ICOM2020 – EXAMEN FINAL

29 de diciembre de 2020

Notas:

1. Uso de prácticos: **se pueden utilizar los trabajos prácticos propios realizados.**

Problema 1: Block chain

Se desea implementar un sistema para manejar transacciones que utilice mecanismos parecidos a los que utiliza la tecnología de cadena de bloques (Block Chain) para asegurar la integridad de las transacciones realizadas.

Para ello, las transacciones (supongamos que son números enteros) son almacenados en bloques. Cada bloque permite almacenar hasta N transacciones. Cuando un bloque es completado, se genera una firma de integridad del bloque (por ejemplo un checksum o CRC), y se crea un nuevo bloque que es referenciado por el anterior. El nuevo bloque almacena la firma de integridad del bloque anterior. Con esto se asegura que si una transacción anterior (almacenado en algún bloque previo) es modificada, esto pueda ser detectado ya que haría que la firma de integridad (que esta almacenada en el bloque siguiente) deje de verificarse.

Utilizando las siguientes definiciones:

```
using Transaction = int;

struct Block_data {
    static const unsigned N = 32;
    Block_data(unsigned pcrc) : prevCRC(pcrc) {}
    unsigned prevCRC;         // checksum del bloque previo, 0 si es el primer bloque
    Transaction transactions[N] = { 0 }; // transacciones del bloque
};

struct Block {
    Block(unsigned crc) : data(crc), pNext(nullptr) {}
    Block_data data;            // bloque de datos
    Block* pNext;               // siguiente bloque en la cadena
};

unsigned genCRC(const Block_data* pBlockData);

class BlockChain {
public:
    // Crea una nueva cadena de bloques y la inicializa como vacía
    BlockChain(); // TODO

    // destruye una cadena de bloques liberando todos
    // los recursos asociados
    ~BlockChain(); // TODO

    // Agrega una nueva transacción a la cadena
    void addTransaction(Transaction t);  // TODO

    // función que devuelve verdadero o falso indicando la validez de la cadena
    bool chainValid();  // TODO

private:
    Block* firstBlock;             // primer bloque de la cadena
    Block* currBlock;                   // bloque en generación
    unsigned currTransacIdx;        // índice de transacción dentro de currBlock
};
```

Suponga que cuenta con la función:

```
unsigned int genCRC(const Block_data *pBlockData);
```

Utilizando el esqueleto en **`BlockChain.cpp`** complete las funciones restantes.

Problema 2: Generador de Laberintos

Se solicita implementar un generador de laberintos completando el código que se provee en **`lab_gen.cpp`** (si necesita o ve conveniente implementar otros métodos o funciones, hágalo). El algoritmo que se debe implementar es el siguiente:

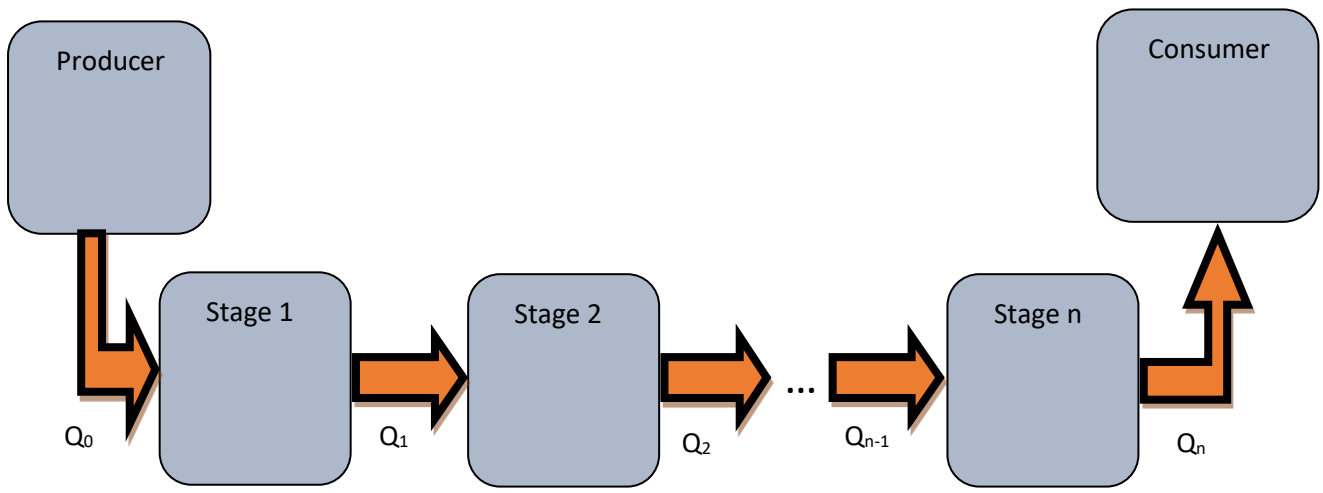
1. El número de filas (**`nf`**) y columnas (**`nc`**) del laberinto debe ser impar, mayor o igual a 3 y deben cumplir que **`nf * nc >= 15`**
2. Se inicia colocando paredes en todas las celdas con índice de fila **`o`** de columna par (recuerde que los índices comienzan en 0). En el resto de las celdas se colocan espacios vacíos.
3. Se elige al azar uno de los espacios vacíos y se lo reemplaza por la entrada **`E`**.
4. Se elige al azar otro de los espacios vacíos y se lo reemplaza por la salida **`S`**.
5. Se realiza el siguiente proceso recursivo, comenzando por la celda de la entrada:
   a. Se recorren, **en forma aleatoria**, todas las celdas vecinas que están a una distancia de **2** celdas en las 4 direcciones (arriba, abajo, izquierda y derecha).
   b. Si el vecino corresponde a una celda válida que no fue visitada anteriormente, se la marca como visitada, se elimina la pared que la separa de la celda sobre la que estamos trabajando y en su lugar se coloca un espacio visitado.
   c. Se repite el proceso para el vecino que acabamos de visitar.
   d. Si no se puede visitar a un vecino, se continúa con el siguiente.
6. Finalmente se recorre todo el laberinto, marcando las celdas visitadas como no visitadas.

Implemente un **main** donde pruebe su generador de laberintos, solicitando al usuario que ingrese el número de filas y columnas. Realice un manejo de excepciones adecuado, utilizando **`throw`**, **`try`** y **`catch`**.
Las siguientes imágenes muestran un laberinto de 5x5 en distintas etapas del proceso de construcción.

```
#####    #####    #####    #####
# # #    # #S#    #o#S#    #  S#
#####    #####    #o###    # ###
# # #    #E# #    #E# #    #E  #
#####    #####    #####    #####
 (2)      (3,4)    (5…)     (6)
```

Problema 3: Simulación de un pipeline

Como parte del diseño de una línea de montaje, se desea simular en forma simplificada el comportamiento de la dinámica del *pipeline*, y analizar el efecto de distintos parámetros en las diferentes etapas (*working stages*) que componen la línea de trabajo y de los canales de comunicación y transporte (*transfer queues*) entre esas etapas.



Se puede pensar que las unidades de trabajo (*Work Unit*) que ingresan por Q_0 colocadas por un generador (**`Producer`**), van sufriendo transformaciones como resultado de las distintas acciones que van realizando los distintos *working stages* a lo largo del *pipeline* y termina saliendo como producto terminado por Q_n, y tomadas por un consumidor (**`Consumer`**)

Cada **`WorkingStage`** puede estar trabajando solo sobre un **`WorkUnit`** en un dado momento, pero las distintas etapas podrían estar ejecutando acciones sobre **`WorkUnit`** distintas con lo que se logra que trabajen en "paralelo", aumentando el throughput del sistema.

Para la simulación se modelan las *transfer queues* como instancias de la clase **`XferQueue`**, colas de tamaño ilimitado y con una interface como la que puede verse en la implementación parcial del sistema completo que hay en **`pipeline.cpp`**.

Desde el punto de vista de un **`WorkUnit`**, este tiene el siguiente ciclo de vida:
- Alguien (**`Producer`**) lo crea y lo coloca en Q_0 con algún tiempo característico entre creaciones.
- En algún momento, cuando el **`stage1`** esté disponible, será tomado por éste, el que le aplicará un conjunto de acciones de transformación. Para la aplicación de cada acción hace falta un tiempo propio del tipo de acción.
- Cuando **`stage1`** complete sus acciones, colocará el **`WorkUnit`** en Q_1 para siga su transformación y así hasta que sea tomado por el **`Consumer`**.

El ciclo de cada **`WorkingStage`** consiste en:
- Chequear la cola de entrada, si no hay **`WorkUnits`**, no hace nada.
- Si hay **`WorkUnits`** pendientes toma uno, y le comienza a aplicar acciones. Cada acción le insume un tiempo determinado.
- Cuando completa todas las acciones de transformación coloca el **`WorkUnit`** sobre la cola de salida.

Para dar la noción de paralelismo en la simulación se utiliza un planificador (**`Scheduler`**) que, conociendo la topología del sistema completo, se encarga de ir haciendo avanzar la simulación a través de la discretización del tiempo en períodos de ancho fijo que denominaremos **`ticks`** y solicitando que cada Stage (**`Procucer, WorkingStage y Consumer`**) vayan avanzando.
Complete la implementación parcial del sistema que está en **`pipeline.cpp`** y pruebe con el **`main`** dado.