Instituto Tecnológico de Costa Rica

Arquitectura de Computadores

Grupo 40

IC3101

Tarea Programada 1

Profesor: Eduardo Adolfo Canessa Montero

Fabian Araya - 2019159110

Binjie Liang - 2023064642

Alejandro Madrigal - 2020219677

Esteban Secaida - 2019042589

IS – 2024

Tabla de Contenidos

Introducción

Para comprender óptimamente el próximo proyecto, es esencial familiarizarse con una variedad de conceptos clave. Para empezar, NASM [1] (Netwide Assembler) es un ensamblador y desensamblador que se ha desarrollado para la arquitectura Intel x86. El código NASM puede escribirse para programas de 16, 32 o 64 bits. Es importante entender que el desarrollo del código se llevó a cabo en Linux [7], un sistema operativo popular en entornos de desarrollo y servidores de código abierto, una estructura de archivos jerárquica, una seguridad robusta y una amplia compatibilidad con hardware son todos los beneficios que ofrece.

En el presente documento, se da la explicación y detalles del desarrollo de un editor y comparador de texto en lenguaje ensamblador NASM para la arquitectura x86_64. Este programa puede mostrar el contenido de un archivo de texto, compararlo con otro archivo y editar el texto. El desarrollo se llevó a cabo en Linux Ubuntu versión 18+, utilizando las llamadas al sistema del sistema operativo.

Es importante recalcar que un editor de texto, es una herramienta fundamental en la informática moderna que permite crear, visualizar y modificar archivos de texto. Funciona como una interfaz entre el usuario y el contenido del archivo, facilitando la edición de texto de manera eficiente.

Para llevar a cabo este proyecto, fue necesario investigar y comprender varios temas importantes de programación en ensamblador. La manipulación de archivos, el manejo de entradas del usuario, las llamadas al sistema y el control de cadenas y buffers son algunos de estos temas. A pesar de que algunos de estos conceptos ya se habían trabajado en tareas anteriores, la magnitud y complejidad de esta tarea requerían un estudio y lógica a bajo nivel más exhaustivo.

Desarrollo

Para la primera parte del desarrollo de este editor, se investigó otros desarrollos de editores, por ejemplo el editor nano, el editor kilo [2], vim, entre otros. Durante la investigación, nos encontramos con un video: desarrollando un editor de texto en C de forma minimalista [4]. Este video nos ayudó bastante para tener una mejor lógica de cómo debería funcionar el programa y minimizar los errores que podría ocurrir. Para la segunda etapa, nos concentramos en hacer los flujos que debía tener el programa. Nos basamos en gran parte de la lógica del video mencionado anteriormente, pero nos encontramos con complicaciones que perjudicaban al funcionamiento fluido del editor, por lo tanto, tuvimos que adaptarlo.

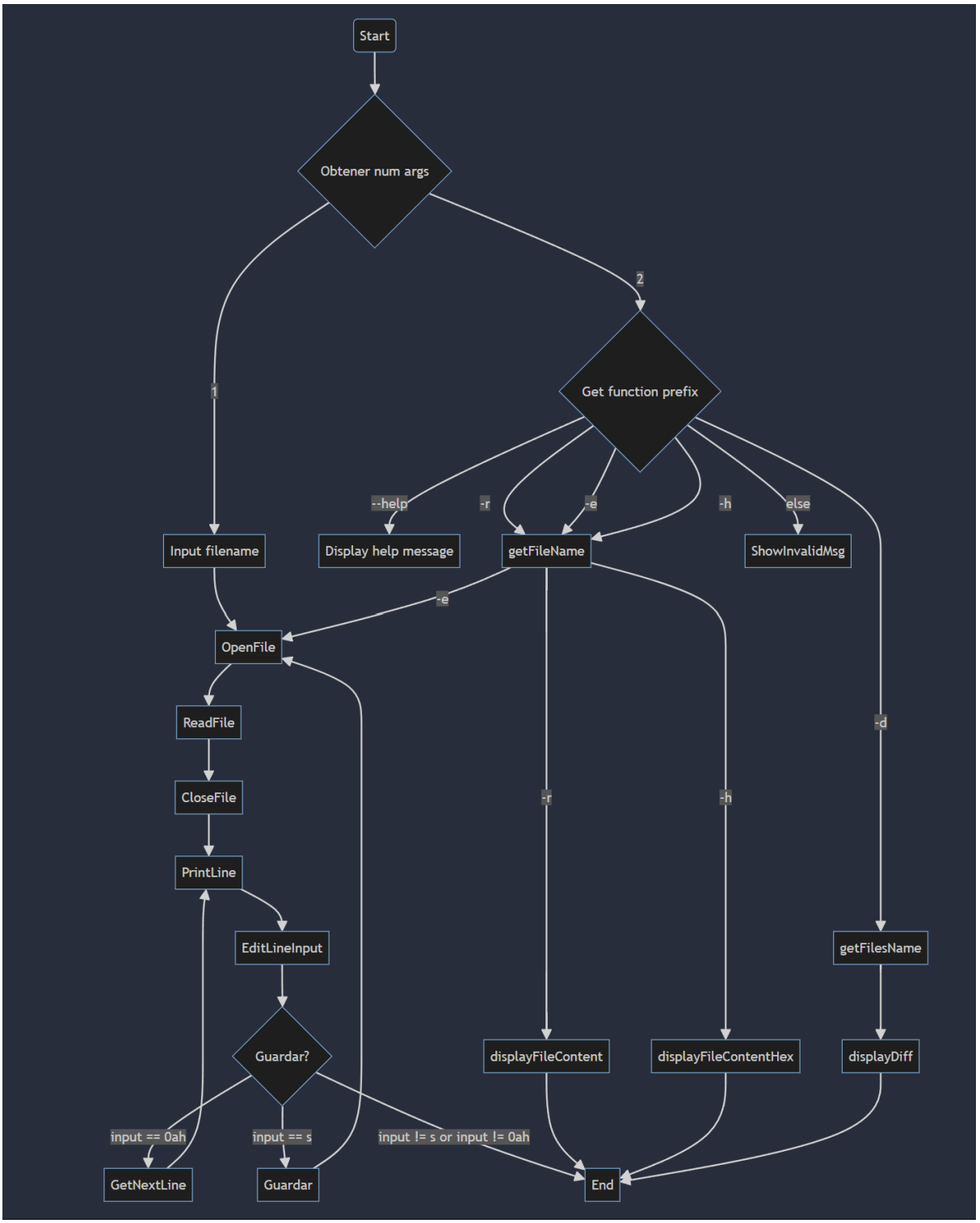

Diagrama de flujo desarrollado en Mermaid [5]

Para el diagrama de flujo, primero comenzamos viendo cuantos argumentos tiene la consola. Es decir, si en la línea de comandos invocamos `./prog arg1 arg2` tenemos que saber cuantos argumentos se ingresaron. Para eso, se hizo una investigación y encontramos una pregunta similar en StackOverFlow [6]. Al iniciar un programa en

ensamblador, este guarda datos de la línea de comandos en el stack. Si le hacemos Pop al stack, nos va a tirar un valor numérico, este primero valor, nos indica cuantos argumentos tiene la línea de comandos, en este caso, con el ejemplo anterior serían 3, *./prog* cuenta como primer argumento, *arg1* cuenta como segundo argumento y *arg3* cuenta como tercero. Al hacer el siguiente Pop en el stack, nos dará el nombre del programa (./prog), el siguiente Pop nos dará el nombre del segundo argumento (arg1) y al hacer el siguiente, nos dará el nombre del tercer argumento (arg2).

Con esta investigación, pudimos resolver la primera incógnita de nuestro desarrollo, que era leer argumentos en la línea de comandos.

Entonces, si el usuario no ingresa argumentos a la línea de comandos después de iniciar el programa, el programa le pedirá que ingrese un nombre de archivo. Abrimos el archivo y este se guardará en el buffer correspondiente y cerramos el archivo.

De aquí en adelante, comienza un ciclo while, donde si el usuario presiona un teclado diferente a ENTER o “s” el programa termina. Pero antes de eso ocurre algo importante, el programa muestra linea por linea, por lo tanto, enseguida le preguntara al usuario que ingrese un texto para modificar la línea, independientemente de lo que escriba o no, al dar ENTER el programa le pedirá al usuario otro input, este input es donde usted tiene que ingresar si desea continuar leyendo la siguiente línea, guardar las modificaciones o salir.

Si presiona ENTER, obtenemos el siguiente puntero de la línea y simplemente hacemos el mismo ciclo. En caso que desee guardar, primero, guardamos el nuevo texto de la línea, segundo, guardamos todo el contenido anterior a la línea actual y tercero guardamos todo el contenido después de la línea actual. De este modo, abrimos el archivo en modo WRITEONLY y escribimos el nuevo texto.

Si el usuario ingresa 2 argumentos, debemos de comparar con la función de comparador de prefijos que hicimos. Si el usuario ingresa --help como argumento, mostramos el mensaje de ayuda/manual de usuario.

Si el usuario ingresa -r como argumento, obtenemos el nombre de archivo del stack, abrimos el archivo para mostrar el contenido.}

Si el usuario ingresa -e como argumento, obtenemos el nombre de archivo del stack, saltamos hacia el proceso de cuando no se tenían argumentos, directamente a abrir el archivo, ya que tenemos el nombre de archivo.

Si el usuario ingresa -h como argumento, abrimos el archivo obtenido del stack, y mostramos el contenido del buffer en formato hexadecimal.

Si el usuario ingresa -d, tenemos que ver si ingreso 4 argumentos, si no, termina el programa, pero si, si lo hizo, mostramos las diferencias de línea por línea del archivo2 respecto al archivo1.

Bloques Funcionales

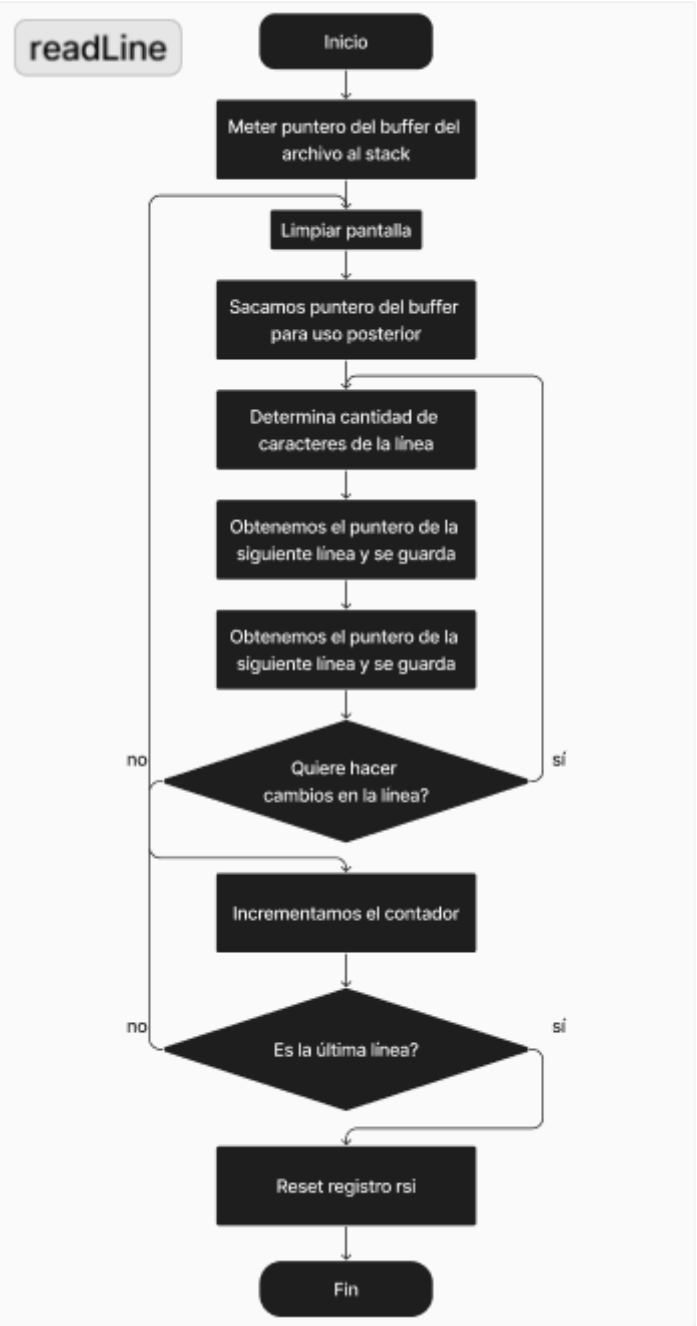

Diagrama de flujo bloque funcional readLine en Figma[3]

Hagámosle énfasis al bloque funcional readLine, el cual es el ciclo principal del editor. Primero, debemos meter al stack el puntero del buffer del archivo. Al entrar al ciclo limpiamos la pantalla primero. Sacamos el puntero del buffer de la pila y

debemos guardarlo también, para uso posterior. Sacamos cuantos caracteres tiene la línea actual para poder imprimirlo Después de eso, sacamos el puntero nuevamente, para obtener el puntero de la siguiente línea (lo guardamos en la pila). Le preguntamos al usuario si quiere hacerle cambios a la línea actual. Si no hace cambios, pues incrementamos el contador de líneas y volvemos al ciclo, donde debemos hacerle pop al puntero del buffer (en este caso ya sería la siguiente línea). Es importante recalcar que si llegamos a la última línea, hagamos un reset al puntero del rsi. Si el usuario decide guardar, eso se explicaría en el siguiente bloque funcional.

Entonces expliquemos el bloque funcional Guardar, que de hecho son 3 funciones. El programa tiene en memoria la línea actual con sus modificaciones, primeramente recibe el puntero de la siguiente línea, entonces al comenzar en la siguiente línea, podemos simplemente almacenar todo su contenido en una memoria nueva. Ahora para guardar el contenido anterior a la línea actual, tenemos el contador de líneas, simplemente metemos al registro rsi el buffer del archivo y vamos viendo si ya llegamos a la línea actual contando la cantidad de “enters”, ya que en memoria tenemos el número de la línea actual. Con todo el contenido guardado en sus respectivas memorias, guardamos cada contenido en un solo buffer, de esta manera abrimos el archivo en modo escritura y sencillamente aplicamos los cambios. Una vez hecho eso, limpiamos las memorias y saltamos nuevamente a abrir el archivo.

Restricciones del programa

- Los archivos deben estar en el mismo directorio.
- Los archivos no deben sobrepasar 4kb, si lo hace, solo mostrará los primeros bytes.
- Los archivos deben estar previamente creados.

Conclusiones

1. La familiarización con NASM y su aplicación en el proyecto fue fundamental para el desarrollo del editor y comparador de texto. El uso de NASM para la arquitectura x86_64 permitió un control preciso y detallado sobre las operaciones a nivel de hardware, lo cual es esencial para el manejo eficiente de archivos y la ejecución de llamadas al sistema en un entorno Linux.
2. La fase de investigación fue crucial para comprender cómo otros editores de texto manejan sus funciones y flujos de trabajo. A pesar de tener una base teórica sólida, la necesidad de adaptación surgió a medida que se enfrentaban problemas prácticos, lo que demuestra la importancia de ser

flexible y estar dispuesto a modificar el enfoque según los desafíos encontrados.

3. El proyecto requirió un profundo entendimiento de varios conceptos de programación en ensamblador, como la manipulación de archivos, manejo de entradas de usuario, llamadas al sistema y control de cadenas y buffers. Estos conceptos, aunque en parte conocidos, exigieron un estudio más detallado y una lógica de programación a bajo nivel más rigurosa debido a la complejidad del proyecto.

Referencias

[1] “NASM.” https://www.nasm.us/
[2] Antirez, “GitHub - antirez/kilo: A text editor in less than 1000 LOC with syntax highlight and search.,” *GitHub*. https://github.com/antirez/kilo
[3] “Figma: The Collaborative Interface Design Tool.” https://www.figma.com/
” https://www.nasm.us/
[4] Nir Lichtman, “Making minimalist text editor in C on Linux,” *YouTube*. Nov. 28, 2023. [Online]. Available: https://www.youtube.com/watch?v=gnvDPCXktWQ
[5] “Mermaid | Diagramming and charting tool.” https://mermaid.js.org/
[6] “NASM assembly reading file provided via command line arg,” *Stack Overflow*. https://stackoverflow.com/questions/49181814/nasm-assembly-reading-file-provided-via-command-line-arg
[7]Torvalds, L., & Diamond, D. (2001). Just for fun: The story of an accidental revolutionary. Harper Business.