Thread Pools

Ejercicio N° 2

Objetivos	● Diseño y construcción de sistemas orientados a objetos ● Diseño y construcción de sistemas con procesamiento concurrente ● Encapsulación de Threads y Sockets en clases ● Protección de los recursos compartidos
Instancias de Entrega	**Entrega 1:** clase 6 (16/04/2019). **Entrega 2:** clase 8 (30/04/2019).
Temas de Repaso	● Threads en C++11 ● Clases en C++11
Criterios de Evaluación	● Criterios de ejercicios anteriores ● Orientación a objetos del sistema ● Empleo de estructuras comunes C++ (string, fstreams, etc) en reemplazo de su contrapartida en C (char*, FILE*, etc) ● Uso de **const** en la definición de métodos y parámetros ● Empleo de constructores y destructores de forma simétrica ● Buen uso del stack para construcción de objetos automáticos ● Ausencia de condiciones de carrera e interbloqueo en el acceso a recursos ● Buen uso de Mutex, Condition Variables y Monitores para el acceso a recursos compartido

Índice

Introducción

Thread Pool

Cuando tenemos que realizar muchas tareas que no comparten recursos entre sí, muy usualmente surge la idea de realizarlas en paralelo.

Un esquema simple para implementar esto consiste en crear (*spawn*) un thread para cada tarea, y de esta forma delegar en el scheduler de nuestro sistema la responsabilidad de decidir qué tarea avanzar en cada ciclo de procesamiento.

Sin embargo, este esquema tiene múltiples inconvenientes, entre los que se destacan:

- La posibilidad de requerir más threads de los que nos permite el sistema (o más habitualmente el framework sobre el cual programemos), trayéndonos problemas difíciles de depurar.
- Que un thread conceptualmente es una secuencia independiente de ejecución, pero en la práctica implica reservar recursos que pueden ser limitados.
- La imposibilidad de determinar prioridades entre las tareas (o que esto dependa del sistema en el cual se ejecute nuestro software).
- El lanzar un thread y el finalizarlo son operaciones costosas.

Teniendo en cuenta estos inconvenientes surge una propuesta que nos permite determinar la cantidad de hilos que utilizará nuestro sistema. Esta propuesta consiste, por un lado, en **utilizar un conjunto fijo de threads (un thread pool**) que permanecen dormidos esperando a que haya alguna tarea para realizar; y por otro lado en **separar el trabajo en tareas** idealmente cortas y que compartan la menor cantidad de recursos posible.

A partir de este esquema, los hilos trabajan tomando tareas de algún contenedor, y ejecutándolas. Se debe notar que ya no se instancian hilos cada vez que se desea ejecutar una tarea, sino que los mismos se reutilizan a lo largo de la ejecución del programa.

Brainfuck

Brainfuck es un lenguaje de programación que no se destaca precisamente por su facilidad. Sin embargo, tiene operaciones sumamente simples.

Los programas empiezan con una **memoria de datos** de lecto-escritura inicializada en ceros y un **puntero de datos** que apunta a su primera posición. Y además, con una **memoria de código** de sólo lectura y su respectivo **puntero de código**, que apunta a su primera posición.

Teniendo en cuenta estos elementos, se definen seis operaciones:

Operación	Descripción
<	Mover el puntero de datos a la izquierda
>	Mover el puntero de datos a la derecha
+	Incrementar el dato apuntado por el puntero de datos en 1 unidad
-	Decrementar el dato apuntado por el puntero de datos en 1 unidad
.	Imprimir el caracter apuntado por el puntero de datos
,	Leer un dato de la entrada y colocarlo en la posición apuntada por el puntero de datos
[	Si el dato apuntado por el puntero de datos es igual a cero, salta hasta la instrucción siguiente al siguiente **]**
]	Vuelve al **[** anterior

(**Notar** que el código que queda entre corchetes funciona como un bucle cuya condición se obtiene del dato apuntado por el puntero de datos)

Descripción

El ejercicio consistirá en implementar un thread pool en C++ que ejecute tareas cortas, que serán scripts del lenguaje *Brainfuck*.

El programa tendrá dos modos de funcionamiento:

1) **Intérprete**: Será simplemente un intérprete de *brainfuck*, con un único thread, que deberá ejecutar scripts del mencionado lenguaje. **Recomendamos fuertemente trabajar en este modo durante la primera semana.**

2) **Thread-pool**: Los scripts a ejecutar serán ingresados a nuestro programa por entrada estándar, indicando su prioridad, y deberán ser agregados a una cola de prioridad bloqueante, para que los threads del pool los ejecuten tomando siempre el más prioritario.

Formato de Línea de Comandos

El formato de la línea de comandos deberá indicar el modo de funcionamiento, y el script a ejecutar en modo intérprete, o bien el número de threads en modo thread-pool:

```
./tp modo [script] [numero_de_threads]
```

Entonces, si queremos ejecutar el programa en modo intérprete, usaremos el comando:

```
./tp interprete un_script.bf
```

En este caso, cuando el script deba leer (instrucción '**,**'), lo hará desde la entrada estándar del proceso; y cuando deba escribir (instrucción '**.**'), lo hará en la salida estándar del mismo.

Mientras que, si queremos ejecutar el programa en modo thread-pool, usaremos:

```
./tp thread-pool 4
```

En el caso de los thread-pool, cada script (ingresado por entrada estándar) deberá indicar, además de su prioridad, qué archivo utilizará para leer y qué archivo para escribir. Para más información sobre cómo se deben ingresar estos datos, ver las secciones siguientes. La cantidad de threads en el pool es un número fijo que se recibe como parámetro.

Códigos de Retorno

El programa retornará 1 si encuentra algún error sintáctico en algún script, 2 si hay algún error en la línea de comandos (por ejemplo, si se le pasa un script al modo de funcionamiento thread-pool). En caso de una ejecución exitosa se deberá retornar un 0.

Entrada y Salida Estándar

Entrada Estándar

En **modo intérprete**, la entrada estándar se utilizará como el archivo de entrada del script a ejecutar.

En **modo thread-pool**, se utilizará para ingresar todos los scripts a ejecutar junto con su metadata (nombre, prioridad y archivos de entrada y salida):

```
(nombre_del_script, prioridad, archivo_entrada, archivo_salida, codigo_brainfuck)
(nombre_de_otro_script, prioridad, archivo_entrada, archivo_salida, codigo_brainfuck)
...
(nombre, prioridad, archivo_entrada, archivo_salida, codigo_brainfuck)
```

Por ejemplo:

```
(hello_world, 3, input.bf, output.txt,
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.
+++.------.--------.>+.>.)
```

Y si quisiéramos ejecutar dos tareas:

```
(hello_world_1, 3, input_1.bf, output_1.txt,
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.
+++.------.--------.>+.>.)
(hello_world_2, 2, input_2.bf, output_2.txt,
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.
+++.------.--------.>+.>.)
```

Nota: El modo thread-pool se ejecuta hasta el end-of-file de su entrada estándar.

Salida Estándar

En **modo intérprete**, la salida estándar se utilizará como el archivo de salida del script a ejecutar.

En **modo thread-pool** no se utilizará la salida estándar.

Salida Estándar de Errores

No se harán chequeos en SERCOM acerca de la salida estándar de errores, el estudiante puede utilizar este flujo para imprimir sus propios mensajes de error descriptivos.

Ejemplos de Ejecución

A continuación veremos algunos ejemplos de ejecución.

Modo intérprete y Hola Mundo

Para usar el modo intérprete necesitamos un script, por ejemplo nuestro archivo **hello_world.bf**:

```
[hello_world.bf]
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.
+++.------.--------.>+.>.
```

Haciendo un seguimiento por partes de este primer ejemplo vemos lo siguiente:

```
[hello_world.bf]
```

La primera línea de este script tiene el nombre del archivo dentro de un bucle que se ejecutará **cero** veces, porque la memoria está inicializada con **ceros**. Este es un truco que se usa para añadir comentarios en *Brainfuck*, y por ende no debemos considerar errores sintácticos los caracteres que no son propios del lenguaje, sino que simplemente debemos ignorarlos.

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]
```

Aquí vemos una suerte de *for de diez iteraciones* que inicializa cuatro posiciones de memoria, quedando las primeras 5 posiciones de la misma (en base decimal) en `[ 000 | 070 | 100 | 030 | 010 ]`. Observar que la primera posición es la que usó como contador del falso *for*. Y las siguientes son 70 (cerca de las mayúsculas en ASCII), 100 (cerca de las minúsculas), 30 (cerca del espacio), y 10 (cerca del signo de exclamación).

Dicho esto, es fácil entender el resto del programa:

```
>++.                            Se mueve hasta las mayúsculas, imprime la H
>+.                             Se mueve hasta las minúsculas, imprime la e
+++++++..                       Se queda en las minúsculas, imprime dos veces la l
+++.                            Se queda en las minúsculas, imprime la o
>++.                            Imprime el espacio
<<+++++++++++++++.>.+++.        Wor
------.--------.>+.>.           ld!
```

Por lo tanto, la salida estándar de nuestro intérprete deberá mostrar:

```
Hello World!
```

Para ejecutar el intérprete con ese script, debemos escribir el siguiente comando:

```
./tp interprete hello_world.bf
```

Contador de Palabras en modo Intérprete

Este segundo ejemplo, el script **word_counter.bf** cuenta la cantidad de líneas (de caracteres '`\n`'), palabras, y bytes en la entrada.

```
>>>+>>>>>+>>+>>+[<<],[
    -[-[-[-[-[-[-[-[<+>-[>+<-[>-<-[-[-[<++[<++++++>-]<
        [>>[-<]<[>]<-]>>[<+>-[<->[-]]]]]]]]]]]]]]]]]]]]
    <[-<<[-]+>]<<[>>>>>>+<<<<<<-]>[>]>>>>>>>+>[
        <+[
            >+++++++++<-[>-<-]++>[<+++++++>-[<->-]+[+>>>>>>]]
            <[>+<-]>[>>>>>++>[-]]+<
        ]>[-<<<<<<]>>>>
    ],
]+<++>>>[[+++++>>>>>>]<+>+[[<+++++++++>-]<.<<<<<]>>>>>>>>]
```

Por lo que, si ejecutamos el script, con la línea:

```
./tp interprete word_counter.bf
```

Y en la entrada estándar escribimos:

```
Hello World!
```

En la salida estándar deberemos ver:

```
      0     2     12
```

Nota: el separador entre los números es un TAB.

Ambos ejemplos en modo thread-pool

Para ejecutar ambos códigos en modo thread-pool, debemos usar el comando:

```
./tp thread-pool 4
```

En este caso no le pasamos un script a interpretar en un archivo, sino que le podemos enviar varios por entrada estándar:

```
(hello_world, 1, empty.txt, hello_world_out.txt,
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.
+++.------.--------.>+.>.)
(word_counter, 2, wc_in.txt, wc_out.txt, >>>+>>>>>+>>+>>+[<<],[
    -[-[-[-[-[-[-[-[<+>-[>+<-[>-<-[-[-[<++[<++++++>-]<
        [>>[-<]<[>]<-]>>[<+>-[<->[-]]]]]]]]]]]]]]]]
    <[-<<[-]+>]<<[>>>>>>+<<<<<<-]>[>]>>>>>>>>+>[
        <+[
            >++++++++++<-[>-<-]++>[<+++++++>-[<->-]+[+>>>>>>]]
            <[>+<-]>[>>>>>++>[-]]+<
        ]>[-<<<<<<]>>>>
    ],
]+<++>>>[[+++++>>>>>>]<+>+[[<+++++++++>-]<.<<<<<]>>>>>>>>])
```

A medida que las tareas se van ingresando al sistema, los threads del pool los van a ir ejecutando tomando siempre el de mayor prioridad en el momento.
En este caso, se va a crear un archivo “hello_world.txt” que contendrá el string “Hello World!”, y otro archivo “wc_out.txt” que tendrá la información correspondiente respecto al contenido del archivo “wc_in.txt”.

Restricciones y Ayudas

La siguiente es una lista de restricciones técnicas exigidas por el cliente:

1. El sistema debe desarrollarse en ISO C++11.
2. Está prohibido el uso de variables globales.
3. Cada thread del pool debe elegir la tarea con la prioridad más alta que esté disponible a la vez que por entrada estándar se están agregando nuevas tareas, por lo cual se debe usar algún mecanismo de sincronización adecuado.
4. Las condiciones de carrera son descalificadoras.

Y a continuación van algunas sugerencias y ayudas para hacer el ejercicio más fácil:

1. El tiempo para implementar el ejercicio es de dos semanas, y a la mitad de ese período está planificada la clase sobre threading, pero el ejercicio está pensado para hacerlo en dos partes. Es altamente recomendable implementar el intérprete íntegro antes de usar ningún thread, durante la primera semana, y hacer muchas pruebas en modo intérprete. De esta manera, si algo falla al agregar threads tenemos cierto grado de seguridad de que no es un problema del intérprete.
2. (o 1 bis): No esperar a la segunda semana para empezar el intérprete. Una segunda semana en la que solamente tenemos que paralelizar algo que ya está funcionando debería ser fácil, mientras que una semana implementando las dos partes no lo será.
3. Prestar especial atención a los ejemplos de código de **condition variables** de la clase de threading.
4. En las referencias hay dos herramientas de la **STL de C++** ([1] y [2]) que combinadas serán de gran ayuda para implementar los mecanismos de sincronización requeridos.

Referencias

[1] Cola de prioridad en C++: http://www.cplusplus.com/reference/queue/priority_queue

[2] Condition Variables en C++: http://www.cplusplus.com/reference/condition_variable/condition_variable

[3] Lenguaje Brainfuck: https://es.wikipedia.org/wiki/Brainfuck

[4] Un intérprete para hacer pruebas (también genera código JavaScript): https://copy.sh/brainfuck/