Web Crawler

Ejercicio N° 2

Objetivos	● Diseño y construcción de sistemas orientados a objetos multithreading. ● Encapsulación en objetos RAII, movibles y no-copiables (salvo excepciones). ● Protección de los recursos compartidos en forma de monitores.
Instancias de Entrega	**Entrega 1:** clase 6 (18/05/2021). **Entrega 2:** clase 8 (01/06/2021).
Criterios de Evaluación	● Criterios de ejercicios anteriores ● Correcto uso de mecanismos de sincronización como **mutex**, **conditional variables** y colas bloqueantes (**queues**). Protección de los objetos compartidos en objetos **monitor**. ● Prohibido el uso de funciones de tipo *sleep()* como *hack* para sincronizar salvo **expresa** autorización en el enunciado. ● Ausencia de condiciones de carrera (race condition) e interbloqueo en el acceso a recursos (deadlocks y livelocks). ● Correcta encapsulación en objetos **RAII** de C++ con sus respectivos constructores y destructores, **movibles (move semantics) y no-copiables** (salvo excepciones justificadas y documentadas). Uso de la librería estándar de C++ (no reinventar la rueda). ● **Pasaje por referencia o por movimiento**. El pasaje por copia no está permitido salvo **justificación y documentación**. En otras palabras, la copia debería ser *excepcional*. ● Uso de **const** en la definición de métodos y parámetros. ● Buen uso del stack para construcción de objetos automáticos.

El trabajo es personal*: debe ser de autoría completamente tuya. Cualquier forma de* ***plagio es inaceptable****: copia de otros trabajos, copias de ejemplos de internet o copias de tus trabajos anteriores (self-plagiarism).*
Si usas material de la cátedra deberás dejar en claro la fuente y dar crédito al autor (a la materia).

Índice

[Introducción](#)

[Descripción](#)

[Formato de Línea de Comandos](#)

[Códigos de Retorno](#)

[Entrada y Salida Estándar](#)

[Ejemplo de Ejecución](#)

[Recomendaciones](#)

[Restricciones](#)

[Referencias](#)

Introducción

Un web crawler es un programa que escanea un sitio web y cómo sus páginas se enlazan entre sí.

Google, entre otros, tiene web crawlers escaneando toda la internet en búsqueda de nuevas páginas web, páginas que ya no existen y nuevas relaciones (links) entre estas.

En la jerga a estos programas también se los conocen como spiders por su comportamiento de andar moviéndose por la web.

Este TP consistirá en implementar en un web crawler multithreading.

Descripción

Para nuestros fines cada URL tiene 3 estados posibles:
- **ready**: la URL fue encontrada pero aún la página web no fue leída ni analizada.
- **explored**: la página web direccionada por la URL fue leída y procesada.
- **dead**: la URL apunta a una página web inexistente (no encontrada).

El web crawler recibirá por parámetro el archivo **'<targets>'** que contiene una lista de URLs iniciales que son puestas en la *queue* (estado **ready**) para ser escaneadas.

Deberá haber una **única queue**, thread safe, con un método **pop bloqueante** en donde el thread principal (*main*) pondrá las URLs iniciales y los threads workers sacaran URLs para procesarlas y pondrán las nuevas URLs descubiertas.

El crawler lanzara **'<w>'** threads (número determinado por línea de comandos) con el fin de realizar el escaneo en paralelo.

Cada thread toma 1 URL de la queue y realiza un fetch de la página web para luego buscar dentro de ellas más URLs, URLs que pondrá en la queue y continuar el ciclo una vez más.

Nuestro crawler sin embargo no realizará ningún fetch real sino que lo **emulara**.

El archivo **'<pages>'** contiene todas las páginas webs y el archivo **'<index>'** contiene en cada línea una URL, un offset y un size que representan la posición dentro del archivo **'<pages>'** y el largo de la página web (ambos números están en hexadecimal).

Así la emulación del fetch consistirá en buscar el offset y el size de la página web según la URL requerida y leer del archivo dicho bloque.

El archivo **'<pages>'** ***no*** puede cargarse a memoria totalmente. Cada thread deberá leer y cargar solo lo que necesite y liberarlo a medida que ya no lo necesite.

En cambio el archivo **'<index>'** si puede cargarse a memoria. Se recomienda cargarlo en una estructura de datos que permita la búsqueda rápida por URL (como **std::map**).

Si el thread hizo un fetch exitoso (la URL se encontró en el índice), este procesa la página web obtenida y luego pone la URL en el estado **explored**.

Si el fetch falla, el thread pone la URL en el estado **dead**.

Con fines didácticos se requiere el estado de las URLs (**ready**, **explored** y **dead**) se mantenga en un **único objeto compartido** entre los threads .

El procesamiento de la página web consiste en buscar todas las URLs de la forma:

```
http://dominio
http://dominio/
http://dominio/foo/bar
http://dominio/?param=42
http://dominio/foo/bar?
```

Esta garantizado que todas las URLs arrancaran con "`http://`", que están separadas de otros textos por espacios o saltos de línea y que serán URL válidas como las mostradas arriba.

Otras URLs como las "`https://`" no deben ser consideradas.

URLs como las siguientes, aunque válidas, no estarán presentes y no deben ser consideradas (lo que simplificará el parsing):

```
http://user:password@dominio
http://dominio:port
```

Adicionalmente, el crawler recibirá por parámetro un dominio **'<allowed>'** a modo de filtro: las URLs cuyo dominios no coincidan con **'<allowed>'** o no sean subdominios de **'<allowed>'** serán URLs que el crawler deberá ignorar **salvo** si las URLs se encuentran en la lista inicial de **'<targets>'**.

Por ejemplo si **'<allowed>'** es "`example.com`", entonces todas las siguientes URLs están permitidas y deberán ser exploradas por el crawler:

```
http://example.com/
http://example.com/foo/bar
http://sub.example.com/foo/bar
http://sub.sub.example.com/foo/bar?param=1
```

Las 2 primeras URLs tienen el mismo dominio mientras que las últimas 2 son subdominios de `example.com` (`sub.example.com` y `sub.sub.example.com`) Para que un dominio sea subdominio de otro tienen que coincidir de derecha a izquierda.

En cambio, las siguientes URLs no están permitidas y deben ser ignoradas:

```
http://fi.uba.ar/
http://fi.uba.ar/example.com
http://example.com.ar/
```

Nota: en la última URL, `example.com.ar` no es un subdominio de `example.com` ya que no coincide de derecha a izquierda (el `.ar` no coincide con el `.com`).

El thread principal (*main*) por su parte y luego de haber cargado la queue inicialmente y de haber lanzado los thread workers, deberá esperar una cantidad de **'<n>'** segundos.

Es el único lugar del TP en el que se permite llamara a *sleep()* o alguna función similar.

Luego de pasado ese tiempo el thread principal deberá **cerrar la queue** de tal manera que los hilos se enteren que no hay más trabajo pendiente salvo el que ya está encolado.

El thread principal luego esperará a que los otros threads (workers) terminen y finalmente imprimirá por pantalla las todas URLs ordenadas por orden alfabético seguidas de su estado.

Todos las casos de prueba públicos y privados son pequeños y el web crawler debería ser lo suficientemente rápido para explorar todos los links y paginas web en el tiempo dado.

De otro modo habrá inconsistencias y salidas no-determinísticas. Tenerlo en cuenta.

Formato de Línea de Comandos

```
./tp <targets> <allowed> <w> <index> <pages> <n>
```

Parametros:

- **<targets>**: el archivo que contiene la lista de las URLs iniciales a escanear.
- **<allowed>**: solo las URLs con este dominio o un subdominio podrán ser escaneadas (salvo las listadas en **<targets>**).
- **<w>**: cantidad de workers (threads) a lanzar.
- **<index>**: el archivo que indexa el archivo **<pages>**.
- **<pages>**: el archivo que contiene todas las páginas web para la emulación del *fetch*.
- **<n>**: tiempo que el thread principal (*main*) esperará antes de cerrar la *queue*.

Códigos de Retorno

El programa deberá retornar 0.

Entrada y Salida Estándar

No habrá ninguna entrada por la entrada estándar.

El programa deberá imprimir por salida estándar las URLs y sus estados ordenados de forma alfabética.

Ejemplo de Ejecución

Supongamos el siguiente archivo **index.txt**:

```
http://taller-de-programacion.github.io  0x3199   0x168
```

```
http://sercom.taller-de-programacion.github.io/sercom   0x3301 0x80
```

Como puede verse cada línea de **index.txt** tiene el siguiente formato:
`<url> <offset> <length>`

Este archivo es lo suficientemente chico para ser cargado en
memoria en alguna estructura de datos que permita el rápido acceso.

El **<offset>** indica en qué posición del archivo **pages.txt** se encuentra
la página web para esa URL y el **<length>** determina su longitud.
Ambos números son en bytes y están en hexadecimal.

El archivo **pages.txt** ***no*** puede cargarse a memoria. Cada
thread deberá leer y cargar sólo lo que necesite y liberarlo a medida que
ya no lo necesite.

Digamos que el archivo **targets.txt** es el siguiente

```
http://taller-de-programacion.github.io
```

Supongamos que ahora lanzamos el web crawler de la siguiente manera.

```
./tp targets.txt taller-de-programacion.github.io 4 index.txt pages.txt 2
```

El programa cargará la queue con las URLs listadas en **targets.txt**,
lanzara 4 workers (threads) y esperará 2 segundos antes de cerrar la queue.

Cuando un worker tome la url `http://taller-de-programacion.github.io` buscará
esta en el índice, obtendrá el offset `0x3199` y length `0x168` y parará
el archivo **pages.txt** en esa subsección todas las URLs de la forma **“**`http://`**”**.

Supongamos que dicho fragmento fuese:

```
<html>
    <head></header>
    <body>
        <a href=" http://sercom.taller-de-programacion.github.io/sercom "
target="_blank">
        <a href=" http://www.taller-de-programacion.github.io/web " target="_blank">
        <a href=" http://www.taller-de-programacion.github/noreal " target="_blank">
       <a href=" https://sercom.taller-de-programacion.github.io/novalid "
target="_blank">
    </body>
</html>
```

Las URLs que deberia ser obtenidas deberían ser:

```
http://sercom.taller-de-programacion.github.io/sercom
```

`http://www.taller-de-programacion.github.io/web`
`http://www.taller-de-programacion.github/noreal`

Sin embargo, no todas las URLs pertenecen al dominio `taller-de-programacion.github.io`.
Solo las siguientes URLs deberan ser procesadas y puestas en la queue
`http://sercom.taller-de-programacion.github.io/sercom`
`http://www.taller-de-programacion.github.io/web`

Finalmente la URL inicial `http://taller-de-programacion.github.io`
pasara del estado **ready** al estado **explored**.

Ahora la queue contiene 2 URLs:
`http://sercom.taller-de-programacion.github.io/sercom`
`http://www.taller-de-programacion.github.io/web`

Y el ciclo vuelve a empezar.

Al finalizar los 2 segundos (como lo fue especificado en la línea de comandos),
el programa deberá cerrar la queue, imprimir las URLs y sus estados
y finalizar.

Esta sería la salida esperada:

```
http://sercom.taller-de-programacion.github.io/sercom -> explored
http://taller-de-programacion.github.io -> explored
http://www.taller-de-programacion.github.io/web -> dead
```

Recomendaciones

Los siguientes lineamientos son claves para acelerar el proceso de desarrollo sin pérdida de calidad:

1. **Repasar las recomendaciones de los TPs pasados y repasar los temas de la clase.** Los videos, las diapositivas, los handouts, las guías, los ejemplos, los tutoriales.
2. **Verificar** siempre con la **documentación** oficial cuando un objeto o método es *thread safe*. **No suponer.**
3. Hacer algún diagrama muy simple que muestre **cuales son los objetos compartidos** entre los threads y asegurarse que estén **protegidos** con un monitor o bien sean thread safe o **constantes**.
   Hay veces que la solución más simple es no tener un objeto compartido sino tener un objeto privado por cada hilo.
4. **Asegurate de determinar cuales son las critical sections.** Recordá que por que pongas mutex y locks por todos lados harás tu código thread safe. **¡Repasar los temas de la clase!**
5. **¡Programar por bloques!** No programes todo el TP y esperes que funcione. ¡Menos aún si es un programa multithreading!
   Dividir el TP en bloques, codearlos, testearlos por separado y luego ir construyendo hacia arriba. Solo al final agregar la parte de multithreading y tener siempre la posibilidad de "*deshabilitarlo*" (como algún parámetro para usar 1 solo hilo por ejemplo).
   ¡Debuggear un programa single-thread es mucho más fácil!
6. **Escribí código claro**, sin saltos en niveles de abstracción, y que puedas leer entendiendo qué está pasando. Si editás el código "*hasta que funciona*" y cuando funcionó lo dejás así, **buscá la explicación de por qué anduvo**.
7. **Usa RAII, move semantics y referencias.** Evita las copias a toda costa y punteros e instancia los objetos en stack. Las copias y los punteros no son malos, pero deberían ser la excepción y no la regla.
8. No te compliques la vida con diseños complejos. **Cuanto más fácil sea tu diseño, mejor**.

Restricciones

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

1. El sistema debe desarrollarse en C++ (C++11) con el estándar POSIX 2008.
2. Está prohibido el uso de variables globales.
3. El método *pop* de la queue debe ser bloqueante.
4. No se puede usar *sleep()* o similar para sincronizar salvo explícita autorización en el enunciado.
5. El archivo **<pages>** no se puede cargar a memoria.
6. Todos las clases implementadas deben ser movibles y no-copiables salvo excepción justificada y documentada.

Referencias

[1] https://www.cplusplus.com/reference/map/map/
[2] http://www.cplusplus.com/reference/fstream/ifstream/
[3] https://www.cplusplus.com/reference/mutex/

[4] https://www.cplusplus.com/reference/condition_variable/condition_variable/
[5] https://en.cppreference.com/w/cpp/thread/condition_variable