Frame of Reference

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 (24/09/2019). **Entrega 2:** clase 8 (08/10/2019).
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

Descripción
- Procesamiento en paralelo

Formato de Línea de Comandos

Códigos de Retorno

Entrada y Salida Estándar
- Entrada Estándar
- Salida Estándar
- Salida Estándar de Errores

Restricciones y Ayudas

Referencias

Introducción

En ciertas aplicaciones es común transferir o almacenar largas secuencias de números.

En algunos casos contamos con que los números están ordenados (como por ejemplo cuando los números representan índices) o están ordenados "*localmente*".

En este último caso los números no tienen un orden pero presentan la característica que números similares están en posiciones cercanas.

Descripción

Considere los siguientes numeros en notacion hexadecimal:

```
0012  0015  0013  0014  0012  0021  0017  0015  001b  001a  0018  0016
```

Si bien la secuencia no está ordenada se puede ver como los 5 primeros números están alrededor del `0013`, luego secuencia crece y los siguientes números está alrededor del `0019`.

Este patrón donde los números fluctúan pero no de forma abrupta sino suavemente puede encontrarse en mediciones, por ejemplo de una señal eléctrica.

Podemos tomar ventaja de este hecho para comprimir la secuencia y optimizar la transferencia o almacenamiento de la misma.

La técnica que usaremos es la de *frame of reference* o marco de referencia.

A la secuencia se la divide en bloques de `N` números. Si la secuencia no es múltiplo de `N` el último bloque se completa repitiendo el último valor de la secuencia.

Por cada bloque se calcula el mínimo y se toma ese valor como referencia para ese bloque.

Supongamos `N = 4`, los números de referencia son marcados con un asterisco:

```
0012  0015  0013  0014 | 0012  0021  0017  0015 | 001b  001a  0018  0016

****                   ****                                   ****
```

A cada número se le resta la referencia de su bloque. Esto garantiza que los números resultantes sean siempre positivos o cero.

Continuando con el ejemplo:

```
0000  0003  0001  0002 | 0000  000f  0005  0003 | 0005  0004  0002  0000

****                   ****                                   ****

0012                   0012                                   0016
```

Obviamente debemos almacenar también las referencias para poder reconstruir la secuencia original por lo

que pasamos de bloques de `N` números a bloques de `N+1`.

La ventaja está en que si los números contiguos mantienen cierta similaridad las diferencias deberían ser pequeñas. Lo que nos permite codificarlas con menos bits.

Para ello veremos cuál es la cantidad mínima de bits que necesitamos para codificar un bloque.

Al tomar como referencia el mínimo garantizamos que estas diferencias son además no negativas por lo que no tenemos que preocuparnos por el signo.

Tomemos como ejemplo el primer bloque:

```
0000  0003  0001  0002
```

El numero (la diferencia) más grande es el número `0003` por lo que necesitamos solo 2 bits para represenarlo.

Entonces, en notación binaria y usando solo 2 bits, el bloque debería resultar en:

```
00  11  01  10
```

Si las muestras iniciales eran de 4 bytes, pasamos de tener 4*4 bytes a tener 4*2 bits, o sea 1 byte.

Por supuesto, dado que la cantidad de bits que necesitamos en cada bloque varía, debemos almacenarla junto con el valor de referencia:

```
00  11  01  10 | 0000  1111  0101  0011 | 101  100  010  000
**               ****                                ***
0012 (2)         0012 (4)                            0016 (3)
```

Nótese como el último bloque resulta en un número de bits **no** múltiplo de 8, 12 bits en particular.

Para facilitar el acceso (lectura) de un bloque, estos deben estar alineados a 8 bits o 1 byte.

En este caso el bloque número tres no esta alineado faltandole 4 bits de padding que se pondran a cero.

El resultado es:

```
00  11  01  10 | 0000  1111  0101  0011 | 101  100  010  000  0000
**               ****                                ***  ^^^^
0012 (2)         0012 (4)                            0016 (3)
```

Finalmente cada bloque es escrito a disco o transferido enviando la referencia (4 bytes), la cantidad de bits usados en cada muestra dentro del bloque (1 byte) y las muestras empaquetadas (n bytes).

Asi, un bloque de N*4 bytes (asumiendo que los números iniciales son de 4 bytes) se representa con 4+1+n bytes.

Procesamiento en paralelo

El presente trabajo consiste en comprimir una secuencia de números usando la técnica de frame of reference mencionada anteriormente procesando `T` bloques en paralelo.

La lectura del archivo de entrada se hará por los `T` hilos en *round robin*.

Esto es, el primer bloque de números los lee el hilo 1, el segundo el hilo 2, el tercer el hilo 3, y así hasta el bloque número `T` lo lee el hilo `T`, el bloque `T+1` lo lee el hilo 1, el `T+2` el hilo 2 y así sucesivamente.

En el caso de `T = 2`, los bloques impares (1, 3, 5) se leerán por el primer hilo mientras que los pares (2, 4, 6) por el segundo.

Siguiendo con nuestro ejemplo:

```
0012  0015  0013  0014 | 0012  0021  0017  0015 | 001b  001a  0018  0016

---------------------- ----------------------- -----------------------

      hilo 1                    hilo 2                   hilo 1
```

Es restricción del enunciado que cada hilo leea del mismo *file descriptor* (`std::ifstream`)

Esto fuerza a que haya una *race condition* que se espera que la implementación pueda solventar y evitar (así tienen un lugar claro de usar un *mutex*, aunque puede que **no sea el único**)

A medida de que cada hilo termina de procesar su bloque este debe ser escrito a disco. Pero la escritura supone una *race condition* que ni siquiera un *mutex* puede resolver (se ve por que?)

Lo que cada hilo tendrá es una cola *thread-safe* (*queue*) en donde irán guardando los bloques procesados (*push*). Asi, el hilo 1 tendrá una cola `q1`, el hilo 2 una cola `q2`.

Será tarea de un hilo extra, el *escritor*, que leerá de esas colas haciendo *round robin* e irá escribiendo a disco de forma ordenada.

Esto es, leerá el primer bloque de la cola `q1`, luego de `q2`, luego de `q3`, así hasta `qT` y el ciclo comenzará otra vez.

Esto garantiza que los bloques escritos a disco estén en el mismo orden de cómo fueron leídos.

Es restricción del enunciado que cada hilo lea y procese de a un bloque a la vez y lo guarde (*push*) en su respectiva cola tan pronto como pueda.

El hilo *escritor* deberá leer de la cola que le corresponda y guardar a disco tan rápido como pueda.

El archivo no puede ser leído en su totalidad en memoria, por lo que cada hilo deberá leer y procesar de a un bloque y las colas tendrán un límite de `Q` items. Al alcanzar ese límite el hilo deberá esperar hasta que haya lugar.

Formato de Línea de Comandos

`./tp <N> <T> <Q> <infile> <outfile>`

Donde

`<N>` es la cantidad de números que componen cada bloque.

`<Q>` es la cantidad de máxima de elementos que cada cola (q1, q2, q3) puede tener.

`<T>` especifica la cantidad de hilos a procesar (sin contar el hilo *escritor*).

`<infile>` es el nombre del archivo de entrada; si es `'-'` se lee de *stdin*.

`<outfile>` es el nombre del archivo de salida; si es `'-'` se escribe en *stdout*.

Códigos de Retorno

Retorna 0 en caso exitoso o 1 si algo fallo como argumentos incorrectos o archivos no existentes.

Entrada y Salida Estándar

Entrada Estándar

Solo si `<infile>` is igual a `‘-’`, contiene la secuencia de números a comprimir.

Salida Estándar

Solo si `<outfile>` is igual a `‘-’`, contiene la secuencia de números comprimidos.

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.

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. Las condiciones de carrera son descalificadoras.
4. Se deben respetar las restricciones del enunciado.

5. Los números de la secuencia de entrada son números de 4 bytes sin signo y *big endian*. Los números resultantes de la compresión (cuando corresponda) debe tratarse también como sin signo y *big endian*.

Referencias

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