Headers

Para el funcionamiento completo del programa, se han considerado dos headers o interfaces: "lista.h" y "definitions.h".

La primera define la estructura que tiene un nodo (puntero de caracteres como valor y un puntero a siguiente nodo) y una lista enlazada: un nodo primero y un último. Después define los métodos que dictaminan su funcionamiento: métodos de inserción/creación, eliminación, obtención e impresión.

La segunda define las rutas de los procesos contador y procesador, y sus clases. También define una estructura de proceso (process_id y clase)

Lista.c

En este apartado de la memoria **haremos hincapié únicamente en aquellos métodos que tengan una mayor complejidad**. Estos serán InsertarN() y EliminarN(). Pero antes de ello, es importante resaltar que la **lista se ha implementado considerando como el índice de primer elemento el 0, y el último como n-1**, tal y como sucede en un array (importante de cara al manager).

InsertarN()

Para insertar un valor cualquiera en una posición N de la lista, presuponemos que se inserta detrás de dicha posición N. Esto ya difiere de los otros métodos de inserción: insertar(), que crea un nodo antes de *pPrimero, o insertarFinal(), que coloca un nodo después de *pUltimo.

Lo primero que haremos será controlar el valor de índice que se le pasa a la función, siendo incorrectos los valores negativos o aquellos que sobrepasen el valor del último índice de la lista (= tamaño - 1).

Una vez tenemos un índice válido, declaramos dos nodos auxiliares, el iterativo para avanzar en la lista, y el clon que nos ayudará a colocar el elemento nuevo. Despúes, discriminamos el valor de *index*:

- Para $index = 0$ llamamos al método insertar() y ahorramos líneas de código.
- Para $index \geq 1$, iremos avanzando las posiciones de la lista ("nodo iterativo = su siguiente nodo"). Una vez el iterador se ha colocado en la posición 'index', clonamos la información del nodo iterativo en el clon, ya que el nodo iterativo con posición 'n' será modificado para el nuevo valor. Entonces, podremos redirigir estos dos nodos, por lo que se habrá simulado una inserción.

EliminarN()

El método eliminarN() hace justo lo contrario al método explicado anteriormente. En vez de insertar un nodo en un índice N, va a eliminarlo, liberando el espacio de memoria asignado a dicho nodo.

En primer lugar, tras recibir el puntero con el nombre de la lista y el índice a eliminar, se va a controlar que el valor del índice no sea negativo (< 0) o mayor al índice del último elemento de la lista. Si se da el caso, saltará una excepción informando del problema y finalizará el programa.

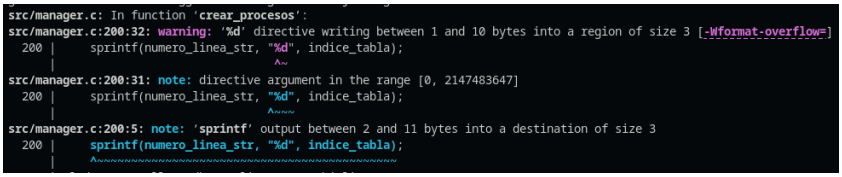
Tras esto, se declara un nodo llamado "borrar" que apunta a NULL. Este nodo se encargará de recoger la información del nodo que se quiere eliminar para eliminarlo y redimensionar la lista. Una vez realizado esto, procedemos a analizar el valor del índice (index):

- Para *index = 0*, simplemente recogemos en el nodo borrar el primer elemento de la lista, liberamos el espacio de memoria asignado a su valor, asignamos como primer elemento de la lista el nodo siguiente y procedemos a liberar el espacio de memoria asignado al nodo borrar.
- Para *index ≥ 1* se declara un nodo iterativo para avanzar en la lista y llegar al nodo deseado. Si durante el recorrido de la lista se llega a una zona de memoria no permitida, se procederá a lanzar una excepción comunicando el error y finalizará el programa. Una vez finalizada la iteración y almacenada la información del nodo, se libera el espacio de memoria asignado al nodo que se desea borrar y se comprueba si el siguiente elemento al que iteraría es nulo (NULL). En caso afirmativo asignamos al último elemento de la lista el último nodo avanzado durante la iteración anterior. En caso contrario, apuntamos a nulo el siguiente elemento del nodo borrar y liberamos el espacio de memoria asignado al nodo.

Consideraciones en manager.c

Para el desarrollo de la práctica, se nos ha proporcionado un esqueleto, donde esta clase manejadora de procesos contadores y procesadores estaba completa. Sin embargo, por lo que hemos comentado al principio de lista.c, se han tenido que realizar un par de modificaciones.

De base, al compilar con *make* (más tarde se explica su funcionamiento), muestran estos avisos.



Debido a que **realizamos la compilación con el compilador *gcc* en su versión 12**, el Wall nos avisa que es posible que se produzca un desbordamiento en *numero_linea_str,* acarreando posibles problemas en los *sprintf().* Modificando su tamaño de 3 a 11, se corrigen los avisos.

Por otro lado, no llega a realizarse una ejecución correcta, ya que no se lanzan los procesos de clase procesador como deberían. Con el código base, se desea que desde el segundo elemento de la lista hasta el último (incluido), se lancen procesos de esta clase, en función del patrón almacenado en la lista. Para nuestra lista, lo corregimos quitando una iteración al bucle *for*:

```
for (int i = 1; i < longitud(patrones); i++);
```

Estos cambios están reflejados dentro de la clase manager.c, en el método crear_procesos().

Procesador.c

Para buscar coincidencias de un patrón con respecto a cada línea del archivo, se ha decidido copiar el contenido del archivo en un búfer de caracteres, puesto que vamos a emplear la función strtok() para dividir el contenido por líneas (resultado en un puntero llamado linea).

Entonces, a cada resultado de la división, lo comparamos con el patrón, a través de la función strstr(). Su salida la vamos a escribir en un puntero auxiliar (se ha requerido inicializar dicho puntero).

Si es nulo su contenido (**\0**), quiere decir que no hubo coincidencia, de lo contrario, se encontró el patrón en la línea. Todo esto lo repetimos hasta que la línea apunta a NULL, es decir, ya no hay más contenido en el búfer.

Contador.c

Como queremos conocer cuántas palabras tiene cada línea recogemos el contenido de la línea y el número de dicha línea. Una vez obtenidos los argumentos, se los vamos a pasar a un método llamado contar().

Definimos tres variables, dos de tipo entero (una para contar el nº de palabras y otra para comprobar si recorriendo la línea nos topamos con una palabra) y un puntero de tipo char con el que recorreremos la línea en busca de palabras.

Iremos recorriendo la línea y visualizaremos si nos encontramos valores nulos (\0), espacios (\t), saltos de línea (\n) o recorrimientos de cursor a la izquierda (\r). Se comprueba si nos topamos con una palabra y en caso afirmativo aumentamos en una unidad la variable que cuenta el número de palabras.

Por último, tras haber recorrido la línea procedemos a imprimir el número de línea y la cantidad de palabras que contiene.

Compilación y ejecución con Makefile

Una vez hemos terminado de programar las clases, es hora de comprobar si funciona lo qure hemos escrito. Para ello usaremos un archivo MAKEFILE. Los archivos MAKEFILE contienen reglas que especifican las acciones a realizar por el comando *make* y directivas que indican a éste algunas formas especiales de interpretación del contenido del archivo MAKEFILE.

El comando *make* funciona basándose en las reglas especificadas en el archivo MAKEFILE. En esas reglas se expresa, para cada archivo que deba construirse, de qué otros archivos dependen.

En este caso, cuando escribamos *make* por terminal, nos va a crear directorios nuevos:

- obj/ Aquí se alojarán los archivos objeto (extensión .o), necesarios para la compilación
- exec/ En este directorio se guardan los ejecutables resultantes de la compilación

En la ejecución de make, se crearán 4 archivos objeto: lista.o, manager.o, procesador.o y contador.o, con opción para depuración (ggdb), usando como includes la carpeta include del proyecto (por eso creamos archivo objeto de la lista) y visualización de todas las advertencias (Wall), y 3 ejecutables: contador, manager y procesador. Por último, se ejecutará el manager, a modo de test, pasando como argumentos los archivos de texto incluidos en data/.

En el archivo MAKEFILE también hay definida una última regla de borrado, denominada clean, para borrar las carpetas creadas anteriormente, incluyendo su contenido. Para llevarlo a cabo, sólo debemos escribir en nuestro espacio de trabajo donde esté MAKEFILE el comando *make clean*.