pueden comunicarse eficientemente a través de la memoria que comparten. Si se necesita que un hilo comunique información a otro hilo basta que le envíe un puntero a esa información. En cambio los procesos pesados necesitan enviar toda la información a otros procesos pesados usando pipes, mensajes o archivos en disco, lo que resulta ser más costoso que enviar tan solo un puntero.

Representado a nivel del sistema operativo con un bloque de control de hilo, *TCB* por sus siglas en inglés, tiene sus propios registros, contador de programa y pila. Un proceso puede contener varios hilos de ejecución, lo que le permite realizar actividades concurrentes, además los hilos que pertenecen al mismo proceso comparten la sección de código, sección de datos, entre otras cosas.



Modelo de un proceso monohilo contra un proceso multihilo

Diferencias entre hilos y procesos

Una de las grandes diferencias entre procesos e hilos es que el primero conlleva gran cantidad de información de estados. Además, los hilos pueden comunicarse. Los procesos esto es algo más complicado (creación de estructuras de comunicación, como sockets). Los hilos comparten recursos, datos y espacios de direcciones. Los procesos NO lo hacen. El tiempo que requiere el sistema operativo para realizar un cambio de un proceso a otro es muy elevado, debido a que debe realizar un cambio de contexto, cambiar el proceso de estado de ejecución a estado de espera, copiar toda la memoria del programa y colocar el nuevo proceso en ejecución. En los hilos este tiempo es despreciable, pues todos pertenecen al mismo proceso y además comparten la memoria.

Ventajas de hilos frente a procesos: Los beneficios de los hilos radican en lo relativo al rendimiento:

- Mucho menor tiempo en crear un hilo nuevo que un proceso nuevo.
- Mucho menor tiempo en terminar un hilo que un proceso.
- Mucho menor tiempo en cambiar entre dos hilos que entre dos procesos.
- Los hilos aumentan la eficiencia de la comunicación.
- Normalmente, en la comunicación entre procesos debe intervenir el núcleo (lento, implica gran pérdida de tiempo). En cambio, entre hilos se puede producir la comunicación sin que intervenga el núcleo.

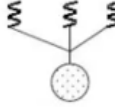
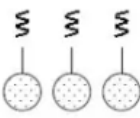
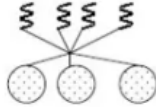
Características de los hilos

A nivel práctico los hilos son implementados a *nivel de usuario* o a *nivel de kernel*. Los hilos a nivel kernel son implementados por alguna biblioteca, por lo que el sistema operativo no es el que le da soporte; su cambio de contexto es más sencillo comparado al cambio de contexto entre hilos de kernel, debido a que supone un cambio en el modo de ejecución, y su planificación puede ser diferente a la estrategia de planificación utilizada por el sistema operativo. Por otra parte, los hilos de nivel kernel son creados, planificados y gestionados por el sistema operativo, lo que permite que aprovechen mejor las arquitecturas multiprocesador.

Los hilos de nivel kernel y usuario se pueden relacionar de, principalmente, tres maneras diferentes:

1. *Many to One*. El modelo asigna múltiples hilos de usuario a un hilo de kernel. Se corresponde a los hilos implementados a nivel usuario, ya que el sistema solo reconoce el hilo de control del proceso. Si un hilo se bloquea lo hace todo el proceso. No podrán ejecutarse hilos en paralelo ni en múltiples CPUs.

2. *One to One*. El modelo asigna cada hilo de usuario a un hilo de kernel. Tiene mayor concurrencia que el modelo *Many to One* ya que evita los bloqueos, sin embargo, está limitado debido a las restricciones en la mayoría de los sistemas operativos respecto a la cantidad de hilos de nivel kernel.
3. *Many to Many*. Este modelo multiplexa una cantidad N de hilos de usuario con una cantidad M de hilos de kernel. Cada proceso tiene asignado un conjunto de hilos de kernel independiente a la cantidad de hilos de usuario creados. No tiene los inconvenientes de los modelos anteriores, ya que puede crear tantos hilos de usuario como se requieran y ejecutarlos en paralelo en diferentes hilos de nivel kernel. Entonces, se tiene un planificador a nivel de usuario que asigna hilos de usuario a hilos de kernel y un planificador a nivel kernel que asignará los hilos de kernel a los CPUs.

HNU:HNN	Descripción	Ejemplos
M:1		Pthreads POSIX
1:1		Windows 2000, OS/2
M:N		Solaris

Diferentes modelos de relación entre hilos de nivel usuario e hilos de nivel kernel

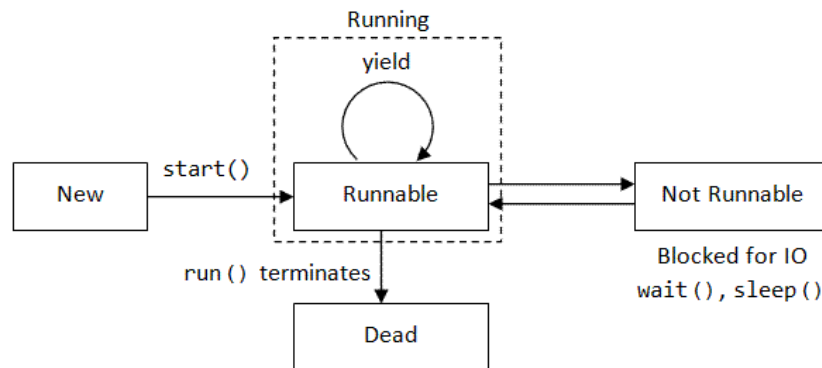
Otra característica importante de los hilos es que, cuando los hilos requieren modificar recursos compartidos, es necesario emplear diferentes métodos de sincronización para evitar colisiones entre sí.

Ciclo de vida de los hilos

Los hilos tienen un ciclo de vida, pasando por diferentes estados:

- **New:** El primer estado de un hilo recién creado. Permanece en este estado hasta que el hilo es ejecutado.
- **Runnable:** Una vez ejecutado pasa a este estado, durante el cuál ejecuta su tarea.
- **Not runnable:** Estado que permite al hilo desocupar la CPU en espera a que otro hilo termine o le notifique que puede continuar, o bien a que termine un proceso de E/S, o bien a que termine una espera provocada por la función `Thread.sleep(100);`. Tras ello volverá al estado Runnable.
- **Dead:** Pasa a este estado una vez finalizado el método `run()`.

El siguiente diagrama resume las transiciones entre los estados del ciclo de vida de un hilo.



Ciclo de vida de un proceso

Es muy importante asegurar que un hilo saldrá de la función `run()` cuando sea necesario, para que pase al estado **Dead**.

Si un hilo se encuentra en estado **Not Runnable**, no podrá hacer el retorno de su función `run()` hasta que no vuelva al estado **Runnable**.

Funciones para el manejo de hilos POSIX

```
int pthread_create(  
    pthread_t *restrict tidp,  
    const pthread_attr_t *restrict attr,  
    void *(*start_routine)(void *),  
    void *restrict arg  
);
```

Función que inicia un hilo nuevo. Como argumentos recibe: un apuntador para almacenar el id del nuevo hilo, utilizado para futuras llamadas para gestionar el hilo; los atributos del nuevo hilo, si se recibe NULL se creará con los atributos por defecto; el apuntador de la función que el hilo ejecutará, y el último argumento son los argumentos que se le pasarán al nuevo hilo. Si el hilo se crea con éxito el retorno de la función es cero, de otra forma es el número de error.

```
pthread_t pthread_self(void);
```

Función que retorna el id del hilo lo manda a llamar. No recibe argumentos.

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Función que verifica la igualdad de los hilos. Si los identificadores de hilo pasados como argumentos no son iguales la función devuelve cero.

```
void pthread_exit (void * rval_ptr);
```

Función que termina el hilo que la ejecuta. Como argumento recibe el apuntador, de cualquier tipo, del valor de retorno del hilo.

```
int pthread_join(pthread_t tid, void ** rval_ptr);
```

Función que espera el término de un hilo. Recibe como argumentos el identificador del hilo del que espera su terminación y un puntero que, en caso de no ser NULL, almacenará el valor pasado a la función *pthread_exit*. Con éxito la función retorna cero, en otro caso retorna el código de error.

```
int pthread_cancel(pthread_t tid);
```

Función que inicia una solicitud de cancelación del hilo identificado por el argumento que recibe. La cancelación del hilo depende del estado de cancelabilidad y el tipo de cancelación, los cuales pueden ser controlados por el hilo objetivo. Si ocurre un error retorna el código de error, de otra forma retorna cero.

```
int pthread_kill(pthread_t thread, int sig);
```

Función que manda una señal, por el argumento *sig*, al hilo identificado por el argumento *thread*. Con éxito, la función retorna cero y el código de error en otro caso.

Fuentes

- <https://sites.google.com/site/tecnologicodetuxtlagutierrez/3-4-ejecucion-de-un-programa>
- <http://tutoriales-isc.blogspot.com/>
- <https://people.ac.upc.edu/marisa/miso/concurrencia.pdf>
- <https://spiegato.com/es/que-es-la-multiprogramacion>

- <https://lsi.vc.ehu.eus/pablogn/docencia/manuales/SO/TemasSOuJaen/DEFINICION/CONTROLDEPROCESO/6Apendice1.MultiprogramacionyTiempoCompartido.htm>
- <https://www.oscarblancarteblog.com/2017/03/29/concurrencia-vs-paralelismo/>
- <https://www.slideshare.net/VanessaRamirez20/hilos-y-procesos-vanessa-ramirez>
- <http://www.jtech.ua.es/dadm/2011-2012/restringido/android-av/sesion01-apuntes.html>
- <https://users.dcc.uchile.cl/~jpiquer/Docencia/SO/aps/node7.html>
- <https://www.fing.edu.uy/tecnoinf/maldonado/cursos/so/material/teo/so05-hilos.pdf>
- <https://sistemasoper2.wordpress.com/2014/10/21/caracteristicas-de-los-hilos>
- http://profesores.elo.utfsm.cl/~agv/elo330/2s07/lectures/POSIX_Threads.html