Módulo gráfico de un Visualizador de Estructuras de un Sistema Operativo Educativo a través de GDB-Stub

Graciela De Luca[1], Martín Cortina[1], Nicanor Casas[1], Esteban Carnuccio[1], Sebastián Barillaro[1], Daniel Giulianelli[1], Pablo Barboza Carvalho[1], Ezequiel Calaz[1], Gabriela Medina[1]

[1] Universidad Nacional de La Matanza,
San Justo, Buenos Aires Argentina
{gdeluca, mcortina , ncasas, ecarnuccio, sbarillaro, dgiunlian}@ing.unlam.edu.ar
pbarbozacarvalho@hotmail.com, gmedina190@gmail.com, aecalaz@gmail.com

Abstract. En este documento se exponen los mecanismos desarrollados para la construcción de un visualizador de estructuras internas de un sistema operativo didáctico, tanto para la comunicación entre el sistema operativo y el visualizador, como así también los módulos gráficos que permiten ver las estructuras internas durante la ejecución. Estos han sido de gran utilidad para obtener como resultado el estado actual de los módulos de la aplicación, que se encargarán de representar gráficamente los componentes del sistema. Se detallan las metodologías empleadas para que las interfaces GUI del graficador puedan representar la información obtenida de SODIUM a través de GDB y scripts desarrollados en el lenguaje Python. El objetivo final es que el visualizador sea de utilidad en el ámbito académico, facilitando el aprendizaje a los estudiantes de materias afines.

Keywords: GDB, GDB-Stub, Python, Front-End, GDM/MI, Estructuras de un Sistema Operativo

1 Introducción

El proyecto de investigación tiene como objetivo final la construcción de una herramienta que facilite el aprendizaje y enseñanza sobre del funcionamiento interno de un sistema operativo convencional.

Esta aplicación está siendo primeramente desarrollada en base al Sistema Operativo SODIUM [7] debido a que el equipo de trabajo ya se encuentra familiarizado con el mismo, pero se espera que el resultado final sea compatible o adaptable a otros sistemas.

Es importante señalar que la premisa fundamental de este programa es la de permitirle al usuario visualizar gráficamente las estructuras lógicas utilizadas por el sistema operativo durante su ejecución a través de una conexión serial.

Seguidamente se describen los mecanismos que se han empleado hasta la fecha a lo largo del proyecto para intentar alcanzar la meta establecida.

Este documento se ha dividido en dos secciones. En la primera parte, se explica cómo se ha implementado en su totalidad el componente GDB-STUB [9] en SODIUM, gracias al cual ahora es posible conectar al mismo el depurador remoto GDB. También se describe el uso que estamos haciendo de una característica muy

interesante de este depurador, que consiste en ser altamente automatizable por medio de scripts. Con esto, logramos obtener el estado de las estructuras internas del sistema operativo desde otra terminal.

En la segunda sección se detalla el estado actual de los módulos gráficos que conformarán el visualizador, y de qué manera empleamos GDB y Python para su construcción.

2 GDB-STUB y scripts de Python

El pilar principal de este proyecto consiste en utilizar GDB para poder acceder al estado de las estructuras del sistema Operativo desde otra terminal. GDB ofrece dos herramientas para debuguear en forma remota: GDB Server y GDB-Stub. Estos mecanismos son utilizados con mucha frecuencia en los desarrollos de sistemas embebidos, donde la única forma de analizar la ejecución de un programa sobre un dispositivo físico es en forma remota. Como SODIUM no posee las características necesarias para implementar GDB-Server dado que carece de muchas bibliotecas que utiliza ese programa para poder funcionar, se decidió desarrollar el módulo GDB-Stub en este Sistema Operativo [2]. Debido a la complejidad de la implementación, ésta fue llevada a cabo en forma gradual. Las etapas iniciales de dicha adaptación se describen en [7] y [8]. Sin la implementación de este módulo en SODIUM la obtención de los estados de las estructuras del sistema operativo durante su ejecución habría sido mucho más extensa y dificultosa, dado que habríamos tenido que desarrollar nosotros mismos el trabajo que realiza el debugger para poder conocer el valor de los datos en tiempo real. Adicionalmente, el resultado final habría carecido de la flexibilidad que esta solución brinda.

Fig. 1. Relación entre los componentes involucrados.

Se puede observar que se sigue una arquitectura cliente-servidor, donde el lado cliente representa al visualizador y el lado servidor representa al sistema operativo.

Del lado servidor se puede ver que el módulo GDB-STUB que se encuentra integrado en el kernel del sistema operativo. En el lado cliente, se encuentra GDB junto al Visualizador y a los script de Python [5]. Más adelante se hablará acerca del módulo de GDB/MI del lado cliente.

Seguidamente se describen brevemente las principales adaptaciones que se realizaron en el código de SODIUM para poder integrar el módulo de GDB-stub.

Funciones agregadas en Sodium para implementar el módulo GDB-Stub.

En el código de SODIUM se comenzó construyendo un módulo *stub* en forma incremental, que incorporaba las funcionalidades esenciales que ofrece el archivo original *i386-stub.c* de GDB. Inicialmente no fueron desarrolladas en su totalidad, debido a conflictos con la implementación preliminar de driver de puerto serie que poseía nuestro sistema operativo.

Luego de varias modificaciones en el código de SODIUM, se consiguió implementar en su totalidad el módulo GDB-Stub en ese sistema. Una de las adaptaciones especiales que se debieron realizar, consistió en la creación de funciones especiales en diversos sitios dentro de los archivos fuentes de SODIUM.

- **set_debug_traps():** Esta función es usada para configurar los handlers que capturan la excepción 3 [4] [7], que ocurre cuando se ejecuta un breakpoint, y la excepción 1, qué sucede cuando se debuggea paso a paso. Dicho de otra forma, dentro de *set_debug_traps()* se invoca a la función encargada de modificar los descriptores dentro de la IDT que están asociados a las excepciones antes mencionadas. De manera que cuando ocurran de dichos eventos, el sistema automáticamente ejecuta los handlers que provee GDB para manejar esas excepciones. Es importante mencionar que fue necesario invocar a *set_debug_traps* al iniciar la función *main()* de SODIUM. Este hecho ocurre después de haber establecido la CPU en modo Protegido y de haber inicializado el driver de puerto Serie, condición fundamental para la comunicación con GDB.
- **exceptionHandler ():** Función intermedia que provee GDB-Stub, cuya tarea es hacer de intermediario entre *set_debug_traps()* y la función que modifica la IDT.
- **breakpoint ():** Internamente, esta función al ser invocada ejecuta un *"int 3"*. Esta línea de código assembler produce una excepción 3, y principalmente es utilizada para conectar GDB y con el sistema operativo SODIUM. Por dicho motivo fue necesario invocarla en el main luego *set_debug_traps.*
- **putDebugChar() y getDebugChar():** Se debió adaptar el driver de puerto Serie desarrollado en SODIUM, generando dos funciones para transmitir de a un carácter a la a vez entre la terminal cliente y Servidor, en lugar de una palabra completa, Esta adaptación fue necesaria dado que el módulo GDB-stub se encuentra diseñada para utilizar de esa manera el protocolo RSP [3].

Además, para lograr el funcionamiento del módulo GDB-Stub en su totalidad en SODIUM, fue necesario realizar un mecanismo híbrido, para poder utilizar el puerto serie por medio de su driver. El cual consistió en permitir a través de interrupciones la detención del sistema operativo por medio de la consola de GDB, De forma tal, que cuando SODIUM reciba en código ASCII un valor en hexadecimal "03" (carácter de control ctrl+c), el kernel invoque automáticamente a la función principal *handle_exception()* [7] para que controle la ejecución del sistema. Posteriormente, una vez que el stub toma el control, la transferencia de datos con GDB se lleva a cabo en modo polling.

Establecer la conexión entre GDB con el Stub de SODIUM en forma remota.

Para poder conectar GDB con SODIUM primeramente es necesario que el usuario inicie el sistema operativo, ya que funciona como un servidor. Una vez que el kernel inicializa los drivers del puerto serie e invoca a la función set_debug_traps(), ejecutará la función breakpoint() que le dará el control al módulo GDB-Stub, así el sistema quedará en una espera activa hasta que se establezca un enlace con GDB.

Por otra parte, el usuario desde el del lado cliente deberá ejecutar un script, que finalmente establecerá la conexión entre SODIUM y el debugger. Por consiguiente, en la siguiente figura se muestran los argumentos utilizados en el script, con los que se ejecuta el programa GDB para que se conecte en forma automática con el sistema operativo.

```
gdb 'target remote localhost:12345' -b $Baudios --symbols='kernel/main.ld' --command='comandos_nuevos.gdb'
```

Figura 2. Ejecución de GDB para establecer conexión con SODIUM

El primer argumento 'target *remote localhost:12345'* establece la conexión entre GDB y el sistema operativo, cuando este último se encuentra esperando activamente en su stub. A este parámetro se le debe indicar la IP y el puerto en donde estará escuchando el servidor. El segundo argumento, '*-b'* indica la velocidad en baudios que se va a utilizar durante la transferencia de los datos. El tercer argumento '*--symbols'* es utilizado para señalar cuál es el archivo que contiene la tabla de símbolos del kernel del sistema operativo. Finalmente el último argumento *--command*, indica en qué archivo se encuentran los comandos especializados de GDB que son creados particularmente para el visualizador. Más adelante en este documento se describe de qué manera ha sido utilizado en SODIUM.

Extensión de comandos GDB para el visualizador de estructuras

Una utilidad importante que ofrece GDB es la de permitirle al programador generar nuevos comandos a partir de los ya existentes en el debugger. Esta característica es esencial para poder desarrollar el visualizador, ya que nos permite generar el enlace entre GDB y el graficador a través de la concepción de nuevas instrucciones según nuestras necesidades. GDB ofrece distintos formas de extender su set de comandos[2]. Pero en este documento se mencionan únicamente los mecanismos que se han utilizado hasta la fecha durante el desarrollo del proyecto. Por ese motivo, a continuación se describen los métodos empleados en el transcurso de la investigación:

- **Archivo de Script de Comandos de GDB**

GDB le permite al usuario definir una secuencia de comandos específicos como una unidad y que luego este conjunto pueda ser ejecutado bajo un nuevo nombre de comando. Para su programación el debugger posee su propio lenguaje de scripting, con sus propias estructuras de control, como por ejemplo if, while, for y funciones. De esta forma el usuario tiene la posibilidad de generar bibliotecas con sus propios comandos de GDB. En la figura 2, se puede observar el argumento *--command* que se

le pasa a GDB, al ser ejecutado desde la consola de Linux. Este parámetro se utiliza para indicarle al debugger el nombre del archivo que contendrá los comandos que fueron personalizados por el usuario.

▪ **Archivo de Script de Python en GDB**

Una de las características más importantes que ofrece GDB es la de poder ejecutar scripts de Python desde su consola, pudiendo extender el conjunto de comandos del debugger utilizando este lenguaje. Para poder aprovechar dicho beneficio, fue necesario recompilar el código fuente de GDB, utilizando el flag --with-python durante su configuración. La forma de utilizar código Python dentro de GDB se puede realizar de dos maneras distintas. La primera forma es invocando al intérprete de Python desde el prompt de GDB. El segundo método es ejecutando el comando de GDB ***source*** junto al nombre del archivo del script de Python, por ejemplo "*source script_python.py"*

Cabe destacar, que GDB ofrece diferentes APIs para invocarlas en Python. Las cuales permiten ejecutar un comando específico de GDB dentro de un script desarrollado en ese lenguaje. De esta manera por ejemplo, se puede conocer desde Python el valor de una estructura del Sistema Operativo SODIUM durante su ejecución. Utilizando para ello la función *gdb.execute("print pstuPCB" ,0,1)*

Por consiguiente en base a las herramientas que ofrece GDB para poder extender sus comandos, se empezaron a desarrollar los módulos gráficos del Visualizador. Por lo tanto, en la segunda sección de este documento, se describen el estado actual del desarrollo de dicho componentes y de qué manera han sido utilizados estos mecanismos de extensión.

3 Desarrollo de Módulos Gráficos del Visualizador de Estructuras de un Sistema Operativo

Como se mencionó anteriormente, el objetivo de este proyecto es generar una herramienta que le permita al usuario poder observar las distintas estructuras que utiliza el sistema operativo SODIUM durante su ejecución. De forma tal, que le facilite a los estudiantes el aprendizaje teórico práctico del funcionamiento de los sistemas operativos convencionales. Por dicho motivo actualmente el desarrollo del visualizador se centra en dos módulos:

▪ **Módulo de visualización de estructuras de Sodium**

Este componente del visualizador se encargará de mostrarle al usuario, a través de interfaces GUI, el estado de las siguientes estructuras que utiliza el sistema operativo en un momento determinado: IDT, GDT, TSS y PCB. Además representará gráficamente el mapa de memoria que maneja el sistema operativo. Mostrando para ello la ubicación exacta de los segmentos en un instante específico.

▪ **Módulo de visualización de Diagrama Temporal**

Este módulo pretende permitirle al usuario poder observar en forma gráfica mediante el visualizador, los diferentes estados que van adquiriendo los procesos que van ejecutándose en SODIUM a lo largo de su vida. De esta se intenta que el

estudiante pueda observar la ejecución de distintos algoritmos de planificación de CPU en forma visual.

Es importante mencionar que en este documento se describe en detalle el desarrollo actual del primer módulo antes mencionado.

Embeber GDB en Python y GDB/MI

Una dificultad encontrada durante el desarrollo, consistió en la imposibilidad de importar la biblioteca que utiliza GDB en un programa desarrollado íntegramente en Python (*"import gdb.py"*). Lo que resultó ser un impedimento para construir un ejecutable del visualizador desarrollado en ese lenguaje, que permita acceder a la información que ofrece SODIUM a través de GDB. Este percance es debido a la existencia de un fallo en el código fuente entre GDB y Python [10], que impide hacer esto, que aún no ha sido solucionado. Por consiguiente, para intentar subsanar dicho obstáculo, se determinó que era conveniente basar el desarrollo de las interfaces gráficas ejecutando los scripts de Python del visualizador desde la consola de GDB. Utilizando para ello el comando *source,* como se había mencionado anteriormente.

Además se descubrió que GDB ejecuta sus comandos en un único hilo de ejecución. Por lo que al ejecutar un script de Python que muestra una interfaz GUI desde la consola del debugger, se imposibilita poder seguir la ejecución normal de SODIUM. Debido a que en ese momento el debugger se encuentra ejecutando el script de Python. Por dicho motivo se decidió analizar los mecanismos que utilizan los Front Ends que usan GDB y extraer su funcionamiento elemental para poder aplicarlo luego al visualizador. Estas herramientas permiten ejecutar una interfaz gráfica y en simultáneo continuar debuggeando una aplicación utilizando GDB/MI [2]. GDB/MI hace de intermediario entre la comunicación de GDB y los Front End, mediante una interfaz basada en texto protocolizado. Por dicho motivo, se está analizando el funcionamiento de un Front End en especial desarrollado en Python, denominado Pyclewn [11]. No obstante paralelamente, se está desarrollando cada módulo gráfico en forma modular como un conjunto de comandos de GDB. En donde cada uno de ellos ejecuta una interfaz GUI que realiza una tarea determinada. Luego una vez finalizado este proceso, se pretende integrar todos los submódulos gráficos aplicando los conceptos que adquiridos sobre el funcionamiento de los Front Ends.

En consecuencia, a continuación se describen el estado actual de las interfaces gráficas que componen el módulo de visualización de estructuras de SODIUM.

Módulo de visualización de estructuras de Sodium

A los fines de poder visualizar gráficamente las estructuras internas del sistema operativo SODIUM, se generó un archivo de comando de script de GDB (llamado "*comandos_nuevos.gdb").* El cual, tiene como objetivo generar un conjunto de comandos personalizados con la intención de extender las utilidades que ofrece el debugger. Como se muestra en la figura 2, el nombre del script debe ser pasado como parámetro a GDB al ser ejecutado desde Linux. Dentro de este archivo, se generaron nuevos comandos utilizando las técnicas de extensión previamente detalladas. En algunas de estas instrucciones, se invocan a archivos con script de Python, que se encargan de mostrar por pantalla la información solicitada por el usuario a través de interfaces GUI. Para ello, el código de las ventanas gráficas fue escrito en PyQT [6],

de acuerdo a como se había señalado en [8]. Además como SODIUM está desarrollado en base a la arquitectura X86, la información es representada en concordancia a lo especificado por INTEL [1]. A partir de lo explicado, en este script se crean las siguientes utilidades:

- **Comandos para visualizar la IDT, GDT, TTS y PCB**

Se creó un set comandos en Python que, al ser invocados mediante la instrucción *guiGDB,* desde el prompt de GDB, mostrarán la siguiente interfaz gráfica al usuario. Permitiéndole así observar la composición de las estructuras internas de SODIUM.

Figura 3. Interfaz GUI principal para visualizar las estructuras de SODIUM

En la figura anterior se pueden observar distintas opciones: PCB, GDT, IDT Y TSS. Si el usuario selecciona una de las alternativas, el visualizador mostrará otra ventana que permitirá al usuario indicar el registro de la estructura que se desea visualizar.

En consecuencia, si el usuario elige el botón GDT podrá observar lo siguiente:

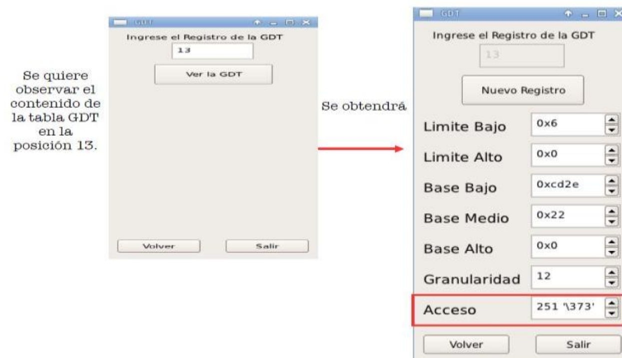

Figura 4. Interfaz GUI con la composición de la GDT

En cambio, si se escoge un descriptor determinado de la IDT se mostrará dicha estructura de la siguiente manera:

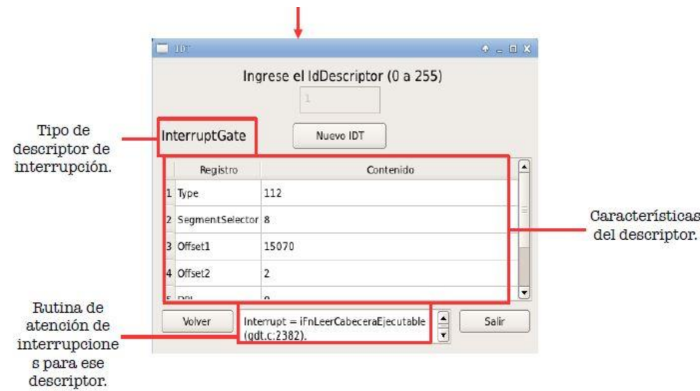

Figura 5. Interfaz GUI con la composición de la IDT

Si por otra parte el usuario desea ver la composición del PCB asociado a un determinado proceso, podrá conocerlo ingresando su número de PID en el campo correspondiente.

Figura 6. Interfaz GUI con la composición del PCB de un proceso

- **Comandos para visualizar el estado del mapa de memoria del sistema operativo en un momento determinado**

Con el objetivo de poder observar la ubicación en memoria de los segmentos que conforman cada componente que administra el sistema operativo, se desarrollaron un set de instrucciones de GDB que permiten ver el estado del mapa en un momento determinado. Por lo tanto, ingresando el comando *mapamemoria* en la consola de GDB, se imprimirá en modo textual el estado del mapa de memoria que tiene la terminal donde se esté ejecutando SODIUM en ese momento. Esto se puede apreciar en la siguiente figura.

Figura 7. Mapa de memoria de Sodium

Como se observa en el gráfico, por cada proceso se imprimen la dirección inicial y final en donde se ubican sus segmentos de código, datos, TSS y SS0 en la memoria principal. Además simultáneamente, se imprimen las direcciones del Kernel de SODIUM (Main.bin), así como también su BSS, Heap y las tablas GDT e IDT.

- **Comandos para establecer nuevos Puntos de instrumentación**

Fue necesario investigar una nueva forma de implementar los puntos de instrumentación en el código de S.O.D.I.U.M., debido a que se dificultaba poder aplicar correctamente la metodología que se había mencionado en [7], capturando desde GDB los mensajes emitidos por la función *vFnLog* del Sistema Operativo. Por consiguiente, se debió generar otro sets de comandos de GDB. Los cuales definen una cierta cantidad de breakpoints en determinadas partes del código de SODIUM, asociándolos a un evento del sistema operativo.

Luego cuando se ejecuta uno de estos breakpoints durante la ejecución, el debugger captura dicha ocurrencia y determina el evento que ha acontecido notificándoselo al usuario a través de una ventana gráfica. Cabe mencionar que esta técnica aún no se encuentra implementada en su totalidad, pudiendo sufrir cambios en un futuro.

4 Conclusiones

Hasta la fecha en esta investigación se consiguieron grandes avances en el desarrollo de las interfaces gráficas del visualizador. Dado que se pudieron mostrar los datos de las estructuras internas que utiliza SODIUM durante su ejecución utilizando PyQT. Gracias a la implementación del módulo GDB-Stub, en el código

del sistema operativo, se pudieron obtener estos avances. De forma tal, que nos permitió controlar totalmente su ejecución. Si bien por el momento cada interfaz gráfica es visualizada en forma separada, ejecutando un comando personalizados de GDB. Se pretende integrar todas las ventanas GUI en una única aplicación, empleando los mecanismos que utilizan los Front Ends que actualmente se están analizando. Pudiendo así conseguir ejecutar las interfaces del visualizador y GDB en forma simultánea. Otro logro obtenido consistió en poder ver en forma textual desde el debugger la composición del mapa de memoria del sistema operativo durante su ejecución. Con lo cual, el paso siguiente consistiría en representar dicha información mediante interfaces GUI. De esta manera se está desarrollando una aplicación que les permita a los estudiantes observar el funcionamiento interno de un sistema operativo real en forma gráfica.

5 Referencias

1. Intel: "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide": pp 65-6,413 (2011)
2. Richard Stallman, Roland Pesch, Stan Shebs "Debugging with gdb" Free Software Foundation, Tercera Edición: pp 319-449 y 461-539 (2015)
3. Bill Gatliff "Embedding with GNU: the GDB Remote Serial Protocol" revista Embedded Systems Programming, Noviembre 1999
4. Prasad Krishnan: "Hardware Breakpoint (or watchpoint) usage in Linux Kernel", IBM Linux Technology Center, Canada: pp 1-10 (2009)
5. Guido van Rossum,"El tutorial de Python", Editorial: Fred L. Drake Jr. Septiembre 2009
6. Mark Summerfiled, "Rapid Gui Programming with Python and QT", Editorial: Prentice Hall, Año: 2007
7. Graciela De Luca, Martín Cortina, Nicanor Casas, Esteban Carnuccio, Sebastián Barillaro Sergio Martín, Gerardo Puyo, " Visualizador de Estructuras de un Sistema Operativo Educativo", Congreso CACIC (2014)
8. Graciela De Luca, Martín Cortina, Nicanor Casas, Esteban Carnuccio, Sebastián Barillaro Sergio Martín, Gerardo Puyo, "Desarrollo de un prototipo para un visualizador de estructuras de un sistema operativo en ejecución a través de la comunicación serial", Congreso WICC(2015)
9. The GNU Project Debugger,https://sourceware.org/gdb/onlinedocs/gdb/Remote-Stub.html
10. The Cliffs of Inanity, http://tromey.com/blog/?p=806
11. Pyclewn, http://pyclewn.sourceforge.net/