Universidad Rey Juan Carlos

# **Práctica:** Concurrencia - Lectores/Escritores

Sistema distribuidos y concurrentes

v.0.5.1

En esta tercera práctica pondremos en práctica los conocimientos adquiridos en clase de teoría sobre concurrencia, haciendo uso de primitivas como semáforos, variables condición, mutex. Desarrollaremos un sistema de lectores y escritores concurrentes que modifican un contador. Existirán un número de clientes (lectores o escritores) que se conectarán a un servidor para leer o modificar un contador. Recuerda que:

- Solo 1 escritor puede estar en la región crítica.
- Mientras se hagan escrituras nadie más puede entrar en la región crítica
- Los lectores pueden y **deben** entrar concurrentemente a la región crítica.

**Paso de Mensajes.**

Para esta práctica todos los mensajes intercambiados mediante sockets entre los procesos utilizarán obligatoriamente la siguiente estructura:

```
enum operations {
      WRITE = 0,
      READ
};

struct request {
   enum operations    action;
   unsigned int       id;
};

struct response {
   enum operations    action;
   unsigned int  counter;
   long  latency_time;
};
```

NOTA: No es posible cambiar, o modificar estas estructuras.

El campo ***action*** podrá contener valores del enumerado `operations`:

- WRITE: el cliente notifica que quiere escribir.

- READ: el cliente notifica que quiere leer.

El campo **id** define el id del cliente que realiza la petición.

El campo ***counter*** es utilizado para recibir el valor del contador.
El campo ***latency_time*** es utilizado para recibir el número de nanosegundos que ha esperado un cliente para acceder a la región crítica en el servidor (lectura o escritura).

# Práctica

Desarrolla 2 procesos independientes (cliente y servidor), con las siguientes características.

## Cliente

Un cliente puede comportarse como lector, si recibe como parámetro “--mode” “reader”, o escritor si recibe como parámetro “--mode” “writer”. Además de lo anterior recibirá un número entero que indicará el número de lectores/escritores concurrentes que creará el cliente.

**Ejemplo ejecución cliente:**

```
./client --ip IP --port PORT --mode writer/reader --threads 100
```

Esto lanzará 100 threads escritores concurrentes (haz pruebas con más valores). Lo que quiere decir que esos threads NO ejecutan secuencialmente.

**Comportamiento escritor:**
Cada escritor mandará un mensaje al servidor indicando que quiere escribir y recibirá como respuesta un mensaje con la operación realizada, el contador y el tiempo de espera en el servidor.

**Comportamiento lector:**
Cada lector mandará un mensaje al servidor indicando que quiere leer y recibirá como respuesta un mensaje con la operación realizada, el contador y el tiempo de espera (en nanosegundos) en el servidor.

Cada thread lanzado y ejecutado por el cliente debe mostrar el siguiente texto al recibir la respuesta del servidor, donde N es un número entre 0 y el parámetro `--threads`que identificará a cada thread:

```
[Cliente #N] Lector/Escritor, contador=X, tiempo=Y ns.
```

## Servidor:

El servidor puede configurarse con prioridad de lectores o de escritores, según indiquemos en el parámetro “--priority”.

**Ejemplo ejecución servidor:**

```
./server --port PORT --priority writer/reader
```

El servidor debe escuchar siempre en todas las interfaces disponibles en la máquina donde ejecuta. El servidor debe implementar un procesamiento de clientes multi-hilo y permitir ejecución paralela (NO secuencial). El servidor nunca debe ejecutar más de **600 threads a la vez**. Si llegan más peticiones debe ser capaz de encolarlas y atenderlas. Según vayan terminando los threads de ejecutar, el servidor debe ser capaz de lanzar nuevos threads. En ningún caso, el servidor debe rechazar conexión de los clientes.

El servidor estará esperando los mensajes de los escritores/lectores.

- Si recibe un mensaje de un escritor aumentará un contador en 1 y escribirá el valor del contador en el fichero `server_output.txt`. Siempre que se modifica el contador el servidor escribirá por la salida estándar:

```
[SECONDS.MICRO][ESCRITOR #N] modifica contador con valor X
```

- Si se recibe el mensaje de un lector el contador no se modifica y se debe mostrar la siguiente traza.

```
[SECONDS.MICRO][LECTOR #N] lee contador con valor X
```

- Asegúrate de imprimir las trazas anteriores dentro de la región crítica.
- SECONDS.MICRO: hace referencia al tiempo absoluto en ese formato
- Tras cada lectura o escritura/actualización del contador, el servidor debe realizar un sleep random dentro del rango de 75-150 ms. Este sleep debe estar dentro de la región crítica.
- Si el fichero `server_output.txt` existe, el servidor debe leer el valor y comenzar a operar con ese contador.
- Además de lo anterior el servidor debe calcular la latencia de espera de cada cliente (escritor o lector) en el acceso a la región crítica.

La latencia de espera se define como el tiempo que pasa desde que un thread intenta entrar en la región crítica hasta el instante exacto que ejecuta la primera línea de la sección crítica.
El servidor responderá al cliente con un mensaje indicando el tipo de operación que realizó, el contador y la latencia de espera. Además deberá respetar la prioridad en que ha sido configurado: prioridad de lectores o de escritores.

Prioridad de lectores: Se atiende siempre a los lectores antes que a cualquier escritor. Lectores deben pasar concurrentemente, no secuencialmente.
Prioridad de escritores: Se atiende siempre a los escritores antes que a cualquier lector. Escritores deben pasar de 1 en 1.

**Consideraciones**

- Para programar los argumento de tu programa consulta la página de manual de *getopt* y *getopt_long* o el siguiente enlace: https://linux.die.net/man/3/getopt_long
- Pueden ser usadas todas las primitivas de concurrencia (semáforos, mutex y variables condición) que se necesiten y establecerlas en el punto que se considere oportuno para garantizar el correcto funcionamiento del sistema.
- Los clientes pueden atenderse concurrentemente hasta un máximo de 600. Esto quiere decir que el servidor, nunca tendrá más de 600 threads creados. Pero debe ser capaz de atender todas las peticiones que se manden.
- Se recomienda lanzar varios escritores y varios lectores, lanzar escritores/lectores en diferentes momentos, etc, para comprobar que el sistema funciona correctamente.
- Toda la lógica de comunicaciones y mensajes debe quedar encapsulada en stubs, como ya se hizo en la práctica 2.
- más conveniente (bloqueante, no-bloqueante).Puedes utilizar pthreads y sockets con la configuración que creas
- La práctica debe compilar y ejecutar en los ordenadores del laboratorio.
- Únicamente deben aparecer las trazas descritas en el enunciado.
- Los procesos deben llamarse “server” y “client”
- Ejecuta cada proceso en una máquina física distinta.
- NO ESTÁ PERMITIDO espera activa ni condiciones de carrera en ningún caso