Proyecto 1: Sistema de monitoreo de desempeño programado en Ensamblador x86_64 para sistemas operativos de la familia Linux

Anasystem

Carlos Andrés Solano Artavia *2013044781*
Juan Pablo Ortiz Jarquin 201218772
Gill Carranza Otárola 2013101387
Randall Bonilla Vargas 2013018028

EL4313 – Lab. Estructura Microprocesadores
2do Semestre 2016
Tecnológico de Costa Rica

***Abstract*—The following document shows the development of the solution to a monitoring program of performance and system information implemented in assembly language x86_64 for linux family operating systems.**

***Resumen*—El siguiente documento muestra el desarrollo de la solución a un programa de monitoreo de desempeño e información del sistema implementado en lenguaje de ensamblador x86_64 para sistemas operativos de la familia linux.**

I. INTRODUCCIÓN

El objetivo principal del proyecto es implementar en un programa que monitoree el rendimiento del sistema, así como desplegar información del sistema de arquitectura x86 de 64 bits con un sistema operativo Ubuntu programado en lenguaje ensamblador x86_64. En este documento se encuentra toda la información pertinente al desarrollo, proceso e implementación de la solución, resultados y retos encontrados a lo largo del desarrollo del proyecto.

II. MARCO TEÓRICO

II-A. Ambiente computacional

En este proyecto se trabaja sobre una arquitectura x86_64, esta arquitectura x86_64 es comúnmente utilizada en la actualidad, es la versión de 64 bits del conjunto de instrucciones x86. Soporta una cantidad mucho mayor de memoria virtual y memoria física de lo que le es posible a sus predecesores, permitiendo a los programas almacenar grandes cantidades de datos en la memoria, también provee registros de uso general de 64 bits y muchas otras mejoras. [9].

II-B. Ambiente de trabajo

Ubuntu es un sistema operativo basado en GNU/Linux y que se distribuye como software libre. Está compuesto de múltiple software normalmente distribuido bajo una licencia libre o de código abierto. Una de las mayores ventajas de los sistemas operativos Linux como Ubuntu es la existencia de comunidades en la web que ayudan al manejo y comprensión de estos SO, también es sumamente estable, flexible y con un manejo óptimo de los recursos [9].

Una de las herramientas utilizadas en el proyecto es un ensamblador, se procede a utilizar NASM (Netwide Assembler) que consiste en un ensamblador libre para las plataformas 80x86 y x86_64. La instalación de NASM se realiza con los siguientes comandos:

- sudo apt-get update.
- sudo apt-get install nasm.

Otra herramienta utilizada es el editor llamado NANO, para la instalación de esta herramienta se utiliza el comando sudo apt-get install nano, de esta forma se observa en la terminal el proceso de instalación que toma sólo algunos segundos.

Por último es necesario un debugger, GDB es una herramienta que permite ejecutar paso a paso un programa sobre el Linux, permite visualizar el estado de diferentes registros, variables y estructuras dentro del microprocesador en cualquier momento (con la utilización de puntos de parada o breakpoints escritos por

el programador en partes específicas donde se desea conocer la información contenida).

La instalación se está herramienta se realiza bajo los siguientes comandos:

- sudo apt-get update.
- sudo apt-get install gdb.

Esto se muestra en la siguiente imagen.

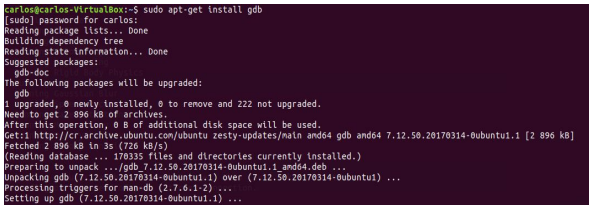

Figura 1. Instalación del paquete GDB.

II-C. Ensamblador

El lenguaje ensamblador, es un lenguaje de programación de bajo nivel. Consiste en un conjunto de mnemónicos que representan instrucciones básicas para los computadores, microprocesadores, microcontroladores y otros circuitos integrados programables. Implementa una representación simbólica de los códigos de máquina binarios y otras constantes necesarias para programar una arquitectura de procesador y constituye la representación más directa del código máquina específico para cada arquitectura legible por un programador [9].

Cada arquitectura de procesador tiene su propio lenguaje ensamblador que usualmente es definida por el fabricante de hardware, y está basada en los mnemónicos que simbolizan los pasos de procesamiento (las instrucciones), los registros del procesador, las posiciones de memoria y otras características del lenguaje. Un lenguaje ensamblador es por lo tanto específico de cierta arquitectura de computador física o virtual [9]. En este proyecto se utiliza el lenguaje ensamblador x86_64.

Toda la información está almacenada en los registros, un registro es un espacio de almacenamiento disponible para el CPU. Una de las principales características de estos, es que pueden ser accesados más rápido que cualquier otro dispositivo de almacenamiento de una computadora.

Los procesadores de la familia x86 cuentan con una serie de registros disponibles para almacenamiento temporal de variables, valores y demás información que utilizan durante la ejecución de instrucciones además de punteros a secciones de memoria como la pila [9].

Podemos mencionar cuatro diferentes tipos de registros:

1. Registros generales.
2. Segmentos de registros.
3. Flags (banderas de estado).
4. Instruction Pointer (IP) , puntero a la próxima instrucción a ejecutar.

Para la arquitectura en cuestión existen los registros que se enmarcan en rojo en la siguiente figura, en esta se muestra también el tamaño de los mismos, tipo y modos de compatibilidad.

Register or Stack	Legacy and Compatibility Modes			64-Bit Mode		
	Name	Number	Size (bits)	Name	Number	Size (bits)
General-Purpose Registers (GPRs)	EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP	8	32	RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8–R15	16	64
128-Bit XMM Registers	XMM0–XMM7	8	128	XMM0–XMM15	16	128
64-Bit MMX Registers	MMX0–MMX7	8	64	MMX0–MMX7	8	64
x87 Registers	FPR0–FPR7	8	80	FPR0–FPR7	8	80
Instruction Pointer	EIP	1	32	RIP	1	64
Flags	EFLAGS	1	32	RFLAGS	1	64
Stack	–		16 or 32	–		64

Figura 2. Registros de la arquitectura x86_64. [11]

Se debe tener un orden y claridad en la utilización de los registros para optimizar la eficiencia del desarrollo de la solución y un mayor provecho de los recursos, es por esto que en la siguiente figura se muestra el nombre y número de los registros y sus propósitos.

Registro		Propósito
#	Nombre	
0	rax	Registro acumulador: Acarrea el resultado o el código de llamada de sistema a usar
1	rbx	Registro base
2	rcx	4to Argumento de llamada de sistema / registro contador
3	rdx	3er Argumento de llamada de sistema / paso de datos
4	rsi	2do Argumento de llamada de sistema
5	rdi	1er Argumento de llamada de sistema
6	rbp	Base Pointer – Indica el inicio del segmento de código en el stack
7	rsp	Stack Pointer – Indica la posición actual de la ejecución en el segmento de código
8	r8	Propósito variado Paso de argumentos adicionales Recolección de resultados
9	r9	
10	r10	
11	r11	
12	r12	
13	r13	
14	r14	
15	r15	

Figura 3. Propósito de los registros de arquitectura x86_64. [11]

III. SOLUCIÓN IMPLEMENTADA

III-A. Descripción de la solución

Para la implementación de la solución se define una estructura para el programa, de forma que permita optimizar y minimizar el procedimiento, además de facilitar la utilización al usuario.

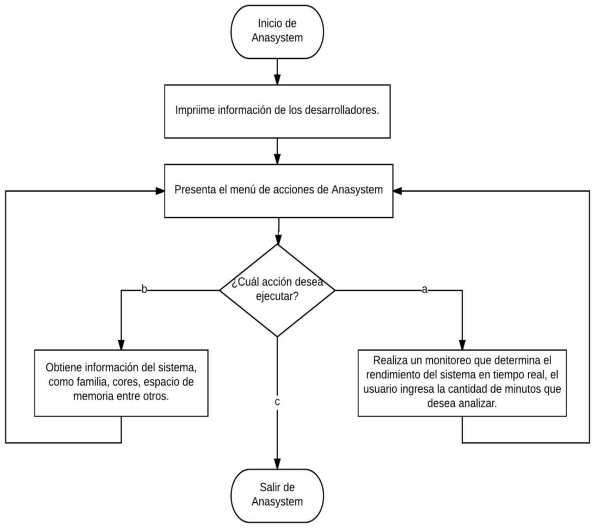

Figura 4. Diagrama de flujo principal de ANASYSTEM

Como se observa en el diagrama de flujo anterior, el programa inicia y muestra en pantalla la información de los desarrolladores, posteriormente se despliega en pantalla el menú de acciones del programa, llamado "Anasystem", en este menú de acciones se presentan 3 opciones, la primera opción recopila información del sistema como características del procesador, espacio de memoria, frecuencia, ram, entre otros, la segunda opción se dirige a una sección de programa que realiza un monitoreo de rendimiento al sistema. La tercera y última opción permite la salida del programa.

Lo anteriormente mencionado, es una visión macro del programa, en las siguientes secciones se detalla cada parte de manera específica.

1. Creación de macros para las rutinas.

Una macro es una sección de rutina que puede ser accesada en cualquier momento de la rutina, mediante el llamado (nombre de macro más parámetros necesarios) de la macro, esto permite ahorrar muchas líneas de código en el programa final, pues en una sola línea de código se realiza un proceso completo, como ejemplo se tiene la macro imprimir_texto_guarda, que despliega en pantalla información y guarda en un archivo .txt los resultados (ver sección III-A.2). El pseudocódigo que describe este proceso se muestra a continuación.

```
Se define el macro
Recibe 2 parámetros
Se hace la llamada al sistema "sys write"
Se mueve al registro rdi el destino: "std out"
Se mueve al registro rsi el primer parámetro: Texto
Mover al registro rdx el segundo parámetro: Tam_texto
Se llama al sistema
Se hace la llamada al sistema "sys write"
Se mueve al registro rdi el destino: puntero con dirección del archivo
Se mueve al registro rsi el primer parámetro: Texto
Mover al registro rdx el segundo parámetro: Tam_Texto
Se llama al sistema
```

2. Creación de archivo txt para guardar resultados.

Como se indica en el instructivo, uno de los objetivos del proyecto es que el usuario pueda acceder a la información de resultados desde un archivo que almacene los valores obtenidos en cada análisis. Para ello se crea un archivo .txt llamado "RESULTADOS", en el cual se almacena toda aquella información que se despliega en pantalla, necesaria para comprender el análisis. Para lograr esto se realiza conforme el siguiente procedimiento.

```
Se inicia bloque de inicio
Se guarda en rdi la variable "resulttxt"
Se carga (2000o+1000o+100o+2o) a rsi y
(700o+40o+4o) para abrir el archivo
RESULTADOS.txt
Se llama el sistema
El puntero "result_fd" al registro rax
```

Durante la ejecución del programa se recopila y almacena la información en el .txt ya mencionado, para finalmente cerrar el documento .txt con el siguiente procedimiento.

```
Se limpia el registro rdi
Se carga un 3 a rax
El registro rsi toma el valor de la dirección del puntero
"result_fd"
Se llama al sistema
```

3. Creación de subrutinas.

Para este proyecto, debido a la complejidad y extensión del procedimiento para lograr una óptima solución, se decide implementar el método de subrutinas, que consiste en la creación de rutinas de ejecución por separado, las cuales son llamadas (mediante call) en la rutina principal, cabe destacar que todas las constantes y variables a utilizar en las subrutinas, deben estar debidamente definidas en la rutina principal, de lo contrario se producirán errores, además se debe tener en cuenta que dichas subrutinas modificarán valores de registros que contengan información antes de el llamado de la subrutina, esto indica que se debe tener un manejo claro y correcto de los registros que se están utilizando.

En este proyecto se crearon únicamente dos subrutinas, una que contiene todo lo necesario para la ejecución del monitoreo de rendimiento (Rendimiento) y otra que contiene lo necesario para la obtención de la información del sistema (Familia), las cuales se explican con detalle en las secciones 4 y 5 respectivamente. En la siguiente figura se explica el funcionamiento de una subrutina y los pasos básicos que debe seguir.

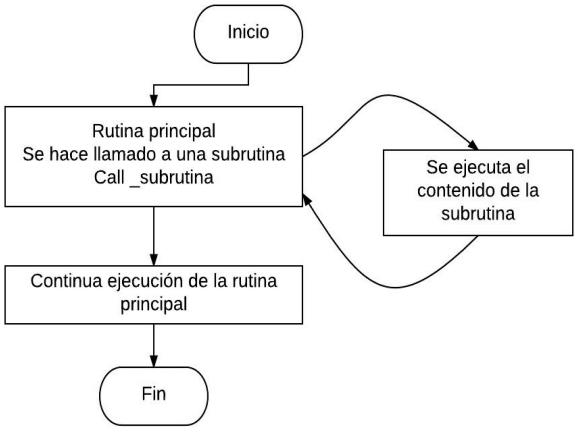

Figura 5. Diagrama de una subrutina.

4. Rutina principal.

La rutina principal es aquella que contiene los macros, la definición de variables y constantes, entre otros. Para este proyecto se crea la rutina principal denominada “Final2” la cual tiene la siguiente estructura:

- Inicialmente se encuentran todos los macros que se utilizaran en las subrutinas y rutina principal, como impresión de texto, lectura de teclado entre otras macros.
- Seguidamente se encuentra el sección . data, que contiene todas las constantes que se van a utilizar, como texto a utilizar, entre otros.
- Luego se encuentra la sección .bss, en esta sección se encuentran las variables que sirven para lectura de teclado, las cuales reservan un espacio de memoria para dicha lectura. También incluye constantes globales, es decir, que tienen un valor definido, puede ser un valor numérico.
- Finalmente se encuentra la sección .text, donde se encuentra todo el código que permite la ejecución del programa, llamadas de macros, llamadas de subrutinas, loops, entre otros.

Para la implementación de la rutina principal, se utiliza el control de flujo, que permite acceder a partes específicas del código, ya sea una de las acciones del menú o secciones dentro de subrutinas, esto es posible mediante la utilización de etiquetas y saltos. A nivel de código, las etiquetas son referencias que apuntan a la ubicación en memoria de una instrucción en particular. El contador de programa (PC) puede apuntarse a una etiqueta en cualquier momento durante la ejecución del programa, mediante los saltos [10]. Esto permite controlar el orden de ejecución del programa.

Como ejemplo de esto se muestra el procedimiento en pseudocódigo del despliegue del menú de acciones, su escogencia y el proceso para saltar a un bloque específico.

```
Se llama el macro de imprimir texto
Se imprime el menú de acciones
Entra a bloque _parche
Se limpia el registro r11
El puntero menu apunta a la dirección de r11
Se llama al macro de leer_pantalla
El registro r11 toma el valor del puntero menu
Se compara r11 con 0x61 (letra a en ascii)
De ser iguales, salta a _jill (Subrutina Familia)
Se compara r11 con 0x62 (letra b en ascii)
De ser iguales, salta a _primero (Subrutina Rendimiento)
Se compara r11 con 0x63 (letra c en ascii)
De ser iguales, salta a _final (Finaliza programa)
Si ninguno es igual, salta a _parche
```

5. Subrutina de información del sistema.

5.1 Obtención de las características del procesador.

Para obtener la información sobre el procesador, ya sea: Familia, modelo, tipo de procesador, stepping y sus extensiones respectivas es necesario asignarle el valor de un "1" al registro EAX y después de esto, definir el rango de los bits a analizar, pues como se puede observar en la imagen que se muestra a continuación, dependiendo del rango de estos bits en el registro EAX, se puede obtener toda la información necesaria sobre el procesador.

Por ejemplo, si se desea obtener el número de modelo del procesador, se debe hacer un llamado al cpuid, es decir; se carga toda la información que éste posee , asignando a la vez un "1" al registro EAX. Seguidamente se le aplica una máscara "0x00F0" a este registro para rescatar del bit 4 al 7, luego se aplica un desplazamiento hacia la derecha de 4 bits (que sería el número de bits equivalente a [4:7] del número del modelo) para ser convertidos a ascii y finalmente proyectado en pantalla para el usuario.

Lo anteriormente descrito se puede comprender con mayor facilidad con la ayuda del pseudocódigo que se muestra a continuación:

```
Sección .data
Se define una variable que contiene los números en
hexadecimal.

Sección .bss
Se define variable "un_byte" con tamaño de un byte

Sección .text
Se le asigna al registro EAX un "1"
Se hace el llamado al cpuid
Mover al registro RAX el contenido del registro r8
Mover el registro EAX a EDX
Aplicar la máscara (0x00F0) a EDX
Desplazar EDX, 4 bits a la derecha
Se carga la tabla definida en la sección de datos
Mover dl al registro al
Aplicar el comando xlat
Guardar el resultado en la variable "un_byte"
Llamar al sistema (interrupción 80)
Imprimir en pantalla
```

Para encontrar el resto de la información presente en la figura X, se realiza el mismo procedimiento que se utilizó para averiguar el número del modelo del procesador, sólo que las máscaras van a ir variando dependiendo de los bits que se necesiten para dar con dicha información y por ende, la cantidad de bits del corrimiento a realizar, también será modificada.

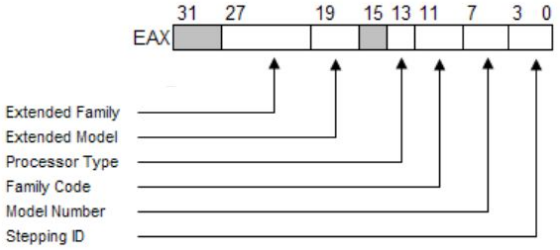

Figura 6. Resultado del CPUID siendo EAX=1.

Finalmente los datos obtenidos en la información del sistema, se pueden consultar en [1], pues en el CPUID es donde se encuentra la información respectiva según sea los datos obtenidos durante el análisis.

5.2 Obtención del tamaño total y disponible del disco duro.

Con el objetivo de conseguir el valor del tamaño total del disco duro presente en el computador, es necesario primero conocer las particiones del disco duro, lo cual se logra con "cat/proc/partitions". Para este caso en análisis fue sólo una, SDA, por lo tanto; si se quiere obtener el tamaño total del disco duro se hace un llamado a la función "*/sys/class/block/sda/size* " y este dará el valor en bloques.

Para una mejor asimilación sobre esta información anteriormente descrita, se adjunta el pseudocódigo capaz de devolver el dato requerido.

```
Sección .data
Se define una variable '/sys/class/block/sda/size'

Sección .bss
Se le designa 100 bits a la variable definida.

Sección .text
Se le asigna a EBX la variable
Se le otorga un "5" a EAX (para abrir archivo)
Se le asigna un "0" a ECX
Llamar al sistema (interrupción 80)
Se le otorga el valor de 3 a EAX (para leer)
Se mueve EAX a EBX
Se le asigna el tamaño de la variable a ECX y EDX
Llamar al sistema
Se le asigna un "4" a EAX(para escribir )
Se le asigna un "1" a EBX
Se mueve la variable a ECX
Llamar al sistema
```

La utilización de los diferentes valores para el registro EAX son indispensables para poder visualizar el resultado en pantalla mediante el comando de interrupción de llamadas del sistema, primeramente se le da la dirección del archivo para abrirlo, luego se lee y finalmente se escribe en pantalla. Como se mencionó anteriormente el resultado del tamaño del disco duro se muestra en las unidades de bloques, por lo que resta hacerle unos cuantos cálculos para poder obtener el resultado en GB.

Para pasar de bloques a bytes, basta con multiplicar la cantidad de bloques por 512, ahora bien, se va a dividir este dato obtenido entre 1024, tres veces, esto por lo siguiente. La primera división hace que el dato pase a KB, si se divide otra vez; se obtendría en MB y finalmente una tercera vez corresponde al tamaño total del disco duro en GB. Estos cálculos se simplifican y se muestran en la Ec.1.

$$Tamaño\ del\ HDD_{GB} = bloques * \frac{512}{(1024)^3} \qquad (1)$$

Para calcular el espacio disponible en el disco duro se repite el mismo proceso del punto anterior. Es importante mencionar que debe haber un cambio del archivo que se va a abrir, pues si se desea obtener el tamaño disponible del disco duro se hace un llamado a la función "*/sys/class/block/sda/sda1/size* ". Una vez que se ejecuta esta función siguiendo el mismo procedimiento para el tamaño total del disco duro, el programa imprimirá en pantalla el tamaño disponible del disco duro en unidades de bloques.

5.3 Obtención del tamaño total de la memoria RAM y el espacio disponible de esta.

A nivel de ensamblador no existen estructuras de datos, pero si existen secciones de memoria donde se escriben las estructuras de datos, así que es necesario ir a estas direcciones, para encontrar el dato solicitado. Lo anterior, se ejemplifica en la figura 6.1, que muestra la estructura de la función sys_info.

Para obtener la información de la memoria RAM(Random Access Memory) se hace uso de la llamada a sistema #99 (0x63H) que se encarga de llamar a "sys_info" y este retorna una estructura de datos con información del sistema.

Una vez definida el área de la función "sys_info" donde se va a trabajar, en este caso la memoria RAM total y disponible, se procede a utilizar el siguiente pseudocódigo, el cual va a brindar la información especificada sobre la memoria RAM:

```
Sección .data
Se define una variable resultado1

Sección .bss
Se le designa 50  bits a la variable definida

Sección .text
se le asigna a RDI, resultado1
Se le asigna a EAX el llamado 0x63
Llamar al sistema (syscall)
Se le otorga el valor de la dirección donde se
encuentra a EAX (para  poder manipularlo)
Se mueve EAX a EDX
Llamar al sistema(syscall)
Se le asigna a R8 el contenido de EDX
Se realiza un shift a RDX (según se requiera )
Se aplica Mascara a EDX
Se realiza un shift a RDX (según se requiera )
Se lee EBX en la [tabla]
se mueve dl a al
Se llama a XLAT
Se mueve AX a un_byte
Se imprime y guarda dígito
.
Se repite proceso para diferentes máscaras y dígitos
```

Para comprender de mejor manera al anterior pseudocódigo, lo primero que se hace es definir "resultado1" con un espacio en memoria que contiene 50 bytes, luego, antes de llamar al sistema, se pasa la dirección de "resultado1" en el registro rdi, y se carga la llamada 0x63 (99 decimal), que corresponde a sysinfo, cuando se llama al sistema (syscall), se recolecta información del sistema y se escribe a partir de la dirección inicial de memoria reservada para "resultado1"

Una vez ensamblarlo y linkearlo, se puede correr desde GDB y poner un punto de chequeo en "_breakpoint"
Cuando GDB se detiene en el "_breakpoint", se puede imprimir los contenidos de "resultado" en formato hexadecimal usando el comando: x/x &resultado
Luego, por cada "enter" que se presiona, se imprime un nuevo bloque de memoria (4 bytes) de manera consecutiva, con lo que se facilita visualizar los contenidos de la memoria (que siguen de uno por medio la estructura mostrada en la figura 6.1) y las direcciones donde estos se encuentran para poder manipularlos y mostrarlos en pantalla mediante la utilización de máscaras y XLAT para recorrer nibble por nibble.

6. Subrutina de rendimiento.

6.1. Obtención del porcentaje de rendimiento del procesador.

Para obtener el porcentaje de rendimiento del procesador, se utilizó el comando "sys_info". De acuerdo a [6], y a la información facilitada por el profesor del curso, dicha llamada es la número 99 (0x63) y devuelve información variada del sistema.

Esta información es devuelta en forma de estructura, e incluye parámetros como la carga experimentada por el CPU del último minuto, de los últimos 5 minutos y de los últimos 10 minutos. También devuelve información sobre el tamaño de la RAM, del disco duro y la cantidad de espacio utilizado en ambas. Sin embargo, esta sección se enfocará en los primeros tres, específicamente en la carga del CPU por minuto.

Es importante tener en cuenta que, en ensamblador no hay estructuras de datos definidas como tales, pero si hay secciones de memoria donde se escriben datos de forma consecutiva, con las cuales se pueden emular dichas estructuras.

En el caso de "sys_info", se obtiene la siguiente estructura.

```
struct sysinfo {
long uptime; /* Cuantos segundos desde que inicio el
sistema */
unsigned long load1         /* Carga del CPU en el
ultimo minuto*/
unsigned long load5         /* Carga del CPU en los
ultimos 5 minutos*/
unsigned long load15        /* Carga del CPU en los
ultimos 15 minutos*/
unsigned long totalram; /* Tamano total de RAM */
unsigned long freeram; /* Memoria RAM disponible */
unsigned long sharedram; /* Memoria RAM
compartida */
unsigned long bufferram; /* Memoria usada en buffers
*/
unsigned long totalswap; /* Memoria de SWAP*/
unsigned long freeswap; /* Espacio de SWAP
disponible*/
unsigned short procs; /* Numero de procesos en
ejecucion*/
unsigned long totalhigh; /* Memoria extendida total*/
unsigned long freehigh; /* Memoria extendida
disponible*/
unsigned int mem_unit; /* Tamano de la unidad de
memoria */
};
```

Figura 6.1. Estructura de datos devuelta por "sys_info"

Al observar esta estructura, se puede observar que el primer reto para el software fue obtener correctamente todos los datos y almacenarlos en una variable con el espacio adecuado. Esto último con el fin de emular una estructura de datos, los cuales se puedan acceder de forma consecutiva en la memoria. A continuación, se muestra la rutina para obtener los datos en pseudocódigo:

```
sección .bss
define variable "resultado" (50 bytes de espacio)

sección .text
Mover a RDI la dirección de "resultado"
Carga en RAX el número 0x63 (99)
Llamar al sistema
Mover a RAX el valor de [Resultado+8]
```

Nótese que, fue necesario definir en la sección .bss se una variable llamada "resultado", en este caso de 50 bytes (consultar [11]) .

Antes de realizar la llamada al sistema, se pasa la dirección de "resultado" al registro rdi, para posteriormente cargar la llamada 99 (0x63), correspondiente a "sys_info".

Al realizar la llamada, se recolecta la información que se muestra en la estructura y se escribe a partir de la dirección inicial de memoria reservada para la variable "resultado".

El siguiente paso fue asociar las posiciones de memoria de la variable "resultado" con los datos obtenidos de la estructura descrita anteriormente. Para lograr esto, se utilizó GDB. Los datos obtenidos se muestran a continuación :

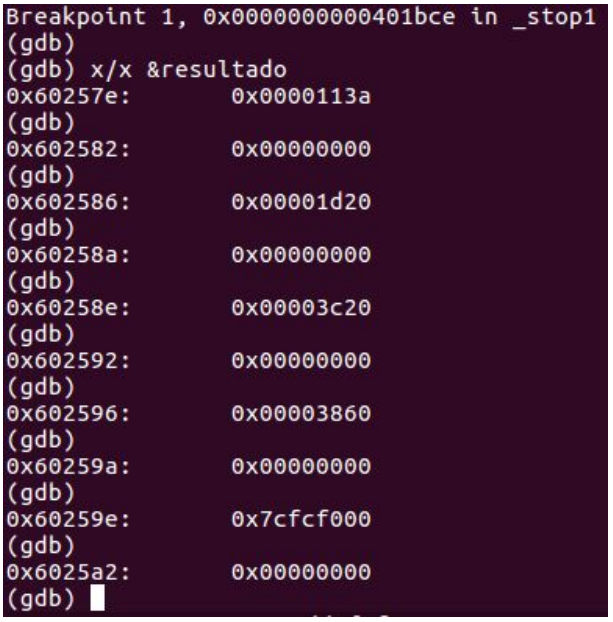

Figura 7. Datos obtenidos de GDB en la variable "resultado".

Al verificar los datos guardados en la variable "resultado", se observa que los datos se guardan de forma alterna, es decir, se guardan ceros después de cada posición de memoria en la que se guarda un dato proveniente de la estructura. .

Si se asume el orden de la estructura de datos, la primera posición corresponde al "uptime" y la tercera corresponde a "CPU Load / Minute" y así sucesivamente.

Por tanto, el dato de interés siempre estará en la tercera posición, 8 unidades después de la posición inicial de la variable "resultado". Como se observa en el pseudocódigo anterior, se guarda el dato que está en la octava posición de memoria en RAX.

Una vez asegurado el contenido correcto de la variable, se investigó sobre el significado del parámetro "CPU Load/min", ya que, como se observa en la depuración con GDB, es un número en hexadecimal que no indica directamente un porcentaje de rendimiento.

Según [3] , el dato obtenido de "sys_info" está en formato de punto fijo (Aritmetic Fixed-Point). Consultando en [8], un dato en Fixed-Point implica que, sólo un número fijo de dígitos, ya sea decimales o binarios, serán utilizados para expresar cualquier otro número. Incluyendo a los números que tienen una parte fraccional, a la par del punto decimal.
Para expresar este valor en en porcentaje, según [3], es necesario dividir el resultado obtenido entre $2^{16} = 65536$.

Finalmente, para obtener el porcentaje, se multiplica el resultado obtenido por 100. La operación y el pseudocódigo se detallan a continuación:

$$\frac{[Resultado + 8] \times 100}{65536}$$

sección .text

Mover RBX el valor de "cien"
Multiplicar RAX por RBX (Resultado en RAX)
Mover a RBX el valor de "scale"
Dividir RAX entre RBX (Resultado en RAX)
Mover a R8 el valor de RAX
Iniciar rutina de conversión a ASCII e impresión en pantalla

Luego de dividir entre el escalamiento, y la multiplicar por 100, es necesario convertir el dato en hexadecimal a un caracter ASCII que sea imprimible en pantalla. Dado que esta función se repite a lo largo de la implementación, se describe más adelante, en el apartado 6.4.

6.2 OBTENCIÓN DE LA FECHA.

Para obtener la fecha, se utilizó la llamada número 96, la cual según [11], corresponde a "sys_gettimeofday".
Esta llamada devuelve, según [4], devuelve la fecha y la hora en formato EPOCH, es decir, un dato de 32 bits con la cantidad de segundos transcurridos desde el 1ro de Enero de 1970 a las 00:00 hasta hoy.

Debido a esto, la parte investigativa es bastante reducida, en comparación al porcentaje de rendimiento, dado que dicho dato solo requiere de conversiones para poder imprimirse en pantalla. En consecuencia, la implementación requirió de mucho más manipulación matemática dentro del programa. Es por esto que en esta sección los pseudocódigos adquirirán una mayor relevancia.

En primer lugar, se describe la llamada al sistema utilizada. El código sigue la misma estructura de las llamadas generalmente utilizadas en el proyecto. Se describe a continuación:

sección .bss

define la variable "fecha" (5 bytes)

sección .text

Mover a RAX el valor de 0x60 (96)
Mover a RDI la dirección de "fecha"
Llamar al sistema

Como se puede observar, solo es necesario asignarle el valor 96 a RAX, y guarda la dirección de "fecha" en el registro RDI.

Utilizando GDB, se puede observar el valor EPOCH devuelto por la llamada al sistema:

```
(gdb) x/x &fecha
0x6025b0:       0x59bf4672
```

Figura 8. Valor EPOCH

A partir de este dato, se puede calcular el año, el mes y el día. También, se puede calcular la hora, los minutos y los segundos. Para esto, se utilizarán varias constantes, las cuales fueron obtenidas de [4] y se detallan a continuación:

Human readable time	Seconds
1 hour	3600 seconds
1 day	86400 seconds
1 week	604800 seconds
1 month (30.44 days)	2629743 seconds
1 year (365.24 days)	31556926 seconds

Figura 9. Equivalencias de EPOCH con tiempo en unidades del SI.

Estas constantes se definieron al inicio del código principal que se detalla en [2] en la sección .data. Para efecto de los pseudocódigos de esta sección, se definirán conforme se vayan requiriendo.

El primer parámetro que se calculó fue el año. Para esto, se utilizó el siguiente procedimiento:

$$\left(\frac{EPOCH}{31556926}\right) - 30$$

En pseudocódigo, se ve de la siguiente manera:

sección .data

define la constante "year"=31556926 (Cantidad de segundos en un año)

sección .bss

define variable "anho" (100bytes)

define la constante "ys70"=30 (cantidad de años transcurridos desde 1970 hasta el 2000.

sección .tex

Mover a RAX el contenido de "Fecha"
Mover a R12D el contenido de la constante "year"
Limpiar RDX
Dividir RAX entre R12D (Resultado en RAX)
Mover el resultado a R12D
Mover a R13D el contenido de la constante "ys70"
R12D-R13D (resultado en R12D)
Guardar el contenido de R12D en "anho".
Mover el contenido de R12D a R8D

Nótese que, es necesario limpiar el registro RDX, dado que la división no daba el valor correcto. Es necesario limpiarlo para garantizar que se obtendrán los valores correctos para poder continuar con el algoritmo.

Al finalizar el algoritmo, el valor del año calculado se guarda en la variable "anho". Esto con el fin de utilizar el año calculado en algoritmos posteriores. Finalmente, el valor del año queda listo en el registro R8D para ser utilizado en la subrutina de impresión en pantalla.

Para obtener el mes y el día, se calcula la cantidad de días del año que han transcurrido. Por ejemplo, al 15 de septiembre 2015, han transcurrido 258 días. Esto permite realizar un solo cálculo y en la rutina de impresión de pantalla obtener el número de días y el mes directamente. Este procedimiento se detalla en el apartado 6.4.

Es importante mencionar que el ajuste de años bisiestos, no es necesario realizarlo todavía, ya que el cálculo parte del número de años transcurridos desde el 70 y no del total de días, por lo que el cálculo no requiere tomar en cuenta este ajuste.

El cálculo utilizado es el siguiente:

$$\frac{EPOCH - [31556926 \times (30 + 17)]}{86400}$$

El pseudocódigo se ilustra a continuación:

sección .data

define la constante "day"= 86400 Número de segundos en un día

sección .bss

define la variable "dia"

sección .text

Mover a R11D el contenido de "fecha"
Mover a R12D el contenido de "anho" o R8D
Mover a R13D el contenido de "ys70"
R12D+R13D (Resultado en R12D)
Mover a EAX el contenido de "year"
Multiplicar EAX por R12D (Resultado en EAX)
Mover el resultado anterior a R12D
R11D - R12D (Resultado en R11D)
Mover R11D a EAX
Mover a R14D el contenido de "day"
Dividir EAX entre R14D (resultado en EAX)
Mover el resultado anterior a R15D
Guardar el contenido de R15D en "dia"

El código realiza la secuencia detallada en la ecuación. Se puede notar como el registro acumulador EAX es actualizado constantemente, ya que se utiliza en las multiplicaciones y divisiones como operador principal. También, se puede notar el uso de varios registros distintos a los de la subrutina del cálculo del año. Esto tiene un objetivo que se discutirá en el apartado de Limitaciones y Recomendaciones.

6.3 OBTENCIÓN DE LA HORA

Para realizar los cálculos de la hora se utilizó el mismo algoritmo detallado anteriormente, el cual es la llamada al sistema que devuelve el formato EPOCH. Se optó por realizar la llamada nuevamente con el objetivo de obtener el dato más actualizado posible.

Posteriormente, se desarrolló todo el cálculo para la hora, minuto y segundo en una sola secuencia. El pseudocódigo de esta secuencia se omite debido a su extensión, pero basta con aplicar las instrucciones desarrolladas en los códigos anteriores, a las operaciones que se detallarán a continuación:

1. Se le resta al EPOCH la cantidad de años (en segundos):

$$Resultado\,1 = \{EPOCH - [31556926 \times (31 + Año)]\}$$

2. Se obtiene la cantidad de meses transcurridos (en segundos):

$$Resultado\,2 = 2629743 \times mes$$

3. Esta cantidad se le resta a lo obtenido en 1 (Resultado 1) :

$$Resultado\,3 = Resultado\,1 - Resultado\,2$$

4. Se obtiene el número de días del mes transcurridos hasta el momento (en segundos):

$$Resultado\,4 = (\#dia - num_{dia}) \times 86400$$

5. El resultado obtenido en 4 (Resultado 4) se le resta al obtenido en 3 (Resultado 3):

$$Resultado5 = Resultado\,3 - Resultado\,4$$

6. Se obtiene la hora:

$$Hora = \frac{Resultado\,5 - Ajuste\ de\ Horario}{60}$$

7. Se obtiene la cantidad de horas transcurridas en segundos:

$$Resultado\,6 = Hora \times 3600$$

8. Al resultado obtenido en 5 (Resultado 5), se le resta el obtenido en 7 (Resultado 6):

$$Resultado\,7 = Resultado\,5 - Resultado\,6$$

9. Se obtienen los minutos:

$$Minutos = \frac{Resultado\,6}{60}$$

10. Se obtienen los segundos:

$$Segundos = Resultado\ 7 - (Minutos \times 60)$$

En la secuencia anterior es importante aclarar varias cosas. La primera es que, a nivel de código, las constantes deben definirse en la sección .data.

Las variables #dia y num_{dia} guaran el valor de la cantidad de días transcurridos desde que comenzó el año y la cantidad de días que han transcurrido al 1er día de cada mes.

Se requiere utilizar un ajuste de la zona horaria, que para Costa Rica es de 9hrs respecto al Meridiano de Greenwich. Nuevamente, no se utiliza el ajuste de días bisiestos, ya que estos están implícitos en la cantidad de años, al menos a nivel numérico.

6.4 IMPRESIÓN EN PANTALLA.

Para la impresión en pantalla se utilizaron las macros de Imprimir_Línea e Imprimir_texto facilitadas por el profesor del curso. A estas se les agregó un algoritmo de conversión de hexadecimal a ASCII.

Para el caso de imprimir el porcentaje, imprimir el número del año, el número del día mes, la hora, los minutos y los segundo se utilizó el mismo algoritmo. El pseudocódigo se detalla a continuación:

ETIQUETA:
Limpiar los registrso R10 y R11
Mover a R10 el dato de interés
Mover a R11 dato de comparación
Comparar R10 y R11 (Compara si R10>R11)
Si se cumple: Salta a la siguiente etiqueta.
Si no se cumple:
Imprime el número de las decenas
Se le resta una constante a R10
Se le asigna a AL el valor de R10b
Se utiliza XLAT para leer la tabla con el caracter correspondiente al remanente de la resta.
Se imprime el caracter.
Saltar a ETIQUETA FINAL

Como este algoritmo se utiliza a lo largo de toda la parte de rendimiento, es de suma importancia limpiar los registros R10 y R11.

En seguida, la comparación permite extender la utilización del algoritmo a varias cifras de dos dígitos. Por eso existe “ETIQUETA”.
Si se establecen varias etiquetas del tipo “mayor_que_xx”, se pueden imprimir un mayor rango de números. Por ejemplo, si el número en R10 es un 22, al realizar la comparación en el algoritmo de los números menores a 10 (el cual es el primero en la secuencia) , deberá saltar a la etiqueta “mayor_que_10”.
Aquí se realiza la nueva comparación y vuelve a saltar a “mayor_que_20”, donde se realiza la comparación nuevamente, pero esta vez, al no cumplirse se lleva a cabo el algoritmo. Finalmente, salta a una etiqueta final.

Para el caso de la fecha, se utilizan etiquetas hasta 40, dato que según [4] indica que el formato EPOCH genera overflow en 2038, igual se implementó para los segundos. Para las horas, se utilizó el formato de 24 horas, por lo que la etiqueta más grande es 30. Para los minutos y segundos se utilizaron hasta 60.

Para el caso de los meses, se utilizó un algoritmo similar al anterior. pero con unos pequeños cambios, estos se detallan a continuación:

ETIQUETA_MES
Limpiar los registrso R10 y R11
Mover a R10 el dato de interés
Mover a R11 dato de comparación
Comparar R10 y R11 (Compara si R10>R11)
Si se cumple: Salta a la siguiente etiqueta.
Si no se cumple:
Imprime el texto del mes (Ej: ENE,FEB,MAR)
Calcula la cantidad de días trancurridos desde que empezó el mes y los guarda en num_{dia}.
Calcula la cantidad de dias totales hasta el momento y los guarda en #dia.
Se le asigna a AL el valor de R10b
Se utiliza XLAT para leer la tabla con el caracter correspondiente al remanente de la resta.
Se imprime el caracter.
Saltar a ETIQUETA FINAL

Es en Código 6.G donde se observa de dónde provienen los valores necesarios para completar los cálculos de la hora, minuto y segundo. Es por esta razón constructiva a nivel de código que la fecha aparece antes que la hora.

III-B. LIMITACIONES Y RECOMENDACIONES

Limitaciones en Familia:

Una de las limitaciones generadas en esta sección "Familia.asm" se debió a que al trabajar con subrutinas las variables se definen en el top en este caso "Final2.asm", al correrlo de manera individual como esta en "INFO6.asm" (facilitado en el repositorio del proyecto), las características de espacio total en el disco las ejecuta exitosamente, sin embargo al ejecutarlo en conjunto dentro de "Final2.asm" este no despliega esta información debido a que nombre_archivo, no tiene ningún contenido, por ende la ejecución de la dirección que contiene este dato no se genera al estar en un subrutina.

Limitaciones en Rendimiento:

La primera limitación encontrada fue la interpretación del comando "uptime" en la consola de Linux, el cual imprime la hora, la cantidad de usuarios con sesión abierta y el CPU Load por minuto, por cinco minutos y por 15 minutos.

El significado de esos tres datos según [7] , representan la cantidad de solicitudes al CPU para realizar una acción. Los decimales representan el porcentaje de esta carga, y son proporcionales al número de Cores de la máquina utilizada. Es decir, si se está trabajando con solamente un Core, 1.00 equivale a un 100% y si son, por ejemplo, 3 Cores, 3.00 sería el 100%.

Es importante mencionar que, en ocasiones, al utilizar el comando "uptime" se retorna un número mayor a 1 (en el caso de estar utilizando un Core). De acuerdo a [13], esto puede interpretarse utilizando un puente de automóviles como analogía.

Cuando el puente está vacío, equivale a que no hay solicitudes de recursos para el procesador. Cuando hay carros circulando a través de él, equivale a un porcentaje $0 < \% < 100$ de utilización para el procesador. Si se tiene el un puente completamente lleno, equivale a un 100% de utilización del procesador. Finalmente, si el puente aparte de estar lleno, hay carros esperando para circular por él , equivale a un porcentaje mayor al 100%, por tanto implica que hay solicitudes en "espera" para ser ejecutadas por el procesador.

Fue importante conocer el funcionamiento de "uptime" para emular su estructura en el códgio de rendimiento. Otra inquietud con la que se trabajó fue sobre "sys_info" la cual trata sobre si el porcentaje cambia proporcionalmente a la cantidad de Cores utilizados por la máquina. Al realizar distintas pruebas, se pudo comprobar que el dato obtenido variará proporcionalmente al número de Cores en la máquina virtual.

La segunda limitación encontrada durante el desarrollo de los algoritmos de cálculo de la Fecha y Hora, es que los valores de las variables de almacenamiento a menudo eran corrompidos. La solución implementada fue aumentar el espacio asignado a las variables (100bytes).

Sin embargo, esto no fue útil para el cálculo de la Fecha, dado que las variables continuaron modificando sus valores sin razón aparente. Esta solución fue funcional para la parte de la Hora.

Para solucionar el problema de variables con la Fecha, se utilizaron dentro de los algoritmos para calcular el año, mes y día, distintos registros, con el fin de guardar los valores que se requerían en rutinas posteriores. Al utilizar esta solución, se recomienda planificar previamente los registros de cada subrutina y tener mucho cuidado para no sobreescribir aquellos registros que almacenan datos importantes.

La solución más recomendable para este problema es, realizar un estudio de las variables en memoria, para determinar en qué momento su valor está siendo modificado y por cual registro o variable. Esta solución no se pudo implementar en este proyecto por cuestiones de tiempo.

Una tercera limitación, fue la implementación de la Macro que convierte los datos en hexadecimal a ASCII. Esta limitación se relaciona con segunda, en el sentido de que, al terminar cualquier subrutina, un dato era movido de un registro a una variable de almacenamiento. Sin embargo al entrar a la macro, el dato era completamente diferente.

Se realizó una verificación exhaustiva del código, sin embargo no se logró determinar el origen del problema y por cuestiones de tiempo se decidió eliminar la macro de la implementación.

Esta decisión facilitó la implementación, en el sentido de que todas las conversiones se hacían inmediatamente después de las operaciones, por lo que se redujeron las pérdidas de valores al mínimo.

Sin embargo, esto produjo que el código se hiciera muy extenso, dado que algo que podía hacerse en una macro, tuvo que hacerse repetidas veces. Lo que dificulto el desplazamiento a lo largo del código y la identificación de errores de sintaxis.

Se recomienda implementar la solución del estudio de la memoria para determinar las posibles pérdidas y corrupción de información en las variables.

IV. RESULTADOS

En esta sección se muestran los resultados obtenidos por los desarrolladores, con el fin de validar la solución implementada.

Como se muestra en el diagrama de flujo de flujo de ANASYSTEM, al iniciar el programa, se despliega en pantalla información de los desarrolladores a modo de bienvenida del programa. Esto se observa de la siguiente manera.

FIgura 10. Inicio de ANASYSTEM

Posterior a esto se debe desplegar en pantalla el menú de acciones de ANASYSTEM, que como ya se mencionó, cuenta con tres opciones a escoger, análisis de rendimiento, información de sistema o salir. Esto se observa en la siguiente figura.

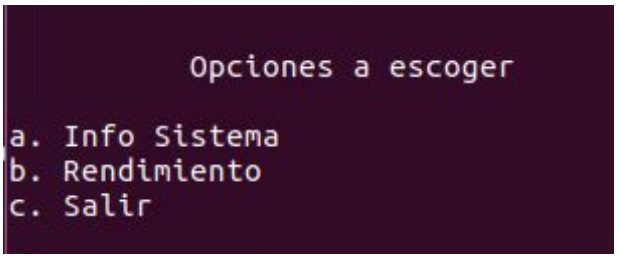

Figura 11. Menú de acciones de ANASYSTEM

Digitando la letra “a” el programa se dirige a “Info Sistema”, esta opción muestra en pantalla datos referentes a familia, núcleos, frecuencia, capacidad de memoria, entre otros, como se observa en la siguiente figura.

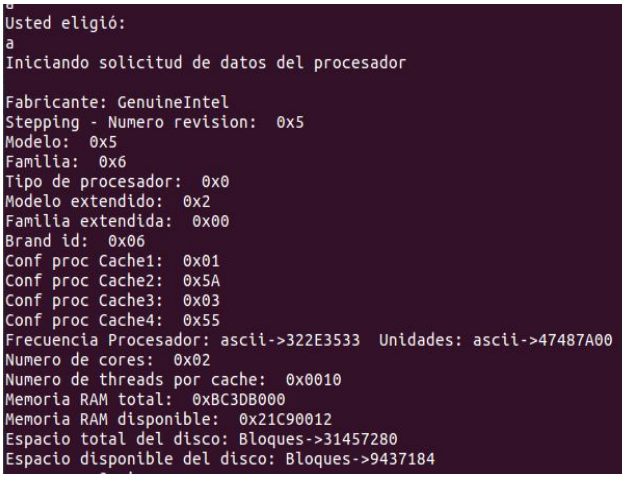

Figura 12. Resultados de la opción “Info Sistema”.

Digitando la letra “b” el programa se dirige a la opción “Rendimiento”, esta opción muestra en tiempo real, el porcentaje de rendimiento del sistema, con fecha y hora en que se realiza, con un periodo de muestreo de 5 segundos. Los resultados de esta opción se observan de la siguiente forma.

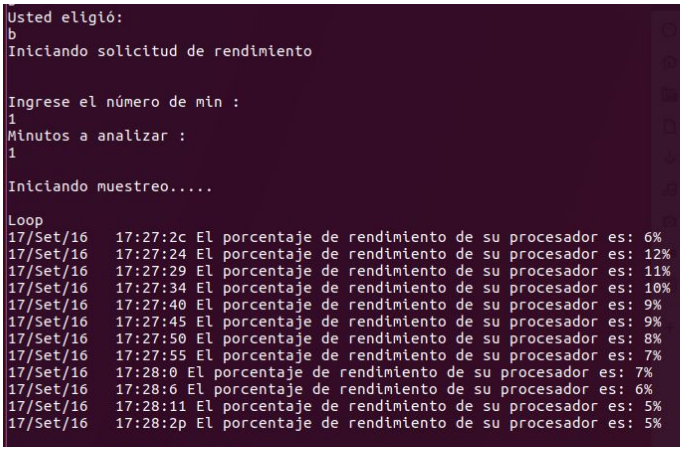

Figura 13. Resultados de la opción “Rendimiento”.

La última opción a escoger en el menú de acciones es salir, esta opción permite terminar el ciclo y salirse del programa. Esto se observa como se muestra a continuación.

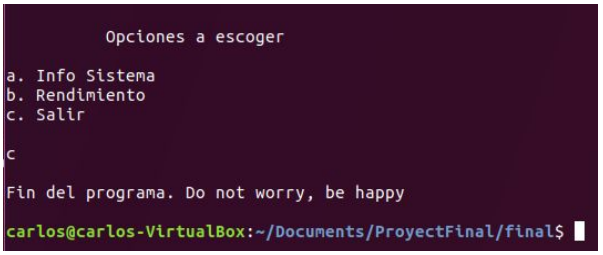

Figura 14. Resultados de la opción “Salir”.

Como requerimiento del proyecto, los resultados de ANASYSTEM deben almacenarse en un archivo txt, para ellos se debe crear el archivo (en este caso llamado RESULTADOS.txt), guardar la información y cerrar el archivo, todo durante la ejecución del programa. En la siguiente figura se muestra la carpeta que contiene el proyecto, inicialmente no debe tener el archivo RESULTADOS.txt, posterior a la ejecución del programa debe estar en dicha carpeta y contener los resultados.

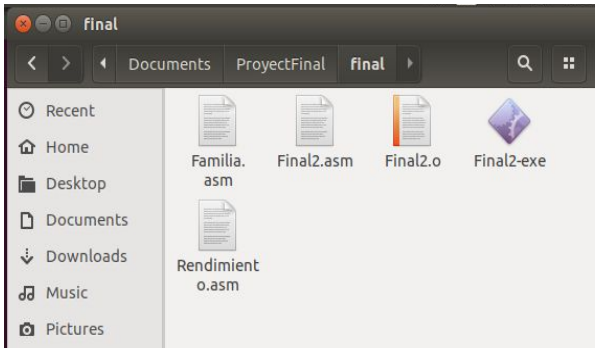

Figura 15. Carpeta que contiene el programa previo a la ejecución de ANASYSTEM.

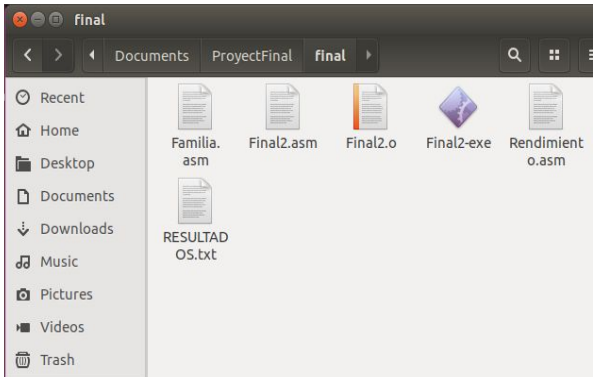

Figura 16. Carpeta que contiene el programa posterior a la ejecución de ANASYSTEM.

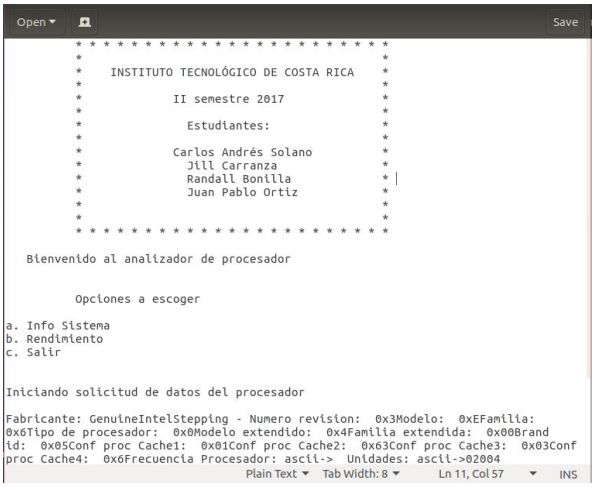

Figura 17. Resultados almacenados en RESULTADOS.txt para la opción “Info Sistema”.

Figura 18. Resultados almacenados en RESULTADOS.txt para la opción “Salir”.

Para la opción “Rendimiento” se presenta un error, pues no guarda de forma adecuada los resultados, esto se debe a un fallo en el macro de imprimir texto guardar, pues cuando recibe un parámetro, la longitud de la variable debe ser exacta, no un espacio guardado en memoria, posiblemente más grande que el contenido en sí. Esto ocasiona que se almacene “basura” que es tomada por el macro ocasionando el error.

V. CONCLUSIONES

- Se realizó una propuesta inicial en equipo, en la cual se tomó en cuenta los requerimientos de alto nivel y se desarrolló tomando en cuenta las habilidades de cada desarrollador.

- Se logró la implementación del programa de monitoreo de desempeño, cumpliendo con los requerimientos generales.

- Se logró la obtención de la información pertinente al sistema, la cual se muestra en pantalla en el análisis.

- Se creó un programa que realiza dos procesos de análisis del sistema, tanto rendimiento como información del sistema, ambos en un solo programa que se ejecuta de forma óptima.

- Se investigó y utilizó el método de subrutinas, de modo que permitió la unión sencilla de código de diferentes desarrolladores, además de código más legible y eficiente.

- El equipo de trabajo implementó de forma clara y ordenada la solución al problema dado.

VI. REFERENCIAS

[1] “Arquitectura x86 CPUID”. *Sandpile.org*. Disponible en el sitio web: http://www.sandpile.org/x86/cpuid.htm

[2] Bonilla, Carranza, Solano & Ortiz. Proyecto1_Lab_Micros,(2017), Repositorio GitHub, https://github.com/randall07/Proyecto1_Lab_Micros

[3] “Can not understand load returned by sys info”. 2012. Disponible en : https://stackoverflow.com/questions/12303141/can-not-understand-load-returned-by-sysinfo

[4] Epoch Converter - Unix Timestamp Converter", Epoch Converter, 2017. [Online]. Disponible en: https://www.epochconverter.com/.

[5] “Identificación del procesador y la instrucción CPUID”. (2009). *INTEL*. Disponible en el sitio web: https://www.microbe.cz/docs/CPUID.pdf

[6] Linux Manual, “sysinfo - return system information”. 2017. Disponible en: http://man7.org/linux/man-pages/man2/sysinfo.2.html

[7] Mombrea, Mathew. “How to interpret the CPU load on Linux”. Mayo 19, 2014. Disponible en: https://www.itworld.com/article/2833435/hardware/how-to-interpret-cpu-load-on-linux.html

[8] N. Gunther, "Understanding load averages and stretch factors", *Linux Magazine*, vol. 83, p. 67, 2017.

[9] P. Ramos, ¨Instrucciones, registros y operadores en x86”. Enero, 2014. Disponible en: https://www.welivesecurity.com/laes/2014/01/28/instrucciones-registros-operadores-x86/

[10] Primeros pasos con el lenguaje ensamblador x86_64 en Sistemas Operativos de base Linux. (n.d.). Cartago: Tecnológico de Costa Rica.

[11] R. Luis Carlos, "Manual de Apoyo", *Tecnológico de Costa Rica*, 2017.

[12] The NASM Development Team, ”NASM-The Netwide Assembler”. 2016. Disponible en: http://www.nasm.us/xdoc/2.12.02/nasmdoc.pdf

[13] “Understanding Linux CPU Load - when should you be worried?”. Julio 7, 2009. Disponible en:http://blog.scoutapp.com/articles/2009/07/31/understanding-load-averages