Tarea 2: Operativos

Ale Parada- -9/10/ 2022

En la presente tarea se especificará el algoritmo creado para poder cumplir con los diferentes puntos a evaluar de la tarea.

Primero que todo, se comienza creando la estructura principal con la cual se trabajará, la cual corresponde a un grafo. Para esto se utiliza una lista de adyacencia que permite ir guardando los vectores correspondientes, además de ir diseñando a partir de los vértices, los caminos entre cada vector con su correspondiente peso.

```
struct Grafo
{
    int V;
    list<int> *adj;
    int peso;
    vector<vector<int>> aux;

    Grafo(int v)
    {
        V = v;
        adj = new list<int>[v];
    }

    void camino(int u, int v, int peso2)
    {
        this->peso = peso2;
        adj[u].push_back(v);
    }
```

El primer punto a considerar es por cada thread vaya recorriendo el grafo, para simplificar esto se crean dos funciones que permiten guardar todos los caminos posibles del grafo entre un vértice a otro, para luego a partir de un random poder mandarle un camino aleatorio entre todos al thread de turno, esto se hace por cada thread creado.

Para realizar lo anterior mencionado se utilizan las siguientes funciones

Primero se parte con la función todos los caminos, que permitirá ir imprimiendo recursivamente todos los caminos del grafo, esto a partir de ir marcando como no visitados a los vértices del camino e ir guardando los caminos obtenidos en la función crear ruta, funcionando de manera compartida con la funcion que se describe a continuación

```
void todosloscaminos(int s, int d)
{
    // marca todos los vertices como no visitados
    bool *visitado = new bool[V];

    //  para guardar caminos
    int *camino = new int[V];
    int path_index = 0; // inicializo en 0

    // marcando como no visitados
    for (int i = 0; i < V; i++)
        visitado[i] = false;

    // recursivo para ir imprimiendo los caminos
    crearRuta(s, d, visitado, camino, path_index);
}
```

En la función crear ruta permite ir verificando que que el nodo final corresponda a los vértices elegidos, en el caso que sean iguales se van guardando en un vector que permitirá imprimir el vector más adelante con los caminos que cumplen la ruta entre un nodo a otro, por otra parte si no se cumple que el vértice no es el destino, se irá repitiendo el proceso recursivamente.

```
void crearRuta(int u, int d, bool visitado[], int camino[], int &index)
{
    visitado[u] = true;
    camino[index] = u;
    index++;

    if (u == d)
    {
        vector<int> nuevoc;
        for (int i = 0; i < index; i++)
            nuevoc.push_back(camino[i]);
        aux.push_back(nuevoc);
    }
    else // si el vertice no es el destino
    {
        // se repite para todos los vertices adj

        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (!visitado[*i])
                crearRuta(*i, d, visitado, camino,
                          index);
    }

    // elimina el vertice del arreglo y lo marca como no visitado
    index--;
    visitado[u] = false;
}
```

La siguiente función permite pasarle el camino aleatorio al thread correspondiente, para esto se crea un grafo de 10 vértices (esto se puede modificar a gusto), en el cual se pasarán las funciones previamente realizadas, en esto se utiliza un vector de vectores que permitirá guardar todos los caminos posibles, para poder aplicar una función random que seleccione un camino de este vector e imprimirlo.

En este punto además se aplica lo correspondiente al semáforo, debido a que hay que restringir cuantos threads utilizar dicha función de los caminos, para esto se utiliza la función sem que permitirá aplicar el

semáforo correspondiente aplicando la zona crítica al camino randomizado.

```
void *pasarcamino(void *arg)
{
    Grafo g(10);
    g.camino(0, 1, 4);
    g.camino(0, 2, 1);
    g.camino(0, 3, 3);
    g.camino(1, 4, 5);
    g.camino(2, 4, 5);
    g.camino(3, 4, 6);
    g.camino(4, 5, 10);
    g.camino(4, 6, 10);
    g.camino(5, 7, 22);
    g.camino(6, 7, 7);
    g.camino(7, 8, 8);
    g.camino(8, 9, 9);

    int s = 0, d = 9;

    sem_init(&hola, 0, 10);
    sem_wait(&hola);
    g.todosloscaminos(s, d);

    int tam = g.aux.size();
    int random = rand() % tam;

    for (auto v : g.aux[random])
    {
        cout << v << " ";
    }
    cout << endl;

sem_post(&hola);
}
```

Finalmente se adjunta el main creado, para realizar los threads, se define un arreglos de estos para luego a partir del primer for adjuntado, ir creando estos, para esto se comienzan con crear 10 threads, este valor se puede ir

modificando.El segundo for adjuntado es para poder repetir cuantas veces el thread iterara con la función adjuntada para imprimir el camino aleatorio, este valor también se puede modificar.

El último for adjuntado, se utiliza la función pthread_join que permite una sincronización entre los threads creado en un orden de 1 a 10, sin embargo como observación puede ocurrir en este proceso un interleaving.

```
int main()
{
    srand(time(NULL));

    for (int i = 0; i < 10; i++) //threads que quiero
    {
        for ( int j = 0; j < 2; j++) //vez que recorre 1 thread en el grafo en este caso son 2
        {
            pthread_create(&(tid[i]), NULL, &pasarcamino, NULL);

        }

    }

    for (int i = 0; i < N; i++) //sincronizacion por thread
    {
        pthread_join(tid[i], NULL);
    }

    return 0;
```

Como dato adicional no se realizó lo correspondiente al punto 3 de la tarea, sin embargo se aplicó una leve sincronización en el apartado de los join