Proyecto 3: Asignación de memoria en un sistema real

Alfonso Murrieta Villegas[1] and Joaquín Valdespino Mendieta[2]

[1] *Universidad Nacional Autónoma de México, Facultad de Ingeniería, Ingeniería en Computación, Sistemas Operativos*

Fecha publicación: 7 de Mayo de 2020

Resumen— Todos los sistemas operativos de uso general utilizan un modelo de memoria basado en la paginación, y casi siempre memoria virtual. Este proyecto, consiste en obtener - extraer información de la memoria de procesos reales. Para ello la primera parte consiste en un programa al cual a través de un PID (Process ID) genera un mapa con toda la información asociada. Por otro lado, la segunda parte, es el presente trabajo escrito con toda la información, aprendizaje y documentación del proyecto.

Palabras clave— Paginación, MMU, Memoria Virtual, Memoria Física, PID

EJERCICIO 1

Descripción General

Para la primera parte del proyecto se desarrolló una re-implementación básica del comando 'pmap', que es básicamente obtener toda la información de un proceso a través de su PID.

Dependencias, Módulos y Lenguaje

Ambiente en el que fue desarrollado:

Recurso	Descripción
S.O.	Ubuntu 18.04
Lenguaje	Python 3.8
Bibliotecas(Python)	Qt5 , os

Para poder compilar el proyecto es necesario tener cualquier versión de python 3, y los respectivos módulos necesarios, como son el caso de os y Qt5.

Para el caso específico de Qt5 se puede utilizar el siguiente comando en linux:

```
apt install qttools5-dev-tools
```

o también puede usarse

```
apt install python3-pyqt5
```

Ejecución del programa

Para poder ejecutar el programa solamente es necesario escribir lo siguiente en terminal o usar el IDE de preferencia:

```
python3 Myapp.py
```

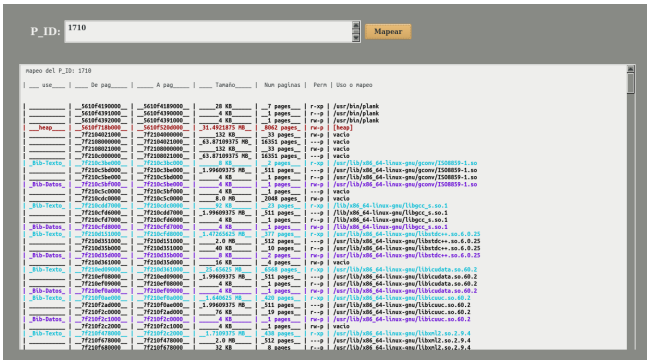

Fig. 1: Interfaz de Usuario

Una vez compilado el código, dentro de la interfaz de usuario, en el box al lado de P_ID escribir el ID correspondiente al proceso que quiera analizar y dar click en Mapear (Ver figura 1):

Descripción del proyecto y código

El proyecto contempla de manera general 3 archivos:

- **LineComplete.py**

  Lleva toda la lógica para la lectura de información, es la encargada de hacer la división de los datos a traves de una serie de atributos, así como guardar cada una de las lineas con la información correspondiente al proceso, por otra parte el calculo del tamaño y el numero de paginas a partir de los atributos base del constructor.

  Está basada en POO, donde sólamente encontramos una clase con 4 métodos, un método constructor de objetos (__init__), otro encargado meramente de la impresión de la información ya separada (troString), además de los 2 métodos de lógica principal gettypeUse y pageInfo.

- **Myapp.py**

  Basada en POO, contiene la clase principal encargada en gran medida de asociar toda la GUI (maps.ui) con LineComptle.py y algunos métodos propios.

  Tiene en total 4 métodos, de los cuales uno es el método constructor, otro que es el encargado de validar el PID que se mande a través del box de la GUI, el método mapping que es el que tiene toda la parte fuerte de lógica y código dentro de la clase, es la encargada de generar 2 archivos .txt donde se muestra el mapa asociado al proceso,por último, el método toPrint que es el encargado de imprimir todo la información del proceso en la GUI.

- **maps.ui**

  Es un archivo XML en UTF8 encargado meramente de tener todo el diseño de la GUI. En caso de querer redisẽnar la GUI, puede hacerse directamente en el archivo o empleando Qt 5 Designer.

- **NOTA:**

  Es importante destacar que la separación y asociación de colores para una mejor compresión de los datos que se mandan y leen en nuestra GUI se encuentran en el método toPrint de la clase Myapp ubicada en el archivo MyApp. la asociación del tipo de uso se realiza de la siguiente forma:
  - Bib-datos si esta contenida en /lib/... y tiene permisos de lectura y escritura
  - Bib-Texto si esta contenida en /lib/... y tiene permisos de lectura y ejecución
  - segmento de Datos si esta contenida en /home/... y tiene permisos de lectura y escritura
  - segmento de Texto si esta contenida en /home/... y tiene permisos de lectura y ejecución

Pruebas de Escritorio

A continuación se presentan los resultados obtenidos mediante nuestro programa, cabe destacar que para mayor detalle en el repositorio hay una carpeta denominada 'evidencias'.

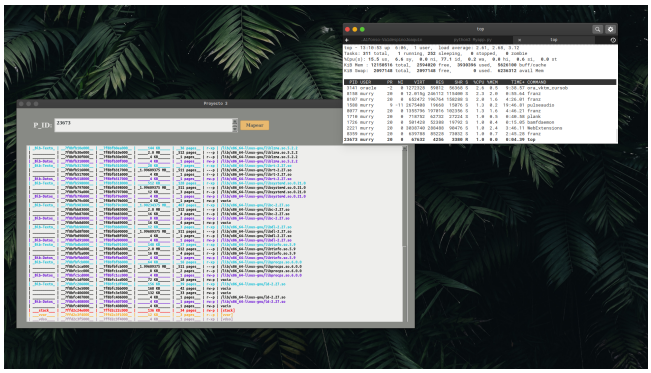

Fig. 2: Sistema: Elementary 5.1 | Arquitectura: x86-64 | Proceso de Franz

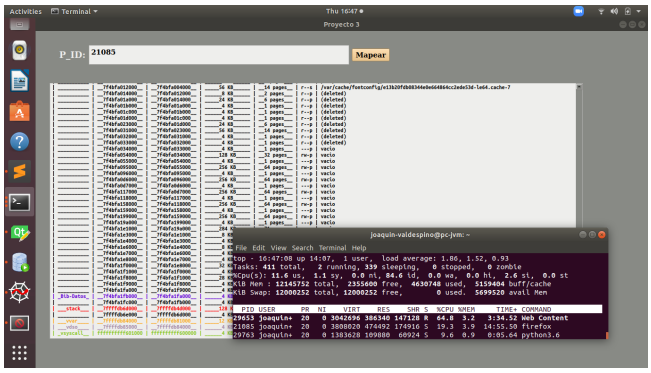

Fig. 3: Sistema: Ubuntu 18.04 | Arquitectura: x86-64 | Proceso de Firefox

EJERCICIO 2

Herramientas auxiliares

Para poder realizar el programa previo fue necesario tener en cuenta los siguientes comandos y herramientas tanto de forma auxiliar, de consulta y de comparación para la interpretación de los datos que poco a poco fuimos obteniendo e interpretando:

1. OBJDUMP

Sirvió particularmente para obtener la información desde archivos binarios, desde el análisis semántico hasta la traducción de este a código ensamblador. Principalmente fue usado para la comparación de los datos de una sección de memoria.

```
gcore -a PID
```

2. GCORE

Sirve sobre todo para la obtención de datos asociados a la memoria dedicada a un proceso, esto a través de su respectivo identificador o PID. Principalmente fue usado para la obtención del proceso en ejecución

```
objdump -s core.PID
```

3. NM

Sirve para dar el tamaño en bytes, el tipo de objeto, nombre y ámbito asociados a archivos de bibliotecas o ejecutables.

```
nm -f sysv fileName
```

4. CAT

Este comando sirve para mostrar elementos que ejecuta ese proceso y la respectiva región de memoria ocupada

```
cat /proc/{PID}/maps
```

Proceso y análisis

A continuación se menciona el análisis del material compartido del profesor además de cómo mediante esto se partió para poder realizar nuestro programa además de verificar nuestra salida.

1. A partir de la obtención del archivo ejecutable y del archivo generado por core.PID se revisaron las secciones de memoria (Ver figura 4 en el anexo), cabe destacar que genéricamente se usará PID en los comandos posteriores.

   ```
   $ ./donde_en_la_memoria
   $ gcore -a PID
   ```

2. Para conocer la ubicación de las cadenas o strings "Yo solo sé que no se nada ..." se empleó en primera instancia

   ```
   $ ./donde_en_la_memoria
   $ objdum -s core.PID
   ```

   El resultado tanto de la instrucción anterior como de la búsqueda se encuentra en la figura 5, donde apreciamos una sección de memoria denominada como "load2". Cabe también destacar que con otras herramientas se válido las direcciones de memoria.

3. Además, también se encontró y diferenció las cadenas de texto de origen respecto a las entregadas por la función printf

   Para esto,el resultado de la búsqueda se encuentra en la figura 6, donde apreciamos una sección de memoria denominada como "load3".

4. Por otro lado, se utilizó readelf para tener una herramienta auxiliar con la cual comparar directamente las direcciones de memoria previa respecto a lo que obtuvimos en nuestro programa (Ver figura 7 y 8).

   Aquí también se verificaron otros datos como el tamaño y permisos.

   ```
   $ readelf -Wa core.PID
   ```

5. Respecto a ubicar funciones, variables en la memoria se empleó nm sobre todo porque es mucho más legible y puntualizada la información obtenida y analizada de un proceso.

   Para ello se empleó el siguiente comando:

   ```
   $ nm -f sysv donde_en_la_memoria
   ```

   Como se puede observar en la figura 9, observamos algunos datos interesantes como la ubicación tanto de funciones como de variables y a su vez de otro tipos de datos denominados como OBJECT, NOTYPE.

   Por otro lado, también notamos algunos aspectos interesantes como que en la última columna de la figura 9 denominada como Section, notamos la diferencia entre las variables cadena1 y cadenatotal , a través de '.data' y '.bss'.

   De hecho, sabemos que se garantiza que la sección .bss será todo ceros cuando el programa se cargue en la memoria. Por lo tanto, cualquier dato global que no se haya inicializado o inicializado a cero se coloca en la sección '.bss' .

   Lo bueno de esto es que los datos de la sección .bss no tienen que incluirse en el archivo ELF en el disco (es decir, no hay una región completa de ceros en el archivo para la sección .bss). En cambio, el cargador sabe de los encabezados de sección cuánto tiene que asignar para la sección .bss, y simplemente lo pone a cero antes de transferir el control a su programa.

6. Por último, para diferenciar tanto símbolos y valores podemos nuevamente revisar la figura9 en la misma columna Section, realmente ahí encontraremos también algunos recursos que ocupa tanto el lanzador como el compilador al momento de ser ejecutado el programa.

   Una forma particular de obtener la tabla de símbolos (Esto como validación al estar buscando en la memoria), es nuevamente mediante el uso de objdump solo que en este caso tendrá que aplicarse directamente en el .o generado por el compilador

   ```
   $ objdump -t donde_en_la_memoria.o
   ```

CONCLUSIONES

Uno de los mayores retos en la creación de los Sistemas Operativos fue la gestión de memoria, sin duda alguna, la complejidad que contempla el que deba considerar cada procesos con sus respectivos subprocesos además de conceptos relacionados como memoria virtual, paginación y MMU, sin duda hace que este sea una de las características más relevantes en un S.O.

Como bien hemos visto en las últimas clases, los sistemas de tipo Linux al manejar prácticamente todo a través

de archivos es como se pudo manejar elementos para este proyecto como objdump .

COMENTARIO | DUDAS

Creemos que la zona de memoria estática permite almacenar variables globales al momento de ejecutar un programa, por otro lado también observamos que hay datos como 'variables locales' en la zona denominada como stack.

La distribución de los datos de nuestros programas nos parece muy distante entre ellos, lo cual nos hace pensar y reflexionar acerca de qué tan factible es esto para el rendimiento.

REFERENCIAS

1. The Geek Stuff. Practical Linux nm Command. Recuperado el 7 de Mayo del 2020, de https://www.thegeekstuff.com/2012/03/linux-nm-command/

2. HowtoForge. Linux objdump Command Explained for Beginners. Recuperado el 7 de Mayo del 2020, de https://www.howtoforge.com/linux-objdump-command/

3. QT. Qt Design Studio Manual. Recuperado el 5 de Mayo del 2020, de https://doc.qt.io/qtdesignstudio/index.html

4. Gunnar Wolf,etal. Fundamentos de Sistemas Operativos. Recuperado el 5 de Mayo del 2020, de http://sistop.org/pdf/sistemas_operativos.pdf

ANEXO DE IMÁGENES

Es importante destacar que todas estas imágenes se encuentran en la carpeta de evidencia

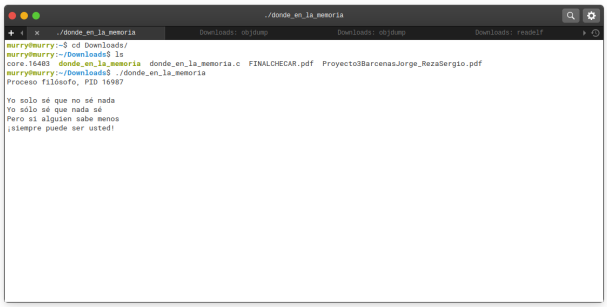

Fig. 4: Compilación y generación del core.PID

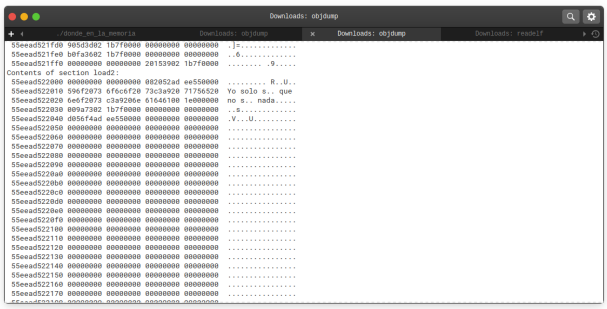

Fig. 5: Uso de objdump, se muestra la arquitectura de la PC así como el inicio del archivo core.16987

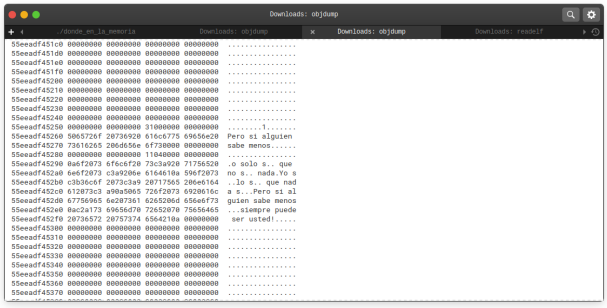

Fig. 6: Uso de objdump para la ubicación de algunas cadenas

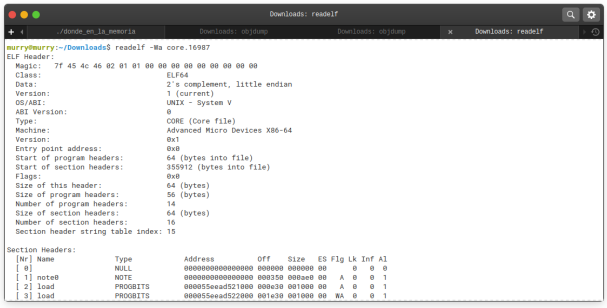

Fig. 7: Uso de readlef como alternativa a objdump para buscar direcciones

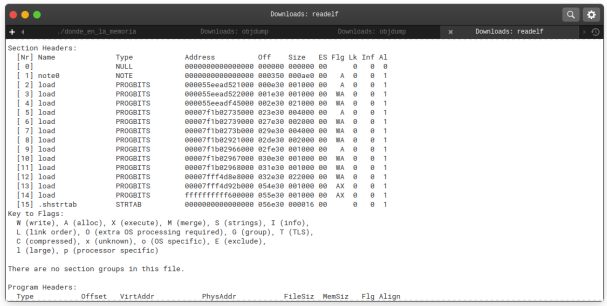

Fig. 8: Uso de readlef compara ver la dirección tanto física como virtual, además del desplazamiento y permisos

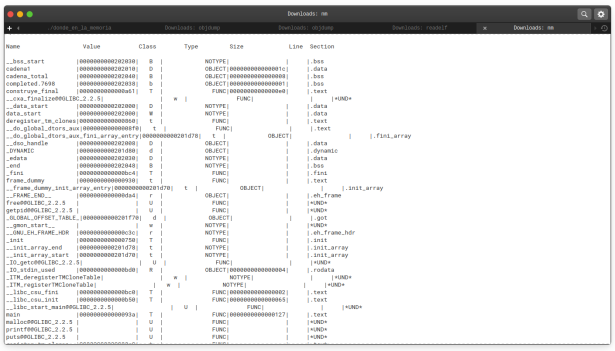

Fig. 9: Uso de nm para analizar el archivo ejecutable obtenido de la compilación, en específico las direcciones de memoria de las cadenas y otros aspectos relentes para el análisis requerido en este proyecto

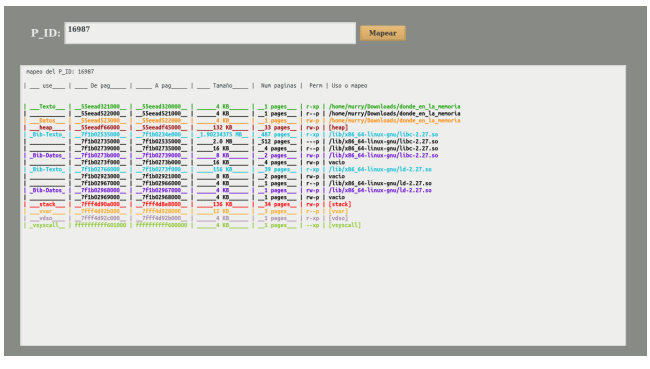

Fig. 10: Resultado final, obtenido con nuestro programa, se observa el buen funcionamiento y resumen respecto a objdump y readlef