Pipeline

Ejercicio N° 2

Objetivos	• Diseño y construcción de sistemas orientados a objetos • Diseño y construcción de sistemas con procesamiento concurrente • Encapsulación de Threads en clases • Protección de los recursos compartidos
Instancias de Entrega	**Entrega 1:** clase 6 (19/09/2017). **Entrega 2:** clase 8 (03/10/2017).
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[1] y Monitores para el acceso a recursos compartido

Índice

Introducción

Una arquitectura en *pipeline* consiste en dividir un problema en etapas que deben ser aplicadas sucesivamente sobre un flujo de datos, usando como entrada de cada etapa la salida de la anterior. Un ejemplo común de esta arquitectura se da cuando encadenamos varios comandos *shell* usando *pipes*:

```
$ ls | grep "a" | sed 's/a/A/'
```

Donde `ls` listará los archivos del directorio actual, `grep` se quedará con los nombres de archivos que tengan algún caracter 'a', y `sed` cambiará cada 'a' por una 'A' mayúscula. De esta manera, el resultado del *pipeline* serán los nombres de los archivos que contengan 'a', con ese caracter pasado a mayúscula.

Se debe notar que cuando ejecutamos esta línea, se estarán ejecutando tres procesos, y cada uno de ellos recibirá la salida del anterior. Esto permite que cuando `ls` emita su primer resultado, `grep` lo pueda procesar sin esperar a que `ls` termine.

Es importante esta paralelización, ya que si cada etapa del *pipeline* tarda un tiempo similar a las demás, el tiempo de procesamiento total tiende a dividirse entre la cantidad de etapas.

Descripción

En este ejercicio se deberá implementar un programa C++ que reciba como parámetros una serie de comandos que deberá ejecutar en *pipeline* sobre un archivo de entrada, y escribirá sus resultados en otro de salida.

El procesamiento se realizará **línea por línea**, esto es: cada línea leída del archivo será procesada por los sucesivos comandos del *pipeline*, donde cada uno emitirá líneas que servirán de entradas para el comando siguiente. El último comando tendrá la responsabilidad de escribir el resultado en un archivo de salida.

Comandos

Comando echo

El comando `echo` dará como resultado su misma entrada: si lee de su entrada una línea con valor `‘line’` deberá escribir en su salida una línea con valor `‘line’`:

```
echo
input: ‘line’ -> output: ‘line’
```

Comando match <*regex*>

Este comando dará como resultado su entrada siempre y cuando coincida con la expresión regular que recibe por parámetro:

```
match ‘(tra)(.*)’
input: ‘trabajo práctico’ -> output: ‘trabajo práctico’
input: ‘pipeline’ -> output: no devuelve nada # NO SE DEBERÁ escribir en la salida
```

La sintaxis de las expresiones regulares que usaremos será la especificada en ECMAScript[2], y por esto es necesario (y obligatorio) investigar y utilizar las regex estándar de C++[3].

Comando replace <*pattern*> <*replacement*>

Este comando reemplazará en la entrada las ocurrencias de pattern (una expresión regular con sintaxis ECMAScript[1]), y devolverá la cadena con el reemplazo hecho como salida.

```
replace ‘\b(e)’ ‘E’
input: ‘hola’ -> output: ‘hola’
input: ‘ejemplo de entrada’ -> output: ‘Ejemplo de Entrada’
```

Notar que en el segundo caso de input, la presencia de `\b` hace que sólo haya un match (y un reemplazo) al principio de cada palabra que empieza con `‘e’`.

Comandos en *pipeline*

Para encadenar comandos en un mismo *pipeline*, usaremos el string ‘`::`’:

```
echo :: replace ‘\b(e)’ ‘E’ :: match ‘^[^E]’
input: ‘hola’ -> output: ‘hola’
input: ‘ejemplo de entrada’ -> output: no hay salida
input: ‘linea de entrada’ -> ‘linea de Entrada’
```

Modo debug y logs

El programa podrá correr en dos modos:

- Si se corre el programa en modo **debug**, se deberá llevar cuenta de todas las salidas intermedias de cada etapa del pipeline, y luego imprimirlas todas juntas por salida estándar de errores al final de la ejecución.
- En modo **normal**, no se debe escribir nada por salida estándar de errores.

Formato de la salida de debug

Se deberá escribir la salida separada en etapas, y cada etapa contará con un título que indique de qué etapa se trata, y las líneas de entrada y salida.

Supongamos que tenemos el siguiente comando y archivos de entrada y salida:

```
Comando a ejecutar:
echo :: replace ‘q’ ‘Q’ :: match ‘algo’ :: replace ‘\b(a)’ ‘A’
```

```
Archivo de entrada:
linea 1, algo
linea 2, que tiene algo
linea 3, que no lo tiene
```

```
Archivo de salida (resultante de aplicar el comando a la entrada):
linea 1, Algo
linea 2, Que tiene Algo
```

Se deberán imprimir entonces por salida estándar de error, las siguientes líneas:

```
(1) echo1
linea 1, algo -> linea 1, algo
linea 2, que tiene algo -> linea 2, que tiene algo
linea 3, que no lo tiene -> linea 3, que no lo tiene

(2) replace1
linea 1, algo -> linea 1, algo
linea 2, que tiene algo -> linea 2, Que tiene algo
linea 3, que no lo tiene -> linea 3, Que no lo tiene

(3) match1
linea 1, algo -> linea 1, algo
linea 2, Que tiene algo -> linea 2, Que tiene algo
linea 3, Que no lo tiene -> (Filtrado)

(4) replace2
linea 1, algo -> linea 1, Algo
linea 2, Que tiene algo -> linea 2, Que tiene Algo
```

Notar:

- El título indica entre paréntesis el número de etapa empezando en uno, el nombre del comando a ejecutar, y el número de etapa del mismo tipo: **“(4) replace2”** indica que es la cuarta etapa, y que es el segundo comando del tipo replace.
- En la etapa **“(3) match1”** se están dejando pasar las dos primeras líneas usando la expresión regular `‘algo’`. Esto significa que la línea puede contener la regex, no hace falta que coincida completamente (ver diferencias entre `std::regex_match` y `std::regex_search`).
- Cuando indicamos las expresiones regulares por consola, usamos comillas simples para evitar que el shell interprete los caracteres especiales antes de llegar a nuestro proceso.

Formato de Línea de Comandos

El formato de línea de comandos se compondrá del nombre del ejecutable, seguido de los comandos a ejecutar separados por el string `‘::’`.

Además, se podrán usar de manera opcional los siguientes flags:

1) **--input filename**: Se usa para indicar el archivo del cual leer el flujo de datos de entrada. Si no se utiliza el flag se deberá leer de entrada estándar.
2) **--output filename**: Será utilizado para especificar el archivo al cual escribir la salida. En su ausencia, se deberá escribir en salida estándar.
3) **--debug**: Si se activa este flag, al terminar la ejecución se deberá imprimir por salida estándar de errores la “salida de debug” explicada anteriormente.

```
Formato:
$ ./tp [--input filename] [--output filename] [--debug] cmd1 :: cmd2 :: … :: cmdN
```

```
Ejemplo usando entrada y salida estándar, en modo normal:
$ ./tp echo :: replace ‘\b(e)’ ‘E’ :: match ‘^[^E]’
```

```
Ejemplo usando entrada y salida estándar, en modo debug:
$ ./tp --debug echo :: replace ‘\b(e)’ ‘E’ :: match ‘^[^E]’
```

```
Ejemplo usando archivos de entrada y salida, en modo normal:
$ ./tp --input in.txt --output out.txt echo :: replace ‘\b(e)’ ‘E’ :: match ‘^[^E]’
```

Códigos de Retorno

El programa deberá devolver 1 si hay algún error en la línea de comandos, o bien 0 si se pudieron ejecutar las etapas del *pipeline*.

Entrada y Salida Estándar

En caso de no haber especificado un archivo de entrada, el programa leerá el flujo de datos de entrada estándar, y en caso de no haber especificado uno de salida, escribirá la salida de la última etapa del

pipeline en salida estándar. Si se indican archivos mediante los flags explicados antes, el *stream* estándar correspondiente no se deberá utilizar.

Ejemplos de Ejecución

En la sección de descripción se incluyó un ejemplo completo de un pipeline. Para mayor entendimiento, se explican algunos ejemplos de comandos separados a continuación.

Ejemplo del comando echo en modo normal usando archivos de entrada y salida

```
Línea de comandos:
./tp --input entrada.txt --output salida.txt echo
```

```
Archivo entrada.txt:
Primera linea
Segunda linea
```

```
Archivo salida.txt
Primera linea
Segunda linea
```

```
Entrada estádnar:
```

```
Salida estándar:
```

```
Salida estándar de errores:
```

Ejemplo del comando match usando entrada y salida estándar, en modo debug

```
Línea de comandos:
./tp --debug match ‘regex’
```

```
Entrada estándar:
Esta línea tiene la regex
Esta línea no la tiene
```

```
Salida estándar:
Esta línea tiene la regex
```

```
Salida estándar de errores:
```

```
(1) match1
Esta línea tiene la regex -> Esta línea tiene la regex
Esta línea no la tiene -> (Filtrado)
```

Nota La cadena '*(Filtrado)*' debe aparecer en la salida.

Ejemplo del comando replace usando un archivo de entrada y salida estándar, en modo debug

```
Línea de comandos:
./tp --debug --input entrada.txt replace 'regex' 'expresión regular'
```

```
Entrada estándar:
```

```
Archivo entrada.txt:
Esta línea tiene la regex
Esta línea no la tiene
```

```
Salida estándar:
Esta línea tiene la expresión regular
Esta línea no la tiene
```

```
Salida estándar de errores:
(1) replace1
Esta línea tiene la regex -> Esta línea tiene la expresión regular
Esta línea no la tiene -> Esta línea no la tiene
```

Restricciones

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

1. El sistema debe desarrollarse en C++11.
2. Está prohibido el uso de variables globales.
3. Cada etapa del *pipeline* tiene que correr en un hilo separado a las demás, y deberán realizar el procesamiento línea a línea. Esto permitirá la ejecución en paralelo de las etapas.
4. Se deberá implementar alguna estructura de datos bloqueante para la comunicación entre etapas, ya que están prohibidas las esperas ocupadas (*busy waitings*).
5. El objeto que reciba los mensajes de *log* del modo **debug** deberá ser uno solo, y por lo tanto deberá estar protegido con algún mecanismo de sincronización. No es válida una solución que lleve logs separados para cada etapa ya que dicha estrategia escapa a los objetivos del ejercicio.

Referencias

[1] Condition variables: http://en.cppreference.com/w/cpp/thread/condition_variable

[2] ECMAScript: http://www.cplusplus.com/reference/regex/ECMAScript
[3] Expresiones regulares en C++ http://www.cplusplus.com/reference/regex