

Threads

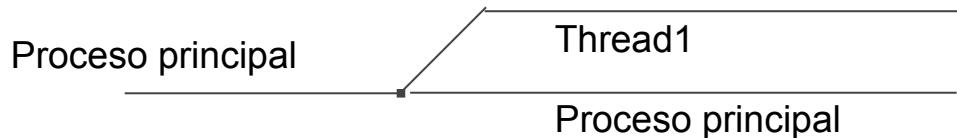
R2004 - 2021

¿Qué es un thread?: Procesos “paralelizables”

Muchas veces nos encontramos haciendo más de un procesamiento al mismo tiempo, en el marco del mismo programa.

De ser así, podemos “paralelizar” los procesos para:

- Hacer más eficiente nuestro programa desde un punto de vista del tiempo
- Ejecutar procesos en simultáneo que no son secuenciales (no necesitan de la finalización de otro proceso para comenzar a funcionar)



¿Por qué un thread y no un segundo proceso?

- Los threads son una herramienta más “liviana” que la división de procesos
 - No tienen un número de PID asignado, lo que implica que desde el punto de vista del sistema operativo solo se ejecuta un proceso, y esto lo hace más ágil.
 - No duplica las variables del proceso, por lo que un thread es más “eficiente” desde un punto de vista de ahorro de recursos.
 - Los threads aprovechan las herramientas multihilo de algunos procesadores para realizar más de una secuencia de código al mismo tiempo, por lo que es más veloz también (para muchos programas de manera imperceptible para el usuario)
- Al estar trabajando en el marco de un mismo proceso, debemos ser cuidadosos porque los distintos threads tienen acceso al mismo juego de variables (globales) que el proceso original, y por lo tanto hay mayor posibilidad de errores.

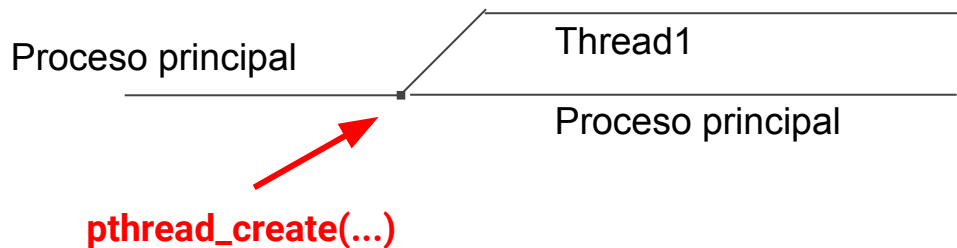


¿Para qué puedo usar threads?

Los threads están pensados para ejecutar un proceso (típicamente, una función), en paralelo con el programa principal. Algunos ejemplos de threads pueden ser:

- Operaciones matemáticas extensas
- Servidores concurrentes
- Procesamiento de señales

Así, el programa principal “divide” una tarea (que suele estar encapsulada en una función) en un thread, que a partir de su ejecución empieza a ejecutarse en forma paralela.



¿Cómo se usan?

1. En este momento "creo" dos threads (similar a fork para procesos)

```
int main () {
```

```
...  
pthread_create( ... , LecturaSocket() ,  
... );  
pthread_create( ... , LecturaUSB() , ... );
```

```
MuestraValores();
```

```
pthread_exit ( NULL );
```

```
}
```

```
void LecturaSocket(void) {
```

```
...
```

```
pthread_exit ( NULL );
```

```
}
```

```
void LecturaUSB(void) {
```

```
...
```

```
pthread_exit ( NULL );
```

```
}
```

3. Todos los hilos de ejecución, inclusive el principal, deben terminar con pthread_exit()

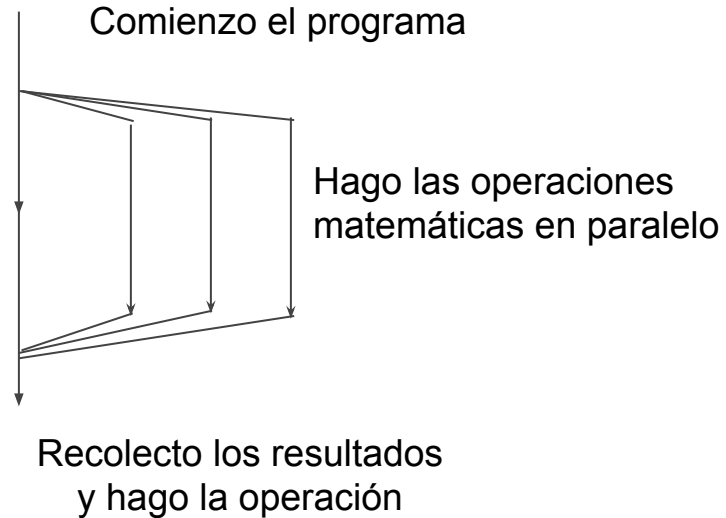
2. Cada uno de los threads realiza su tarea (en "paralelo" con el thread ppal), y cuando termina invoca a pthread_exit()

Ejemplo:

Sea $f(x) = (g(x) \times h(x)) / y(x)$

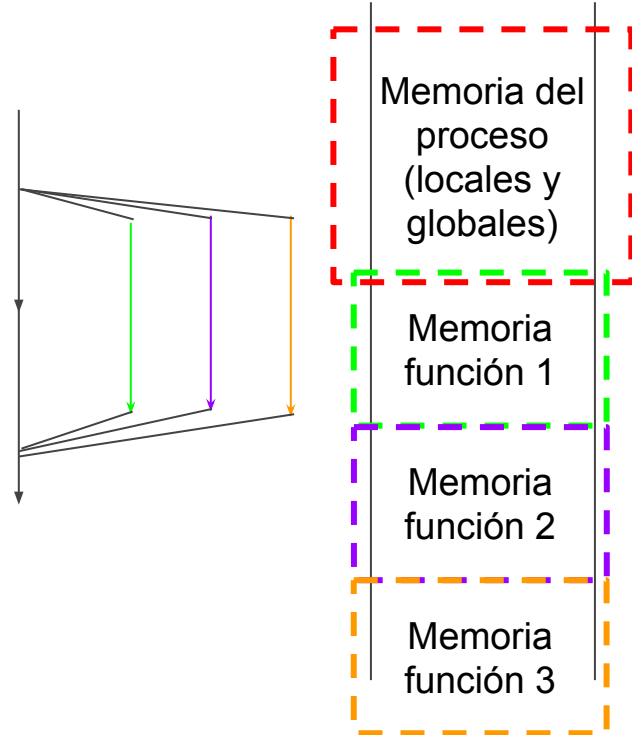
```
int main () {  
    ...  
  
    pthread_create( thread1 , FuncionG() , ...);  
    pthread_create( thread2 , FuncionH() , ...);  
    pthread_create( thread3 , FuncionY() , ...);  
  
    ... //ACA HAGO OTRAS COSAS //...  
  
    pthread_join( thread1 , FuncionG() , ...);  
    pthread_join( thread2 , FuncionH() , ...);  
    pthread_join( thread3 , FuncionY() , ...);  
    ...  
    pthread_exit ( NULL );  
}
```

pthread_join detiene la operación de un thread y se queda esperando que otro thread termine (invoque a pthread_exit)



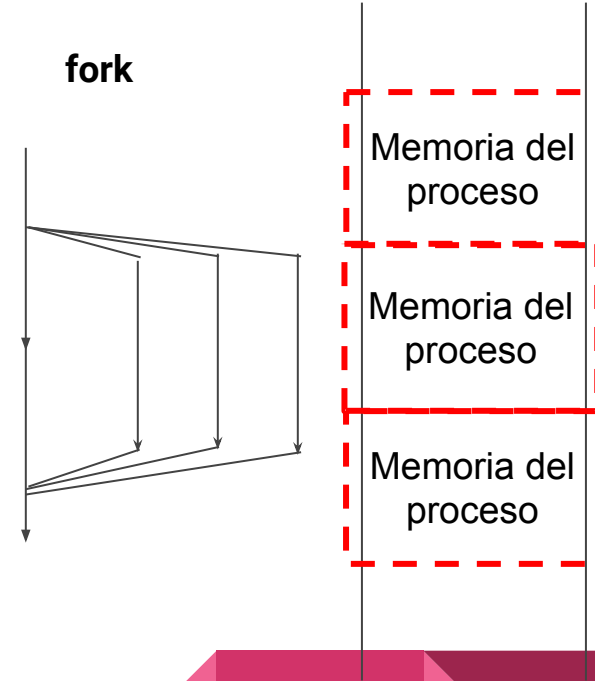
Threads: Manejo de memoria

Threads



Los threads NO DUPLICAN la memoria del proceso que los invoca, como hace fork. Al estar cada hilo “encapsulado” en una función, cada uno tiene sus variables locales, y pueden acceder a las variables compartidas (globales) del proceso

fork



Threads vs Procesos


Semejanzas: Los hilos operan, en muchos sentidos, igual que los procesos.

- Pueden estar en uno o varios estados: listo, bloqueado, en ejecución o terminado.
- También comparten la CPU.
- Cada hilo tiene su propia pila y contador de programa.
- Pueden crear sus propios hilos hijos.

Diferencias: Los hilos, a diferencia de los procesos, no son independientes entre sí.

- Como todos los hilos pueden acceder a todas las direcciones del procesos, un hilo puede leer la pila de cualquier otro hilo o escribir sobre ella.

Ventajas: de los hilos sobre los procesos.

- Se tarda mucho menos tiempo en crear y terminar un nuevo hilo en un proceso existente que en crear un nuevo proceso.
 - Se tarda mucho menos tiempo en conmutar entre hilos de un mismo proceso que entre procesos.
 - Los hilos hacen más rápida la comunicación entre procesos, ya que al compartir memoria y recursos, se pueden comunicar entre sí sin invocar al kernel del SO.
- 

Prototipos y su funcionamiento

```
#include <pthread.h>
```

Header con todos los
prototipos para usar POSIX
threads

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

Crea un nuevo thread, que
invoca la rutina
start_routine

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

Espera a que termine
thread y continua la
ejecución como un solo
thread

```
void pthread_exit(void *retval);
```

Finaliza el thread desde
donde se la llama

pthread_t: tipo de datos para almacenar el ID del thread

pthread_attr_t: se utiliza si queremos que el thread tenga un atributo en especial (sino es NULL)

Profundizando: pthread_create()

```
int pthread_create(
```

```
pthread_t *thread,
```

Los threads tienen un identificador, que nos permite saber cuando terminó en que estado se encuentra, o mandarle una “señal”. este identificador es una variable de tipo pthread_t, y se instancia con la invocación a pthread_create (debo pasarle la dirección de una variable tipo pthread_t)

```
const pthread_attr_t *attr,
```

Cada thread puede ser inicializado con “atributos” específicos, que se pueden enviar en su creación. Es una funcionalidad que no vamos a utilizar, por lo que este parámetro será NULL (se inicializa con atributos por defecto)

```
void *(*start_routine) (void *),
```

```
void *arg);
```

En caso de querer enviarle argumentos a la función, aquí paso la dirección a los argumentos

Los threads se ejecutan en el marco de una función. Aquí paso la dirección de inicio de la función, que se referencia con el nombre de la misma, y que debe recibir y devolver un void *, para que se empiece la ejecución del hilo en este punto

Profundizando: pthread_join() y pthread_exit()

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

El identificador del thread que estoy esperando a que termine

En caso de que haya un valor devuelto, lo recibo en retval

```
void pthread_exit(void *retval);
```

En caso de que quiera devolver un valor, lo devuelvo aquí

Envío de argumentos y devolución de resultados

```
#include <pthread.h>
```

```
struct args{  
    float a;  
    int b;  
};
```

```
int main ( void )  
{  
    float **resultado;  
    float valor;  
    struct args aux;  
    aux.a = 3;  
    aux.b = 5;
```

```
    pthread_t th1;
```

```
    pthread_create( &th1, NULL, Funcion , (void *) &aux );
```

```
    ...  
    pthread_join( th1 , (void **)resultado );  
    valor = **resultado;
```

```
}
```

2. Recupero el valor de los argumentos, accediendo a la dirección de la estructura

1. Envío los argumentos a la función a través de una estructura (envío la DIRECCIÓN casteada como un void * porque así me lo pide el prototipo)

```
(void *) Funcion ( void * arg ){  
    float res;  
    float val1 = (struct args *)arg->a;  
    int val2 = (struct args *)arg->b;  
    ...  
    pthread_exit( (void *) &res );
```

3. Devuelvo la DIRECCIÓN de la ubicación del valor de retorno, también casteado como un void *

4. Obtengo el valor de retorno de la función a través de un **

Ejemplo:

```
#include <stdio.h>
#include <pthread.h>

#define N 15
int a = 0;

void * hola(void *arg);

int main(int argc, char *argv[])
{
    pthread_t id[N];

    for(int i = 0; i < N; i++)
    {
        pthread_create(&id[i], NULL, hola, NULL);
    }

    for(int i = 0; i < N; i++)
    {
        pthread_join(id[i], NULL);
    }

    pthread_exit(NULL);
}

void * hola(void * arg)
{
    printf("Hola soy el thread nro: %d\n", a++);
    pthread_exit(NULL);
}
```

¿Que veremos en pantalla?



¿Cómo compartimos los recursos?

- Al estar todos los hilos compartiendo el mismo entorno de ejecución (todos los threads pueden ver las variables globales del proceso), hay que acceder en forma ordenada a los recursos compartidos.
- Si bien se puede usar una lógica de mensajes con IPCs, los threads poseen funciones propias para ordenar el acceso a los recursos compartidos, que mantienen la lógica de ser livianos, para favorecer la velocidad y el menor gasto de recursos.
- Esta señalización se hace mediante ***mutex***, cuya operatoria es similar a los semáforos de los IPCs.



Utilizando los mutex para el acceso a recursos

```
int a = 0;
```

Recurso global a ser compartido

```
pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
```

1. Genero una variable de tipo pthread_mutex_t, y le pongo un valor inicial (que indica que el recurso a proteger esta "desbloqueado")

```
pthread_mutex_lock (&mutexA);
```

2. Cuando quiero acceder al recurso, primero lo "bloqueo" (pido el acceso, y si lo obtengo, dejo bloqueado el mutex)

```
printf("Hola soy el thread nro: %d\n", a++);
```

3. Una vez bloqueado, utilizo el recurso compartido (en este caso, lo leo y luego lo modifico)

```
pthread_mutex_unlock (&mutexA);
```

4. Cuando termino de usarlo, lo desbloqueo (para que lo use otro)

```
pthread_mutex_destroy(&mutexA);
```

5. Cuando no se necesita más, se destruye el mutex

Threads - bloqueo de recursos

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                       const pthread_mutexattr_t *attr);  
  
0
```

→ Inicializa el mutex

```
pthread_mutex_t mutexVar = PTHREAD_MUTEX_INITIALIZER;
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

→ Destruye el mutex (lo elimina de la memoria)

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

→ Bloquea el mutex si no lo tiene nadie. Si alguien lo tiene bloqueado, el hilo espera hasta que quien lo tenga bloqueado lo libere

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

→ Libera el mutex

Ejemplo con mutex

```
#include <stdio.h>
#include <pthread.h>

#define N 15
int a = 0;

void * hola(void *arg);
pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;

int main(int argc, char *argv[])
{
    pthread_t id[N];

    for(int i = 0; i < N; i++)
    {
        pthread_create(id + i, NULL, hola, NULL);
    }

    for(int i = 0; i < N; i++)
    {
        pthread_join( id[i], NULL);
    }

    pthread_mutex_destroy(&mutexA);
    pthread_exit(NULL);
}
```

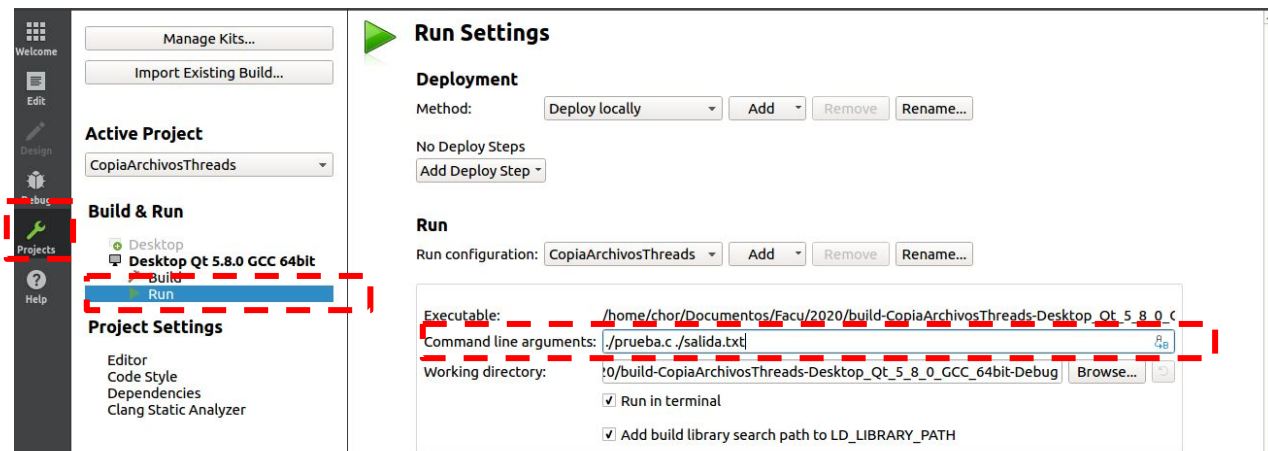
```
~
void * hola(void * arg)
{
    pthread_mutex_lock (&mutexA);
    printf("Hola soy el thread nro: %d\n", a++);
    pthread_mutex_unlock (&mutexA);

    pthread_exit(NULL);
}
```

Ejemplo2: CopiaArchivosThreads.zip

Este ejemplo toma 1 archivo que se envía por línea de comandos y genera un thread que copia su contenido en un segundo archivo, que también se envía por línea de comandos.

Para agregar parámetros en el Qt, tengo que entrar a la pantalla Project, en la sección Run, y escribir los parámetros. Recordemos que el punto de ejecución del programa (donde se crea el ejecutable) es en la carpeta build-NOMBRE_PROYECTO-Desktop-Qt



Ejercicio

Modificar el ejemplo `CopiaArchivosThreads` para que el thread le transmita por medio de variables globales al hilo principal los datos del tamaño del archivo y la cantidad de bytes copiados. El hilo principal será el encargado de mostrar estos datos por pantalla y no el hilo disparado. NO olvide sincronizar el acceso a las variables.

