




Hoja de Ejercicios 5. Threads

Ejercicios Prácticos


Los ejercicios marcados con el icono  son prácticos y deben realizarse en el laboratorio. Los ejercicios prácticos de esta hoja requieren el entorno de usuario básico (shell) y de desarrollo (compilador, editores y depurador). Además algunos ejercicios necesitan acceso de superusuario que está disponible en las máquinas virtuales de la asignatura.

 **Ejercicio 1.** Escribir un programa que cree un número de threads indicado por el primer argumento, de forma que:

- Cada thread se le asignará un identificador 0,1,2... que imprimirá por la salida estándar y usará para hacer un sleep(3) de los mismos segundos.
- El thread principal esperará a que terminen todos los threads, mostrando el identificador del thread que termina.


 **Ejercicio 2.** Modificar el programa anterior para que todos los threads esperen el tiempo suficiente para consultar sus identificadores en el sistema. Ejecutar el programa con 4 threads y completar la siguiente tabla con el comando ps, usar las opciones -L (mostrar threads) y la opción de formato -o con los campos adecuados. Identificar el thread principal en la tabla.

| PID | TID | TGID | PGID | CMD |
|-----|-----|------|------|-----|
| | | | | |
| | | | | |
| | | | | |
| | | | | |

 **Ejercicio 3.** Escribir un programa que realice la suma paralela de los N primeros números naturales. Los argumentos del programa fijarán:

- Primer argumento, el número de threads (N_t).
- Segundo argumento, el tamaño de bloque (T_b) que determina cuántos números sumará cada thread.

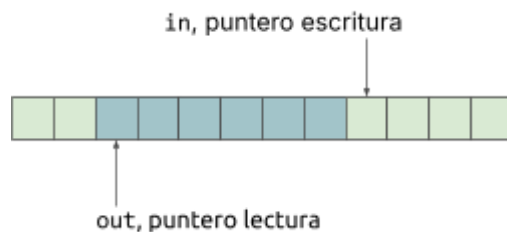
Cada thread se le asignará un identificador ($i = 0,1,2,\dots$) y sumará los enteros en el rango $[T_b \cdot i - (i+1) \cdot T_b - 1]$ que agregará en una variable compartida, suma. El thread principal sincronizará todos los threads y mostrará la suma.

 **Ejercicio 4.** Escribir un programa con P threads productores (primer argumento) y C threads consumidores (segundo argumento) con las siguientes características:

- El productor escribirá en el buffer el elemento producido y el consumidor simplemente lo

imprimirá en el terminal junto con las estadísticas del buffer (in, out y count). Los tiempos de producción y consumo serán de 1 y 2 segundos respectivamente.


- El buffer compartido será un array circular de tamaño fijo que estará representado por una estructura similar a la siguiente:




```
typedef struct _buffer
{
    int count;
    int in;
    int out;

    int data[BUFFER_SIZE];
} buffer_t;
```

- Implementar el sistema con dos variables de condición que estarán asociadas a los siguientes predicados:
 - Thread consumidor: *"puedo consumir"*, $\text{count} > 0$
 - Thread productor: *"puedo producir"*, $\text{count} < \text{BUFFER_SIZE}$
- Cada productor producirá un número fijo de elementos (NUM_ELEMENTS).

 **Ejercicio 5.** Añadir una condición de finalización al ejercicio 4 usando el **patrón centinela o píldora envenenada**, de la siguiente forma:

- El thread principal sincronizará la ejecución de todos los productores para asegurarse de que todos los elementos se han producido.
- A continuación escribirá los elementos (píldoras envenenadas, por ejemplo valor -1) en el buffer, tantos como consumidores se hayan creado. Nota: Debe preservar el acceso concurrente al búfer, igual que el thread productor.
- Cuando un consumidor lea la píldora envenenada del búfer terminará **después** de ajustar los índices y señalar la variable de condición.

 **Ejercicio 6.** Escribir un programa que cree L lectores (primer argumento) y E escritores (segundo argumento), de forma que:

- Los threads de tipo *Lector* imprimirán por pantalla un entero compartido y esperarán 0.1s con la llamada `usleep(3)`. Este acceso lo repetirán 5 veces.
- Los threads de tipo *Escritor* incrementarán en 1 la variable y esperarán 0.25s. Este acceso lo repetirán 3 veces.
- El thread principal arrancará primero los thread de tipo *Escritor* y sincronizará la finalización de todos los threads lectores y escritores.

Ejercicio 7. Diseña una implementación de los mutex de lectura/escritura de la biblioteca Posix Threads (`pthread_rwlock_[rd|wr][lock|unlock]`) usando mutex normales y variables de condición. La implementación debe dar preferencia a los escritores.

 **Ejercicio 8.** Utilizando como base el programa del ejercicio 4, diseñar e implementar una

aplicación concurrente en C que explore un directorio del sistema de ficheros y compute el **tamaño total** de los **ficheros regulares** encontrado.

La aplicación aceptará dos argumentos, el primero será el directorio del que se quiere explorar; y el segundo será el número de threads Consumidores. Los threads de la aplicación implementarán la siguiente lógica:

Thread productor

- Abre el directorio indicado y recorre sus entradas usando `opendir()` y `readdir()`.
- Inserta las **rutras completas** de los ficheros o subdirectorios en un búfer circular compartido (usar `snprintf()`)

Threads consumidores

- Extraen rutas del búfer. Además la llamada al sistema para calcular el tamaño no debe mantener bloqueado el búfer compartido, así que el thread copiará la ruta en una variable local con `strcpy()`.
- Una vez realizada la copia, actualizados los índices del búfer y señalizada la variable de condición usará `lstat()` para obtener información del fichero. Si es un fichero regular acumulará su tamaño en una variable global protegida por otro mutex.
- Si el elemento es una "píldora venenosa" (cadena vacía) finalizará su ejecución.

Thread principal

- Creará el thread productor y los threads consumidores.
- Cuando el productor termina de recorrer el directorio (`pthread_join`), inserta una píldora venenosa por cada consumidor para indicar el fin del trabajo.
- Finalmente espera (`pthread_join`) la terminación de todos los consumidores y muestra el tamaño total acumulado de los ficheros procesados.

Comprobar el funcionamiento del programa con diferentes niveles de concurrencia y la orden `time(1)` para evaluar el tiempo de ejecución en cada caso. Se pueden utilizar estructuras similares a las siguientes para gestionar los datos del programa

```
#define BUFFER_SIZE 10

typedef struct _buffer_t
{
    pthread_mutex_t mutex;

    pthread_cond_t  produce;
    pthread_cond_t  consume;

    int count;
    int in;
    int out;
}
```

```
typedef struct _lint_mutex
{
    pthread_mutex_t mutex;
    long            total_size;
} lint_mutex_t;
```

| | |
|--|--|
| <pre>char data[BUFFER_SIZE][PATH_MAX]; } buffer_t;</pre> | |
|--|--|