Proyecto Visualización de Estructuras Internas de un Sistema Operativo en Ejecución como Herramienta Didáctica

Universidad Nacional de La Matanza
Unidad Académica donde se encuentra acreditado: Departamento Ingeniería e Investigaciones Tecnológicas
Código: C2-ING-013
Título del Proyecto: Visualización de estructuras internas de un sistema operativo en ejecución como herramienta didáctica.
Programa de Investigación: CyTMA2
Directora del Proyecto: De Luca, Graciela Elisabeth
Codirector del Proyecto: Cortina, Martín
Integrantes del Proyecto: Barillaro, Sebastián; Carnuccio, Esteban Andrés; Casas Nicanor Blas; Martín, Sergio Miguel; Puyo Gerardo Hernán; Volker, Mariano
Fecha de inicio: 2014/01/01
Fecha de finalización: 2015/12/31

Palabras claves: Sistema Operativo, visualizador de estructuras internas, comunicación serial, GDB-stub, Interfaz Gráfica

Área de conocimiento: Ingeniería de comunicaciones, electrónica y control.
Código Área de conocimiento: 1800
Disciplina: Otros - Sistemas Operativos
Código Disciplina: 1899
Campo de Aplicación: Otros - Sistemas Operativos
Código Campo de Aplicación: 1899
Otras dependencias de la UNLaM que intervinieron en el Proyecto:
Otras instituciones intervinientes en el Proyecto:
Otros proyectos con los que se relaciona: continuación del proyecto “Construcción de un Sistema Operativo didáctico “, S.O.D.I.U.M., Universidad Nacional de La Matanza.

Resumen

En el presente proyecto se desarrolló un visualizador de las estructuras internas de un sistema operativo en ejecución con propósito educativo, que posee la capacidad de comunicación con el mismo, durante su ejecución mediante el puerto serie. Este permite observar en tiempo real el comportamiento de los procesos y el sistema operativo, a través de formatos de visualización de la asignación de memoria, tablas básicas del sistema y planificadores de CPU. Se analizaron las técnicas que emplean los depuradores modernos, en cuanto a la implementación de mecanismos de Puntos de Parada mediante el soporte que provee la arquitectura IA32.Para asegurar la interoperabilidad con el depurador GDB se incorporó un módulo remoto denominado GDB-stub, que resuelve la comunicación a nivel lógico. Se implementa para ello el protocolo RSP de comunicación bidireccional a través del puerto serie entre diferentes máquinas, haciendo uso de un driver de desarrollo propio. Esto permite alterar el estado de ejecución del sistema operativo, pudiendo obtener los estados y estructuras internas del mismo. El visualizador consta de módulos que permiten ver gráficamente las estructuras internas del sistema, mediante la información obtenida de S.O.D.I.U.M. a través de GDB y posibilitando en el futuro reconfigurarlo para funcionar con otros sistemas operativos.

Palabras Clave: Sistema Operativo, visualizador de estructuras internas, comunicación serial, GDB-stub, Interfaz Gráfica.

Keywords: Operating System, Visualizer of Internals Structures, Serial Communication, GDB-stub, Graphical Interface.

1 Estructura

En este apartado se presenta la estructura del presente informe la cual toma como base la propuesta en forma general de la guía de informes de avance/finales, realizando agregados (indicándose con una –A– después del título) o quitando aquellos títulos que no aplican en la presente temática (indicándose con una –NA– después del título).

1 Estructura **– A –**

2 Introducción

- 2.1 Selección del Tema
- 2.2 Definición del Problema
- 2.3 Justificación del Estudio
- 2.4 Limitaciones
- 2.5 Alcances del Trabajo
- 2.6 Objetivos
- 2.7 Hipótesis

3 Desarrollo:

- 3.4 Material y Métodos **– A –**
- 3.5 Lugar y Tiempo de la Investigación
- Descripción del Objeto de Estudio **– NA –**
- Descripción de Población y Muestra **– NA –**
- 3.6 Diseño de la Investigación
- Instrumentos de Recolección y Medición de Datos **– NA –**
- Confiabilidad y Validez de la Medición **– NA –**
- Métodos de Análisis Estadísticos **– NA**
- 3.7 Etapas Ejecutadas **– A–**
  - ETAPA 1:
  - ETAPA 2:
  - ETAPA 3
  - ETAPA FINAL

4 Resultados

5 Conclusiones

6 Bibliografía

7 Producción científico-tecnológica

8 ANEXOS

2 Introducción

Los sistemas Operativos Modernos son cada vez más complejos, con gran cantidad de tareas en ejecución, brindan muchos servicios y administran los recursos disponibles. Esto hace difícil la comprensión de la forma de ejecución de las tareas, como se realiza o mejora la administración de recursos y la modificación o creación de nuevos servicios. Para poder enseñar esto, partimos del análisis y la reconfiguración de un sistema operativo didáctico (1) (S.O.D.I.U.M.) desarrollado por el equipo de investigación. Este posee múltiples algoritmos de administración reconfigurables y distintas formas de brindar servicios e interrupciones para permitir a los alumnos realizar comparaciones. Aun así existen dificultades debido a que es difícil de entender el comportamiento del sistema, las que se pretenden minimizar mediante la visualización (2) (3) de las estructuras internas durante la ejecución y estadísticas conceptualmente interesantes de un sistema operativo ya se dispone del código fuente y objeto mientras el mismo se encuentra en ejecución. Se pretende que de esta forma se pueda comprobar didácticamente el funcionamiento de los algoritmos de planificación y administración de recursos utilizados, brindando de esa forma una herramienta importante a la hora de enseñar la teoría relacionada a sistemas operativos y arquitectura de computadoras. Además esto permite tener la posibilidad de controlar dicho sistema operativo de forma remota, pudiendo detener y reanudar su ejecución cuando el usuario lo desee, dando lugar a una inspección más profunda de su estado actual.

2.1 Selección del Tema

Los estudiantes asimilan mejor el conocimiento cuando se les brinda inicialmente un marco teórico introductorio acerca de un tema específico y luego complementan ese aprendizaje mediante actividades prácticas. Esta metodología es muy utilizada actualmente en diferentes cursos de informática, con muy buenos resultados. Sin embargo existen ocasiones en que el alumno no llega a comprender en su totalidad los conceptos teóricos que le son impartidos, por lo que lleva más tiempo asimilar esas nociones y en algunos casos puede resultar frustrante. Por otra parte, diversos estudios aseveran que el aprendizaje visual (3) a través de distintos esquemas es una técnica muy poderosa que ayuda a los estudiantes a comprender más rápidamente los conceptos que sus educadores les enseñan. En ese contexto, surgió la necesidad de crear un software que les facilite a los estudiantes de Informática la adquisición de conocimientos acerca del funcionamiento interno de un Sistema Operativo, de forma tal, que les ayude a comprender de qué manera trabajan y se administran los componentes, estructuras y procesos de un sistema operativo en forma gráfica.

2.2 Definición del problema

La necesidad de permitir la visualización de diversas estructuras internas y estadísticas conceptualmente interesantes, fue la motivación para el desarrollo de un visualizador, que ejecutara en como un proceso externo al sistema operativo, pudiendo conectarse con éste a través del puerto serie. Debido a que ya poseíamos un sistema operativo del cual se disponía del código fuente y objeto mientras el mismo se encuentra en ejecución, éste resultó sumamente útil para comenzar el desarrollo.
De esta forma se puede comprobar didácticamente el funcionamiento de los algoritmos de planificación y administración de recursos utilizados, tener la posibilidad de controlar dicho sistema operativo de forma remota, permitiendo detener y reanudar su ejecución

cuando el usuario lo desee, dando lugar a una inspección más profunda de su estado actual.
El módulo visualizador debe manejar datos estadísticos bidimensionales, tablas, diagramas de bloques y estructuras de nodos, además un panel desde donde exista la posibilidad de ejecutar comandos de control sobre el sistema operativo. De esta manera se generan una serie de *vistas* que podrán ser posteriormente adaptadas a nuevas necesidades. Dicho módulo se conecta con el sistema operativo a través de una interfaz bidireccional, permitiendo así su interacción.

2.3 Justificación del estudio.

La enseñanza de Sistemas Operativos aporta conocimientos y conceptos muy importantes para el perfil de un Ingeniero en Informática. Incluso si posteriormente no se dedica al desarrollo de sistemas operativos, se encontrará con situaciones del mundo real típicas de sistemas complejos como paralelismo y condiciones de carrera, administración de recursos, manejo de prioridades, sincronización e interacción con dispositivos físicos, por lo que tendrá oportunidades de sobra para aplicar los conocimientos adquiridos.
Creemos que llevando a los alumnos a inspeccionar, analizar y participar activamente en el desarrollo del S.O.D.I.U.M. se logra una mayor transferencia de conocimientos que limitándonos a la enseñanza teórica de los temas relacionados o al estudio de otros sistemas operativos más complejos.
Así es que desde un principio el S.O.D.I.U.M. fue pensado como un Sistema Operativo Didáctico, en el que se privilegia la claridad conceptual sobre la velocidad de ejecución. Esta plataforma les permite a los alumnos no sólo dedicarse a comprender, por ejemplo, un algoritmo determinado de planificación, sino diseñar sus propias variantes y ponerlas en funcionamiento.
Sin embargo, hasta el momento no era posible lograr una visibilidad conceptual adecuada sobre el funcionamiento interno del sistema operativo dado que las herramientas de depuración actualmente disponibles son de muy bajo nivel. Esto dificulta al alumno la comprensión del funcionamiento del sistema luego de haber introducido sus cambios. Por ese motivo, se pretendió desarrollar una herramienta para visualizar el comportamiento interno de un sistema operativo.
Aunque no se pretenda realizar un cambio funcional en el sistema, la posibilidad de verlo en ejecución en forma gráfica hará más efectiva la transmisión de conocimientos teóricos y prácticos sobre el funcionamiento interno de un Sistema Operativo tradicional.

2.4 Limitaciones.

El proyecto se limita a desarrollar un sistema visualizador de estructuras utilizando el entorno de desarrollo Sodium-Devkit (4), creado por el equipo de investigación y posee todas las herramientas necesarias para su construcción e implementación. El sistema se dividirá en dos partes: El Cliente y el Servidor. El Servidor estará representado por el Sistema Operativo a analizar y el Cliente por el Visualizador. Para la transferencia de datos entre las dos terminales se utilizará la comunicación a través del puerto seriales (5), (6) debido a su sencillez, la velocidad máxima que puede alcanzar la transmisión de datos es de 115200 baudios, la cual es configurable. Se pretende hacer el desarrollo en base al Sistema Operativo S.O.D.I.U.M. y generar las bases para que pueda ser adaptado a otros sistemas similares. S.O.D.I.U.M. fue desarrollado para funcionar en una arquitectura x86, con lo cual el visualizador deberá estar configurado para poder recibir datos desde esa plataforma. Además su código fuente se encuentra desarrollada con los lenguajes ANSI C y Assembler.

2.5 Alcances del Trabajo

El sistema Visualizador que permite observar en forma gráfica la composición de determinadas estructuras de un Sistema Operativo durante su ejecución, ofreciéndole al usuario la posibilidad de controlar su ejecución en forma remota desde otra terminal, implementándose el protocolo RSP[1] (7) para la transferencia de datos entre el Sistema Operativo y el Visualizador, creando una aplicación que les facilite a los estudiantes el aprendizaje teórico y práctico acerca del funcionamiento interno de un Sistema Operativo. Para ello, se diseñaron y desarrollaron los módulos gráficos que con conformarán el sistema visualizador, mediante la utilización de herramientas tales como, como lenguajes de programación, editores de código, depuradores, protocolos de comunicación necesarias para el desarrollo y comunicación del sistema. El protocolo de comunicación, tiene la función de realizar la comunicación serial para la transmisión de datos entre el visualizador y el Sistema Operativo.

2.6 Objetivos

- Implementación total del Protocolo RSP para la transferencia adecuada de comandos e información desde el Sistema Operativo S.O.D.I.U.M. (Servidor) hasta la aplicación Visualizador (Cliente), y viceversa.
- Análisis de factibilidad de integración del depurador GDB (8) al Visualizador en el lado Cliente.
- Implementación de un mecanismo que permita controlar totalmente la ejecución del Sistema Operativo en forma remota desde el Visualizador.
- Desarrollo de un módulo parametrizable del visualizador para ejecutar en un host remoto conectando al sistema operativo y represente su funcionamiento interno gráficamente.
- Generar decodificación de mensajes recibidos desde el sistema operativo inspeccionado y viceversa.
- Generar documentación necesaria para que dicho módulo pueda ser integrado a otros sistemas operativos.

2.7 Hipótesis

- Es posible detener la ejecución total o individual a nivel de tareas del sistema operativo por medio del soporte de hardware sin detrimento de continuar la ejecución de un driver de puerto serie gracias a la capacidad de generar interrupciones al recibir información del chip UART16550 (9).
- La velocidad de transferencia del puerto serie será suficiente para mantener el paso a la cadencia de generación de eventos de interés del sistema operativo.
- Existe un punto intermedio entre mostrar información correcta y a su vez conceptualmente rica en cuanto a lo que la teoría de sistemas operativos se refiere.
- Es factible implementar un protocolo de comunicación entre el Visualizador y el Sistema Operativo para controlar su ejecución.
- Es posible implementar el Visualizador en estructuras multiplataforma de manera que pueda funcionar tanto en el Sistema Operativo Linux como en Windows.
- Se pueden observar gráficamente datos del Sistema Operativo S.O.D.I.U.M. desde un programa externo a este.

[1] RSP: Remote Serial Protocol.

3 Desarrollo.

3.1 Material y Métodos

Se ha utilizado bibliografía propia y de otros autores, conjuntamente un el sistema operativo S.O.D.I.U.M. desarrollado por este grupo de investigación, permitiendo organizar el desarrollo del Sistema Visualizador en dos partes. Por un lado la construcción e implementación del sistema operativo S.O.D.I.U.M. funcionando como Servidor, y por otro el desarrollo del Visualizador como Cliente. Los componentes del Sistema Operativo fueron escritos utilizando los lenguajes ANSI C y ASSEMBLER. En cambio para los módulos gráficos que componen el visualizador se utilizó el lenguaje Python, y su biblioteca PyQt (10) (11). Además se utilizaron las herramientas que provee el depurador GDB (8) , como medio de comunicación entre S.O.D.I.U.M. y el Visualizador.
La construcción de todo el sistema fue realizado dentro de un entorno de desarrollo, denominado Sodium-Devkit, el cual contiene una imagen del sistema operativo Linux, conjuntamente con todas las herramientas necesarias para poder programar y ejecutar el Sistema Operativo S.O.D.I.U.M. y el Visualizador. Para la ejecución de la imagen se utilizaron las máquinas virtuales Vmware y Virtualbox.

3.2 Lugar y Tiempo de la Investigación.

El equipo de investigación levó a cabo el desarrollo y pruebas del visualizador en el laboratorio 266, el que pertenece al Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de La Matanza. Este laboratorio está equipado con las máquinas para desarrollo de software con sistemas operativos Windows, Linux y S.O.D.I.U.M. para la realización de pruebas de instalación, acceso a Internet, acceso a la biblioteca digital del MINCyT y máquinas virtuales instaladas con las herramientas de desarrollo, realizando el diseño y desarrollo desde el comienzo del año 2014 hasta el fin del año 2015 *Figura 1* Etapas Previstas.

3.3 Diseño de la Investigación

Como puede verse en el GANTT siguiente (Tabla 1 Gantt 1er. Año y Tabla 2 Gantt 2do. Año) las tareas del proyecto se encuentran divididas por Etapas. En la figura 1 se muestran las 4 etapas a realizar a lo largo del proyecto, la Etapa 1 corresponde al primer año, la Etapa 2 se desarrolla durante el primer y segundo año del proyecto, las siguientes Etapas corresponden íntegramente al segundo año del proyecto.

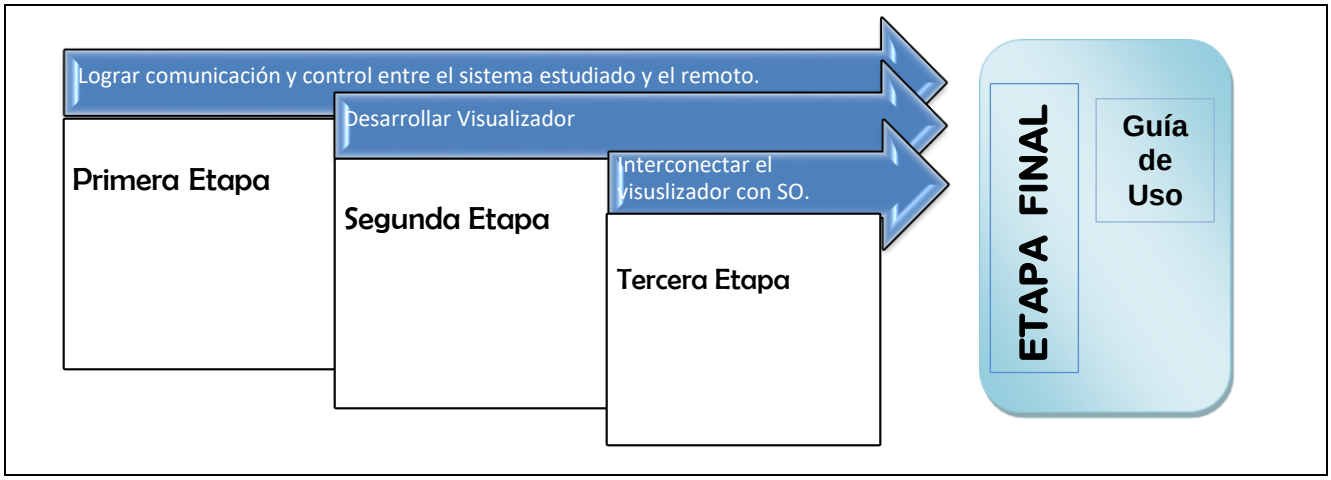


Figura 1 Etapas Previstas

Tabla 1 Gantt 1er. Año

Actividades / Responsables 1er Año	Mes 1	Mes 2	Mes 3	Mes 4	Mes 5	Mes 6	Mes 7	Mes 8	Mes 9	Mes 10	Mes 11	Mes 12
Primera etapa: Lograr comunicación y control entre el sistema estudiado y el remoto.	X	X	X	X	X	X	X	X	X	X	X	X
Investigar características de otros proyectos orientados a la visualización de sistemas complejos.	X	X	X									
Investigar uso y capacidad de *buffering* de chip UART16550.		X	X									
Investigar protocolos de comunicación serie existentes y adoptar o definir el que utilizaremos.			X	X								
Determinar capacidad máxima sostenida de transferencia mensajes para nuestro protocolo de comunicación.				X	X							
Investigar la factibilidad de desarrollar un plug-in para máquina virtual capaz de ejecutar acciones especificadas al momento en el que el sistema operativo atraviese un punto de instrumentación.					X	X						
Desarrollar una o ambas alternativas de conectividad entre el host que ejecuta el sistema operativo y el módulo visualizador que ejecuta en el sistema remoto.					X	X	X	X	X			
Segunda etapa: Desarrollar visualizador										X	X	X
Investigar distintos frameworks existentes capaces de representar grafos, gráficos n-dimensionales y analizar estadísticas.										X		
Definir los tipos de visualizaciones y datos estadísticos que soportaremos.										X	X	
Establecer requerimientos funcionales y no funcionales. Plataformas soportadas											X	X

Tabla 2 Gantt 2do. Año

Actividades / Responsables 2do Año	Mes 1	Mes 2	Mes 3	Mes 4	Mes 5	Mes 6	Mes 7	Mes 8	Mes 9	Mes 10	Mes 11	Mes 12
Segunda etapa: Desarrollar visualizador	X	X	X	X	X	X						
Casos de uso y mapa de navegabilidad del sistema.	X	X										
Diseño de arquitectura interna del visualizador.		X	X									
Desarrollar navegación y panel de control.			X	X								
Desarrollar la capacidad de visualización de los tipos definidos anteriormente				X	X	X						
Escribir manual de usuario					X	X						
Tercera etapa: Interconectar el módulo visualizador con el sistema operativo de estudio.							X	X	X			
Desarrollar capacidad de transmisión vía puerto serie en visualizador. Negociación de velocidad de transferencia, puertos de comunicación.							X					
Parseo de mensajes recibidos desde el sistema operativo inspeccionado y viceversa							X	X				
Pruebas de integración								X	X			
Etapa final										X	X	X
Generación de vistas adaptables a otros sistemas operativos.										X	X	
Empaquetar el desarrollo en un instalador.											X	
Generación de guía de uso y aplicación.											X	X
Conclusiones												X

3.4 ETAPAS

3.4.1 Primera Etapa

3.4.1.1 Investigar características de otros proyectos orientados a la visualización de sistemas complejos

En esta etapa se realizó una revisión de material bibliográfico sobre la visualización interna de sistemas en ejecución y un análisis de las herramientas disponibles ya desarrolladas.

El análisis detallado de los distintos proyectos orientados a visualización de sistemas complejos existentes fue presentado en el congreso II Congreso Argentino de Ingeniería- CADI 2014 (13)

El énfasis fue puesto en herramientas que permitan la detención y visualización de un software en ejecución. También se analizaron las formas posibles y su costo en tiempo y en complejidad para la transmisión de datos, entre el visualizador y el sistema operativo. Esto llevó a definir los elementos necesarios para permitir la visualización de diversas estructuras internas y de estadísticas en un sistema operativo.

.

3.4.1.2 Investigar uso y capacidad de buffering de chip UART 16550

Se diseñaron un conjunto de formatos para permitir la visualización de las diversas estructuras internas y las estadísticas del sistema operativo mientras éste se encuentra en ejecución. En una primera instancia, este conjunto se aplicó al sistema operativo S.O.D.I.U.M. (desarrollado por este equipo de investigación), con el objetivo de posibilitar la comprobación en forma didáctica del funcionamiento de los algoritmos de planificación y administración de recursos utilizados, brindando de esa forma una herramienta importante a la hora de enseñar la teoría relacionada a sistemas operativos y arquitectura de computadoras.
Se decidió la implementación del driver del puerto serie en S.O.D.I.U.M. porque el costo de overhead del mismo, no impacta significativamente en los tiempos de ejecución del Sistema Operativo.
Debido a las características didácticas del sistema, los tiempos de ejecución no son una variable de ponderación significativa en el diseño. Es preferible una visualización llevada al mayor detalle posible aunque implique un detrimento en la velocidad de ejecución. Sin embargo, cabe considerar que una ejecución extremadamente lenta podría provocar la pérdida de concentración deseada.
Por otro lado, mediante la utilización de ese puerto serie se logra controlar dicho sistema operativo de forma remota. Permitiendo así, detener y reanudar su ejecución cada vez que el usuario lo desee. El GDB[2] permitió depurar remotamente al sistema, dando lugar a una inspección más profunda de su estado en tiempos de ejecución.
Asimismo se incorporó la utilización de máquinas virtuales permitiendo de esta forma realizar el reemplazo de computadoras reales. Facilitando así el desarrollo del proyecto. Con esto se logró que el entorno de visualización pueda interactuar con S.O.D.I.U.M., mediante cualquiera de estos métodos, la interconexión de los puertos series a través de una computadora remota o a través de distintas máquinas virtuales.
El primer inconveniente que se detectó es que en el sistema operativo S.O.D.I.U.M. no existía el driver para puerto serie. Hubo que desarrollarlo completamente debiendo definirse la información básica y necesaria para la construcción del mismo. Teniendo en cuenta que para la comunicación serial se utilizan *interrupciones o polling*[3], fue necesaria información técnica de la UART[4].

Características del driver de puerto serie.
La función principal de esta controladora (UART) es la de tomar cada byte leído del bus de datos paralelo y serializarlo en un tren de bits, enviándolos de a uno por vez través

[2] **G**NU **D**e**b**ugger

[3] Operación de consulta constante hacia un dispositivo de hardware para crear una actividad sincrónica sin el uso de interrupciones

[4] "**U**niversal **A**synchronous **R**eceiver-**T**ransmitter"

del canal de comunicación. A su vez, la misma contempla el proceso inverso en su canal de recepción. La norma de comunicación asociada es la RS-232 (13) que especifica las características eléctricas y mecánicas de las interfaces así como velocidades admitidas, control de flujo y uso de bits adicionales para señalización y detección de errores. Esto permite, en particular al chip (16650A), la opción de configurar la generación de interrupciones a la CPU cuando se reciben 1, 4, 8 o14 bytes (14) (15)

La norma RS-232 especifica un par de líneas de datos (una para enviar y otra para recibir) y seis para control, con las que implementa el control de flujo. De esta manera, el equipo receptor puede indicarle al equipo emisor que está listo para recibir, o que se abstenga de continuar enviando datos. Así se logra adaptar la comunicación al estado de ambos equipos.

El driver serial implementado actualmente en S.O.D.I.U.M. fue construido sin aprovechar el uso del buffer FIFO que poseen los chips UART más modernos (como el 16550A mencionado anteriormente).

A lo largo de las pruebas efectuadas, se identificó el riesgo de que la tasa inicial de transmisión de datos entre terminales que se logró obtener, pudiera no ser suficiente para cubrir nuestras necesidades. Dicho riesgo llevó a relevar el modelo del chip UART utilizado en una muestra de computadoras convencionales de distintas marcas, determinando al fin que sí es común encontrar la capacidad de buffering en las mismas, por lo tanto hemos verificado que posteriormente se podrá implementar el uso de dicha capacidad en S.O.D.I.U.M. mejorando así la tasa de transferencia del driver ya desarrollado, sin mayores inconvenientes.

Según la documentación de VMware, el software ofrece cuatro puertos serie a los sistemas virtualizados. Cualquiera de estos puertos puede ser direccionado a un puerto físico real del equipo, o a un archivo, o interconectado entre dos computadoras virtuales. La UART virtualizada es un modelo compatible con el chip 16550A, que ofrece un buffer FIFO de 16 bytes. Los cuatro puertos series comparten las IRQ 3 y 4 de a pares (14).

En el caso de Bochs, también son cuatro los puertos serie virtualizados. Permite elegir el modo de operación de manera similar a VMware: null, raw, mouse, file, term (sólo para Unix), FIFO y Socket (sólo para Windows). En todos los casos, el chip emulado ofrece compatibilidad con el muy difundido 16550A (16)

El driver presenta un conjunto de funciones que permiten: inicializar los puertos COM, inicializar la configuración de comunicación (velocidad de transmisión, formato de envío de datos, entre otros), transmisión, recepción y tratamiento de errores. Todas estas funciones fueron implementadas contemplando el uso de *interrupciones* o *polling*, utilizando los ***Registros de direcciones de los puertos de E/S*** (Tabla 3 Direcciones estándar en la mayoría de las arquitecturas en base hexadecimal), para definir los puertos COM y una línea de interrupciones IRQ para llamar la atención del procesador y los ***Registros utilizados para la comunicación*** *(*Tabla 4 Registros para la comunicación*).*

Tabla 3 Direcciones estándar en la mayoría de las arquitecturas en base hexadecimal

Puerto	Registro	(IRQ)
COM1	3F8	4
COM2	2F8	3
COM3	3E8	4
COM4	2E8	3

Tabla 4 Registros para la comunicación

Registros E/S	Descripción
3F8 / 2F8	Registro de datos Emisión / Recepción
3F9 / 2F9	Registro de habilitación de interrupciones
3FA / 2FA	Registro de identificación de interrupciones
3FB / 2FB	Registro de control de la línea
3FC / 2FC	Registro de control del MODEM
3FD / 2FD	Registro de estado de la línea
3FE / 2FE	Registro de estado del MODEM

3.4.1.3 Investigar protocolos de comunicación serie existentes y adoptar o definir el que utilizaremos.

- **Transferencia de datos**

Para este módulo se utilizó una transmisión configurada en 9600 bps con formato 8-N-1 (8 bits de datos | sin paridad | 1 bit de parada). El cable empleado fue de tipo NULL MODEM con un terminal RS-232 en un extremo, cumpliendo con la norma especificada en (13), y un terminal USB adaptado en el otro. El terminal USB permite la conexión con cualquier dispositivo (laptop, tablet, PC, etc) que carezca de puerto serie.

- **Implementación de puerto serie en S.O.D.I.U.M.**

Para poder conseguir detener la ejecución del sistema Operativo, en determinado momento, se utilizaron las herramientas que presenta GDB para depurar en forma remota. Esto posee ventajas debido a que no hay que generar nuevas estructuras propias de S.O.D.I.U.M. y permite, a futuro, interactuar con otros sistemas operativos que utilizan la misma herramienta.
Se utilizaron dos terminales, de las cuales una de ellas puede llegar a estar simulada (terminal virtual). La primera terminal la denominamos *Cliente* que es en donde se está ejecutando la aplicación de visualización. En esta máquina se instaló el GDB con la finalidad de permitirnos detener al sistema operativo S.O.D.I.U.M. en forma remota. La segunda terminal, la que se denomina *Servidor*, es donde se está ejecutando el sistema operativo. La que puede estar representada de dos formas:

Computadora Física: dónde el sistema operativo puede ejecutarse en una PC de escritorio conectada físicamente por cables USB con el *cliente* a través del protocolo serie (Figura 2(a) Representación gráfica de la ejecución del visualizador y S.O.D.I.U.M. en dos terminales. (b) Representación de la ejecución en la misma terminal.).

Máquina Virtual*:* necesaria para el caso particular en que el usuario no cuente con disponibilidad para utilizar más de una máquina para poder probar el funcionamiento (Figura 2(a) Representación gráfica de la ejecución del visualizador y S.O.D.I.U.M. en dos terminales. (b) Representación de la ejecución en la misma terminal.). Como consecuencia, se han elegido dos plataformas para ejecutar este sistema operativo: Bochs (17)y VMware (18) La elección recayó sobre éstas por dos razones: la primera debido a que es de mayor uso por los alumnos que cursan la materia y la segunda se consideró que de esa manera se limita la cantidad de virtualizadores a fin de facilitar la integración al sistema por los docentes responsables y las pruebas correspondientes con el sistema unificado.

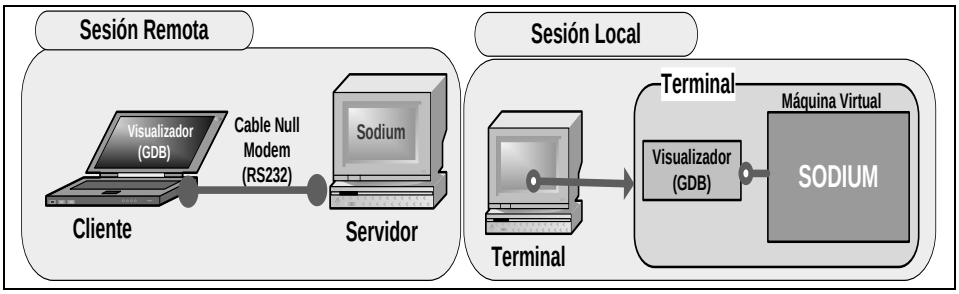


Figura 2(a) Representación gráfica de la ejecución del visualizador y S.O.D.I.U.M. en dos terminales. (b) Representación de la ejecución en la misma terminal.

3.4.1.4 Determinar capacidad máxima sostenida de transferencia mensajes para nuestro protocolo de comunicación.

Se efectuaron distintas pruebas entre el ordenador *cliente* y S.O.D.I.U.M. en una misma máquina, mediante la utilización de algún programa que permita emular una consola para la conexión con máquinas remotas, utilizando para ello distintos protocolos de comunicación.

Se configuró el programa de forma que utilice el protocolo RAW, el cual se eligió debido a la gran cantidad de extensiones que permite y la cantidad de sistemas que lo tienen como parte del mismo, además éste no comprime los datos al archivar la imagen, como lo hace JPG, conteniendo todos los datos de la misma tal como fue captada.

El protocolo RAW debe estar conectado a un puerto determinado para la establecer la comunicación con S.O.D.I.U.M.

A los fines de poder realizar la conexión entre ambos, la máquina virtual en donde se ejecute el Sistema Operativo debe estar configurada para permitir comunicarse con cualquier IP utilizando el mismo puerto tanto el *Cliente* como el *Servidor*, especificado a través del puerto serie COM1.

Por otro lado fue necesario incluir una serie de modificaciones para conseguir la comunicación entre el emulador y S.O.D.I.U.M., utilizando el entorno de desarrollo antes descripto. Para ello se instaló un servidor SSH en Linux y posteriormente se

abrió el número de puerto previamente elegido en el firewall del Sistema Operativo Linux.

Un punto importante para acotar, es que para establecer una correcta conexión entre las dos terminales, primeramente se debe inicializar S.O.D.I.U.M. en el *Servidor* y posteriormente ejecutar el *cliente*. En caso contrario, al no estar S.O.D.I.U.M. esperando conexiones, se produce un error de conectividad en el *cliente.*

En las pruebas iníciales del proyecto se configuraba a la UART 16550A para que realice la transferencia de datos entre el cliente y el Servidor a 9600 baudios, velocidad estándar del puerto serie. Sin embargo con el avance del proyecto de investigación esta velocidad fue incrementada a la máxima soportada por este chip. La cual es de 115200 baudios. Con lo cual se determinó conveniente que esa sea la velocidad estándar de transferencia entre el Sistema operativo S.O.D.I.U.M y la aplicación cliente utilizando el protocolo RSP[5] (19)

3.4.1.5 Investigar la factibilidad de desarrollar un plug-in para máquina virtual capaz de ejecutar acciones especificadas al momento en el que el sistema operativo atraviese un punto de instrumentación.

Al comenzar el proyecto se realizó un estudio inicial sobre las distintas alternativas a desarrollar para conseguir comunicar el Sistema Operativo S.O.D.I.U.M con la aplicación Cliente a través de una comunicación serial. Con lo cual inicialmente se plantearon dos hipótesis:

- Desarrollar un plug-in para máquina virtual Bochs que permitirá controlar la ejecución del Sistema Operativo mientras este se está ejecutando en ella.
- Utilizar las herramientas que ofrece GDB para controlar aplicaciones ejecutadas en forma remota.

Después de analizar y estudiar las dos hipótesis planteadas, se descartó la primera de ellas. Debido a que si, al comenzar el proyecto, se centraba la investigación en únicamente desarrollar un plug-in para ejecutar el Sistema Operativo dentro de una máquina virtual, se estaría limitando a que el Visualizador solamente funcione en la misma computadora que se esté ejecutando S.O.D.I.U.M. Además para poder desarrollar un plug-in de la máquina virtual, era necesario tener que modificar y recompilar todo el código fuente de la V.M a los fines de poder generar un módulo interno que funcione en ella. Por consiguiente se hubiera descartado, desde el inicio del proyecto, la posibilidad de investigar si era factible desarrollar al visualizador y S.O.D.I.U.M para que también se puedan ejecutar en distintas máquinas. Por ese motivo, se determinó conveniente seguir la segunda hipótesis planteada. De esta manera se investigó y utilizaron las distintas herramientas que ofrece GDB para conseguir controlar la ejecución de aplicaciones remotas desde el Visualizador que se pretendía desarrollar .Por ende se realizó una profunda investigación sobre el funcionamiento interno del depurador GDB, la cual es descripta en los siguientes aparatados.

RSP protocolo de comunicación remota que utiliza GDB

El protocolo de comunicación remota utilizado por GDB es el denominado RSP que posee características especiales porque está basado en el conjunto de caracteres ASCII. Esto significa que todos los caracteres transmitidos ente *Cliente y Servidor* se

5 Remote Serial Protocol

encuentran dentro del rango imprimible, lo que facilita su estudio, documentación, y depuración.

El protocolo RSP (7) posee caracteres de inicio y terminación de mensaje claros. La parte útil de cada mensaje comienza por un símbolo “$” y termina con un símbolo “#”, agregando control de errores. Luego del símbolo que indica la finalización de cada mensaje enviado se escribe un número que es el resultado de la suma, módulo 256, del valor de cada uno de los caracteres comprendidos.

La recepción de un mensaje es confirmada inmediatamente por el receptor, indicando “+” si el mismo se recibió íntegramente, “-“si se detectó un error en el formato del mensaje o durante la ejecución del mismo. Siguiendo a dicho carácter, si es necesario, se transcribe el resto de la respuesta, o un código y descripción de error.

Una transacción típica consiste en la emisión de un comando por parte del depurador hacia el sistema remoto y su respectiva respuesta. La única excepción a este comportamiento se da cuando el sistema remoto emite un mensaje hacia el depurador con el objetivo de que este último lo muestre en su salida de consola. Esta acción puede llevarse a cabo en cualquier momento siempre y cuando no exista una transacción todavía en proceso.

Los datos respondidos por el sistema depurado pueden, opcionalmente, estar comprimidos mediante la técnica RLE[6], con lo que se puede reemplazar secuencias de más de tres caracteres consecutivos similares por la emisión de un carácter, un signo “ * ”, y un contador numérico de repeticiones.

De esta forma se realizó un módulo elemental de GDB que permite la detención del sistema operativo en forma remota.

3.4.1.6 Desarrollar una o ambas alternativas de conectividad entre el host que ejecuta el sistema operativo y el módulo visualizador que ejecuta en el sistema remoto.

En el apartado anterior se mencionó la decisión de desarrollar únicamente la hipótesis de utilizar las herramientas que ofrece GDB para poder comunicar y controlar la ejecución de aplicaciones en forma remota para el resto del curso del proyecto de investigación.

Elección del Ciclo de Vida del Proyecto de Investigación.

El desarrollo durante todo el proyecto presentó grandes desafíos técnicos y funcionales, debido a que las especificaciones eran volátiles y fueron surgiendo a lo largo de la construcción del Visualizador. Por ese motivo se determinó conveniente emplear un modelo de ciclo de vida incremental con prototipo desechable para su desarrollo.

Desarrollo del Prototipo desechable de la Interfaz la gráfica.

En esta etapa, la construcción del prototipo permitió determinar con mayor precisión las funcionalidades y características que debía presentar el sistema. Dado que resultó ser la primera aproximación funcional gráfica del Servidor comunicándose con el Sistema Operativo S.O.D.I.U.M. como cliente. Su desarrollo e implementación fue detallado en (14). Como consecuencia, este nos permitió obtener un enorme retroalimentación acerca de cómo debería ser la comunicación entre el Visualizador y el Sistema Operativo.

---

6 Run-length encoding

En el momento en que fue desarrollado el prototipo, la investigación estaba siendo orientada en desarrollar e implementar módulos que emplearán el protocolo RSP (15) (7) para la transferencia de datos, tanto del lado Cliente como del Servidor. Por ese motivo en S.O.D.I.U.M., se decidió ir implementando en forma gradual un módulo muy elemental, que simulará algunas de las funcionalidades esenciales del Gdb-Stub (8). De esta forma se podía controlar en forma remota la ejecución del S.O. Por otro parte del lado del Cliente, el equipo de investigación empezó a desarrollar un módulo que pretendía simular el trabajo que hace GDB para decodificar los mensajes con el protocolo RSP enviados por Gdb-Stub. De esta manera se trataba de evitar utilizar depurador en el Visualizador. Un punto importante, es que para la transferencia de datos entre el prototipo de la interfaz gráfica y el sistema operativo se utilizó la comunicación serial. Aprovechando para ello el driver de puerto serie desarrollado en S.O.D.I.U.M. (16)y en el lado Cliente, los módulos Miniterm y PySerial (17) que están desarrollados en Python.
Sin embargo, después un tiempo de investigación y desarrollo, se desistió continuar con la hipótesis de no usar GDB. Debido a la complejidad de su implementación. Por ese motivo se determinó que resulta más conveniente seguir implementando el módulo Gdb-Stub en S.O.D.I.U.M. en forma gradual, y posteriormente comunicarlo en forma remota con el depurador GDB. De esta forma se evitaría tener que desarrollar nuestro propio módulo decodificador de mensajes RSP en el lado Cliente, debido a que GDB lo hace automáticamente.

Implementación Inicial de GDB-Stub:

La implementación del módulo Gdb-Stub en el Sistema Operativo S.O.D.I.U.M demoró más de lo estipulado en la planificación inicial. Si bien su adaptación fue iniciada en esta etapa del primer año de la Investigación. Su implementación fue concluida exitosamente en el segundo año del proyecto. Debido a la complejidad de su adaptación e interrelación con otros componentes del sistema. Se consideró conveniente describir su implementación en apartados posteriores.

3.4.2 Segunda Etapa

3.4.2.1 Investigar distintos frameworks existentes capaces de representar grafos, gráficos n-dimensionales y analizar estadísticas.

Entre las bibliotecas que implementan interfaces gráficas de usuario en Python, las principales son Tkinter, WxPython y PyQT (23)
Las bibliotecas Tkinter y WxPython fueron descartadas, debido a la cantidad de elementos gráficos, el comportamiento de la interfaz o a la falta de un editor gráfico de ventanas en aplicaciones, multiplataforma.
Se resolvió emplear el binding de Python que posee el Framework QT, Pyqt, para la construcción de la interfaz gráfica. Esta biblioteca (24) brinda facilidades para el desarrollo de interfaces GUI, un completo conjunto de elementos gráficos y un flexible y potente control del comportamiento de la interface, posee un mecanismo de conexión de señales y eventos, sencillo, rápido, de apariencia nativa, y se puede separar el diseño de la interface.

3.4.2.2 Definir los tipos de visualizaciones y datos estadísticos que soportaremos.

En primer lugar, cabe destacar que no siempre es de interés toda la información contenida en una estructura si el objetivo de su extracción es lograr una abstracción del detalle de la misma a través de la visualización de sus interrelaciones con otras.

El sistema operativo S.O.D.I.U.M. fue desarrollado para ejecutar en equipos con arquitectura x86. Dada su naturaleza de sistema operativo de estudio, se buscó ejercitar todas las características que esta arquitectura provee, como ser protección, segmentación fija, segmentación paginada, cambios de contexto por hardware y manejo de interrupciones y excepciones. Por consiguiente, para poder utilizar la memoria, CPU y dispositivos externos se emplean distintas estructuras que están intrínsecamente relacionadas. Lo que se pretende mostrar por medio del visualizador es el estado de dichos datos y su variación en el tiempo ante distintos eventos en tiempo real.

Cabe destacar, el contraste, con otros sistemas operativos, cuyo objetivo es ser fácilmente portables a distintas arquitecturas de hardware, buscan mantener la mayoría de sus mecanismos de administración de recursos agnósticos a la arquitectura, utilizando únicamente el mínimo denominador común de características presentes en las mismas, como ser paginación. Otro mecanismo provisto por la arquitectura x86 (25) que es frecuentemente ignorado, tanto por compatibilidad como por eficiencia, es el cambio de contexto por hardware.

Seguidamente, se describen las estructuras que utiliza S.O.D.I.U.M. que van a ser representadas gráficamente y cómo se va a exponer su interrelación por el visualizador.

- **GDT (GLOBAL DESCRIPTOR TABLE)**
  La arquitectura Intel ofrece la posibilidad de utilizar unas tablas de descriptores alojadas en memoria para poder ubicar los segmentos que se utilizan en el sistema. Una de esta estas tablas es la GDT, apuntada por el registro GDTR de la CPU (25) Dicha estructura contiene esencialmente los descriptores de códigos, datos y stacks utilizados por el sistema operativo para ubicar los segmentos correspondientes en la memoria principal. Para hacer uso de esta herramienta, S.O.D.I.U.M. administra esta tabla utilizando una estructura en forma de vector. Cada elemento de dicho vector contiene fundamentalmente, tipo de descriptor (código, datos o stack), dirección base del segmento, tamaño del segmento y nivel de privilegio *DPL*. En consecuencia, en base a estos ítems, S.O.D.I.U.M. consigue acceder a la información de cada proceso en memoria en un momento determinado. Para esto se utilizan otros elementos que se comentarán seguidamente.
- **PCB (PROCESS CONTROL BLOCK)**
  Esta estructura contiene toda la información necesaria de un proceso en particular que un sistema operativo utiliza para poder llevar a cabo su administración durante su tiempo de vida. De forma tal que el SO pueda asignarle eficientemente los recursos que este emplea. Siendo así, S.O.D.I.U.M. maneja dicha herramienta para gestionar los distintos procesos que utiliza el sistema. Este componente se encuentra conformado en un vector de estructuras que almacenan distinta información. Entre las más importantes se encuentran el ID del proceso, el ID del padre, estado, nivel de prioridad y los índices de descriptores de segmentos de GDT asignados a código, datos y stack.
  La visualización gráfica de esta información es trascendental para que el estudiante pueda comprender el funcionamiento del sistema.
- **IDT (INTERRUPT DESCRIPTOR TABLE)** Otra tabla que el Sistema Operativo S.O.D.I.U.M. debe gestionar, aplicando la tecnología que ofrece Intel, es la IDT. Esta estructura contiene los descriptores de segmentos en donde se encuentran alojadas las rutinas de atención de las interrupciones. Esta tabla consiste en un vector de estructuras de 256 posiciones. Cada posición del vector presenta un selector de segmento para la GDT, Offset, tipo y nivel de privilegio *DPL*. En consecuencia, cada vez que ocurre una interrupción en modo protegido, el sistema operativo S.O.D.I.U.M. utiliza esa estructura para ubicar la rutina de atención asociada a dicho evento.

- **MAPA DE MEMORIA**
  Una de las características importantes que se pretende desarrollar en el visualizador es que este permita observar en tiempo de ejecución la ubicación de los componentes que conforman al sistema operativo y procesos en la memoria principal. El croquis del diseño preliminar de cómo se pretende mostrar el estado de la memoria en un momento dado se ve en la Figura 3 Croquis del Mapa de Memoria que representará el visualizador

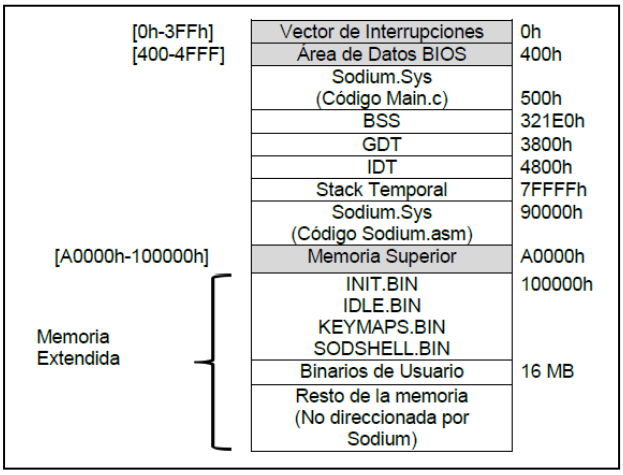


Figura 3 Croquis del Mapa de Memoria que representará el visualizador

Todas las estructuras anteriormente mencionadas son algunos de los componentes que se pretende que el visualizador de S.O.D.I.U.M. muestre inicialmente, dado que es muy importante poder ver gráficamente su contenido. De esta manera, el usuario podrá observar e interactuar con dichos componentes a través los diagramas que los representan.

Formato de las Estructuras IDT, GDT, TSS y PCB del Sistema Operativo S.O.D.I.U.M.

En este apartado se exponen los diseños de las estructuras, dentro del código fuente del sistema operativo S.O.D.I.U.M., que se representan gráficamente en el visualizador:

- **IDT:**

En S.O.D.I.U.M. la estructura de la tabla IDT se puede ver en la Figura 4.

- **GDT:**

La tabla tiene 8192 (máximo) entradas, cada una de 8 bytes:

La estructura tiene el formato que se puede ver en la Figura 5.

- **PCB:**

La estructura del PCB en el S.O se puede ver en la Figura 6.

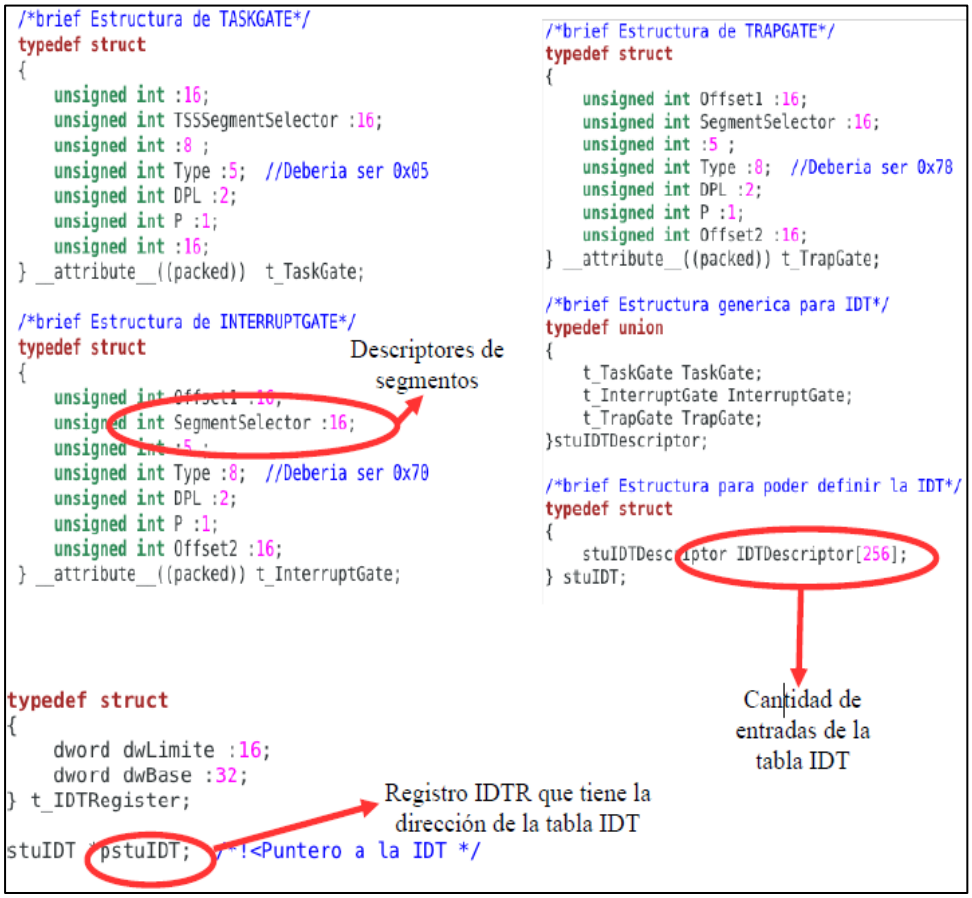


Figura 4 Formato de la estructura de la IDT

```
typedef struct {
   stuGDTDescriptor stuGdtDescriptorDescs[8192]; /*!< Vector de posiciones de la GDT
} stuEstructuraGdt;

stuEstructuraGdt *pstuTablaGdt; /*!< puntero a la GDT */

typedef struct {
  unsigned short usLimiteBajo;    /*!< limite bajo 0..15   */
  unsigned short usBaseBajo;      /*!< base bajo 0..15     */
  unsigned char ucBaseMedio;      /*!< base media 16..23   */
        /**
                (Type es 1 byte (bits 8-11) indica si el descriptor es de codigo, de datos o de sistema.
                Bit 'S' que indica con cero, si el descriptor es de sistema o, con uno, si es de codigo
                datos.
                DPL  Define nivel de previlegio del segmento.
                Bit 'P' indica si el segmento está en memoria principal o no.
        */

  unsigned char ucAcesso;         /*! byte de acceso*/

  unsigned int bitLimiteAlto:4;    /*!< limite alto 16..19   */

        /**
                La estructura de un descriptor de segmento es:
                Bit 20: AVL, Disponible para el uso del software del sistema.
                Bit 21: deberia ser siempre cero.
                Bit 22: D/B especifica distintas opciones segun sea el segmento de codigo, de stack o de
                datos.
                Bit 23: G Bandera que indica la granularidad. Determina si el limite del
                segmento se especifica de a un Byte o de a 4 KByte.
        */

  unsigned int bitGranularidad:4;

  unsigned char usBaseAlto;        /*!< base alto24..31    */

} NOALIGN stuGDTDescriptor;
```

Figura 5 – Formato de la Estructura de la GDT

```
typedef struct _stuPCB_ {
    unsigned long int ulId; /*!< id del proceso */
        unsigned int iPrivilegio;
        unsigned long ulTiempoListo; /*!< tiempo en que el proceso pasa al estado de listo */
    unsigned int uiIndiceGDT_CS; /*!< indice del descriptor de segmento de codigo de este proc en la GDT */
    unsigned int uiIndiceGDT_DS; /*!< indice del descriptor de segmento de datos de este proc en la GDT */
        unsigned int uiIndiceGDT_ES; /*!< indice del descriptor de segmento de codigo de la
bibliotecadinamica en la GDT */
    unsigned int uiIndiceGDT_TSS; /*!< indice de la TSS de este proc en la GDT */
    unsigned int uiIndiceGDT_SS0; /*!< indice del descriptor del segmento de stack ss0 de este proc en la
GDT */
    void (* vFnFuncion) (); /*!< puntero a funcion (proceso) */

    unsigned long ulParentId; /*!< id del padre */
    unsigned long ulUsuarioId; /*!< id del usuario */
    int iPrioridad; /*!< para futuro uso */
    int iEstado; /*!< ver estados mas arriba (PROC_XXX) */
    unsigned long lNHijos;
    int iExitStatus;
    unsigned long ulLugarTSS;
    char stNombre[25];
    unsigned int uiTamProc;
    struct stuTablaPagina * pstuTablaPaginacion;
    unsigned int uiDirBase,

            uiLimite;
    stuMemoriasAtachadas memoriasAtachadas[MAXSHMEMPORPROCESO];

    //agregado
    unsigned long ulTiempoEspera; /*!< Tiempos para algoritmos tipo HRN */
    unsigned long ulTiempoServicio; /*!< Tiempo de servicio del proceso */
    //fin agregado
    /*!< Estructura usada para guardar los tiempos de proceso usados para la syscall "times"*/
    struct _stuPCB_ * pstuPcbSiguiente;
    /*!< Array con los contadores utilizados como temporizadores para la SC "times"*/
    tms stuTmsTiemposProceso;
    /*!< Tiempo durante el cual el proceso tiene que permanecer "dormido" */
    itimerval timers[3];
    /*!< Puntero a donde se guardara el resto del tiempo que debia estar detenido el proceso y no lo h
    long lNanosleep;
    /*!< Indica si el proceso esta siendo rastreado.*/
    timespec *puRestoDelNanosleep;
    /*!< Indica si el proceso esta siendo rastreado.*/
    long lPidTracer;
    unsigned int uiTamanioTexto; /*!< Bytes de codigo ejecutable */
    unsigned int uiTamanioDatosInicializados; /*!< Bytes de datos globales inicializados (incluye BSS
final)*/
    unsigned int uiTamanioStack; /*!< Bytes de stack (libres+usados) */

    unsigned int uiTamanioOverhead; /*!< Bytes de Overhead (frag. interna)*/

    unsigned int uiEsperaTeclado; //Indica que el proceso está esperando una entrada de teclado.
    unsigned int uiEsperaNet; //Indica que el proceso está esperando una entrada/salida de red.
    stuHeaderELF stuCabeceraELF;
    stuEsperarMemoria stuEsperaMemoria;
    unsigned int uiBaseSs0;
    unsigned int uiPosStack;
    unsigned int uiEsperaIO; //Indica que el proceso está esperando una entrada de

        stProfile Profile;
        int iTeclaRead;
        int IOState;
        int NetState;
} stuPCB;
```

Figura 6- Estructura del Formato del PCB

3.4.2.3 Establecer requerimientos funcionales y no funcionales. Plataformas soportadas.

En este apartado se mencionaran los requisitos funcionales y no funcionales en los que basó el desarrollo y la implementación del sistema Visualizador. Cabe destacar que esta lista de requisitos fue siendo modificada a lo largo de todo el proyecto. Por lo que versión final es la que se presenta a continuación. Inicialmente para generar y confeccionar el modelo de casos de uso base del proyecto, fue necesario generar un

prototipo desechable, que fue anteriormente mencionado. De esta forma se pudo establecer los requisitos iníciales del sistema Visualizador. A partir del desarrollo del prototipo, se pudo determinar el alcance y las funcionalidades del Visualizador. Por ende se consiguió establecer los módulos que lo conforman, en donde cada uno ellos cumplirán un objetivo específico

Requisitos funcionales:

a) El sistema deberá estar dividido en dos partes:
   - **El servidor:** Estará conformado por el Sistema Operativo, que contendrá las estructuras a representar gráficamente por el visualizador
   - **El cliente:** Estará conformado por el sistema Visualizador, que será el encargado de representar gráficamente las estructuras internas del Sistema Operativo

b) El sistema visualizador deberá permitirle al usuario detener y reanudar la ejecución del Sistema Operativo que se está analizando en un momento determinado.
c) El sistema visualizador deberá permitirle al usuario depurar el código fuente del Sistema Operativo durante su ejecución.
d) El sistema visualizador deberá permitirle al usuario visualizar, a través de un componente gráfico, la composición de los elementos que componen la estructura de la tabla PCB en un momento determinado durante la ejecución del Sistema Operativo.
e) El sistema visualizador deberá permitirle al usuario visualizar, a través de un componente gráfico, la composición de los elementos que componen la estructura de la tabla GDT en un momento determinado durante la ejecución del Sistema Operativo.
f) El sistema visualizador deberá permitirle al usuario visualizar, a través de un componente gráfico, la composición de los elementos que componen la estructura de la tabla IDT en un momento determinado durante la ejecución del Sistema Operativo.
g) El sistema visualizador deberá permitirle al usuario visualizar, a través de un componente gráfico, la composición de los elementos que componen la estructura de la tabla TSS en un momento determinado durante la ejecución del Sistema Operativo.
h) El sistema Visualizador deberá permitirle al usuario visualizar en tiempo de ejecución la ubicación de los componentes que conforman al sistema operativo y procesos en la memoria principal gráficamente.
i) El sistema deberá permitirle al usuario ver en forma gráfica, con un diagrama temporal, los distintos cambios de estados que sufren los procesos durante su ejecución en el Sistema Operativo.
j) El sistema Visualizador deberá permitirle al usuario establecer puntos de instrumentación, ejecución de determinadas secciones de código fuentes del Sistema Operativo.
k) El sistema Visualizador deberá notificarle al usuario la ocurrencia de un punto de instrumentación.
l) El sistema Visualizador deberá poder ser configurado para visualizar el contenido de diferentes Sistemas Operativos, a través de un panel de control.

Requisitos No funcionales:

a) La aplicación Visualizador deberá poder ser ejecutado en entornos Linux y Windows
b) La aplicación Cliente y Servidor deberá comunicarse a través del puerto Serie, chip UART16550A.
c) El protocolo de comunicación utilizado entre Visualizador y el sistema Operativo deberá ser RSP.
d) El Servidor, sistema operativo, podrá ejecutarse dentro de una máquina Virtual comunicándose a través del protocolo serie con el cliente, que se estará ejecutando en la misma computadora.
e) El Servidor, sistema operativo, podrá ejecutarse en una PC de escritorio conectada físicamente por cables USB con el *cliente* a través del protocolo serie, que se estará ejecutando en otra computadora
f) El Visualizador deberá utilizar las herramientas que provee el depurador GDB para controlar la ejecución del Sistema Operativo.
g) La velocidad de transferencia de datos entre el Visualizador y el Sistema Operativo deberá estar configurada a 115200 baudios.
h) durante la ejecución del Sistema Operativo, observar el estado del contenido de una determinada variable de su código fuente
i) El sistema Visualizador deberá ser desarrollado utilizando el lenguaje Python y PyQT.
j) Para el desarrollo inicial del Visualizador se deberá usar como sistema Operativo de referencia a S.O.D.I.U.M.

3.4.2.4 Casos de uso y mapa de navegabilidad del sistema.

Modelo de Casos de Uso:

Al igual que los requisitos funcionales el modelo de casos de usos fue siendo modificado en el transcurso del desarrollo del proyecto.
A continuación se muestra el Modelo de Casos de Uso del Visualizador (Figura 7 Modelo de casos de uso del Visualizador de Estructuras (Cliente)) y se describen brevemente cada uno de sus módulos.

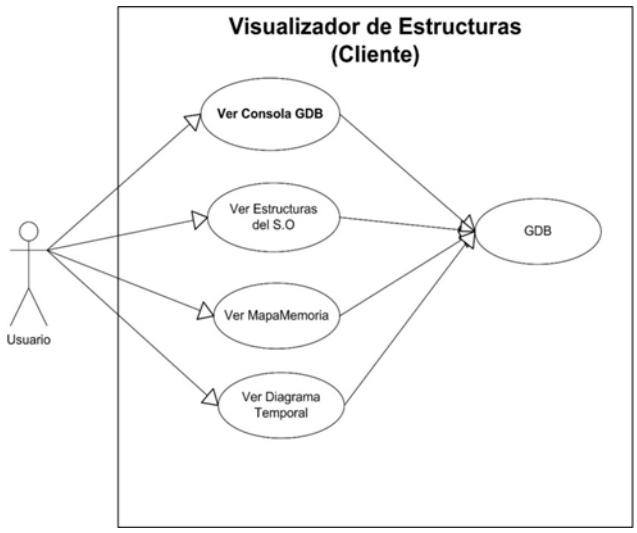


Figura 7 Modelo de casos de uso del Visualizador de Estructuras (Cliente)

- ***Módulo gráfico para controlar GDB :***

Este módulo representaría el Front-End necesario para depurar S.O.D.I.U.M. a través de la interfaz GDMB/MI del depurador GDB

- ***Módulo de visualización de estructuras de S.O.D.I.U.M. :***

Este componente del visualizador se encarga de mostrarle al usuario interfaces GUI con el estado de las estructuras IDT, GDT, TSS, y PCB que utiliza S.O.D.I.U.M. en un momento determinado.

- ***Módulo de visualización del Mapa de Memoria:***

Este módulo tiene como finalidad representar en forma gráfica el estado del 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 componente pretende que el usuario pueda observar en forma gráfica los diferentes estados que van a ir adquiriendo los procesos a lo largo de su vida, a medida que vayan ejecutándose en el sistema operativo. De esta forma se intenta que el estudiante pueda visualizar la ejecución de distintos algoritmos de planificación de la CPU en forma visual.

Mapa de navegabilidad del Sistema

El mapa de navegabilidad del sistema sufrió constantes cambios a lo largo del proyecto. Por lo que en este apartado se detalla el mapa de navegabilidad resultante obtenido al final del proyecto de investigación.
El diseño final del visualizador se elaboró en base al funcionamiento interno de tres archivos que contiene su código fuente

- **GDB.py**
- **GDBMI.py**
- **Archivos de las Interfaces_GUI.py**

Estos archivos fueron el pilar fundamental para poder construir el funcionamiento del Visualizador, con los conocimientos que fueron adquiridos después del estudio Pyclewn. De esta forma se pudo solucionar las limitaciones que se presentaron al usar Python desde GDB, que se descubrieron durante la primera versión de los módulos de Visualización. En consecuencia el gráfico de la Figura 8 Interrelaciones existentes entre las clases que componen el Visualizador esquematiza, de una forma muy elemental, las interrelaciones existentes entre las clases que contienen estos tres archivos. Para mayor detalle sobre cómo se realiza esta interrelación Véase en **ANEXO** 5 Apartado 6.

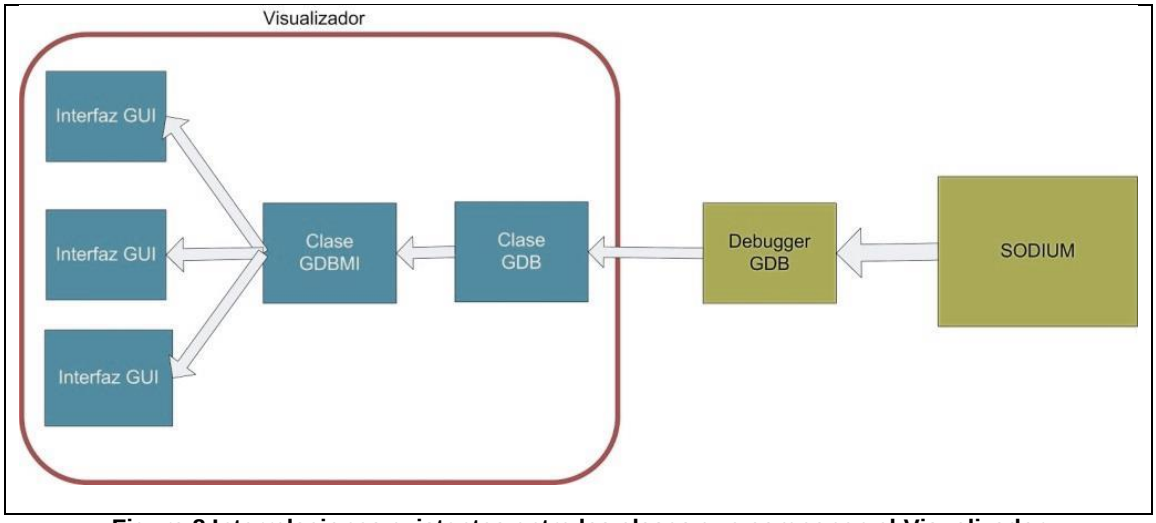


Figura 8 Interrelaciones existentes entre las clases que componen el Visualizador

3.4.2.5 Diseño de arquitectura interna del visualizador.

La arquitectura interna del sistema Visualizador sufrió constantes cambios a lo largo del proyecto. Por lo que en este apartado se detalla el la arquitectura resultante obtenida al final del proyecto de investigación

En la Figura 9 Diagrama de Clases del Visualizador, se expone las clases principales, desarrolladas Python, que conforman el sistema visualizador. Para mayor claridad, en el diagrama no se representaron todas las entidades que constituyen el sistema. Sino que se muestran las más importantes.

Se puede observar que la clase principal del Visualizador es la denominada *VentanaPrincipal_Codigo.* Esto se debe a que su código corresponde a una interfaz GUI del tipo contenedor o MDI. Por ese motivo contiene el método *main*, punto de entrada para iniciar el visualizador. Esta clase además es la encargada de inicializar los hilos de GDBMI y GDB, y de crear todas las interfaces gráficas cuando el usuario lo solicite

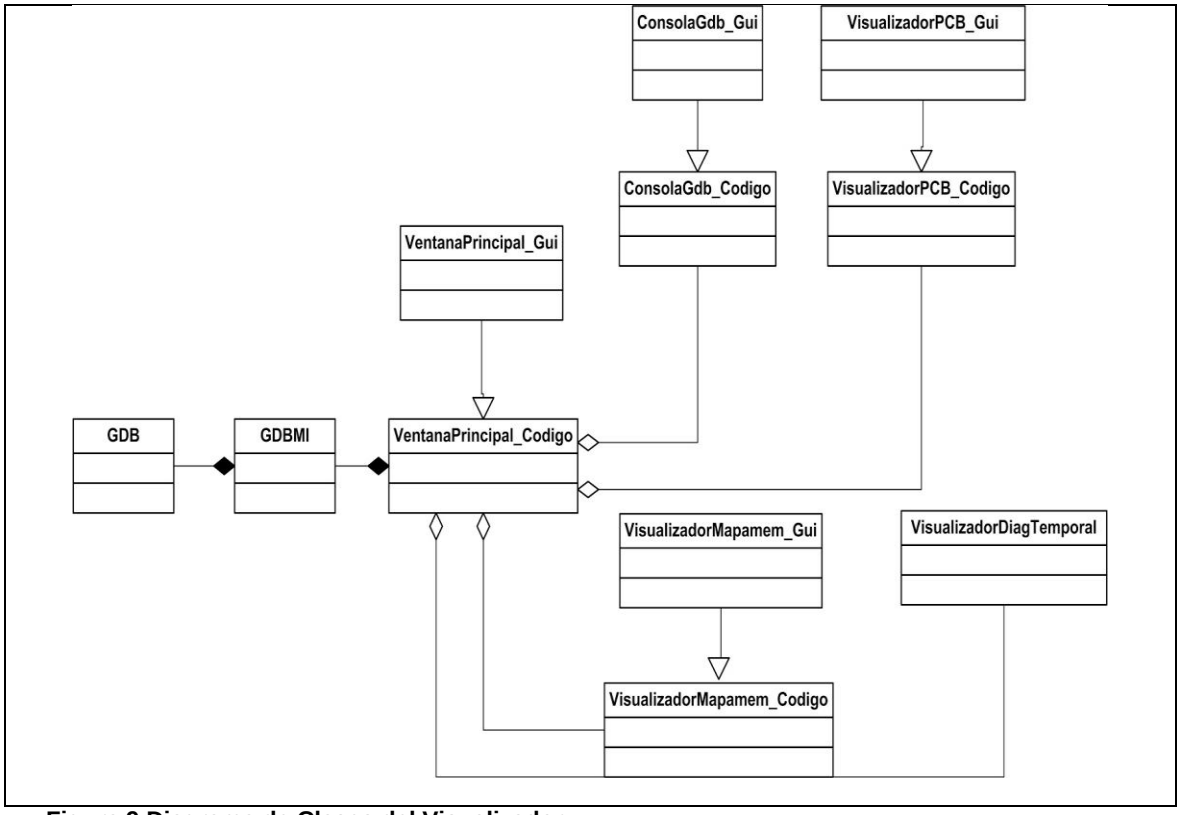


Figura 9 Diagrama de Clases del Visualizador

.

3.4.2.6 Desarrollar navegación, panel de control y capacidad de visualización de los tipos definidos anteriormente

Durante esta etapa se desarrolló la versión final del mapa de navegabilidad y la visualización de los tipos de estructuras del sistema operativo definidas anteriormente. Es importante mencionar, que inicialmente en este apartado, se mencionarán las distintas dificultadas encontradas al llevar a cabo esta fase del proyecto. Por dicho motivo, primeramente se describirá como se ha realizado la Primera versión de los módulos de Visualización de Estructuras y Mapa de Memoria del Sistema Operativo y finalmente se describirá su versión final. Esta última es la versión a la que le corresponde los diagramas anteriormente descriptos.

Implementaciónfinal de GDB-Stub

Una de las tareas que a lo largo del proyecto demandó mayor tiempo de trabajo, fue poder comunicar correctamente el depurador GDB con el Sistema Operativo S.O.D.I.U.M. a través de su propio módulo Gdb-Stub. Por ese motivo, para poder llevar a cabo esta adaptación, fue necesario efectuar una extensa investigación y desarrollo en el Sistema Operativo. Este trabajo fue descrito en los artículos (16) (13) (18) (14) (19). Adicionalmente fue necesario configurar con la correcta versión de GDB el entorno utilizado para el desarrollo de S.O.D.I.U.M., denominado Sodium-Devkit (28). El cual contiene una imagen del S.O Linux, además de todas las herramientas necesarias para poder desarrollar y ejecutar el Sistema Operativo S.O.D.I.U.M. junto al Visualizador. Esta imagen puede ser ejecutada en una máquina virtual Vmware o Virtualbox. Por ese motivo en el **Anexo 4 Sección A** se muestra un pequeño instructivo con los pasos que se realizaron para poder instalar GDB 7.9 dentro de esta

imagen. El archivo que contiene la imagen con el entorno de desarrollo Sodium-Devkit, puede ser descargado desde el siguiente sitio web:

http://so-unlam.com.ar/wiki/index.php?title=PUBLICO:Sodium_Developer_Kit

Además se investigó la manera de instalar GDB en Windows. De esta forma se pretendía que el usuario pueda utilizar el visualizador desde este Sistema Operativo. No obstante, como se explicará más adelante en los siguientes apartados, esto no pudo llevarse a cabo. Sin embargo a continuación, en el **Anexo 4 Sección B,** se describen los pasos que se realizaron para poder instalar GDB en la plataforma Windows.

Establecimiento de conexión entre GDB y S.O.D.I.U.M.

Durante la implementación completa del Gdb-stub de S.O.D.I.U.M. se pudo determinar cuáles eran los pasos correctos que se debían seguir para establecer la conexión entre S.O.D.I.U.M. y GDB, a través de su Stub. Esta guía varía dependiendo del Sistema Operativo en donde se ejecute GDB. Mientras que S.O.D.I.U.M. siempre deberá estar ejecutándose dentro de la imagen de la máquina virtual.

Conexión con GDB desde Sodium-Devkit :

I. **Compilar S.O.D.I.U.M.**
   - Utilizar el siguiente comando desde el directorio raíz donde se encuentre el código fuente de S.O.D.I.U.M. :
     Makeclean test

II. **Ejecutar GDB:**
   - Desde la consola de Linux ejecutar el comando:
     gdb

III. **Establecer la conexión S.O.D.I.U.M. :**
   Una vez que se abra la consola de gdb, se deberán ejecutar los siguientes comandos del depurador
   - Cargar la tabla de símbolos del S.O :
     filevSodium/kernel/main.ld
   - Establecer la velocidad de transferencia de datos utilizadas en el puerto serie:
     set serial baud<velocidad en baudios>
   - Establecer la conexión remota con S.O.D.I.U.M. a través del puerto 12345:
     targetremote<ip>:12345

Durante el desarrollo fue necesario comentar la línea de código, dentro del script *generar_lanzador.sh*, que establece la conexión automática de S.O.D.I.U.M. con GDB. Esto se realizó con la finalidad de evitar la conectividad de forma predeterminada al compilar S.O.D.I.U.M., y poder así efectuar las pruebas pertinentes. Posteriormente, una vez que se logró determinar cuál era la correcta secuencia de comandos de GDB para efectuar la conexión a través del Stub, este archivo fue modificado con los cambios descritos en (19).

En el Anexo 4 Sección C, se describen la secuencia de pasos que fueron necesarios para poder establecer la conexión entre S.O.D.I.U.M., ejecutándose dentro de la máquina virtual, y GDB desde el S.O Windows.

.

Primera versión de los módulos de Visualización de Estructuras y Mapa de Memoria del Sistema Operativo

Inicialmente se desarrollaron los módulos de visualización de estructuras y mapa de memoria haciendo uso de las herramientas que ofrece GDB para extender su set de comandos (8). De esta forma se consiguió visualizar, en un momento determinado durante su ejecución, ciertos componentes del S.O. utilizando dos mecanismos de extensión que ofrece el depurador:

- Archivo de Script de Comandos de GDB.
- Archivo de Script de Python en GDB.

Con el primer mecanismo, GDB le permite al usuario definir una secuencia de comandos específicos como una unidad y luego este conjunto puede ser ejecutado bajo un nuevo nombre de comando. Para su programación el depurador 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 el segundo mecanismo, GDB ofrece la posibilidad de poder ejecutar scripts de Python desde su consola, pudiendo extender el conjunto de comandos del depurador utilizando este lenguaje. Para poder aprovechar dicho beneficio, fue necesario recompilar el código fuente de GDB, utilizando el *flag--with-python* durante su configuración. Cabe destacar, que GDB ofrece diferentes API[7]s 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 S.O.D.I.U.M. durante su ejecución. Utilizando para ello la función:
gdb.execute("printpstuPCB" ,0,1)

Es importante mencionar que GDB provee dos formas distintas de cargar y utilizar los nuevos scripts de comandos. Tanto los realizados en Python (segundo mecanismo citado), como en archivos extendidos de comandos del depurador (primer mecanismo).

- **Como parámetro desde Linux:** La primera forma es ejecutar a GDB, desde la consola de Linux, enviando el parámetro --command como argumento junto al nombre del script que contiene la extensión de los comandos del depurador.

- **Comando source:** El segundo método indicado es ejecutando el comando *source* en la consola de GDB, junto al nombre del archivo del script de Python o con los comandos extendidos que se quieran utilizar. Un ejemplo de este caso sería, "*source script_python.py"*. o "*sourcecomandos_nuevos.gdb*"

a. Archivo de comando "*comandos_nuevos.gdb"*

En consecuencia a lo explicado en el apartado anterior, se aprovecharon las herramientas que brinda GDB, de forma tal que se pudo generar la primera versión de los módulos de estructuras y de mapa de memoria. Para ello primeramente se generó un archivo de comando de Script de GDB, llamando "*comandos_nuevos.gdb"*. El cual tuvo como objetivo generar un conjunto de comandos personalizados con la intención de extender las funcionalidades que ofrece el depurador. Uno de estos comandos generados dentro de este archivo, fue la instrucción *mapamemoria.* Creada con la

[7] **API** *Application Programming Interface*

finalidad de poder observar la ubicación en memoria de los segmentos que conforman cada componente que administra el sistema operativo en un momento determinado. En la Figura 10- Bloque de código del comando Mapamemoria se muestra el bloque de código que define este nuevo comando:

```
define mapaMemoria
    echo  ******************************************************\n
    echo Estructura\t\tInicio\tFin\n
    #Kernel Sodium
    printf "MAIN.BIN\t\t000000\t%x\n",(void*)uiTamanioKernel-1
    printf "BSS\t\t\t%x\t%x\n",uiTamanioKernel,(void*)(uiTamanioKernel+uiTamanioBSS)-1
    printf "Heap Memoria Baja\t%x\t%x\n",ptrHeapMemoriaBaja,(void*)(ptrHeapMemoriaBaja+liTamanioHeapMemoriaBaja)-1
    printf "Heap Memoria Alta\t%x\t%x\n",ptrHeapMemoriaAlta,(void*)(ptrHeapMemoriaAlta+liTamanioHeapMemoriaAlta)-1

    #Estructuras Sodium
    printf "PCB\t\t\t%x\t%x\n", pstuPCB,(void*)pstuPCB+sizeof(stuPCB)*CANTMAXPROCS-1
    printf "GDT\t\t\t%x\t%x\n", pstuTablaGdt,(void*)&(pstuTablaGdt->stuGdtDescriptorDescs[8192])-1
    printf "IDT\t\t\t%x\t%x\n", pstuIDT,(void*)pstuIDT+sizeof(stuIDT)-1
    echo ROM-BIOS\t\ta0000\tfffff\n
    busca_ES
    echo  \n******************************************************\n\n

    echo Procesos de Usuario\n

    muestraProcesosUsuario
end
```

Figura 10- Bloque de código del comando Mapamemoria

Bloque de código del comando Mapamemoria

Resultados Por otra parte, *GuiGDB*es otro comando extendido bastante importante creado dentro del script *"comandos_nuevos.gdb".* Esta instrucción invoca y ejecuta un script de Python *"MenuPrincipalGuiGDB.py"*, que se encargan de mostrar por pantalla la información de las estructuras del S.O que solicita el usuario a través de interfaces GUI. El código de las ventanas gráficas fue escrito en PyQT, de acuerdo a como se había señalado en (14). Además como S.O.D.I.U.M. está desarrollado en base a la arquitectura X86, la información es representada en concordancia a lo especificado por INTEL (21). En la Figura 11- Bloque de código del comando GuiGDB se muestra el bloque de código que define esta instrucción:

```
define GuiGdb
    source Servidor/GuiGDB/MenuPrincipalGuiGdb.py
end
```

Figura 11- Bloque de código del comando GuiGDB

b. Ejecución de los comandos extendidos de GDB

Los comandos personalizados mencionados anteriormente solamente pueden ser ejecutados desde la consola del depurador, luego de haberse establecido la conexión entre S.O.D.I.U.M. y GDB al iniciar el Sistema Operativo. Además las instrucciones generadas únicamente podrán ser ejecutadas por el depurador si solo si la ejecución del sistema operativo fue detenida por el usuario.

Al ejecutar 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 S.O.D.I.U.M. en ese momento En la Figura 12 se puede observar un ejemplo de la ejecución de esta instrucción:

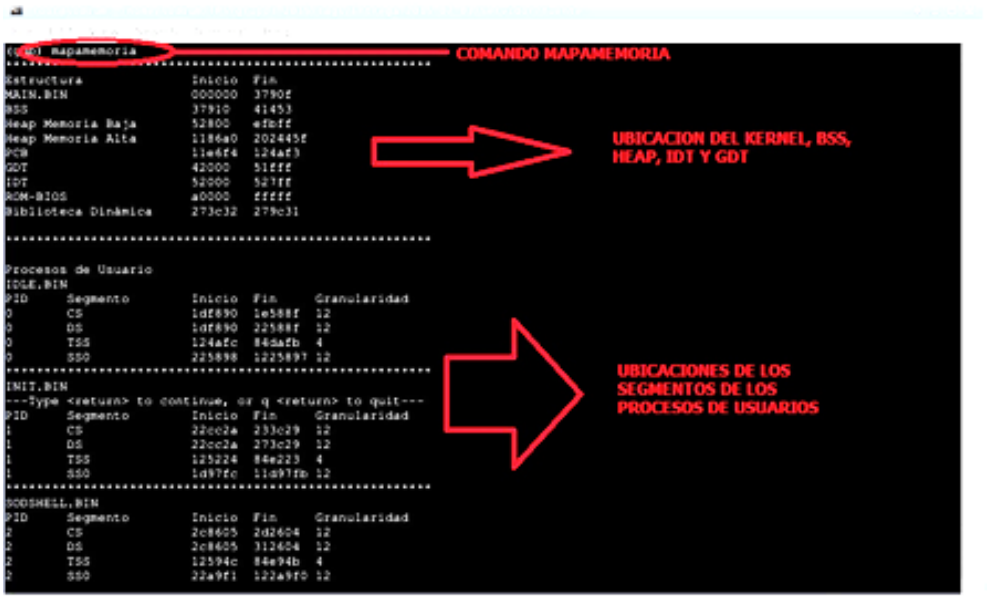


Figura 12 Resultado de la ejecución del comando Mapamemoria desde GDB

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 S.O.D.I.U.M. (Main.bin), así como también su BSS, Heap y las tablas GDT e IDT. Estas ubicaciones son calculadas a partir de los valores que contienen ciertas variables y estructuras del S.O S.O.D.I.U.M. , en el momento que se ejecutó el comando *mapamemoria*. Gracias a que GDB permite inspeccionar el valor de una variable de código durante su ejecución. Característica fundamental para el visualizador

Por otra parte, al ejecutar desde la consola de GDB el comando *GuiGDB*, el depurador mostrará la siguiente interfaz gráfica al usuario. Permitiéndole de esta forma observar la composición de las estructuras internas de S.O.D.I.U.M.

Figura 13 Ventana de selección en la Primera Versión del Módulo Visualizador de Estructura

En la Figura 13 Ventana de selección en la Primera Versión del Módulo Visualizador de Estructurase 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. A modo de

simplificación, únicamente se mostrará en forma gráfica el contenido del PCB de un proceso específico. Dado que para las demás estructuras se utiliza un mecanismo similar:

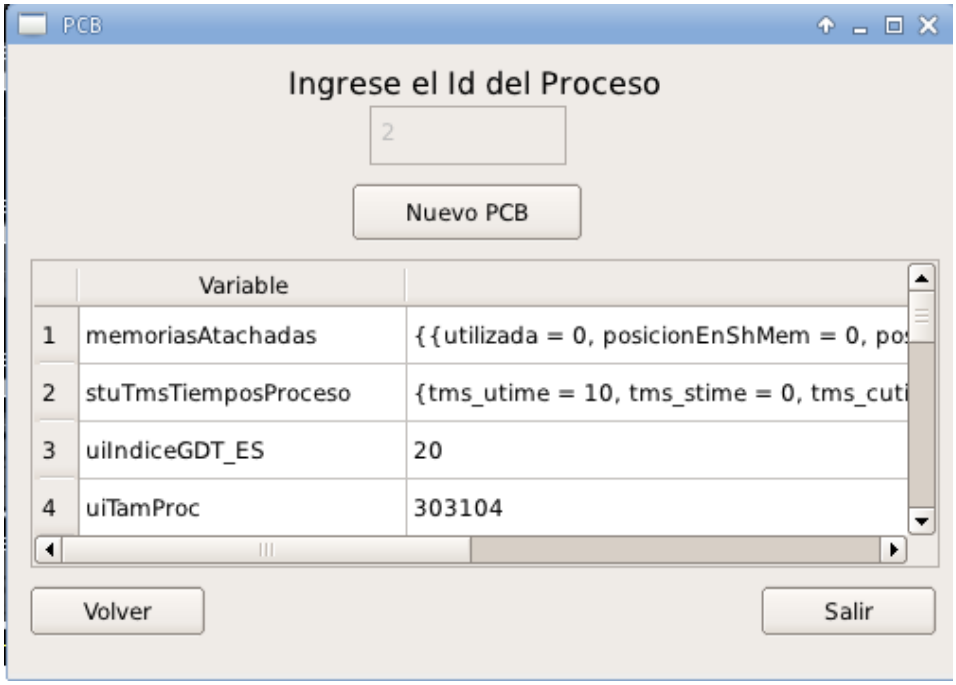


Figura 14 - Ventana gráfica que muestra el contenido del PCB, en la primera versión del Módulo visualizador

Cómo se puede observar en la imagen anterior Figura 14, de acuerdo al formato especificado en la figura 3, se está representando en forma gráfica el contenido de una determinada posición de la estructura de PCB. El depurador internamente, a través de un script de Python, ejecuta la sentencia *printpstuPCB* de la siguiente manera para poder capturar su contenido:

var = gdb.execute(str("print pstuPCB["+idProceso+"]."+word),False,True)

El resultado que devuelve GDB es representado en formato texto, por lo tanto fue necesario almacenarlo en una variable para posteriormente separar los campos de la estructura PCB y luego mostrarlos en forma independiente.

Dificultades encontradas al usar Python desde GDB:

Durante el desarrollo de los distintos módulos que componen el visualizador se presentaron varios inconvenientes al ejecutar un script de Python desde la consola de GDB (19). Estas limitaciones imposibilitaron la ejecución del depurador mientras se mostraba gráficamente una interfaz GUI. Por ese motivo se desistió en seguir ejecutando scripts de Python desde la consola de GDB, con la finalidad de poder mostrar los datos de S.O.D.I.U.M. Esta decisión fue tomada debido a que el uso de esta metodología era bastante limitada.
Para poder resolver esta dificultad se determinó la conveniencia de investigar acerca de la interfaz GDB/MI (8) que ofrece GDB. Dado que esta herramienta es muy utilizada por diferentes *Front Ends* para poder depurar aplicaciones de usuario por medio de

GDB en segundo plano, mientras en simultáneo se muestran distintas ventanas gráficas que representan el estado de la ejecución del proceso en ese instante. Según (29) las aplicaciones más importantes que usan esta interfaz son *Eclipse CDT*, *Netbeands, Codelite, QtCreator, GNU Emacs, WinGDB y Pyclewn*. Por ese motivo se optó aprovechar las utilidades que ofrece GDB/MI para que por medio de su utilización, se puedan mostrar varias interfaces GUI con información de S.O.D.I.U.M. y al mismo tiempo poder seguir con la ejecución normal del Sistema Operativo. A continuación, en la Figura 15 se muestra la relación que existe entre GDB/MI y los *Front Ends*.

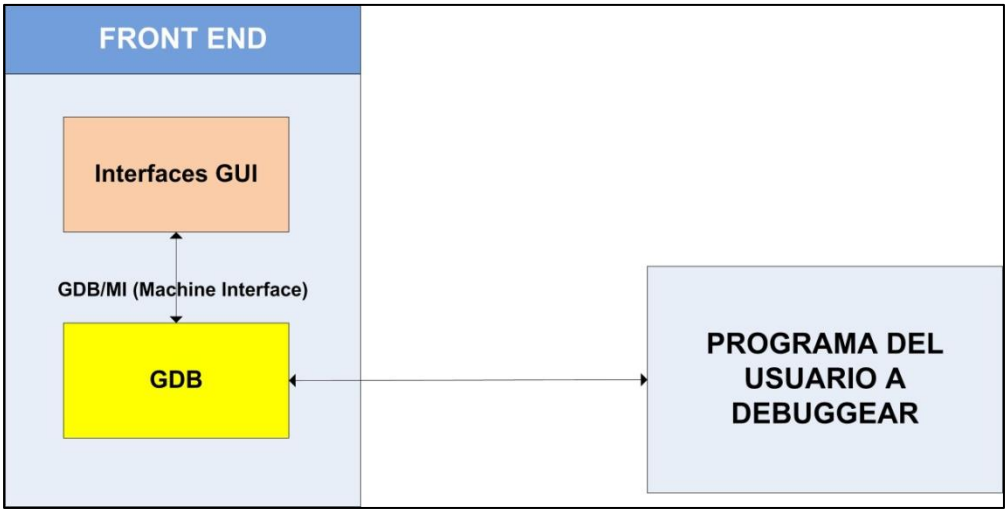


Figura 15 - Relación existente entre GDB/MI y los Front-Ends

Al ser GDB/MI una interfaz con GDB basada en línea de texto, los datos de entrada/salida que intercambia GDB/MI con el depurador deben estar bajo cierto formato (8). En consecuencia esta característica puede ser aprovechada por el visualizador para obtener información que produce la ejecución de un comando de GDB. Por ejemplo, gracias a esta sintaxis se puede determinar el evento que ha ocurrido durante la ejecución del Sistema Operativo, utilizando los registros asincrónicos de GDB/MI. A partir la información que se obtiene de GDB/MI, en el caso de un breakpoint, se puede determinar en qué línea de código ocurrió, el punto de parada y luego asociarlo a un evento específico del S.O, entre otras cosas. Esta información puede ser de mucha utilidad en visualizador para la generación de *Puntos de Instrumentación* (16) (13). A modo de ejemplo en la Figura 16, se puede observar que tipo de información que devuelve GDB/MI al programador una vez que se ejecuta un *breakpoint* durante la ejecución de S.O.D.I.U.M.

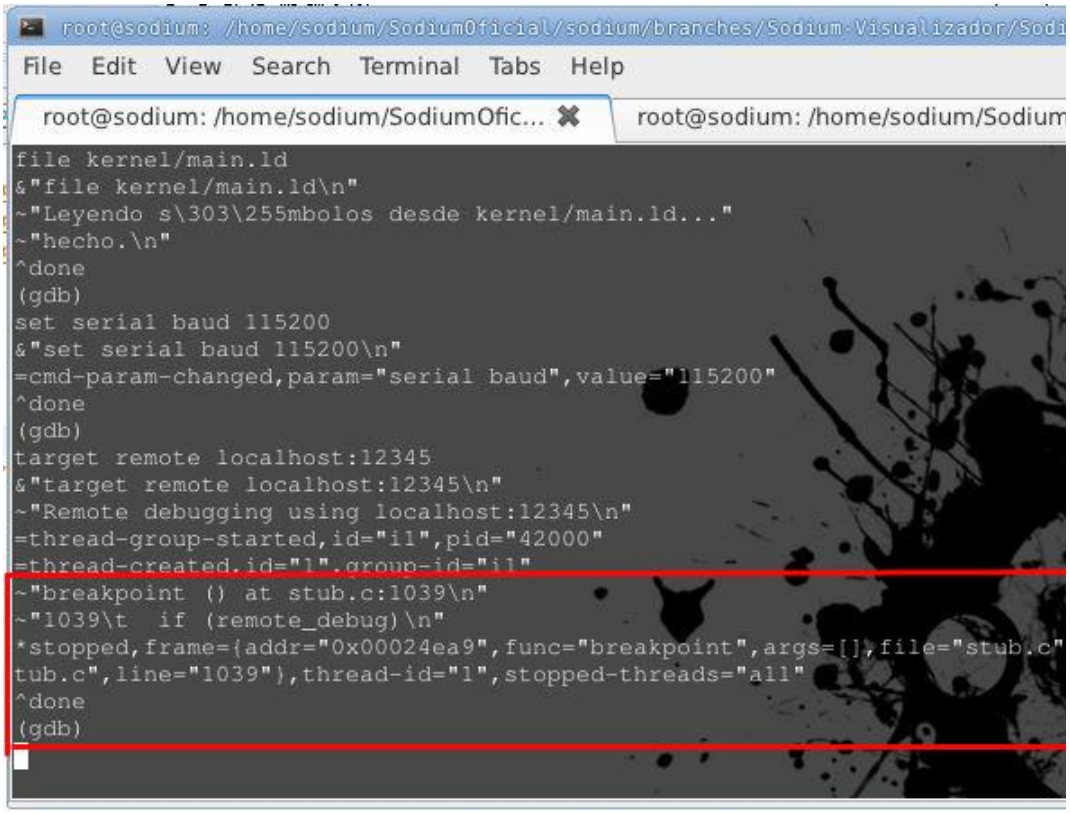


Figura 16- Información de la ocurrencia de un breakpoint devuelta por el depurador al utilizar GDB/MI

En el **Anexo 4 Sección D,** se describe la forma de utilización de la Interfaz GDB/MI a través de su sintaxis.

Análisis del funcionamiento del *Front-End PyClewn*

Como se mencionó anteriormente, se determinó necesario realizar un análisis sobre los mecanismos que utilizan los distintos *Front Ends* existentes en el mercado con GDB/MI. El propósito de este estudio fue para abstraer su funcionamiento básico, con la finalidad de poder sobrepasar las limitaciones explicadas previamente. Como consecuencia del estudio preliminar, se decidió realizar un análisis más profundo de la aplicación Pyclewn (30). Dado que durante las etapas iníciales del desarrollo del visualizador se utilizó Python para su programación y además es el único Front-End escrito en este lenguaje, las demás aplicaciones de este estilo se encuentran desarrolladas íntegramente en C y C++. De esta forma se mantuvo la premisa de que todo el código del Visualizador, sea escrito utilizando un solo lenguaje de programación para que su ejecución sea más eficiente en cuanto a rendimiento.

Introducción al análisis de PyClewn

Pyclewn es un *Front-Ends* para los depuradores GDB y PDB. El cual emplea el editor de texto *Vim* (24) como mecanismo de comunicación con el usuario, por lo que la salida del depurador es redireccionada a la ventana de *Vim,* consola del Sistema Operativo. Los comandos del depurador son mapeados como comandos predefinidos de Vim y se completan si está disponible la línea de comandos del editor de texto. A continuación Figura 17 se muestra una captura de pantalla del *Front-EndsVim* depurando S.O.D.I.U.M.

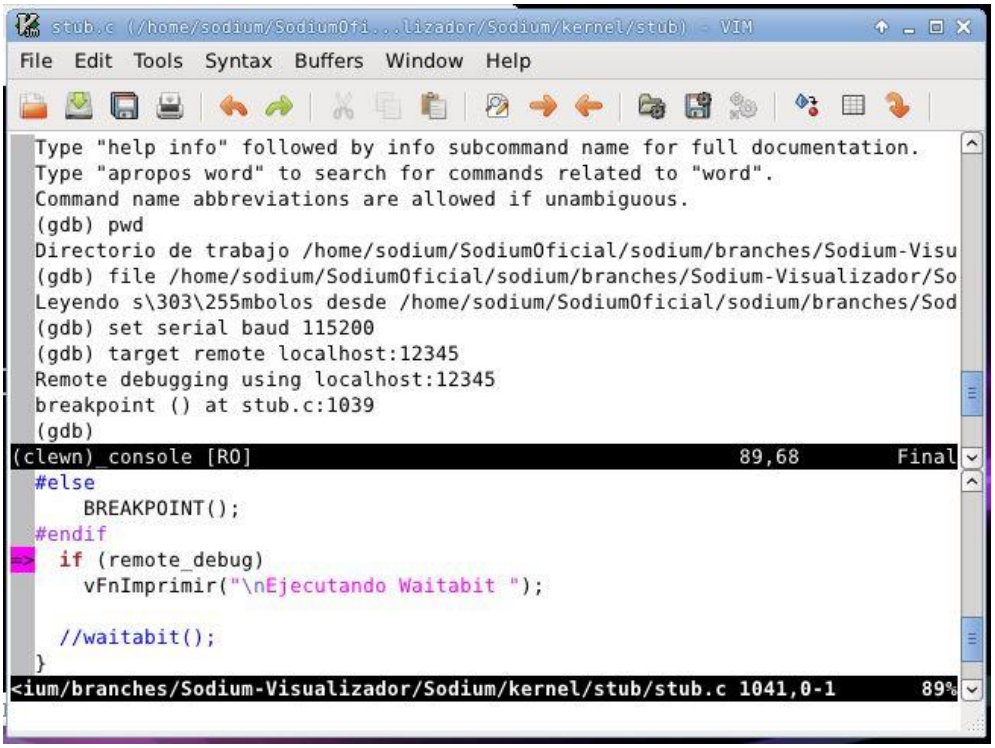


Figura 17 Captura de Pantalla de Pyclewn depurando S.O.D.I.U.M.

-

a. **Instalación de Pyclewn:**

A los fines de poder efectuar el análisis sobre el funcionamiento de Pyclewn, y que sea más accesible realizar la metodología de ingeniería inversa, se tuvo que instalar la aplicación en la máquina virtual VMware en donde se ejecuta el SDK de S.O.D.I.U.M. (28). De esta forma se pudo estudiar en profundidad como internamente se ejecutaba cada línea de código de Pyclewn cuando este depuraba la ejecución de S.O.D.I.U.M. La instalación de este programa fue bastante compleja, dado las dependencias que Pyclewn utiliza. Por dicho motivo en el **Anexo 4 Sección E** se generó un pequeño instructivo con los pasos que se debieron realizar en el Sistema Operativos Linux del SDK para poder instalar este software.

b. **Esquema del funcionamiento de Pyclewn**

La versión de Pyclewn analizada fue la 2.1. Su funcionamiento se basa fundamentalmente en la comunicación asincrónica entre procesos por medio de *eventloops* y los mecanismos de socket asincrónicos que ofrece Python, denominados *Asyncore y Asynchat* (25) (26). En el siguiente gráfico se esquematiza el mecanismo que emplea Pyclewn para ejecutar GDB usando GDB/MI.

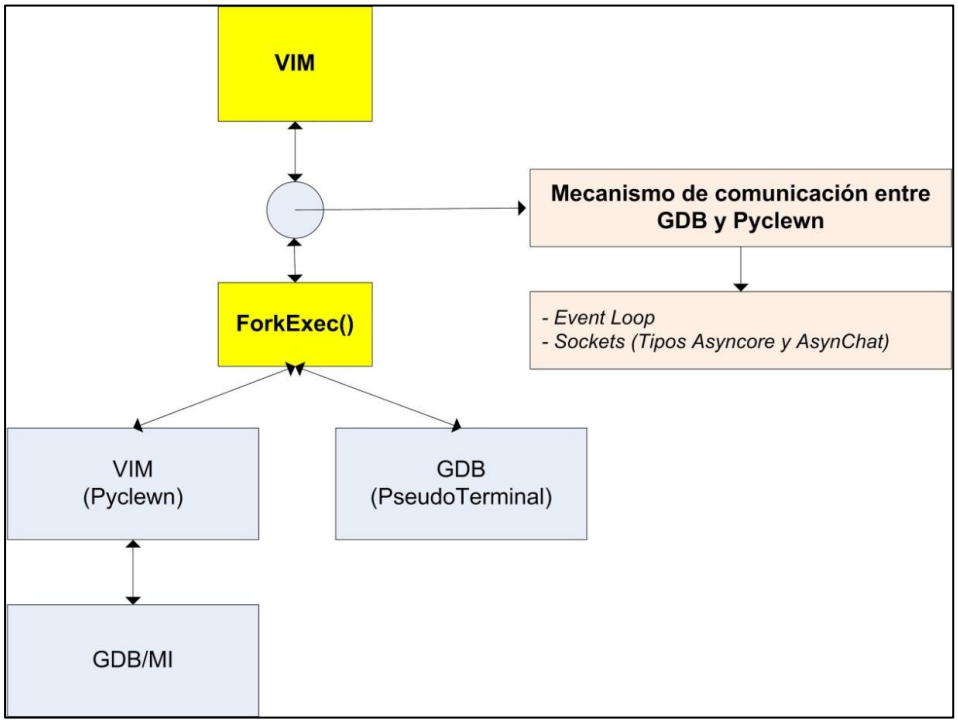


Figura 18 Mecanismo que emplea Pyclewn para ejecutar GDB usando GDB/MI

-

En Pyclewn, Vim es el editor de texto que actúa como Front-End para poder utilizar GDB a través de GDB/MI. Convirtiéndose así en el único punto de entrada que el usuario podrá usar para acceder a esta herramienta. Para poder abrir el editor será necesario ejecutar el script de Python*Pyclewn.py.* Posteriormente Vim será ejecutado como proceso principal y luego durante su ejecución creará un proceso hijo (en el método forkexec(), en posix.py) que ejecutará en segundo plano a GDB utilizando una pseudoterminal. De esta forma el depurador y Vim (Pyclewn) pueden intercambiar información, y por lo tanto se podrá enviar comandos a ejecutar en GDB y recibir su respuesta a través del editor. Cabe destacar que GDB es ejecutado a través del módulo ***subprocess*** de Python (27) El cual permite ejecutar y controlar comandos de Linux, como si hubieran sido ejecutados desde su consola. Por lo tanto GDB se ejecutará siempre a través de ***subprocess,*** invocando el interpreter GDB/MI como se detalla en el **Anexo 4 Sección D**. La captura de los eventos ocurridos tanto en *Vim* (ingreso de comandos por la consola del editor) cómo en GDB (envió de comandos a ejecutar al depurador y recepción de los resultados) es a llevado cabo por medio de un *eventloop*. Este mecanismo detecta un suceso dentro de un proceso y luego se lo comunica a otro por medio de un socket asincrónico, para que luego este haga su procesamiento. Por lo tanto, cada vez que un usuario ingrese un comando para ser ejecutado en GDB en Vim, este evento será capturado por el *eventloop* y luego se lo notificará al depurador para que ejecute este comando. Es importante mencionar, que todos los comandos que ejecuta GDB son realizados a través de llamada "*Gdb –interpreter-exec*", **Anexo 4 Sección E.** En Pyclewn además se emplea el protocolo llamado *Netbeans* (35), necesario para poder controlar el editor Vim.
Por otra parte, el código fuente de Pyclewn está conformado por distintos archivos, cada uno de ellos tiene una funcionalidad específica. Por eso a continuación se mencionan los más importantes y su interrelación a través de la figura 13.

Versión final de los módulos de Visualización de Estructuras y Mapa de Memoria del Sistema Operativo

A continuación se describe el desarrollo de la versión final de los distintos módulos que componen el Visualizador de Estructuras, describiendo en detalle el funcionamiento interno de cada una de las interfaces gráficas que lo conforman. Para ello se explicarán todos los pasos que el usuario deberá seguir para poder interactuar con el sistema.

Desarrollo del Módulo de GDB/MI de S.O.DI.U.M.

Gracias al análisis sobre Pyclewn se pudo comprender cómo un Front End usa GDB/MI para ejecutar comandos de GDB y permitir mostrar en simultáneo varias interfaces GUI mientras se depura el programa usuario. Como resultado del análisis se determinó que para desarrollar el Visualizador era necesario crear como mínimo dos procesos, uno principal y otro que ejecute a GDB en segundo plano a través de ***subprocess***, De forma tal, que el segundo proceso sea el encargado de recibir los comandos que envía el primero para ser ejecutados por el depurador, y que posteriormente éste deba devolverle el resultado obtenido tras haber ejecutado dicha operación en GDB. Por ese motivo fue necesario utilizar un mecanismo de comunicación entre ambos procesos. Inicialmente se pensó en la utilización de socket asincrónicos, cómo Pyclewn. Pero se decidió buscar otra alternativa, dado a que este mecanismo es ineficiente en términos de velocidad de transmisión. Por consiguiente se diseñaron los módulos esenciales del visualizador y su interrelación, tomando para ello los conceptos básicos obtenidos del estudio de Pyclewn.El nuevo diseño del visualizador se elaboró en base al funcionamiento interno de tres archivos que contiene su código fuente que fue descrito anteriormente en el aparatado del diseño del mapa de navegabilidad del sistema.

Ejecución del Visualizador de Estructuras del Sistema Operativo S.O.D.I.U.M.

El usuario para poder inicializar el visualizador, primeramente deberá abrir la imagen de la máquina virtual que se encuentra dentro del Sodium-Devkit. Una vez inicializado Ubuntu, se deberá abrir la consola de comandos de Linux e ingresar al subdirectorio donde se encuentra el directorio raíz del proyecto. Véase **Anexo 5 Apartado 1**. En ese sitio se tendrá que ejecutar un script de la siguiente forma: "*./sodium.sh"*

Al ejecutar este comando automáticamente se comenzará con la compilación del código fuente de S.O.D.I.U.M. y luego se abrirá una nueva consola en donde se ejecutará el S.O. Es importante mencionar que S.O.D.I.U.M. se ejecuta dentro de otra máquina virtual. Por lo que esta nueva ventana corresponderá a la VM Bochs, en donde correrá el sistema operativo ya compilado.

Luego de abrirse la ventana de Bochs, automáticamente se le mostrará al usuario la ventana principal del Visualizador de la siguiente forma:

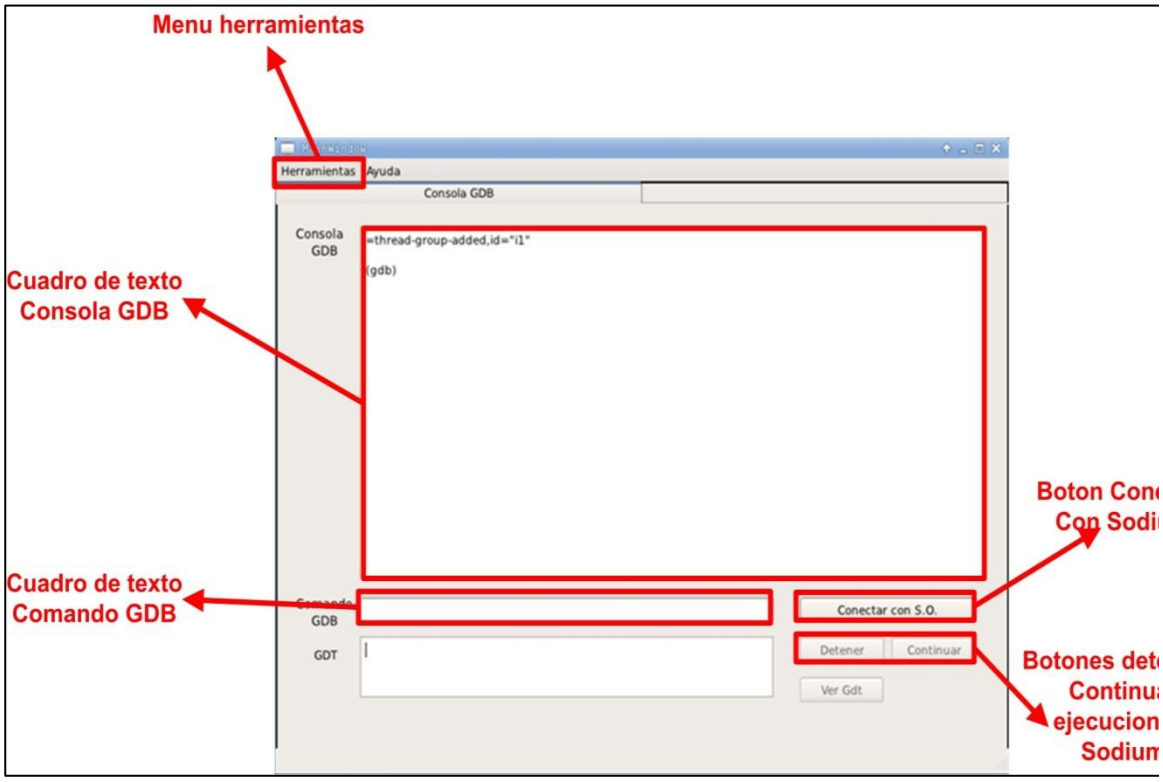


Figura 19 Ventana Principal del Visualizador

Esta interfaz GUI es en cierta parte el Front-End puro de GDB del Visualizador, debido a que por medio de la ventana "*ConsolaGDB"* el usuario podrá interactuar directamente en forma manual la herramienta GDBMI que ofrece el depurador GDB. Esta pantalla correspondería a la implementación del módulo grafico para controlar GDB, mencionado anteriormente en el apartado Módulos gráficos del Visualizador de Estructuras del Sistema Operativo. En consecuencia, de acuerdo a la Figura 19 anterior, desde el **cuadro de texto comando GDB** el usuario podrá enviarles comandos para que ejecute el depurador. Los resultados de cualquier operación serán siempre mostrados en formato GDBMI en el **cuadro de texto consola GDB.**
Una vez que el sistema muestre la interfaz principal del depurador, en la ventana de Bochs se empezará a cargar S.O.D.I.U.M. . En un momento dado el S.O se quedará esperando una conexión remota con el depurador GDB, como muestra la Figura 20.

Para establecer la conexión entre S.O.D.I.U.M. y el Visualizador, el usuario deberá presionar en el botón de la Figura 19 llamado "Conectar con S.O". Una vez establecida la conexión, el S.O continuará su carga normal.

En el **Anexo 5 Apartado 7.3**, se describen los mecanismos utilizados en el código del Visualizador para poder establecer la Conexión con S.O.D.I.U.M.

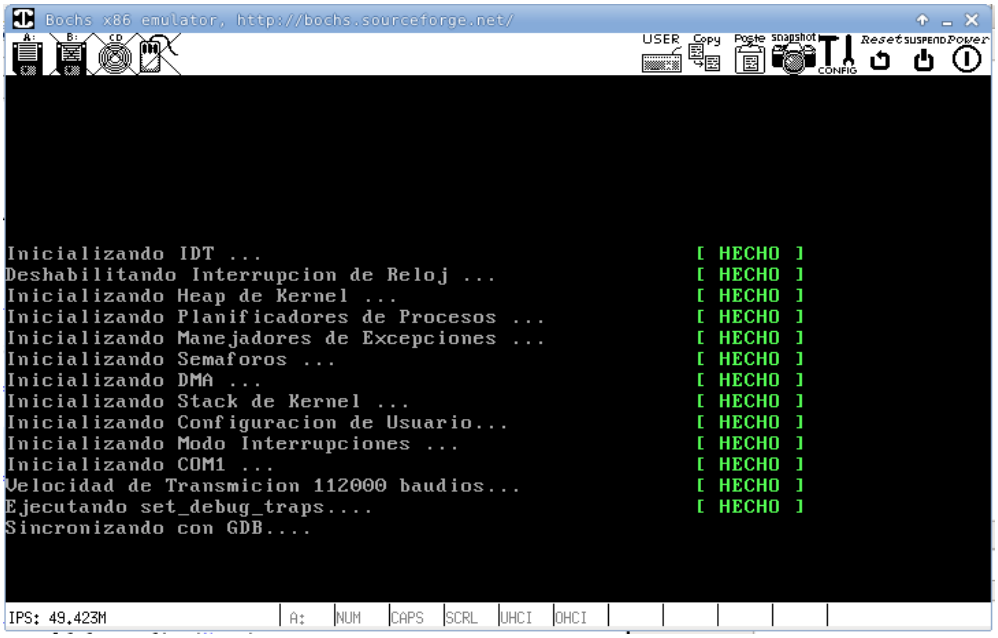


Figura 20 Sincronización entre GDB y S.O.D.I.U.M

Detención y reanudación de la ejecución del Sistema Operativo S.O.D.I.U.M

Si en algún momento el usuario desea detener la ejecución de S.O.D.I.U.M. deberá presionar el *botón "Detener",* indicado en la figura 12. Al suceder esto, el objeto de *ConsolaGDB* le enviará una orden al hilo de GDBMI para que detenga la ejecución del Sistema Operativo . La siguiente figura muestra cómo puede ver la detención de S.O.D.I.U.M.

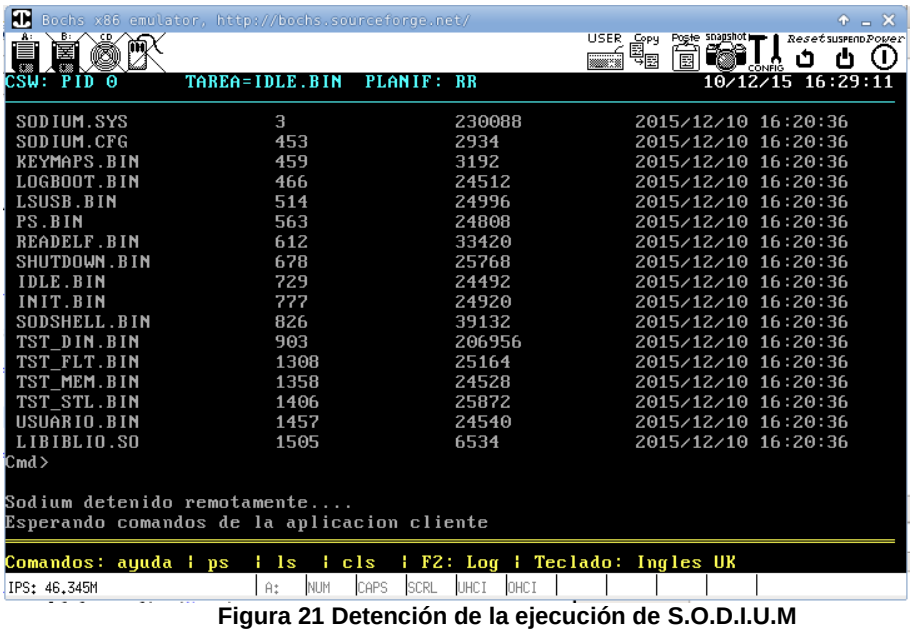


Figura 21 Detención de la ejecución de S.O.D.I.U.M

Luego, cuando el usuario desee reanudar la ejecución del S.O, deberá presionar en el botón "*Continuar".* En ese momento el objeto de *ConsolaGDB* le enviará un comando al depurador para que reanude la ejecución de S.O.D.I.U.M.

Funcionamiento del Módulo Visualizador de Estructuras del S.O

En esta sección se describe el funcionamiento de las interfaces GUI que componen el *Módulo Visualizador de estructuras*. Primeramente el usuario deberá detener la ejecución del S.O para poder visualizar el contenido de una de sus estructuras. Luego tendrá que ir al menú *Herramienta* de la Ventana Principal y seleccionar la opción deseada. Si el usuario no detiene la ejecución de S.O.D.I.U.M, se verán inhabilitadas las opciones para abrir la interfaces encargadas de la visualización de las estructuras.

Figura 22 Menú de Herramientas

Interfaz gráfica "Ver PCB"

La siguiente interfaz nos permite ingresar el índice del PCB del proceso que si quiera visualizar.

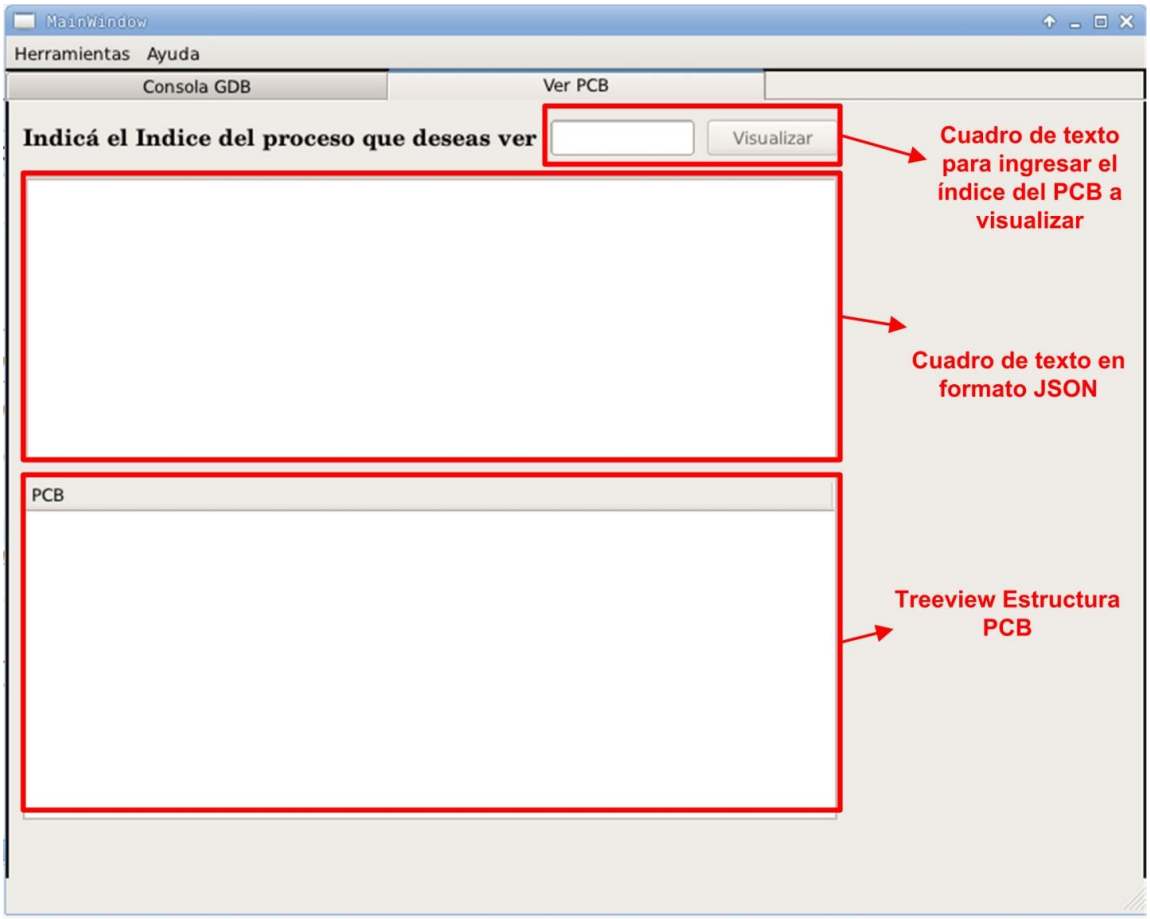


Figura 23 conformación de la Interfaz GUI Ver PCB

Dentro de esta ventana gráfica el usuario primero deberá ingresar, en el cuadro de texto correspondiente, el índice asociado al PCB del proceso que desee ver y luego presionar el botón *Visualizar.* Al suceder esto, el objeto de esta interfaz gráfica le enviará al hilo de GDBMI una orden al depurador, solicitándole la composición en ese momento de los campos del PCB del proceso seleccionado por el usuario. En el Anexo 5 Apartado 7.4 se describe de qué forma se ejecuta internamente este comando.

A continuación en la siguiente figura, se muestra el resultado obtenido de GDB en formato JSON (27), en un cuadro de texto y su asociación en un Treeview.

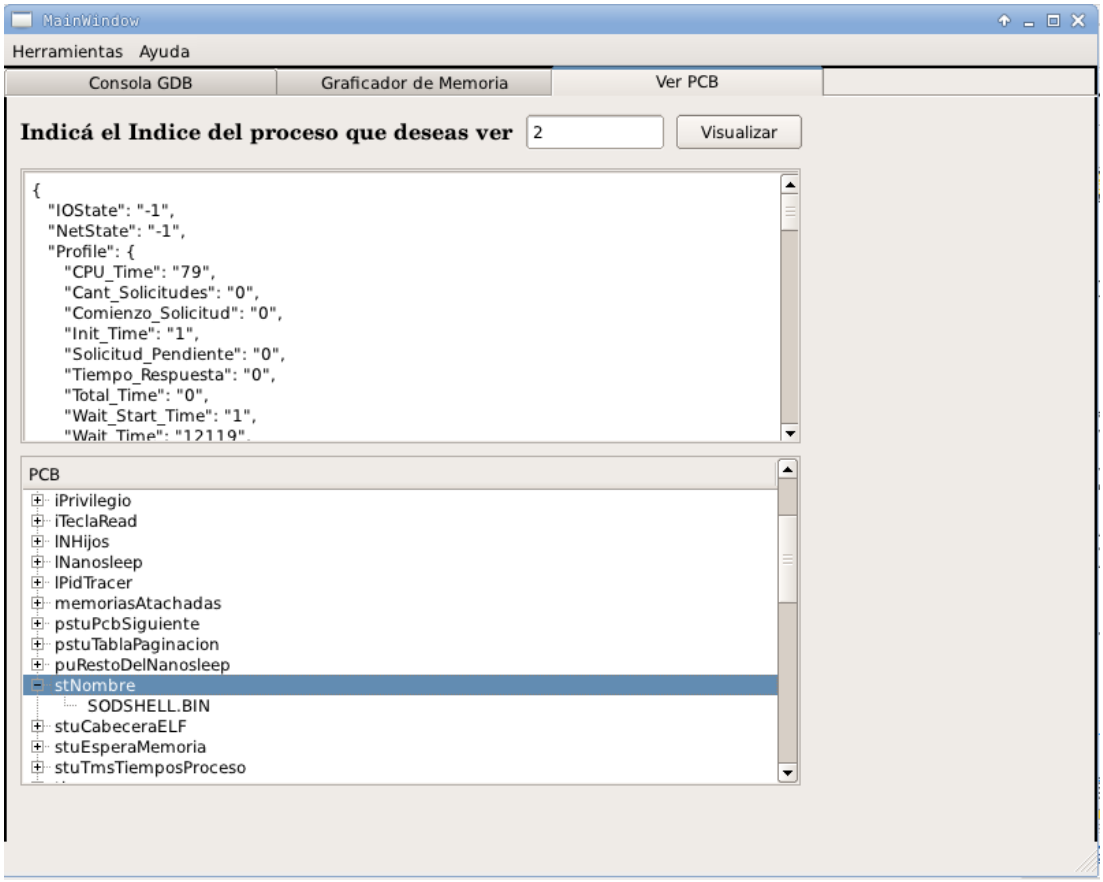


Figura 24 Interfaz Ver PCB con datos de la estructura

Como se puede apreciar en esta figura, los campos que componen la estructura del PCB son los mismos que fueron mostrados en el apartado Formato de las Estructuras IDT, GDT, TSS y PCB del Sistema Operativo S.O.D.I.U.M. .

Interfaz gráfica “Ver IDT”:

La siguiente interfaz nos permite ingresar el índice de la tabla IDT que se desea visualizar. La composición y funcionamiento de esta ventana es muy similar al de “Ver PCB”, con la diferencia de que en este caso se muestra el contenido de una posición determinada de la IDT.

A continuación en la siguiente figura, se muestra el resultado obtenido del GDB en formato JSON y en un Treeview:

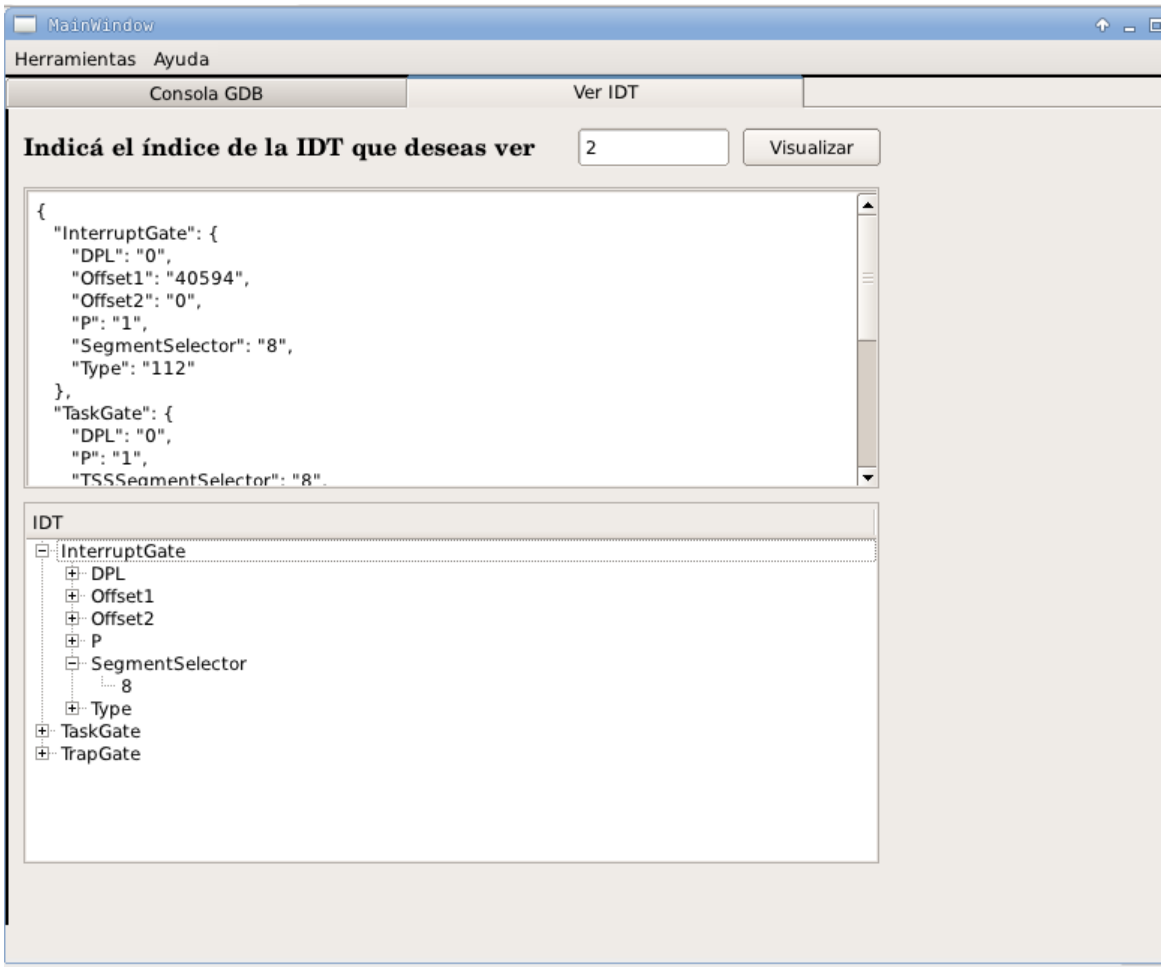


Figura 25 Interfaz Ver IDT con datos de la estructura

Interfaz gráfica “Ver GDT”:

La siguiente interfaz nos permite ingresar el índice de la tabla GDT que se desea visualizar. La composición y funcionamiento de esta ventana es muy similar al de “Ver PCB”, con la diferencia de que en este caso se muestra el contenido de una posición determinada de la GDT.

A continuación en la siguiente figura, se muestra el resultado obtenido del GDB en formato JSON y en un Treeview

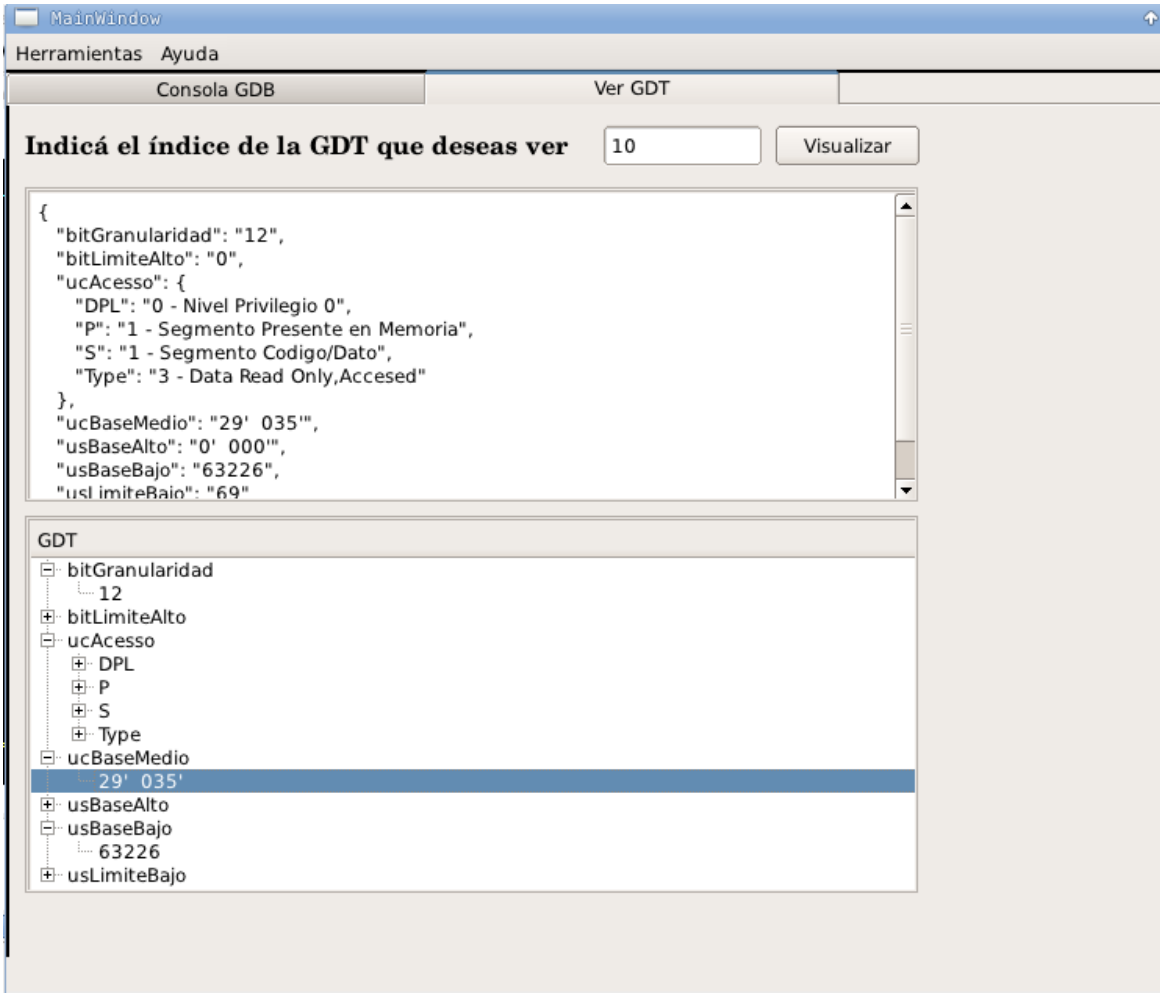


Figura 26 Interfaz Ver GDT con datos de la estructura

Interfaz gráfica “Ver TSS”:

La siguiente interfaz nos permite ingresar el índice de la tabla TSS que se desea visualizar. La composición y funcionamiento de esta ventana es muy similar al de “Ver PCB”, con la diferencia de que en este caso se envía el siguiente comando a ejecutar en el depurador

A continuación en la siguiente figura, se muestra el resultado obtenido del GDB en formato JSON y en un Treeview:

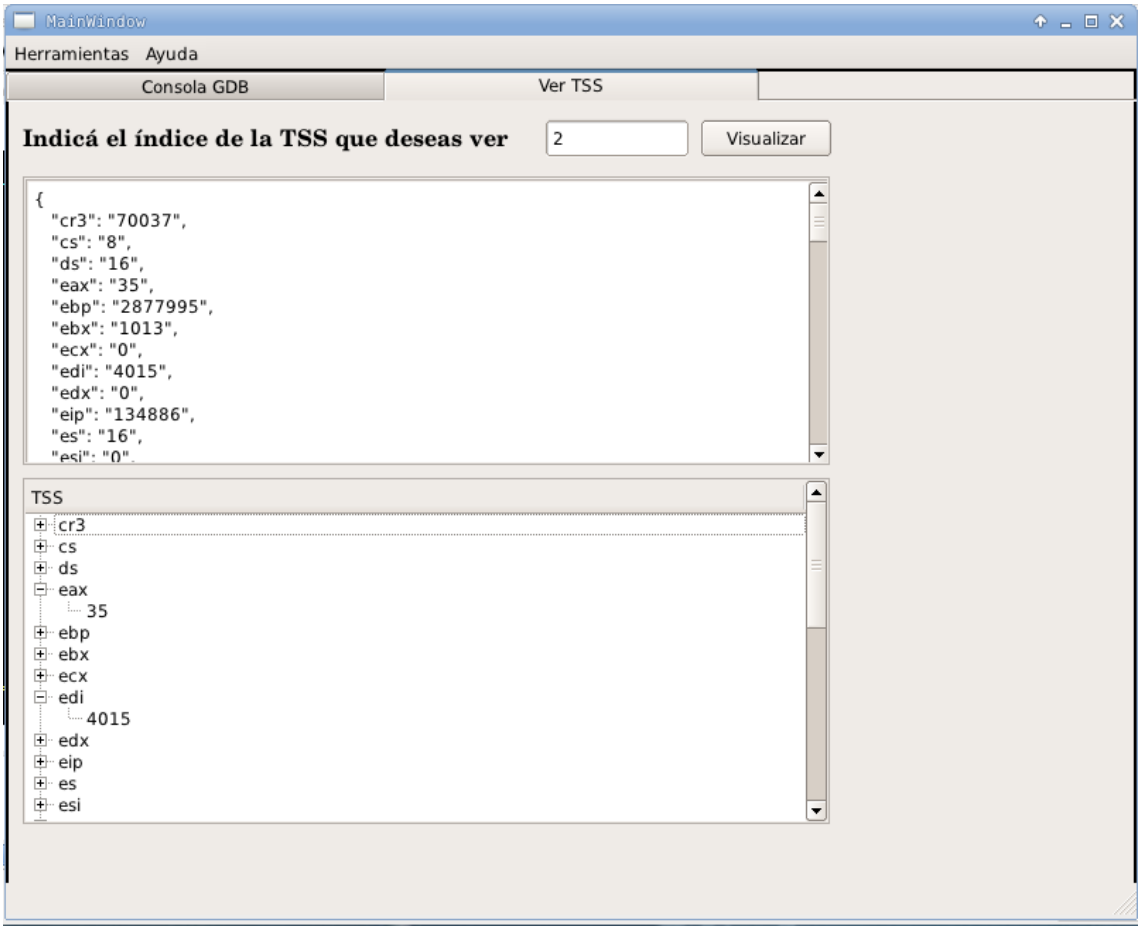


Figura 27 Interfaz Ver TSS con datos de la estructura

Interfaz gráfica "Ver Mapa Memoria":

La siguiente imagen pertenece a la interfaz GUI encargada de representar gráficamente el estado de la memoria de la computadora en donde se esté ejecutando S.O.D.I.U.M en un momento determinado. Esta pantalla corresponde a la implementación Módulo de visualización del Mapa de Memoria, mencionado en un apartado anterior.

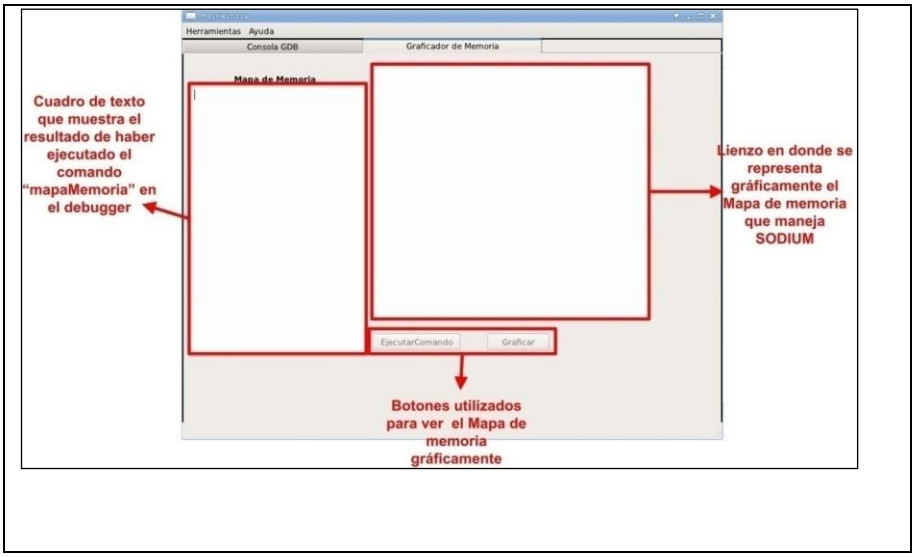


Figura 28 Conformación de la Interfaz Ver Mapa memoria

Con la finalidad de poder representar gráficamente el estado de la memoria, se debió redefinir el comando *Mapamemoria* de GDB, definido dentro del archivo *"comandos_nuevos.gdb".* En el Anexo 5 Apartado 7.5 se describe la readaptación de esta instrucción en el código fuente.
Una vez que se haya mostrado por pantalla esta ventana gráfica, el usuario deberá primero presionar en el Botón *ejecutarComando.* Al realizar dicha acción el objeto de la Interfaz *GUI*, le ordenará al hilo GDBMI que ejecute en el depurador el comando *mapamemoria.* Al recibir el resultado de la operación, la interfaz automáticamente almacenará y mostrará en el cuadro de texto pertinente, la impresión de las tuplas con los datos de los segmentos de memoria.

Luego, al terminar mostrar todas las tuplas devueltas por el depurador el visualizador habilitará el botón *Graficar.* Por lo que si el usuario presiona ese botón, automáticamente se dibujarán en el lienzo de la Interfaz las ubicaciones de todos los bloques de memoria donde se encuentran cargados en ese momento. En la siguiente figura se muestra un ejemplo de este caso:

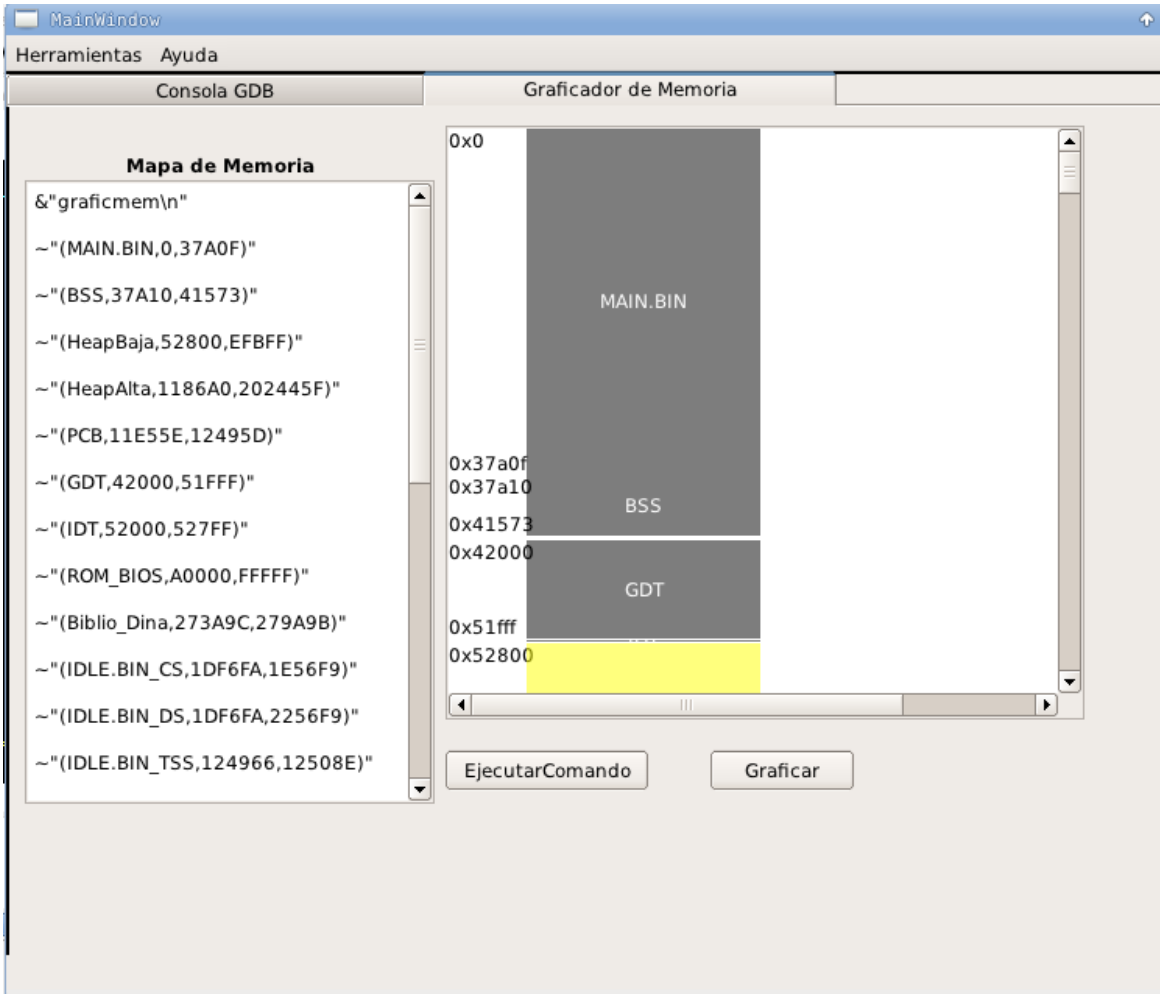


Figura 29 Interfaz Ver Mapamemoria mostrando gráficamente los segmentos

Se han utilizado las clases del módulo Graphics view de PyQT, para conseguir graficar los segmentos de memoria. Las cuales permiten dibujar cualquier figura geométrica a través de Canvas (10). Cuando el usuario selecciona el botón *Graficar*, el contenido de las tuplas será convertido a un diccionario, para luego dibujar en el lienzo cada segmento de memoria como un bloque gráfico representado con un color característico. En cada uno de ellos detallan su dirección de Inicio y Fin, como se muestra en la imagen subsiguiente.

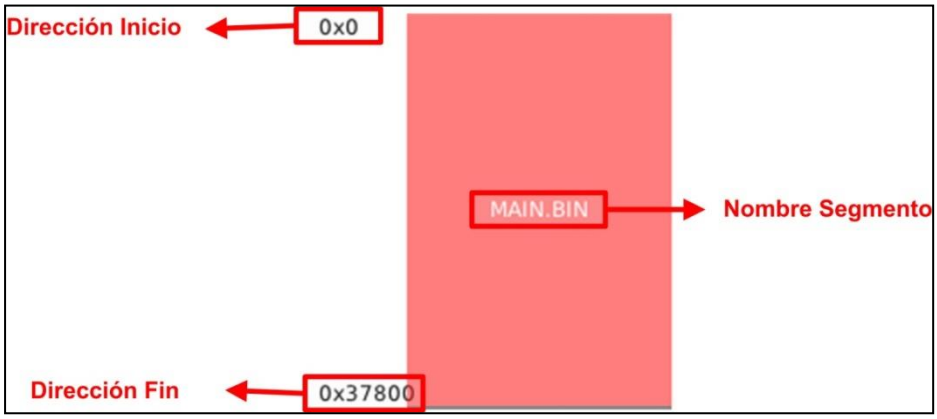


Figura 30 Representación gráfica de un bloque de memoria

En caso de que el usuario quiera modificar o simplemente ver sus propiedades, deberá seleccionar dicho bloque con el mouse. Al suceder esto se mostrará por pantalla una ventana secundaria que contendrá dicha información. Se debe tener en cuenta que el único ítem que no se puede alterar es el Nombre del bloque, ya que se decidió que éste sea único y no modificable por cada bloque existente en la aplicación.

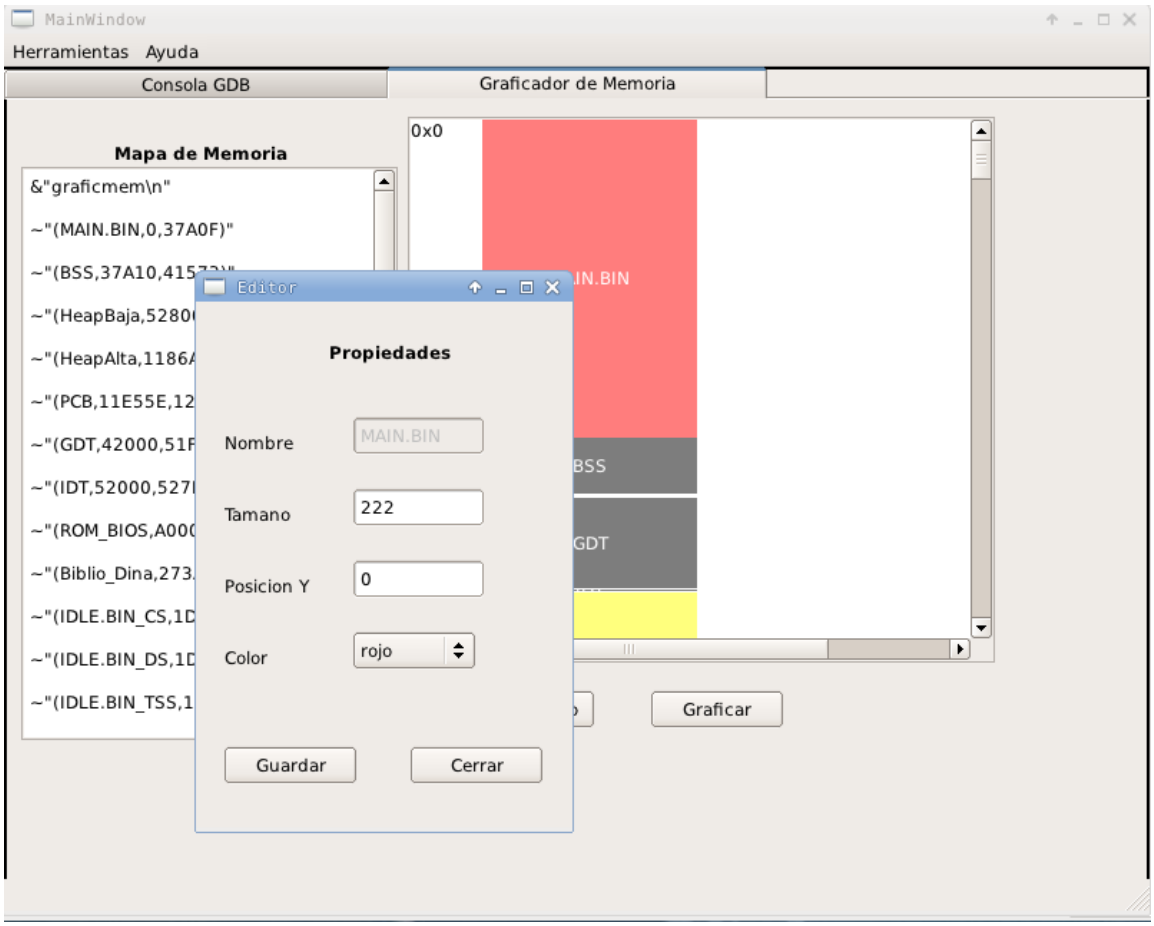


Figura 31 Ventana secundaria de la Interfaz Mapamemoria

Una vez hecho los cambios pertinentes, se debe seleccionar el botón Guardar. Con lo cual, una ventana de dialogo informará que dichas modificaciones fueron exitosas y

luego, para visualizar dichos cambios, se deberá seleccionar en el botón *Cerrar* y después en el botón *Graficar*. Finalmente el visualizador actualizará los bloques en el lienzo y se verán las modificaciones solicitadas.

Debido a que el sistema operativo S.O.D.I.U.M presenta procesos que se pueden superponer en memoria, se tuvo que contemplar esas situaciones y lograr replicar el caso. Por ese motivo se tuvo especial cuidado en la forma de superponer gráficamente los bloques, agregando una transparencia para que sea más visible esta situación especial.

Interfaz gráfica "Ver Planificador – Diagrama Temporal":

Esta sección corresponde a la implementación del *Módulo de visualización de Diagrama Temporal,* mencionado en un apartado anterior.
Como en todo Sistema Operativo, en S.O.D.I.U.M. los procesos van adquiriendo distintos estados a lo largo de su vida. Para ello el sistema necesita un mecanismo que le permita controlar estos cambios. La estructura PCB (36) contiene todos los datos para administrar un proceso, como se muestra en el **Anexo 5 Apartado 3.3**. Uno de los campos definidos dentro de esta estructura, corresponde a los posibles estados que puede adquirir un proceso: PROC_LISTO, PROC_EJECUTANDO, PROC_ESPERANDO, PROC_ELIMINADO, PROC_DETENIDO, PROC_NO_DEFINIDO. Con lo cual se desarrolló el visualizador del Diagrama Temporal, de forma que el usuario pueda observar gráficamente los cambios de estados de los procesos que se producen durante la ejecución de S.O.D.I.U.M. No obstante para evitar problema de rendimiento durante la ejecución del Diagrama Temporal, se determinó conveniente graficar únicamente las siguientes transiciones ocurridas en el S.O:

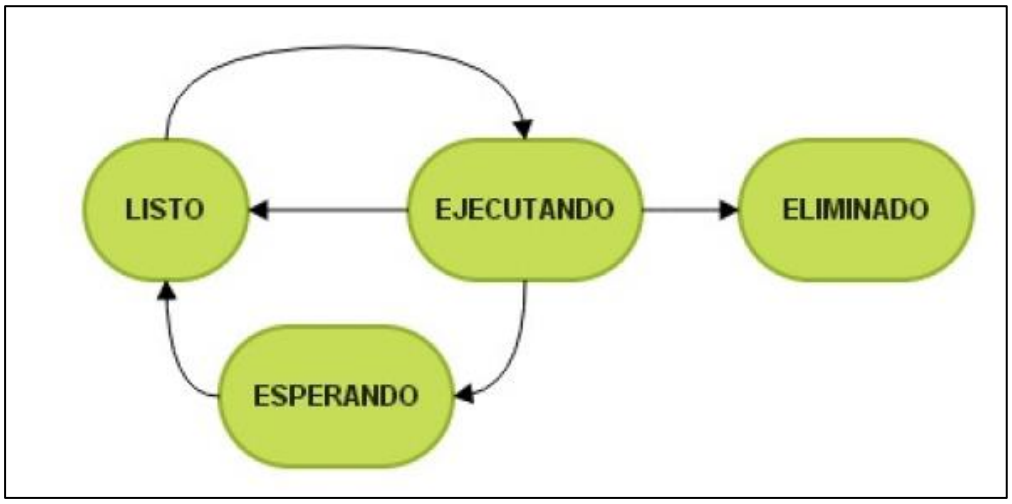


Figura 32 Transiciones de estados de los procesos de usuario que pueden ocurrir en el S.O S.O.D.I.U.M

Cada uno de estos estados es representado utilizando un color determinado. El criterio de la tonalidad elegida es la que se muestra a continuación:

Figura 33 Color asignado a cada estado de un proceso

Esta ventana puede ser accedida por usuario al seleccionar en menú de Ayuda del Diagrama Temporal.

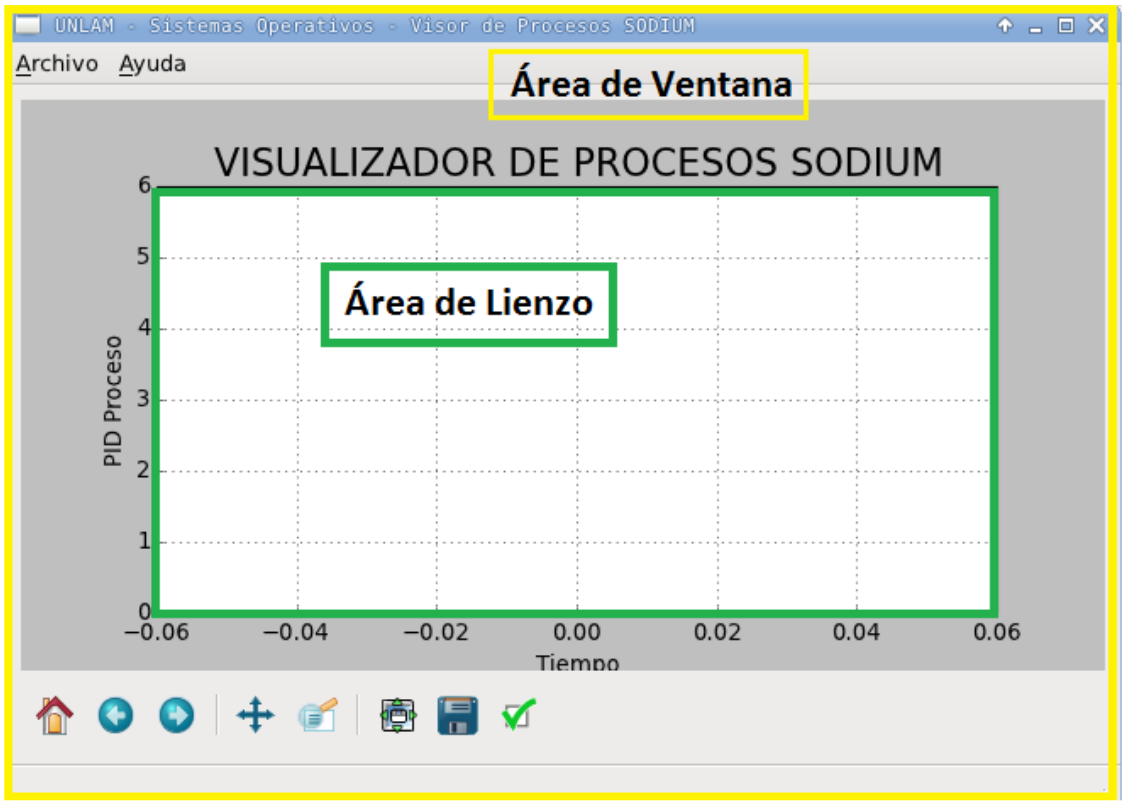


Figura 34 Conformación de la Interfaz Diagrama Temporal

Con la finalidad de graficar el diagrama temporal durante la ejecución del S.O, se utilizó un gráfico de barras que se va ir dibujando en un lienzo a medida que se van produciendo, en tiempo real, los cambios de estado durante la ejecución. Para ello se utilizó la biblioteca MatplotLib realizada en Python (30). La cual debió ser instalada en la imagen de Sodium-Devkit. Los pasos realizados para su configuración fueron detallados en el **Anexo 4 Sección F.**

En la siguiente imagen se muestra un ejemplo del funcionamiento del diagrama Temporal:

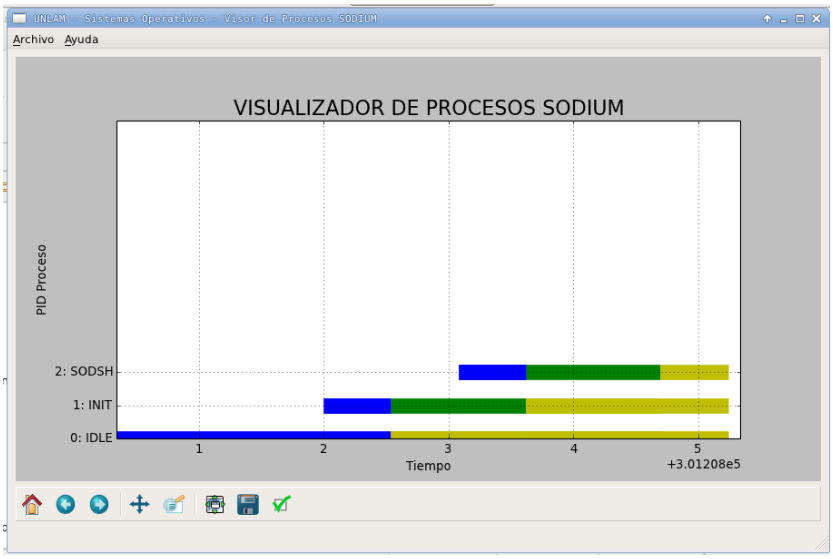


Figura 35 Funcionamiento del Diagrama Temporal

En el eje Y del diagrama, se detallan los nombres de cada proceso usuario que se encuentra activo en el sistema. En cambio en el eje X se muestra el tiempo en que se producen las transiciones de los cambios de estado.

Definición de un Punto de Instrumentación:

Se decidió crear Puntos de Instrumentación con el objetivo de poder detectar en el visualizador la ocurrencia de ciertos eventos en el S.O. Estos consisten en establecer determinados *breakpoints* en el código de fuente de S.O.D.I.U.M y asociarlos a un suceso cuando ocurran durante la ejecución del sistema. Como se explicó anteriormente, a través del uso de la herramienta GDBMI del depurador, los *breakpoint* pueden brindarnos mucha información que los pueden identificar. Dado que permiten determinar en qué línea de un determinado archivo del código fuente se detuvo la ejecución del S.O. Característica que fue aprovechada para diseñar y desarrollar el diagrama temporal de visualizador. En consecuencia, se generaron un listado de breakpoints, definidos dentro del archivo *breakpoint.xml,* en determinados lugares dentro del código fuente del S.O donde se producen los cambios de estados de los procesos. Dentro de este archivo, cada Punto de Instrumentación deberá ser definido en un Nodo denominado B*reakpoint* mediante los siguientes atributos:

- **Archivo:** Nombre del archivo en donde se debe definir el *breakpoint*
- **Línea:** Número de línea del archivo en donde se debe establecer el punto de parada
- **Función:** Nombre de la función donde se define el *breakpoint*
- **Evento:** Mensaje descriptivo del suceso ocurrido que representa ese Punto de Instrumentación.

En consecuencia a continuación, se muestra un ejemplo con la definición de un Punto de Instrumentación dentro del archivo breakpoint.xml.

<breakpoint archivo="dispatcher.c" línea="100" función="vFnDispatcher" evento="1 - Listo a Ejecutando"/>

Establecimiento y eliminación de los Puntos de Instrumentación:

En el código del Visualizador se utilizó el Patrón *Singleto*n (31), para definir una clase llamada *Configuracion_Xml,* que al ser instanciada lee los Puntos de Instrumentación del archivo XML y los almacena en un diccionario Python. La creación de este objeto *Singleton* se crea dentro de la clase de la interfaz *ConsolaGDB,* cuando se abre el Visualizador y se muestra la Ventana Principal. Por otro lado los breakpoints asociados a los Puntos de Instrumentación, son establecidos por el depurador cada vez que el usuario abre la Interfaz GUI del Diagrama Temporal. No obstante, cuando el usuario cierre dicha ventana gráfica, se le ordena a GDB que los elimine. Fue desarrollado de esta forma, como consecuencia de que se produce una merma de rendimiento bastante considerable causada por la ejecución continúa de *breakpoint*s en el Sistema Operativo La causa de la lentitud se debe a que por cada detección de un Punto de Instrumentación, se detiene la ejecución del Sistema Operativo y posteriormente se realiza la extracción de ciertos datos de S.O.D.I.U.M para poder graficar el estado de los procesos. Este mecanismo será explicado más en detalle en el siguiente apartado. Por ese motivo S.O.D.I.U.M y el Visualizador se ejecutarán más lento mientras se encuentre abierta la interfaz GUI del diagrama Temporal. Pero volverán a su velocidad de ejecución normal, una vez que el usuario cierre dicha ventana

En el Anexo 5 Apartado 7.6 se describe de qué manera en el código del Visualizador se detectan los Puntos de Instrumentación durante la ejecución del sistema operativo.

3.4.2.7 Escribir manual de usuario

En el **Anexo 4** se encuentra detallada la guía de compilación**,** configuración**,** instalación y uso del sistema de visualización generado en este proyecto.
También se puede encontrar información en el sitio WEB del grupo de investigación http://so-unlam.com.ar/wiki/index.php?title=PUBLICO:S.O.D.I.U.M , conjuntamente con el código del sistema operativo S.O.D.I.U.M., el Visualizador y todo lo necesario para su utilización.

3.4.3 Tercera Etapa

3.4.3.1 Desarrollar capacidad de transmisión vía puerto serie en visualizador. Negociación de velocidad de transferencia, puertos de comunicación.

El desarrollo de la capacidad de transmisión y negociación de la velocidad vía puerto serie fue llevado a cabo en diferentes etapas anteriores, durante la implementación del puerto serie, adaptación de Gdb-Stub en S.O.D.I.U.M y desarrollo del Visualizador en Python. En consecuencia, este punto de la planificación del proyecto fue descrito en apartados previos.

3.4.3.2 Análisis de mensajes recibidos desde el sistema operativo inspeccionado y viceversa.

Como se mencionó en apartados anteriores, se utilizó el protocolo RSP para la comunicación serie entre El sistema Operativo y el *Front-End* de GDB desarrollado en el visualizador de estructuras. Como GDB/MI utiliza un formato especial para los mensajes, transmitidos por Gdb-Stub, con la información de las estructuras internas de

S.O.D.I.U.M, fue necesario realizar una decodificación de los mismos en los diferentes módulos que conforman el Visualizador de Estructuras. Dicha transcripción fue descrita en apartados anteriores y en el **ANEXO 4 Y 5**.

3.4.3.3 Pruebas de integración

A lo largo del desarrollo del sistema Visualizador, se fueron realizando distintas pruebas de su funcionamiento. Además dicho sistema fue utilizado como herramienta de estudio en cada etapa de su construcción por los alumnos de la Cátedra Sistemas Operativos a lo largo de su cursada, realizándose las modificaciones necesarias

3.4.4 Etapa Final

3.4.4.1 Generación de vistas adaptables a otros sistemas operativos.

Las vistas se desarrollaron para el sistema operativo S.O.D.I.U.M., pudiéndose a posterior realizar modificaciones en el kernel de otros sistemas operativos abiertos y el visualizador.
La implementación actual del Visualizador fue desarrollado para funcionar únicamente junto al S.O S.O.D.I.U.M. , ya que como se mencionó anteriormente en este informe, las estructuras graficadas fueron pensadas dependiendo de ese Sistema. No obstante, si se realizan las modificaciones pertinentes en su código fuente se podría llegar a adaptar a otros Sistemas Operativos.

3.4.4.2 Empaquetar el desarrollo en un instalador.

Todo el sistema Visualizador se encuentra empaquetado en el entorno de desarrollo de S.O.D.I.U.M, que posee ya instaladas las herramientas necesarias para su correcto funcionamiento. El cual puede ser descargado desde el sitio web del grupo de investigación a través de la siguiente URL.

http://so-unlam.com.ar/wiki/index.php?title=PUBLICO:Sodium_Developer_Kit

El código fuente del S.O S.O.D.I.U.M y del Visualizador, ya se encuentra almacenado dentro de la imagen de la máquina virtual Sodium-Devkit que se puede descargar desde la página web antes mencionada. Por ende, dentro del SDK la ubicación de los archivos fuentes del proyecto se encuentra en el siguiente directorio de Linux "/home/Sodium/Sodium-Visualizador".
El código fuente que se entrega junto con el presente documento, se encuentra organizado en distintos directorios. En el **ANEXO 5** se describen las funcionalidades de cada subdirectorio, como así también la de sus archivos más importantes.

3.4.4.3 Generación de guía de uso y aplicación.

La guía de uso y aplicación fue anteriormente mencionada en el apartado "Empaquetar el desarrollo en un instalador". No obstante a continuación se describen algunas consideraciones a tener cuenta en el uso y aplicación del Visualizador de Estructuras internas de un Sistema Operativo.
Al inicio del Proyecto se había planteado la hipótesis que el visualizador sea multiplataforma, para que pueda ser ejecutado tanto en un Sistema Operativo Linux como en Windows. Por ese motivo se eligió al lenguaje Python para su construcción, dado que permite crear código de programas que funcionen de igual manera en ambas plataformas. Sin embargo, a lo largo del proyecto se descubrió que muchas bibliotecas Python utilizadas en el desarrollo funcionan en Linux pero no en Windows. Además, para poder utilizar el depurador GDB en Windows, es necesario instalar programas adicionales para su funcionamiento, como MINGW (32). MINGW, es un programa que

emula funcionamiento de ciertos componentes del S.O Linux en una plataforma Windows. Con lo cual el desarrollo del Visualizador para Windows sería diferente en ciertos aspectos al que se realizó en Linux. Por ese motivo se determinó conveniente implementar y desarrollar el visualizador en un entorno controlado. Para eso se empleó el entorno de desarrollo Sodium-Devkit, que utiliza Linux como S.O base. De esta manera el usuario puede utilizar y programar la versión actual de Visualizador junto a S.O.D.I.U.M sin necesidad de realizar configuraciones adicionales en su sistema
Al desarrollar el Visualizador en forma gráfica con GDB hubo que realizar adaptaciones en el código de S.O.D.I.U.M que impidieron poder visualizar las estructuras del S.O durante su ejecución desde una maquina física. Con lo cual el visualizador final únicamente podrá ser ejecutado junto a S.O.D.I.U.M dentro de una máquina virtual, a través de Sodium-Devkit.

Como se mencionó, cuando el usuario abre la interfaz del Diagrama Temporal, se reduce considerablemente la velocidad de la ejecución del Sistema Operativo S.O.D.I.U.M y del Visualizador. Esto se debe a la constante detenciones de su ejecución como consecuencia de los *breakpoints* asociados a los Puntos de Instrumentación. Es muy conveniente optimizar el código del Visualizador para mejorar la velocidad de su ejecución y que de esta forma pueda ser más rápido la representación gráfica de los cambios de estado de los procesos.

Durante la ejecución del Sistema raras veces se generan problemas de sincronización entre el Visualizador y el depurador. Esto es debido a problemas de diseño para la sincronización entre los comandos que se envían a GDB y los que se están ejecutando. Actualmente todos interfaces gráficas se ejecutan desde el hilo principal del Visualizador. Por lo que una posible solución a los problemas de sincronización mencionados, sería que todas las ventanas graficas se ejecuten desde su propio hilo de ejecución.

4 Resultados

Durante el segundo año del proyecto se consiguió desarrollar el sistema Visualizador de estructuras para un sistema Operativo determinado. Obteniendo de esta forma grandes logros. Dado que se pudo conseguir en parte con la meta planteada al iniciar el proyecto. Durante la investigación se consiguió aplicar los conocimientos adquiridos en este periodo para poder comunicar S.O.D.I.U.M. , funcionando como un Servidor, a una aplicación cliente desarrollada totalmente Python para obtener el estado de los componentes que conforman ese Sistema Operativo, en tiempo real, durante su ejecución. Para eso se utilizó la herramienta que ofrece el depurador GDB, conocida como *GDB-Stub*, utilizada para depurar sistemas remotos. La cual, se debió implementar en el código fuente del S.O, realizando para ello extensos adaptaciones en el mismo para su correcto funcionamiento. Al poder integrar este módulo al S.O, se obtuvo en un enorme avance en el proyecto. Dado que esta utilidad nos permitió controlar totalmente a S.O.D.I.U.M. desde el Visualizador y de no poder haber conseguido este logro hubiera sido muy difícil continuar con el desarrollo del proyecto. Otro beneficio obtenido fue comprender como funcionan los *Front-Ends* de GDB que utilizan las herramientas editoras de código. Al estudiar el funcionamiento de *Pyclewn,* se consiguió destrabar una gran limitación encontrada que se nos presentó durante el proyecto. Dado que, como se mencionó, la ejecución de script de Python desde GDB resultaba bastante restrictiva al momento de mostrar información de S.O.D.I.U.M gráficamente. Pero gracias al uso de la herramienta GDB/MI se pudo superar dicho obstáculo. Gracias al visualizador desarrollado ahora se pueden apreciar en forma gráfica la composición de diversas estructuras del S.O S.O.D.I.U.M, observar la

composición de su mapa de memoria y ver de qué forma los procesos cambia sus estados durante la ejecución del Sistema. Cuestiones que eran imposibles de apreciar sin este programa.

5 Conclusiones

De acuerdo con las premisas plateadas inicialmente, podemos concluir que el Visualizador de Estructuras funciona tal cual lo esperado para el sistema operativo S.O.D.I.U.M. permitiendo la detención, observación y reanudación del sistema. También es importante aclarar que se consiguió implementar gráficamente la información propuesta como necesaria para la comprensión del sistema.
Para la ejecución desde otras máquinas del visualizador, se consiguió que éste ejecute desde Linux, quedando detallada la forma de ejecutarlo en Windows, pero debido a inconvenientes con las bibliotecas de Python se dejó documentada la información de la configuración para Windows, dejando la ejecución en este último para su continuación en un nuevo proyecto.
También se sentaron las bases de conocimiento para la ejecución del visualizador en otros sistemas operativos abiertos, debiendo realizarse cambios en el kernel de estos que deberán también realizarse a tal fin. Estos corresponderán a un nuevo proyecto.
Cabe destacar que el uso del visualizador y el análisis de su estructura fueron probados con pequeños grupos alumnos de sistemas operativos en la ejecución y en su compilación, los que aportaron modificaciones al sistema desde su particular punto de vista.
El sistema completo permite su utilización por parte de otras universidades, encontrándose disponible en http://www.so-unlam.com.ar para su descarga.

6 Bibliografía

1. *Desarrollo de un sistema operativo didáctico.* **Casas, Nicanor, De Luca, Graciela y Martín., Cortina.** Corrientes, Corrientes Argentina : RedUNCI, 2007. ISBN 950-656-109-3.
2. *Using Visualization to understand the Behavior of Computer Systems.* **Bosch, Robert P. Jr.** Palo Alto, Californa USA : Stanford University, 2001.
3. *Integrated Standard Environment for the Teaching and Learning of Operating Systems Algorithms Using Visualizations.* **Alharbi, A., Henskens, F. y Hannaford, M.** s.l. : Computing in the Global Information Technology -ICCGI- Fifth International Multi-Conference on, 2010.
4. **GRUPO-SODIUM UNLaM.** so-unlam. [En línea] http://www.so-unlam.com.ar/sodium/sdk.html.
5. *Serial HowTo.* **Lawyer, David.** s.l. : Greg Hankin, 2011. Vol. Capítulo 18.
6. Puertos Serie. *http://wiki.osdev.org/Serial_Ports.* [En línea]
7. *Embedding with GNU: the GDB Remote Serial Protocol.* **Gathiff, Bill.** 1999, Embedded Systems Programming, pp. 108-113.
8. **Stallman, Richard.** *Debugging with GDB.* Boston : Free Software Foundation, 2015. págs. 271-274.
9. **Durda, F.** www.freebsd.org. *Serial and UART Tutorial.* [En línea] 29 de 04 de 2014. https://www.freebsd.org/doc/en/articles/serial-uart/..
10. **Summerfield, Mark.** *Rapid Gui Programming witf Python and QT.* Massachusetts. : Prentice Hall, 2007. págs. 1-579.
11. **Von Rossum, Guido.** *El tutorial de Python.* Reston, VA / Estados Unidos : Fred L. Drake Jr., 2009. págs. 1-115.
12. *Visualizador de Estructuras de un Sistema Operativo Real con Fines Educativos.* **De Luca , Graciela, y otros, y otros.** San Miguel de Tucumán : Universidad del Norte

Santo Tomás de Aquino. UNSTA, 2014. II Congreso Argentino de Ingeniería- CADI 2014. ISBN: 978-987-1662-51-7.
13. **Bennett, Jeremy.** *Howto: GDB Remote Serial Protocol. Writing a RSP Server.* San Francisco, California : Embecosm, 2008.
14. **Freebsd.org.** freebsd. *Serial and UART Tutorial.* [En línea] Serial and UART Tutorial. http://www.freebsd.org/doc/en/articles/serial-uart/. [En línea] .
15. **OS Dev .ORG.** Puertos Serie. *http://wiki.osdev.org/Serial_Ports.* [En línea] 2015. http://wiki.osdev.org/Serial_Ports..
16. Serial and UART Tutorial. *UART.* [Online] 2014. http://www.freebsd.org/doc/en/articles/serial-uart/.
17. **Sourceforge.net.** Bochs Emulator Project. *sourceforge.* [En línea] 2014. http://bochs.sourceforge.net/doc/docbook/user/bochsrc.html.
18. **WMware Inc.** WMWARE INC. [Online] 2014. http://www.dpunkt.de/leseproben/1686/Kapitel_2.pdf.
19. **GNU Project. org.** GNU Project. *Debugger.* [Online] 2014. https://sourceware.org/gdb/onlinedocs/gdb/Remote-Stub.html.
20. *Desarrollo de un prototipo para un visualizador de estructuras de un.* **De Luca, Graciela, y otros, y otros.** Salta, Argentina : Wicc 2015, 2015. XVII Workshop de Investigadores en Ciencias de la Computación. 978-987-633-134-0.
21. *Mecanismos de visualización de estructuras de un sistema operativo en ejecución a través de la comunicación serial.* **De Luca, Graciela, y otros, y otros.** Ushuia, Tierra del Fuego : WICC 2014, 2014. XVI Workshop de Investigadores en Ciencias de la Computación. 978-950-34-1084-4.
22. **Liechti, Chris.** *PySerial-Documentation- Release 2.6.* 2011.
23. **python.org.ar.** Interfaces gráficas (GUI). *python.org.ar/InterfacesGraficas.* [En línea] 2015. Interfaces gráficas (GUI)", http://python.org.ar/InterfacesGraficas.
24. **Summerfiled, Mark.** *Rapid Gui Programming with Python and QT.* s.l. : Prentice Hall, 2007.
25. **Intel.** *Intel® 64 and IA-32 Architectures Software Developer's Manual",.* s.l. : Intel, 2011.
26. *Visualizador de Estructuras de un Sistema Operativo Educativo.* **De Luca, Graciela, y otros, y otros.** La Matanza, Buenos Aires : Universidad Nacional de La Matanza, 2014. CACIC 2014 - XX Congreso Argentino de Ciencias de la Computación - CACIC 2014. 978-987-3806-05-6.
27. *Módulo gráfico de un Visualizador de Estructuras de un Sistema Operativo Educativo a través de GDB-Stub.* **De Luca, Graciela, y otros, y otros.** Junín, Buenos Aires, Argentina : Universidad de La Plata, 2015. XXI CONGRESO ARGENTINO-CACIC 2015. 978-987-3724-37-4.
28. **SODIUM.** SO-UNLAM. *SODIUM.SDK.* [En línea] 2009. http://www.so-unlam.com.ar/sodium/sdk.html.
29. **Foundation, Free Software.** sourceware.org. *FrontEnds.* [En línea] 18 de 12 de 2014. https://sourceware.org/gdb/wiki/GDB%20Front%20Ends.
30. **Pyclewn.** Pyclewn. [En línea] 2015. http://pyclewn.sourceforge.net/.
31. **Robbins, Arnold y Hannah, Elbert.** *Learning the Vi and Vim Editors.* Estados Unidos : O' Reilly Media, 2008. págs. 145-315.
32. **Van Rossum, Guido.** *The Python Library Reference.* s.l. : Python Software Foundation, 2015. págs. 681-687.
33. **Hellman, Doug.** *The Python Standard Library by Example.* Michigan, Estados Unidos : Addison-Wesley, 2011. págs. 619-634.
34. **Hellmann, Doug.** *Python Module of the Week.* 2015. págs. 215-224,513-545.
35. **Prieur, Gordon.** vimdoc.sourceforge.net. [En línea] 29 de Septiembre de 2010. http://vimdoc.sourceforge.net/htmldoc/netbeans.html.
36. **Stalling, William.** *Operating Systems Internals and Design Principles.* New Jersey, Estados Unidos : Prentice Hall, 2012. págs. 106-153.

37. **Tosi, Sandro.** Capítulo 6."Embedding Matplotlib in Qt 4". *Matplotlib for Python Developers.* s.l. : Packt Publishing, 2009.
38. **Summerfield, Mark.** *Python in Practice: Create Better Programs Using Concurrency, Libraries, and Patterns.* Indiana, Estados Unidos : Addison Wesley, 2014. págs. 26-29.
39. **Shpigor, Ilya.** *Instant MinGW Starter.* Bimingham. UK : Pack Publishing, 2013.
40. **J.Bennet.** *Howto: GDB Remote Serial Protocol. Writing a RSP Server.* San Francisco, California : Embecosm, 2008.
41. **Hankin, Greg and David, Lawyer.** Capítulo 18 What are UARTs? How Do They Affect Performance. *The Linux Document Project.* [Online] Febrero 2011. http://www.tldp.org/HOWTO/Serial-HOWTO.html.
42. **forge.net, source.** Bochs Emulator Project. [En línea]

7 Producción científico-tecnológica

7.1 Publicaciones

a) Artículos

AUTOR (ES): De Luca, Graciela; Cortina, Martín; Casas, Nicanor; Carnuccio, Esteban; Martín, Sergio
TÍTULO Mecanismos de visualización de estructuras de un sistema operativo en ejecución a través de la comunicación serial. ISBN 978-950-34-1084-4

FUENTE Workshop de Investigadores en Ciencias de la Computación **WICC 2014**
VOLUMEN 1.
TOMO.
NÜMERO 6354
PÁGINAS 45-49
EDITORIAL Universidad Nacional de Tierra del Fuego.
LUGAR Ushuahia (Tierra del Fuego, Argentina).
FECHA 2014

AUTOR (ES): De Luca, Graciela; Cortina, Martín; Casas, Nicanor, Carnuccio, Esteban; Barillaro Sebastián; Martín, Sergio; Puyo, Gerardo
TÍTULO Visualizador de Estructuras de un Sistema Operativo Real con Fines Educativos
FUENTE II Congreso Argentino de Ingeniería – **CADI 2014 ISBN: 978-987-1662-51-7**
VOLUMEN
TOMO. CAP.8
NÜMERO T498
PÁGINAS
EDITORIAL CONFEDI.
LUGAR San Miguel de Tucumán, Tucumán.
FECHA 2014

AUTOR (ES): De Luca, Graciela; Cortina, Martín; Casas, Nicanor; Carnuccio, Esteban; Barillaro, Sebastián; Martín, Sergio; Puyo, Gerardo
TÍTULO Visualizador de Estructuras de un Sistema Operativo Educativo
FUENTE **CACIC 2014**

VOLUMEN 1.
TOMO.1
NÜMERO 6733
PÁGINAS 294-303
EDITORIAL Universidad Nacional de La Matanza. ISBN 978-987-3806-05-6
LUGAR San Justo, Buenos Aires
FECHA 2014

AUTOR (ES): De Luca, Graciela; Cortina, Martín;Casas, Nicanor; Carnuccio, Esteban; Martín. Sergio
TÍTULO Mecanismos de visualización de estructuras de un sistema operativo en ejecución a través de la comunicación serial.
FUENTE Workshop de Investigadores en Ciencias de la Computación WICC 2015
VOLUMEN 1.
TOMO.
NÜMERO 6354
PÁGINAS 1-5
EDITORIAL Universidad Nacional de Salta ISBN 978-987-633-134-0
LUGAR Salta
FECHA 2015

AUTOR (ES): De Luca Graciela; Cortina Martín; Casas Nicanor; Carnuccio Esteban; Martín. Sergio
TÍTULO Módulo gráfico de un Visualizador de Estructuras de un Sistema Operativo Educativo a través de GDB-Stub
FUENTE CACIC 2015
VOLUMEN 1
TOMO.
NÜMERO 7589
PÁGINAS
EDITORIAL UNNOBA
LUGAR Junin, Buenos Aires
FECHA 2015

b) Capítulos de libro

AUTOR (ES): Incorpore a todos los autores del trabajo, con su apellido y el nombre separados por una coma. Es importante que ingrese los autores en el orden en que figuran en la publicación. Especifique si es necesario, la función de cada autor entre paréntesis. Ej.: Apellido Nombre. Puede ser autor, editor, compilador.
TÍTULO del trabajo.
FUENTE: el título del libro en que fue publicado el trabajo.
ISBN: Indicar el código de ISBN
PÁGINAS inicial y final del trabajo. Ej. : 18- 25
PÁGINAS totales del documento mayor.
COLECCIÓN: indique si el libro (documento mayor) pertenece a una colección o serie.
EDITORIAL, FECHA, LUGAR DE EDICIÓN, EN PRENSA como se indica para artículos.

c) Libros

AUTOR (ES): Ingrese todos los autores del trabajo como se indicó para capítulos. Es importante que ingrese los autores en el orden en que figuran en la publicación. Especifique si es necesario, la función de cada autor entre paréntesis. Ej. : Apellido Nombre. Puede ser autor, editor, compilador.
TÍTULO del libro.
SUBTÍTULO: en caso de tener subtítulo ingresarlo en este campo.
PÁGINAS totales del libro.
ISBN: Indicar el código de ISBN.
COLECCIÓN: indique si el libro pertenece o forma parte de una colección o serie.
EDITORIAL, FECHA, LUGAR DE EDICIÓN, EN PRENSA como se indica para artículos.

d) Congresos Internacionales, Nacionales, Simposios, Jornadas, otros

AUTOR (ES): Incorpore el apellido y el nombre de todos los autores del trabajo. Es importante que ubique los autores en el orden en que figuran en la publicación. Especifique si es necesario, función de cada autor entre paréntesis. Ej. : Apellido, Nombre (compiladora. Puede ser autor, editor, compilador).
TÍTULO del trabajo.
SUBTÍTULO: en caso de tener subtítulo ingresarlo en este campo.
TIPO: aclare si es una conferencia, simposio / ponencia o una comunicación libre (póster o exposición)
REUNIÓN: Incorpore el nombre que identifica a la reunión (conferencia, congreso, simposio, workshop, taller, jornadas.) En que se presentó el trabajo que se está cargando.
LUGAR: Ingrese el nombre del lugar geográfico donde se realizó la reunión.
FECHA REUNIÓN: Ingrese la fecha en que se llevó a cabo la reunión.
RESPONSABLE: Indicar nombre de la Institución, Sociedad u Organismo responsable de la reunión.
TIPO DE TRABAJO: Aclare si es un artículo completo, artículo breve o resumen.
FUENTE: Nombre del libro si es diferente del nombre de la Reunión o evento.
EDITORIAL, FECHA, LUGAR DE EDICIÓN, EN PRENSA como se indica para artículos.

4.2.2. Actividades tecnológicas (Presentar certificaciones que avalen las actividades).

a) Convenios , Asesorías o Servicios a Terceros

PARTICIPANTE: el nombre del investigador responsable de la transferencia. Si hay más de un participante (a diferencia de las publicaciones) no ingresar los nombres de los otros participantes.
FUNCIÓN: Indicar si es asesor, investigador, técnico, consultor, prestador de un servicio.

TIPO: Indicar el tipo de transferencia: convenio I+D; Asesoría técnica; servicios a terceros.
OBJETO: de la transferencia tecnológica. Si por acuerdos de confidencialidad acordados con la contraparte no se pudieran declarar alguno de los objetos / temas, consignar CONFIDENCIAL
IMPACTO: (Impacto socioeconómico. Estimación personal o institucional del beneficio social y /o económico en la contraparte o destinatario.

CAMPO: (Campo de aplicación) Utilizar código de SPU-ME empleado en la presentación de protocolo.
DESTINATARIO: Persona o Institución a las que se les brindan el servicio o con quien se firma el convenio.
MONTO: Monto percibido por el servicio.
AUTORIZACIÓN: Autorización dela Universidad/Organismo y el número de resolución.
ORGANISMOS: Organismos participantes en la transferencia tecnológica. CONICET, Unidad de Vinculación Tecnológica (UVT).

b) Registro de Propiedad

b.1) Patentes o Modelos de Utilidad / Otros
TÍTULO: Ingrese el título del trabajo.
AUTOR: Ingresar el apellido y nombre del autor separados por coma.
TITULAR: indicar la firma, empresa u organismo.
PAÍS: país que otorga el registro de la patente.
FECHA SOLICITUD: Fecha en que fue solicitada la patente. (Prioridad - Formato ISO).
NÚMERO DE SOLICITUD: Ingresar el número de la solicitud de la patente.
FECHA DISPONIBLE: Ingrese la fecha en que toma estado público. (Formato ISO.
NÚMERO: Ingresar el número de la patente otorgado por el INPI.
FECHA DE CONCESIÓN: Fecha en que fue concedida la patente. (Formato ISO).
PERIÓDO: Indicar el período por el que se concedió la patente.
IDIOMA: Idioma de la patente.
CÓDIGO: (Código sección / clase): Utilizar codificación INPI.
PATENTES RELACIONADAS: (Familia) Mencionar otros números de patentes concedidos o solicitados por el mismo objeto o título en otras jurisdicciones.
PATENTE COMERCIALIZADA: (o licenciada) Especifique la contraparte.

Acompañar todas las referencias documentales, indicando titulares o empresas, fecha de vigencia y demás datos que certifiquen la transferencia.

8 ANEXOS

ANEXO 1 Publicaciones

1) CACIC 2015 -Módulo gráfico de un Visualizador de Estructuras de un Sistema Operativo Educativo a través de GDB-Stub.

2) WICC2015 -Desarrollo de un prototipo para un visualizador de estructuras de un sistema operativo en ejecución a través de la comunicación serial.

3) CACIC 2014 -Visualizador de Estructuras de un Sistema Operativo Educativo

4) CADI 2014 Visualizador de Estructuras de un Sistema Operativo Real con Fines Educativos

5) WICC2014 -Mecanismos de visualización de estructuras de un sistema operativo en ejecución a través de la comunicación serial.

1)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

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.

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.

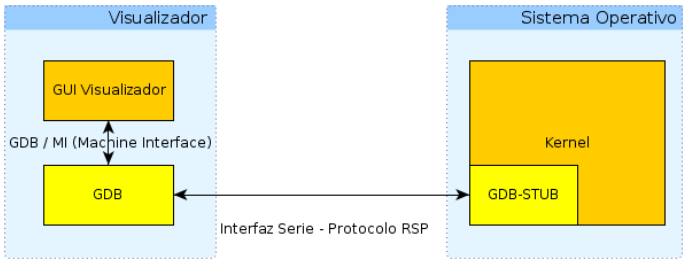


Fig. 1. Relación entre los componentes involucrados.

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.

1 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:

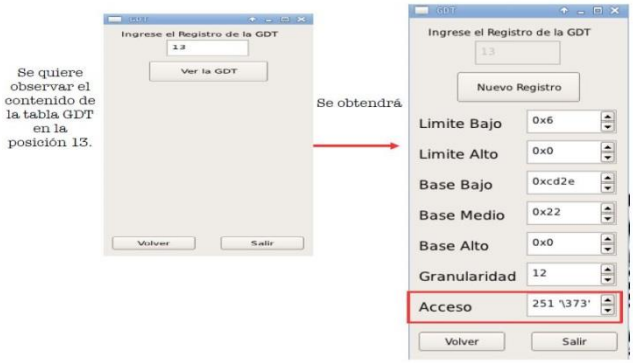


Figura 4. Interfaz GUI con la composición de la GDT

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

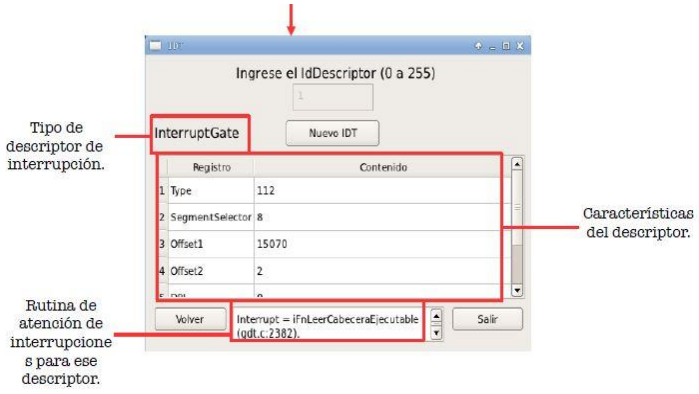


Figura 5. Interfaz GUI con la composición de la IDT

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.

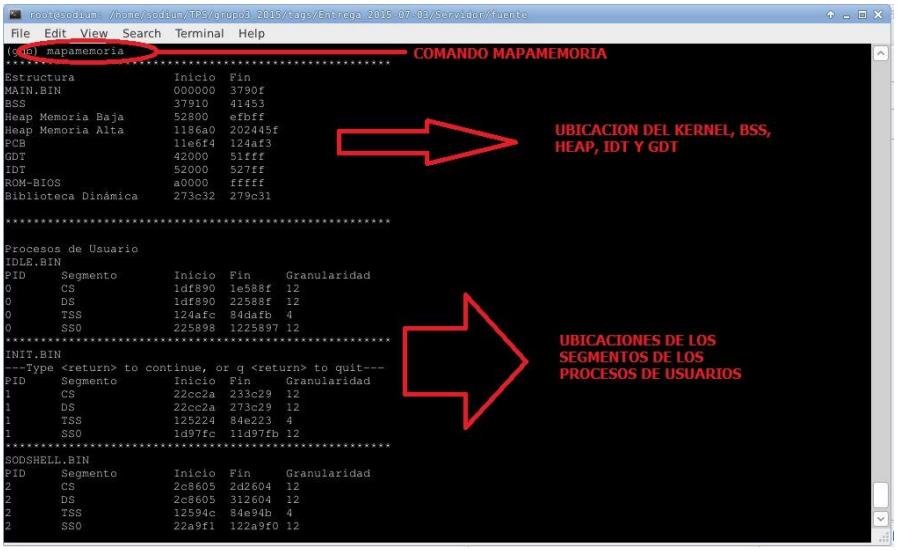


Figura 7. Mapa de memoria de Sodium

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.

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.

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/

12.

2)Desarrollo de un prototipo para un visualizador de estructuras de un sistema operativo en ejecución a través de la comunicación serial.

Graciela De Luca, Martín Cortina, Nicanor Casas
Esteban Carnuccio, Sebastián Barillaro, Sergio Martín

Departamento de Ingeniería e investigaciones Tecnológicas
Universidad Nacional de La Matanza
Florencio Varela 1903 - San Justo, Buenos Aires, Argentina
Te. (54-11) 4480-8900
{gdeluca; mcortina; ncasas;ecarnuccio; sbarillaro}@ing.unlam.edu.ar.

Resumen

En este artículo se exponen los avances relativos conseguidos durante la investigación en curso acerca del desarrollo de una interfaz de depuración remota. Si bien la construcción del visualizador será genérica, primeramente está desarrollada para el sistema operativo SODIUM[8]. Este documento se centra principalmente en las metodologías y herramientas utilizadas para desarrollar un prototipo básico sobre la interfaz de la aplicación cliente, la cual permite comunicarnos en forma remota con el S.O mencionado.

Primeramente se presenta el análisis efectuado sobre distintos lenguajes de programación y herramientas de creación de ventanas GUI con el fin de seleccionar el más adecuado para el desarrollo del prototipo visualizador. A continuación se detallan su diseño, funcionamiento e interacción con el sistema operativo SODIUM a través de dispositivos de puerto serie. Finalmente se describe como se ha implementado un protocolo de comunicación del lado del Servidor, para poder aplicarlo en la aplicación Cliente gracias al prototipo desarrollado.

Palabras clave: Interfaces GUI, Pyqt, Pyserial, Estructuras de datos, RSP, GDB, UART

Contexto

Esta Línea de Investigación es parte del proyecto “Visualización de estructuras in-ternas de un sistema operativo en ejecución como herramienta didáctica”, dependiente de la Unidad Académica del Departamento de Ingeniería e Investiga-ciones Tecnológicas perteneciente al pro-grama de Investigaciones CyTMA2 de la Universidad Nacional de La Matanza, el cual es continuación de otros proyectos, entre ellos del proyecto SODIUM.

Introducción

2 Elección del lenguaje a utilizar

Inicialmente se evaluaron las herramientas de programación para desarrollar una aplicación del lado cliente. Los lenguajes considerados inicialmente fueron C, C++, Java, Perl y Python. Se excluyeron los lenguajes no orientados al desarrollo de aplicaciones de escritorio y sin capacidad de generación de interfaces gráficas con el usuario y se determinó que el uso de Python para desarrollar la interfaz gráfica del visualizador sería el más conveniente. Debido a que ofrece la posibilidad de crear aplicaciones multi-plataforma y posee una funcionalidad encargada de gestionar la memoria automáticamente sin necesidad de

[8]SODIUM.-Sistema Operativo del Departamento de Ingeniería de la Universidad de La Matanza

ser peticionada en forma explícita. Se descartó Java debido a que no permite variar de tipo durante la ejecución del programa sin una conversión.

Python [1] puede ser utilizado tanto para scripting como para programación orientada a objetos, posee tipos dinámicos, tipos primitivos de alto nivel como listas, diccionarios y tuplas, y una abundante biblioteca de módulos, que brindan valiosas funcionalidades como cálculo científico, manejo de ventanas, multipro-cesamiento, manejo de puerto serie, funciones de red, etc.

3 Elección del framework.

Entre las bibliotecas que implementan interfaces gráficas de usuario en Python, las principales son Tkinter, WxPython y PyQT. [7]

Se descartaron las bibliotecas Tkinter y WxPython debido a la cantidad de ele-mentos gráficos, el comportamiento de la interfaz o a la falta de un editor gráfico de ventanas en aplicaciones, multiplata-forma.

Se resolvió emplear el binding de Python que posee el Framework QT, Pyqt, para la construcción de la interfaz gráfica. Esta biblioteca [3] brinda facilidades para el desarrollo de interfaces GUI, un completo conjunto de elementos gráficos y un flexi-ble y potente control del comportamiento de la interface, posee un mecanismo de conexión de señales y eventos, sencillo, rápido, de apariencia nativa, y se puede separar el diseño de la interface.

4 Elección del modelo de ciclo vida del desarrollo

El desarrollo presenta desafíos técnicos y funcionales, con especificaciones al mo-mento volátiles, que irán siendo forjadas conforme a los resultados de diversos es-tudios de factibilidad, estaremos siguien-do un modelo de desarrollo incremental con prototipado desechable para el visualizador. El módulo de comunicación del extremo servidor seguirá un ciclo de vida incremental, acompañando las necesidades del visualizador.

5 Diseño y construcción del Prototipo #1

Para el uso de la interfaz serie, se encontró un módulo de bajo nivel denominado Pyserial [4], ya desarrollado con el que se consiguió efectuar la conexión y transferencia de datos entre SODIUM, funcionando como Servidor, y un programa escrito en Python, actuando como Cliente. Posteriormente se empleó el módulo Miniterm, que utiliza interna-mente Pyserial con funcionalidades de más alto nivel.

6 Funcionamiento del sistema Cliente/ Servidor Inicial

En principio se confeccionó un prototipo muy elemental de una aplicación que actúa como cliente. Actualmente la interfaz GUI desarrollada permite la visualización de cadenas de caracteres recibidas del servidor, como así también el envío de datos a través de ella. El prototipo brinda la posibilidad al usuario de detener y reanudar la ejecución de SODIUM en cualquier momento a través de botones, lo que permite observar en forma gráfica la ocurrencia de ciertos sucesos en SODIUM y empezar a controlar su ejecución. En la Figura 1 se muestra el prototipo desarrollado.

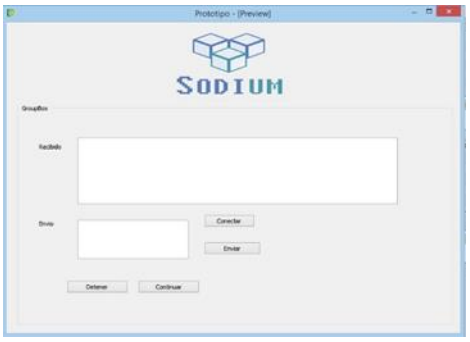


Figura 1- Interfaz gráfica del prototipo 1

En la figura 2 se pueden ver los componentes principales de la aplicación. El prototipo está conformado por dos componentes, el módulo Miniterm y la interfaz GUI. Miniterm es el encargado de efectuar la administración del puerto serie en el terminal cliente, comunicándose por medio de señales con la interfaz gráfica desarrollada en Pyqt.

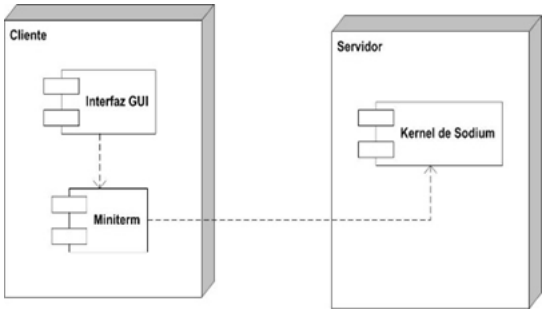


Figura2-Diagrama de Componentes del prototipo

7 Detalle de la interacción entre el cliente y el servidor

El S.O, una vez iniciado, buscará una petición de conexión del cliente. A partir de ese momento el usuario podrá interactuar con la interfaz presionando el botón "Conectar" para solicitar que se establezca la conexión.

En el instante en que SODIUM detecte una solicitud de conexión, establecerá el enlace y continuará con su ejecución normal a la espera de peticiones de datos por medio del puerto serie.

El visualizador, después que confirma el vínculo, finaliza la configuración de su UART [5] y queda a la espera de peticiones de datos de parte de la Interfaz GUI o del servidor. Para evitar que la interfaz gráfica se bloquee durante el uso del puerto serie, en el código de la aplicación cliente se genera un hilo que será responsable del manejo del mismo.

Como se puede observar en la figura 2, en la interfaz existen dos cuadros de texto. El primero con la etiqueta "Recibido", muestra las cadenas de caracteres que el S.O envía al prototipo. De esta manera le informa al usuario la ocurrencia de eventos que suceden en tiempo real. El otro cuadro de texto, con la etiqueta "envío", es el sitio donde el usuario puede ingresar caracteres ASCII para enviarlos al servidor.

8 Implementación del Protocolo RSP del lado del Servidor

En un principio se evaluó la posibilidad de incorporar en SODIUM el módulo GDB-Stub que ofrece GDB y realizar las adaptaciones necesarias para que el mismo funcione en nuestro sistema, pero resultó más simple aislar las rutinas básicas de GDB-Stub y en base a ello generar un módulo propio muy elemental, el cual denominamos "*Stub-Sodium"*. Este componente creció paulatinamente hasta implementar gran parte de la funcionalidad de GDB-Stub. Por lo tanto, tenemos una implementación parcial pero funcional del protocolo RSP [6] en el lado Servidor, con la que podemos leer y escribir remotamente en posiciones arbitrarias de memoria, detener y reanudar la ejecución del SO, y también consultar el estado estructuras internas del mismo una vez detenido. También se implementó el envío de mensajes iniciados por el servidor que indicarán la ocurrencia de eventos al cliente, funcionalidad necesaria para implementar puntos de instrumentación.

Actualmente se está implementado el protocolo del lado del cliente, utilizando para ello el prototipo desarrollado.

9 Traducción del contenido de estructuras del sistema operativo estudiado para su representación en el visualizador

En primer lugar, cabe destacar que no siempre es de interés toda la información contenida en una estructura si el objetivo de su extracción es lograr una abstracción del detalle de la misma a través de la visualización de sus interrelaciones con otras. Llamaremos a esta abstracción del detalle, la representación canónica.

10 Representación canónica o normalizada de una estructura

Entendemos como representación canónica de una estructura de un sistema operativo a una suficientemente genérica como para describirla adecuadamente en su calidad funcional, sin tener en cuenta los detalles de implementación del sistema operativo estudiado. Esto permitirá su análisis y vuelco en los distintos diagramas que proveerá la herramienta de visualización.

A modo de ejemplo, podemos citar la estructura que representa una entrada de la

tabla GDT propia de la arquitectura i386 declarada en lenguaje C que respeta estrictamente su organización en memoria. Seguidamente, vemos su potencial equivalente en lenguaje Python, donde tendremos en cuenta únicamente los datos de interés para su visualización.

C(dependiente de arquitectura i386)

```
typedef struct {
 ushort usLimiteBajo;
 ushort usBaseBajo;
 uchar ucBaseMedio;
 uchar ucAcesso;
 uint bitLimiteAlto:4;
 uint bitGranularidad:4;
 uchar usBaseAlto;
} __attribute__((packed))
stuGDTDescriptor;
```

Python (normalizada):

```
class stuGDTDescriptor:
 def __init__(self):
 self.base = 0
 self.limite = 0
 self.tipo = ''
 self.priv = 0
 self.pres=False
```

11 *Transmisión del contenido de las estructuras*

Con respecto a la necesidad de transmitir el estado de las estructuras administradas por el sistema operativo en estudio hacia la interfaz del visualizador, optamos por transmitir el contenido completo de las mismas, sin alteraciones, a través de la interfaz serie.

Esta alternativa aporta como ventaja principal la simplicidad del código del stub. El mismo debe conocer únicamente información acerca de la ubicación y sólo en algunos casos el tamaño de las estructuras del sistema y de las estructuras propias de la arquitectura del hardware ya que no debe encargarse de extraer o interpretar el contenido de las mismas. La simplicidad de este módulo no sólo aporta robustez a la solución, sino que además permite encapsular la responsabilidad de interpretar el estado interno del sistema operativo en el visualizador.

Prevemos que será necesario tener en cuenta la extremidad (término más conocido como "endianness") de los bytes recibidos por el visualizador, así como tener un conocimiento exacto de los campos de las estructuras que han sido transferidas. La extracción de los campos de las estructuras transferidas no será trivial en los casos en los que las mismas se encuentren empaquetadas, o se encuentren desalineados con respecto al ancho de las palabras del bus de direcciones.

En los casos en los que se requiera navegar por una lista enlazada de estructuras, prevemos también que el visualizador se verá obligado a solicitar un nodo por vez, extrayendo y evaluando el puntero al siguiente elemento entre cada adquisición.

Líneas de Investigación, Desarrollo e Innovación

Actualmente, nos encontramos trabajando sobre los siguientes aspectos:

-Implementación del protocolo RSP del lado de la aplicación Cliente

-Diseño y desarrollo de mecanismos a utilizar para visualizar las estructuras del sistema operativo SODIUM

-Diseño y desarrollo de las interfaces gráficas del visualizador

Formación de Recursos Humanos

El grupo de trabajo consta de dos investigadores de categoría IV, un investigador categoría V, dos investigadores con varios años de experiencia y un investigador inicial.

Resultados y objetivos

Estamos actualmente finalizando la implementación del Sodium-Stub, y ya desarrollamos los primeros prototipos del visualizador, uno de los cuales ya es capaz de conectarse a nuestro sistema operativo SODIUM, detener y reanudar su ejecución, y consultar el estado de una estructura interna.

En etapas posteriores desarrollaremos además una guía de adaptación del visualizador a otros sistemas operativos de código abierto para poder analizar el funcionamiento de los mismos y utilizar esta herramienta como elemento didáctico para la enseñanza de Sistemas Operativos.

Referencias

[1] Guido van Rossum,"*El tutorial de Python*",Editorial: Fred L. Drake Jr. *Septiembre 2009*
[2] Raúl González Duque,*"Python para todos"*,Editorial: Autoedición, Año: 2010
[3] Mark Summerfiled, *"Rapid Gui Programming with Python and QT",* Editorial: Prentice Hall, Año: 2007
[4] Chris Liechti,*"pySerial Documentation", Versión:2.6, Diciembre 2011*
[5]David S.Lawyer: "Serial HowTo" Greg Hankin
[6]Bill Gatliff "*Embedding with GNU: the GDB Remote Serial Protocol*", revista Embedded Systems Programming, Noviembre 1999
[7] "Interfaces gráficas (GUI)", http://python.org.ar/InterfacesGraficas
[8]http://www.linuxchix.org/content/courses/security/raw_sockets

3)Visualizador de Estructuras de un Sistema Operativo Educativo

Graciela De Luca, Martín Cortina[1], Nicanor Casas[1], Esteban Carnuccio[1],
Sebastián Barillaro[1], Sergio Martín[1], Gerardo Puyo[1]
[1] Universidad Nacionald de La Matanza,San Justo, Buenos Aires Argentina
{gdeluca, mcortina , ncasas, ecarnuccio, sbarillaro, smartin, gpuyo}@ing.unlam.edu.ar

Abstract. El presente trabajo describe los avances conseguidos durante el desarrollo de una aplicación que permitirá la visualización remota de estructuras lógicas y el control de un sistema operativo para uso académico, con el objetivo de facilitar la asimilación de conceptos que, en primera instancia, suelen ser demasiado abstractos. Este será suficientemente flexible como para representar el estado interno de cualquier sistema operativo, aunque inicialmente se basará en el Sistema Operativo SODIUM.
En este documento describiremos el análisis efectuado sobre los chips UART de los dispositivos seriales con la finalidad de optimizar el funcionamiento del driver serie de SODIUM y el diseño preliminar de algunas de las estructuras que el visualizador representará. Finalmente se describirán los mecanismos que se están implementando en SODIUM con el objetivo de conseguir detener su ejecución en forma remota mediante puntos de parada.
Keywords: Interrupciones, instrumentación, puntos de parada, SODIUM, comunicación serial, RSP, GDB-Stub, UART, sistemas operativos, visualización de estructuras, GDT, IDT, PCB, control de ejecución.

Introducción

Cualquier sistema visualizador debe poseer la capacidad de generar diagramas en base a grandes cantidades de datos de forma inmediata y de una manera que les permita a los usuarios interpretar fácilmente dicha información [1]. En consecuencia, el visualizador que se está desarrollando en este proyecto procurará recibir información sobre los distintos componentes y estructuras que conforman el sistema operativo durante su ejecución y, en simultáneo, generar gráficos entendibles para que los estudiantes puedan comprender su funcionamiento. La arquitectura Intel presenta distintos componentes que son utilizados por los sistemas operativos para el funcionamiento de una PC. La interrelación entre ambos actores es frecuentemente es de difícil comprensión. Los sistemas operativos arman distintas estructuras para administrar los recursos de una computadora en base a los elementos que proveen sus fabricantes.

El Sistema Operativo SODIUM fue construido con fines educativos para que pueda ser utilizado por estudiantes y docentes para el aprendizaje práctico del funcionamiento de un SO tradicional. Hasta la fecha, SODIUM no presenta un visualizador gráfico que permita observar sus estructuras, componentes y funcionamiento de sus algoritmos. Con propósito de brindar una nueva herramienta de estudio se lleva a cabo este proyecto de investigación, cuya meta es desarrollar un Visualizador gráfico que pueda ser utilizado en un principio por SODIUM y posteriormente por otros Sistemas Operativos.

El presente documento describe los distintos mecanismos que se han implementado en el proyecto para poder ir alcanzando el objetivo planteado al inicio de la investigación

Visualizador externo del Sistema Operativo

Como bien se mencionó anteriormente, este proyecto de investigación tiene como objetivo principal generar un software que permita observar diferentes estructuras del Sistema Operativo SODIUM, implementándolo posteriormente para su uso en otros sistemas. La comunicación entre SODIUM y el programa visualizador se efectuará a través de dispositivos seriales. Por consiguiente, fue necesario desarrollar un driver que permita la comunicación entre distintas terminales empleando los puertos COM disponibles en una computadora [7]. En la figura siguiente se detalla cómo se efectuará la implementación final de la aplicación en las distintas unidades.

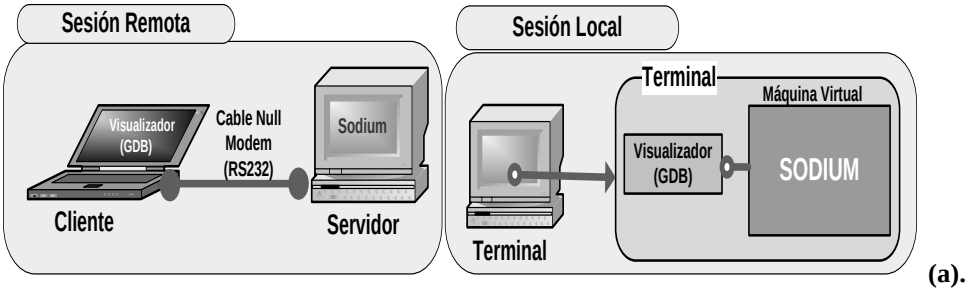


Fig. 1. (a) Representación gráfica de la ejecución del visualizador y SODIUM en dos terminales. **(b)** Representación de la ejecución en la misma terminal

Características del driver de puerto serie implementado

Por su simpleza, la comunicación serie fue uno de los medios de comunicación punto a punto entre computadoras y dispositivos más difundidos en los albores de la micro computación. La función principal de esta controladora (UART) es la de tomar cada byte leído del bus de datos paralelo y serializarlo en un tren de bits, enviándolos de a uno por vez través del canal de comunicación. A su vez, la misma contempla el proceso inverso en su canal de recepción. La norma de comunicación asociada es la RS-232 [5], que especifica las características eléctricas y mecánicas de las interfaces así como velocidades admitidas, control de flujo y uso de bits adicionales para señalización y detección de errores. Esto permite, en particular al chip (16650A), la opción de configurar la generación de interrupciones a la CPU cuando se reciben 1, 4, 8 ó 14 bytes [6].

La norma RS-232 especifica un par de líneas de datos (una para enviar y otra para recibir) y 6 para control, con las que implementa el control de flujo. De esta manera, el equipo receptor puede indicarle al equipo emisor que está listo para recibir, o que se abstenga de continuar enviando datos. Así se logra adaptar la comunicación al estado de ambos equipos.

El driver serial implementado actualmente en SODIUM fue construido sin aprovechar el uso del buffer FIFO que poseen los chips UART más modernos (como el 16550A mencionado anteriormente).

A lo largo de las pruebas efectuadas durante esta fase del desarrollo, se identificó el riesgo de que la tasa inicial de transmisión de datos entre terminales que logramos obtener pudiera no ser suficiente para cubrir nuestras necesidades. Dicho riesgo nos llevó a relevar el modelo del chip UART utilizado en una muestra de computadoras convencionales de distintas marcas, determinando al fin que sí es común encontrar la capacidad de *buffering* en las mismas. Con esta conclusión, sabemos que posteriormente podremos implementar el uso de dicha capacidad en SODIUM mejorando así la tasa de transferencia del driver ya desarrollado.

Según la documentación de VMware, el software ofrece 4 puertos serie a los sistemas virtualizados. Cualquiera de estos puertos puede ser direccionado a un puerto físico real del equipo, o a un archivo, o interconectado entre dos computadoras virtuales. La UART virtualizada es un modelo compatible con el chip 16550A, que ofrece un buffer FIFO de 16 bytes. Los cuatro puertos series comparten las IRQ 3 y 4 de a pares [10].

En el caso de Bochs, también son 4 los puertos serie virtualizados. Permite elegir el modo de operación de manera similar a VMware: null, raw, mouse, file, term (sólo para Unix), FIFO y Socket (sólo para Windows). En todos los casos, el chip emulado ofrece compatibilidad con el muy difundido 16550A [11].

Diseño preliminar del Visualizador del Sistema Operativo

El sistema operativo SODIUM fue desarrollado para ejecutar en equipos con arquitectura x86. Dada su naturaleza de sistema operativo de estudio, se buscó ejercitar todas las características que esta arquitectura provee, como ser protección, segmentación fija, segmentación paginada, cambios de contexto por hardware y manejo de interrupciones y excepciones. Por consiguiente, para poder utilizar la memoria, CPU y dispositivos externos se emplean distintas estructuras que están intrínsecamente relacionadas. Lo que se pretende mostrar por medio del visualizador es el estado de dichos datos y su variación en el tiempo ante distintos eventos en tiempo real.

Cabe destacar, en contraste, que otros sistemas operativos, cuyo objetivo es ser fácilmente portables a distintas arquitecturas de hardware, buscan mantener la mayoría de sus mecanismos de administración de recursos agnósticos

a la arquitectura, utilizando únicamente el mínimo denominador común de características presentes en las mismas, como ser paginación. Otro mecanismo provisto por la arquitectura x86 que es frecuentemente ignorado, tanto por compatibilidad como por eficiencia, es el cambio de contexto por hardware.

Seguidamente, se describen algunas de las distintas estructuras que utiliza SODIUM que van a ser representadas gráficamente y cómo se va a exponer su interrelación por el visualizador en su etapa inicial.

GDT (GLOBAL DESCRIPTOR TABLE)

La arquitectura Intel ofrece la posibilidad de utilizar unas tablas de descriptores alojadas en memoria para poder ubicar los segmentos que se utilizan en el sistema. Una de esta estas tablas es la GDT, apuntada por el registro GDTR de la CPU [4]. Dicha estructura contiene esencialmente los descriptores de códigos, datos y stacks utilizados por el sistema operativo para ubicar los segmentos correspondientes en la memoria principal. Para hacer uso de esta herramienta, SODIUM administra esta tabla utilizando una estructura en forma de vector. Cada elemento de dicho vector contiene fundamentalmente Tipo de descriptor (Código, datos o stack), dirección base del segmento, tamaño del segmento y nivel de privilegio DPL. En consecuencia, en base a estos ítems, SODIUM consigue acceder a la información de cada proceso en memoria en un momento determinado. Para esto se utilizan otros elementos que se comentarán seguidamente.

PCB (PROCESS CONTROL BLOCK)

Esta estructura contiene toda la información necesaria de un proceso en particular que un sistema operativo utiliza para poder llevar a cabo su administración durante su tiempo de vida. De forma tal que el SO pueda asignarle eficientemente los recursos que este emplea. Siendo así, SODIUM maneja dicha herramienta para gestionar los distintos procesos que utiliza el sistema. Este componente se encuentra conformado en un vector de estructuras que almacenan distinta información. Entre las más importantes se encuentran el ID del proceso, el ID del padre, estado, nivel de prioridad y los índices de descriptores de segmentos de GDT asignados a código, datos y stack.

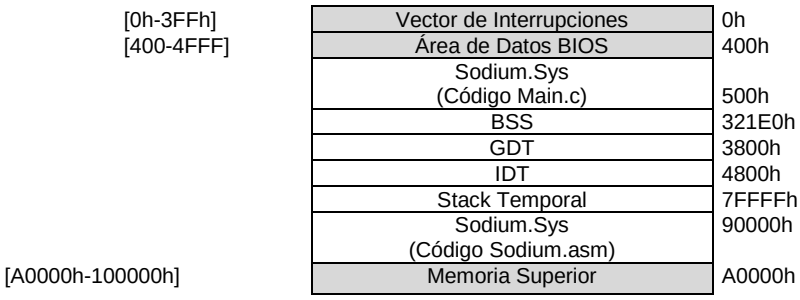
La visualización gráfica de esta información es trascendental para que el estudiante pueda comprender el funcionamiento del sistema.

IDT (INTERRUPT DESCRIPTOR TABLE)

Otra tabla que el Sistema Operativo SODIUM debe gestionar, aplicando la tecnología que ofrece Intel, es la IDT. Esta estructura contiene los descriptores de segmentos en donde se encuentran alojadas las rutinas de atención de las interrupciones. Esta tabla consiste en un vector de estructuras de 256 posiciones. Cada posición del vector presenta un selector de segmento para la GDT, Offset, Tipo y nivel de privilegio DPL. En consecuencia, cada vez que ocurre una interrupción en modo protegido, el sistema operativo SODIUM utiliza esa estructura para ubicar la rutina de atención asociada a dicho evento.

MAPA DE MEMORIA

Una de las características importantes que se pretende desarrollar en el visualizador es que este permita observar en tiempo de ejecución la ubicación de los componentes que conforman al sistema operativo y procesos en la memoria principal. La siguiente figura muestra un croquis del diseño preliminar de cómo se pretende mostrar el estado de la memoria en un momento dado.



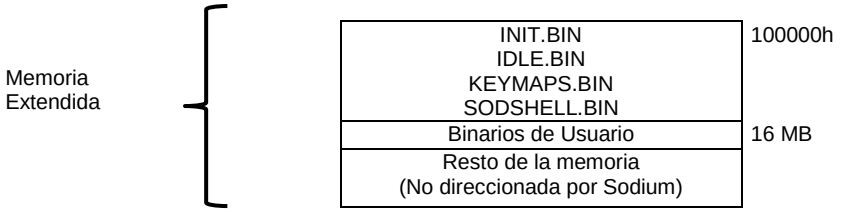


Fig. 2. Croquis del Mapa de Memoria que representará el visualizador

Todas las estructuras anteriormente mencionadas son algunos de los componentes que se pretende que el visualizador de SODIUM muestre inicialmente, dado que es muy importante poder ver gráficamente su contenido. De esta manera, el usuario podrá observar e interactuar con dichos componentes a través los diagramas que los representan [1] [2]. En la figura siguiente se muestra una de las relaciones que se pretenderá graficar entre las tablas PCB, GDT y la ubicación de los distintos componentes en la memoria principal.

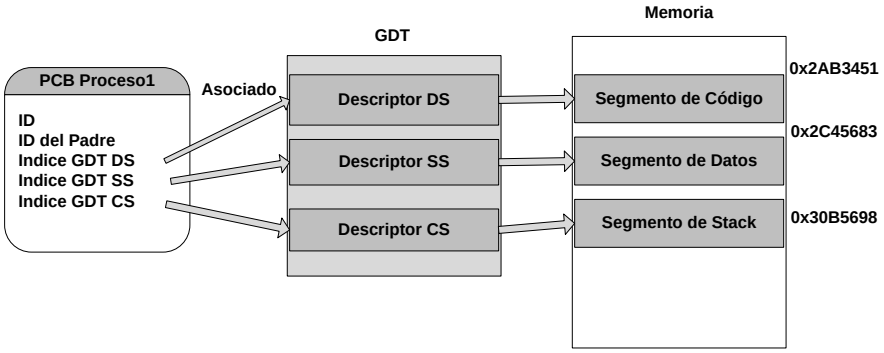


Fig. 3. Relación entre estructuras de SODIUM

Logueo al inicio de SODIUM

Se desarrolló en SODIUM una funcionalidad que permite enviar mensajes arbitrarios a la terminal remota con un formato específico de logueo. Este módulo fue confeccionado basándose en el comando FTRACE de Linux. Estos mensajes tienen como fin notificar al usuario la ocurrencia de distintos eventos de SODIUM. Actualmente, se informan en la terminal remota los acontecimientos ocurridos desde el inicio de SODIUM hasta que la consola del sistema operativo es activada. Sin embargo, se pretende que a futuro utilizar este tipo de mensajes para informar al visualizador de la ocurrencia de sucesos del sistema operativo. Denominaremos "Puntos de Instrumentación" a las regiones de código que generen este tipo de mensajes.

El formato de logueo utilizado es el siguiente:

Criticidad	*Fecha*	*Hora*	*Evento*	*Descripción*
0	24/07/2014	21:23:33	1024	ES_LA

Fig. 4. Formato de logueo remoto

Criticidad: Indica el nivel de importancia del mensaje de log. Varía de 0 a 5 siendo 5 el nivel más crítico.
Fecha: Indica la fecha en que se generó el evento en el sistema operativo.
Hora: Indica la hora en que ocurrió el suceso.
Evento: Es el tipo de evento. Este número entero es conocido tanto por el sistema operativo como por el visualizador. Por ejemplo, el número 1024 puede indicar el evento de carga satisfactoria de un mapa de caracteres para la consola activa.
Descripción: Contiene la descripción del hecho acontecido. Siguiendo con el ejemplo anterior, puede contener el nombre de la distribución de teclado cargada.

Detención de la ejecución del Sistema Operativo

Uno de los principales objetivos planteados desde el inicio del proyecto consistió en que el usuario tuviera la posibilidad de detener la ejecución del Sistema Operativo desde la interfaz de visualización, a fin de poder observar

el estado de las distintas estructuras que lo componen en un momento determinado. Esta característica es muy importante, dado que otorga a los estudiantes la facultad de interactuar con SODIUM en forma remota, pudiendo así reafirmar los conocimientos teóricos impartidos al respecto.

Utilización de puntos de parada

En la investigación en curso, se hizo hincapié en la detención del Sistema Operativo SODIUM en forma remota. Por dicho motivo, primeramente se analizó el funcionamiento de otros depuradores existentes y de los mecanismos que éstos utilizan para implementar puntos de parada en distintos programas de usuario. En consecuencia, se observó que algunas de estas herramientas emplean dos tipos diferentes de puntos de parada [9][3]: los implementados por software y los implementados por hardware

Si bien en SODIUM se desarrollaron las rutinas de manejo de excepciones generadas tanto por breakpoints de software como de hardware, se determinó que era conveniente trabajar únicamente con el primer tipo en la etapa inicial del desarrollo de la detención remota del Sistema Operativo. Esta decisión se fundamentó en que los puntos de parada desarrollados por software no presentan limitaciones de uso e implementación respecto de su contraparte.

Inicialmente las invocaciones de la excepción número 3, asociada a los puntos de parada por software, podían ser realizadas únicamente desde el código del Kernel del Sistema Operativo SODIUM. Esta restricción se debió a que las excepciones eran ejecutadas con un nivel de privilegio con valor 0, correspondiente al DPL de la Interrupt Gate asociada. En consecuencia, un programa de usuario no podía utilizar un punto de parada de este tipo durante su ejecución. Por este motivo, se determinó cambiar el nivel de privilegio del DPL de esta excepción asignándole el valor número 3. De esta manera se consiguió que un programa de usuario pueda ejecutar una instrucción que genere una interrupción de punto de parada durante su ejecución en SODIUM.

Adaptación de GDB-Stub

Con la finalidad de controlar la ejecución del Sistema Operativo SODIUM en forma remota, se analizó el funcionamiento del módulo que ofrece GDB, denominado GDB-Stub [8]. Se indagó en la comprensión de sus funcionalidades, con el objeto de obtener los conocimientos necesarios para emplearlos luego en el proyecto. En base al estudio llevado a cabo, se determinaron los cambios necesarios que se debieron efectuar en SODIUM a los fines de conseguir detener su ejecución desde otra terminal. En los siguientes apartados se describirán los mecanismos que se están empleando con este propósito.

Proceso de detención de SODIUM en forma remota

A raíz del análisis mencionado, se descubrió que la tarea principal de GDB-Stub se centra en su función “handle_exception”, cuya funcionalidad elemental consiste en detener el sistema y luego esperar en forma activa las peticiones de los usuarios que estén conectados remotamente a ella. En consecuencia, se desarrolló en SODIUM un módulo que emula su funcionamiento. En la figura 5 se visualiza el funcionamiento básico de este componente:

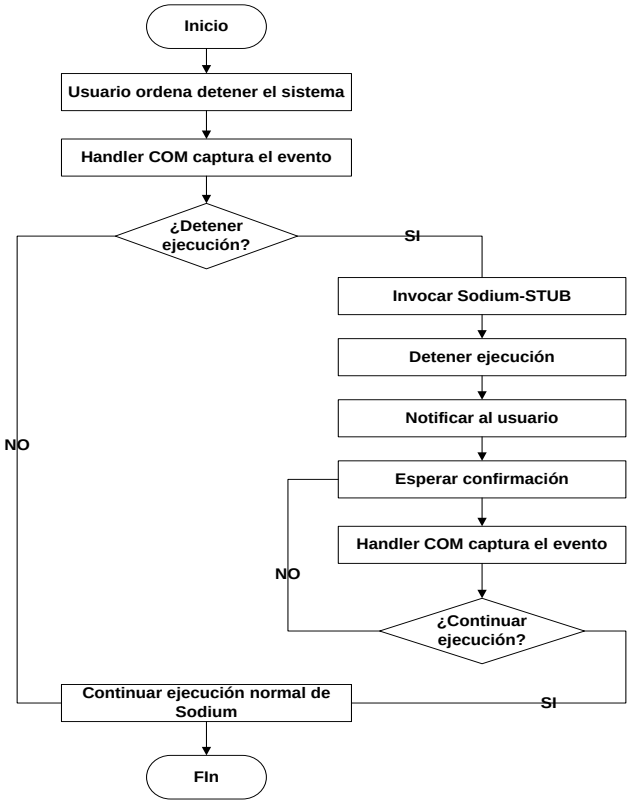


Fig. 5. Diagrama de flujo sobre la detención de la ejecución del sistema

En el gráfico anterior se puede observar el diagrama de flujo esencial de la detención del sistema en forma remota. Primeramente el usuario envía, desde otra terminal, un mensaje a SODIUM indicándole que detenga su ejecución. Como la comunicación entre el cliente y el sistema operativo se efectúa de acuerdo al protocolo RS-232, este aviso es recibido por el controlador de puerto serie [7]. Cuando la UART recibe un byte desde otro dispositivo, genera una interrupción que es capturada por el manejador asociado al puerto COM que se está utilizando. Actualmente en esta investigación se está empleando el puerto COM1, por lo que se estará activando la IRQ (Interrupt Request) número 4. Una vez capturada esta interrupción, se determina si la señal enviada por el cliente es de detención. En caso afirmativo, se invoca al módulo desarrollado en SODIUM que emula el funcionamiento de "handle_exception" (de ahora en más SODIUM-Stub). Esta función detiene la ejecución del Sistema Operativo y envía un mensaje de notificación al usuario comunicándole que el sistema se detuvo y queda en espera de las peticiones del cliente. Finalmente, el usuario envía un mensaje solicitando reanudar la ejecución del sistema operativo, y este es recibido por el manejador del puerto serie. Acto seguido, el manejador invoca a SODIUM-Stub y éste restablece la ejecución normal de SODIUM.

El procedimiento anterior describe el funcionamiento actual del módulo encargado de la detención del Sistema Operativo. Cabe señalar que este componente aún continúa optimizándose, por lo que puede sufrir variaciones en un futuro.

Visualización de estructuras cuando SODIUM se encuentre detenido

Se pretende que el sistema operativo transfiera el estado de cada una de sus estructuras internas al visualizador en el instante en que el usuario detenga su ejecución. De tal forma, el usuario podrá interactuar con las mismas observando en detalle su composición. Con esta característica se pretende generar una herramienta educativa que consiga transferir a los estudiantes los conocimientos teóricos y prácticos acerca del funcionamiento interno de un Sistema Operativo tradicional.

Manejadores y puntos de parada

Al inicio del desarrollo de la detención remota del sistema, se había planificado que SODIUM ejecute automáticamente un punto de parada por software al recibir un mensaje de solicitud de parada desde una terminal

remota. Luego, una vez que el manejador capturara la excepción 3 generada, invocaría al módulo SODIUM-Stub para llevar a cabo las peticiones del usuario. No obstante, posteriormente se determinó que era más eficiente llamar directamente al módulo SODIUM-Stub, en lugar invocarlo a través de la generación de una excepción. Esto fue debido a que se producía una disminución en el rendimiento del sistema dado por el anidamiento de manejadores. Por dicho motivo, se decidió evitar la utilización de la excepción 3 al momento de recibir un mensaje de detención y que en su lugar se invoque directamente a la función SODIUM-Stub, tal como se muestra en la figura 4.

Es importante mencionar que el estado actual de este mecanismo todavía presenta una merma en la productividad del sistema, dado que aún se continúa ejecutando con anidamientos de manejadores. Por dicho motivo, actualmente se está trabajando para conseguir subsanar dichas dificultades. Además, cabe señalar que, si bien no se emplearán en la detención del sistema operativo los mecanismos desarrollados para el manejo de puntos de parada, estos serán resguardados en el código fuente de SODIUM para futuros proyectos.

Conclusión:

Hasta la fecha, en la investigación en curso, se consiguió desarrollar una característica fundamental en el funcionamiento del Visualizador que consiste en lograr la posibilidad de detención del Sistema Operativo SODIUM en forma remota. De esta manera se le otorga al usuario la posibilidad de detener el sistema en el preciso momento en que desee analizar el estado del mismo. Además de lo anteriormente dicho, se pudo establecer preliminarmente el tipo de estructuras que el Visualizador mostrará al usuario por medio de diferentes representaciones gráficas. Conjuntamente, a través del envío de mensajes asincrónicos hacia el visualizador, se busca informar al usuario en forma descriptiva la ocurrencia de distintos eventos del sistema. Como se mencionó anteriormente, estas funcionalidades deben seguir optimizándose, de forma tal que puedan ser empleadas satisfactoriamente por los estudiantes y educadores. No obstante, se consiguió construir una parte fundamental en el desarrollo del Visualizador del sistema Operativo.

Referencias

1. Robert P. Bosch Jr.:“Using Visualization to Understand The Behavior of Computer System”: pp 13-16 (2001).
2. Farzaneh Zareie y Mahsa Najaf-Zadeh: “OSLab: A Hand-on Educational Software for Operating System Concepts Teaching and Learning”: Research WebPub: pp 1-3 (2013)
3. Prasad Krishnan: “Hardware Breakpoint (or watchpoint) usage in Linux Kernel”, IBM Linux Technology Center, Canada: pp 1-10 (2009)
4. Intel: “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3: System Programming Guide”: pp 65-6,413 (2011)
5. Dallas Semiconductor:”Fundamentals of RS-232 Serial Communications, Application note 83”, Sunnyvale, (1998)
6. David S.Lawyer: “Serial HowTo” Greg Hankin: Capítulo 18: (2011)
7. Graciela De Luca, Martín Cortina, Nicanor Casas, Esteban Carnuccio, Sergio Martín, “Mecanismos de visualización de estructuras de un sistema operativo en ejecución a través de la comunicación serial”, Congreso WICC (2014)
8. The GNU Project Debugger, https://sourceware.org/gdb/onlinedocs/gdb/Remote-Debugging.html#Remote-Debugging
9. http://x86asm.net/articles/debugging-in-amd64-64-bit-mode-in-theory/index.html
10. VMware Inc., http://www.dpunkt.de/leseproben/1686/Kapitel_2.pdf
11. Bochs Emulator Project, http://bochs.sourceforge.net/doc/docbook/user/bochsrc.html

4)Visualizador de Estructuras de un Sistema Operativo Real con Fines Educativos

Graciela De Luca[1], Martín Cortina[1], Nicanor Casas[1], Esteban Carnuccio[1], Sebastián Barillaro[1], Sergio Martín[1], Gerardo Puyo[1]

[1] Universidad Nacionald de La Matanza,
San Justo, Buenos Aires Argentina
{gdeluca, mcortina , ncasas, ecarnuccio, sbarillaro, smartin, gpuyo}@ing.unlam.edu.ar

Resumen. El presente trabajo se centra en los avances relativos al desarrollo de una interfaz de depuración remota. Si bien el desarrollo de la interfaz y el graficador es genérico, inicialmente se basará en el sistema operativo S.O.D.I.U.M[9], del cual tenemos completo conocimiento y control. Para asegurar la interoperabilidad de nuestro desarrollo con el depurador GDB[10], se está analizando e incorporando un módulo remoto denominado gdbstub, que resuelve la comunicación a nivel lógico, implementando el protocolo RSP. Se analizan también las técnicas utilizadas por los depuradores modernos en cuanto a la implementación del mecanismo de *breakpoints* y el soporte que la arquitectura IA32 provee para facilitar dicha tarea. En función de esto estudiaremos también las responsabilidades que debe tener un manejador de excepciones de depuración por hardware.

Palabras Clave: Visualizador - Sistema Operativo Educativo – S.O.D.I.U.M. - Breakpoints - Gdbstub - Comunicación Serial

1 Introducción

S.O.D.I.U.M. es un sistema operativo realizado con propósitos didácticos, que permite durante su ejecución la reconfiguración, cambiando los algoritmos utilizados por los administradores del sistema. Esto tiene como propósito didáctico permitir la comprensión y el análisis del funcionamiento interno del sistema operativo, pudiendo de esta manera adquirir competencias en la evaluación de los algoritmos y la elección de los sistemas operativos comerciales estableciendo rendimientos y comportamientos de acuerdo al entorno en el que se realiza la ejecución. Para esto la investigación se centra en la construcción de una aplicación con fines didácticos que permita la visualización gráfica del funcionamiento interno, estructuras de datos del sistema, valores de las variables, estados de los procesos en S.O.D.I.U.M., con el fin de modificar mínimamente la ejecución del sistema, se decidió realizar esta visualización en otra máquina, estudiando para este propósito los diferentes protocolos utilizados a tal fin. La enseñanza teórica tradicional de un sistema operativo se basa principalmente en el estudio de una gran cantidad de bibliografía relacionada con el tema. La forma tradicional de la enseñanza universitaria está basada en proporcionar conocimientos teóricos y luego pasar a la práctica para aplicar la teoría aprendida, con esta propuesta se podrá estudiar distintos casos propuestos por el docente, analizando el comportamiento interno del sistema operativo, sin interferir prácticamente en su ejecución.
En consecuencia, el desarrollo de una herramienta que permita visualizar el funcionamiento de un sistema operativo facilitará la enseñanza de los profesores a sus alumnos, dado que se podrá analizar el comportamiento de los distintos módulos del mismo en menor cantidad de tiempo. Otro de los beneficios, es que podrán visualizarse gráficamente los mecanismos utilizados por un sistema operativo real al momento de resolver problemas específicos. De esta forma el alumno podrá aprender más rápidamente su funcionamiento, pudiendo ver el estado del sistema en un determinado momento. La enseñanza tradicional de un sistema operativo se basa en tres pilares, los estudiantes modifican o amplían parte de un sistema operativo; ellos escriben código para demostrar aspectos de la tecnología en un sistema operativo comercial; además ejecutan código que simulan partes de la tecnología de un sistema operativo.
El sistema operativo S.O.D.I.U.M. se construyó siguiendo la primera premisa. Como consecuencia a cada grupo se les asignó una investigación y el desarrollo de una sección determinada de este sistema operativo, con lo que su complejidad fue aumentando a medida que fueron pasando los años. A su vez, se fue incrementando la necesidad de poder visualizar gráficamente el funcionamiento, para apreciar el

[9] Sistema Operativo del Departamento de Ingeniería de La Universidad de La Matanza
1.1 [10] GNU Project Debugger

esfuerzo de tanto tiempo de trabajo y que luego éste pueda ser utilizado por otras instituciones educativas. Por lo tanto la investigación en curso se dispone a desarrollar un visualizador que permita la observación de su comportamiento

2 Estado del arte

Las actuales aplicaciones que permiten visualizar el comportamiento de sistemas complejos, se diferencian en la manera en que estos han sido implementados, debido a que se desarrollan de acuerdo al entorno en el que van a ser utilizados.

- **Simulación de máquina**. Software que simula todos los componentes Hardware que constituyen una computadora con la finalidad de poder ejecutar un sistema. Gracias a esto se puede acceder totalmente al Hardware, en forma no intrusiva, y al estado del software utilizando simulaciones.
- **Instrumentación en Tiempo Real**: Otros sistemas usan el detalle del Hardware físico, Firmware e instrucciones de Software para observar el rendimiento de los sistemas en Tiempo Real.

Los Sistemas de visualización descriptos en [2], [3] y [4] simulan una parte del funcionamiento de un sistema operativo para su enseñanza y aprendizaje. Esto trae como desventaja que el alumno puede no llegar adquirir los conocimientos necesarios sobre el funcionamiento completo de un sistema operativo. Dado que solo se estudia determinadas funcionalidades de dicha plataforma. Además, no se estaría trabajando con Hardware real, ya se emplean en entornos simulados.
Por otra parte se encuentra el Sistema Rivet [1], que si bien trabaja con hardware físico en tiempo real y permite la creación rápida de prototipos para mostrar datos puntuales, no está orientado al ámbito educativo sino más bien al análisis de rendimiento de Sistemas.

En consecuencia a lo anteriormente mencionado, se pretende que el Visualizador del sistema operativo S.O.D.I.U.M. sea una aplicación destinada para el estudio educativo sobre el funcionamiento completo de un sistema operativo tradicional. Por lo que se procurará representar gráficamente las características más importantes que el sistema ya posee. El cual además, incorporará las funcionalidades más significativas de los visualizadores previamente descriptos. Para ello, se procura que pueda ser utilizado tanto en entornos simulados como reales, brindando de esta forma mayor libertad a los usuarios, ya que se estará otorgando la posibilidad de que dicho visualizador pueda ser ejecutado tanto en la misma terminal en donde se estará ejecutando S.O.D.I.U.M. como en otra distinta. De esta forma se obtiene una gran diferencia con respecto de los visualizadores existentes, ya que se estaría trabajando tanto en un entorno real como simulado, pudiendo visualizar el completo funcionamiento de un sistema operativo real. Cabe mencionar, que para poder efectuar la comunicación entre la maquina servidor, donde se estará ejecutando S.O.D.I.U.M. y la maquina cliente, donde se estará ejecutando el visualizador del sistema, se desarrollaron distintos mecanismos de comunicación serial. Los cuales permiten el intercambio de datos entre programa visualizador y el sistema que se desea graficar.

3 Visualizador del Sistema Operativo S.O.D.I.U.M.

En el escrito [5], se describe de qué forma se planea construir la aplicación del Visualizador del S.O.D.I.U.M. . En él se plantea que dicho software sea desarrollado de manera tal que pueda ser utilizado por los usuarios de dos formas distintas:

a.- Ejecutando S.O.D.I.U.M. en una máquina virtual
b.- Ejecutado S.O.D.I.U.M. en una máquina física.

Utilizando la primera alternativa mencionada, el alumno podrá ejecutar una imagen del sistema operativo en una máquina virtual dentro de una misma plataforma, y al mismo tiempo interactuar con este utilizando el programa visualizador. Seguidamente, en la figura 1 se muestra como se implementaría la aplicación en la misma terminal. Si bien un entorno simulado no presenta, en su totalidad, las mismas características que una maquina real, se determinó que era conveniente desarrollar esta opción para el caso en particular en que los educadores y estudiantes no posean dos terminales en donde realizar las pruebas pertinentes con el Visualizador de S.O.D.I.U.M. En consecuencia, los usuarios podrán ejecutar el paquete completo del Visualizador en la misma terminal, utilizando la máquina virtual Bochs para poder ejecutar una imagen compilada de S.O.D.I.U.M. dentro de un entorno de trabajo bajo Linux. Esto es importante debido a que la utilización de dicha VM permite trabajar emulando casi en su totalidad las mismas prestaciones que ofrece el hardware de una máquina real en una computadora totalmente distinta.

Por otra parte se está desarrollando la posibilidad de ejecutar S.O.D.I.U.M. en una máquina real conectada a otra terminal en donde se estará ejecutando el Visualizador. Por dicho motivo, en la figura 2 se podrá observar esta situación en particular.

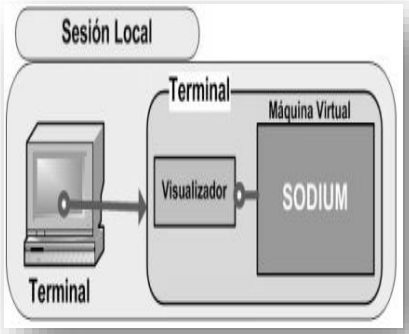
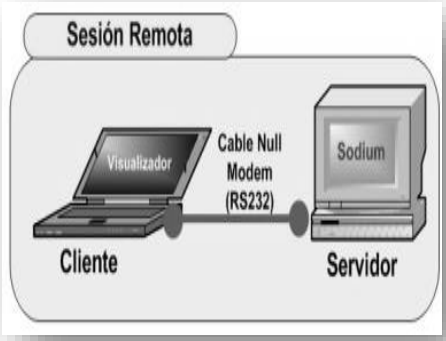


Fig. 1. Representación gráfica de la ejecución del visualizador y S.O.D.I.U.M. en la misma terminal



1 **Fig. 2.** Representación gráfica de la ejecución del visualizador y SODIUM en dos terminales.

4 Conexión entre terminales

Como se mencionó en [5], con el objeto de conseguir la comunicación entre visualizador y sistema operativo, se desarrolló un driver que permite el intercambio de datos a través de los puertos serie. Por consiguiente, se consideró necesario establecer un contrato en la configuración de dicha conexión entre la aplicación cliente y servidor, con la finalidad de conseguir correctamente la transferencia de información. Utilizándose inicialmente una transmisión con el formato estándar 8N1, a una velocidad de transferencia de 9600 bps y utilizando un cable Null-Modem con conectores RS-232. Este acuerdo fue necesario dado que la rapidez de la transacción de datos depende fundamentalmente de factores físicos, tales como el modelo del chip UART utilizado y la longitud del cable serial.

En consecuencia, se están realizando distintas pruebas utilizando un hardware determinado, con el objetivo de poder ir aumentando gradualmente la velocidad de transferencia de información. Intentando ver, la factibilidad de alcanzar la máxima velocidad posible de transferencia de 115200bps a través de medios seriales.

5 Visualización de Estructuras del Sistema Operativo

Según [1], cualquier sistema visualizador debe ser capaz de gestionar grandes cantidades de datos, manipulándolos y realizando cálculos que permitan generar rápidamente prototipos para que el usuario pueda observar y comprender la información obtenida. Por consiguiente, se pretende que el visualizador reciba información sobre los componentes y estructuras del sistema operativo durante su ejecución, de forma que al recibir estos datos la aplicación generé automáticamente gráficos comprensibles por el estudiante.

La interacción entre el usuario y el visualizador es la característica fundamental en el diseño de todo visualizador de un sistema complejo, dado que de esta manera el alumnado puede llegar a entender más rápido los conceptos. Por consiguiente, como el visualizador es un aplicación externa, se determinó que era conveniente investigar la forma de implementar un mecanismo que permita al usuario controlar la ejecución del sistema operativo remotamente. Así, el visualizador ofrecerá la posibilidad de detener las operaciones que realiza S.O.D.I.U.M. y ver el estado de sus componentes en el momento en que se desee.

La primera parte del análisis, consistió en comprender el funcionamiento básico de los mecanismos utilizados por los debuggers para controlar la ejecución de cualquier programa.

6 Análisis de técnicas utilizadas por los Debuggers

La tarea principal es detener la ejecución de los programas en determinados momentos establecidos por los usuarios, donde se podrá analizar su estado y el del procesador en ese instante, lo que se pretende realizar en S.O.D.I.U.M. imitando la esencia de dicho comportamiento.

El corazón de todo depurador es el breakpoint, también conocido como punto de parada. Los cuales pueden ser clasificados según el mecanismo utilizado durante su desarrollo [6]. Los Breakpoints desarrollados por software son los más utilizados por los debuggers existentes. Esto es debido a su simplicidad y alcance durante su implementación. Si bien estos no presentan las mismas utilidades que los que ofrecen los desarrollados por Hardware, presentan el beneficio de poder ser utilizados en gran cantidad, mientras que los otros se limitan a utilizar únicamente cuatro puntos de paradas, como consecuencia de que dicha cantidad es determinada por los registros de la CPU DR 0, 1, 2 y 3. Otra diferencia destacable, es que para su implementación solo es necesario reemplazar un solo byte en la dirección de inicio de la instrucción donde se desea detener la ejecución, por el Opcode de la instrucción assembler “Int3”. Mientras que para la utilización de los breakpoint por hardware, es indispensable indicar en los registros DR0-DR3 la dirección en donde se desea detener la ejecución, e indicar además las condiciones que deberán cumplir esas direcciones en el registro DR7. Gracias al empleo del último registro nombrado, los HW Breakpoints pueden ser utilizados para detener la ejecución de un programa al momento de ejecutar una instrucción. Así como también, cuando se lean o escriban datos en una dirección particular de memoria. Esta característica puede llegar a ser provechosa, para la situación especial en que se desea detener la ejecución de S.O.D.I.U.M. cuando se escriban o lean datos en las direcciones asignadas a los puertos COM durante la comunicación con el Visualizador.

En consecuencia, algunos debuggers solamente implementan breakpoints por Software. Sin embargo, también existen depuradores que utilizan ambas clases de puntos de paradas, como por ejemplo GDB.

6.1 Ejecución de Breakpoints implementados por Software

Esto se puede realizar en el código del programa de dos formas distintas. La primera de ellas es insertando en el código fuente la instrucción assembler INT 3, pero este mecanismo únicamente puede ser utilizado antes de la compilación de la aplicación. La segunda posibilidad es la que llevan a cabo los debuggers, comentada en el punto anterior, que es reemplazar el Opcode de la instrucción a detener por el de Int3 (0xCC) durante la ejecución [7][8]. Por lo que después que se produce dicha sustitución, el sistema operativo deberá retroceder el registro EIP en un byte con la finalidad de ejecutar dicha instrucción. Una vez que se produce la ejecución de dicho comando, de cualquiera de las dos formas antes descriptas, se producirá una excepción 3, siendo capturado por el handler del sistema operativo luego de este evento.

6.2 Ejecución de Breakpoints implementados por Hardware

Una vez configurados, los registros de parada, el procesador compara la dirección de la instrucción en ejecución con el valor contenido en los registros DR0-DR3, y si existe coincidencia posteriormente evaluará las condiciones de debug declaradas en el registro DR7. En el caso de las comparaciones sean satisfactorias, la CPU emitirá una excepción 1, la que deberá ser capturada por el handler en el Kernel del sistema operativo.

Hasta el momento en las pruebas iniciales en la investigación en curso, se consiguió capturar dichos sucesos desarrollando los handlers correspondientes, de forma tal, que pudieron ser invocados utilizando la inserción de la instrucción INT en el código fuente de los programas utilizados en S.O.D.I.U.M. Actualmente, se está analizando la factibilidad de implementar la segunda posibilidad para la ejecución de los SW Breakpoints, que es la inserción del Opcode de la instrucción INT 3 durante la ejecución del

sistema operativo, además se está determinando si es viable el desarrollo de los HW Breakpoints configurando los registros de la CPU.

7 Análisis de Gdbstub

GDB ofrece un módulo denominado Gdbstub [9] que permite analizar el funcionamiento de programas en entornos remotos. Esto es particularmente deseable donde estos entornos, por sus características de implementación (escases de recursos, ausencia de consola local, problemas de accesibilidad, etc.), no son capaces de darle al desarrollador la posibilidad de trabajar localmente.

Este módulo se provee en forma de código fuente en lenguaje C como parte del kit de desarrollo del depurador GDB, del cual además se disponen varias implementaciones, cada una especializada para interactuar con una arquitectura de procesador en particular.

El equipo desde donde se efectúa el control del programa principal es llamado *host* (huésped), mientras que la computadora donde está funcionando la aplicación a analizar es conocida como *target* (objetivo). Cabe mencionar, que entre ambas terminales se establece un vínculo por medio de una conexión serie.

El propósito concreto de este módulo es el de implementar la capa lógica de comunicación entre el depurador GDB que se ejecuta en el equipo *host* y el programa o sistema a ser depurado en el *target*. Para ello se utiliza el protocolo RSP, que ya fue comentado en [5].

Al aislar al implementador de la necesidad de resolver el desarrollo de la capa lógica de este protocolo de comunicación, se gana estabilidad y robustez en la solución, permitiendo además enfocar el esfuerzo sobre la interacción específica de este módulo con el sistema operativo en sí mismo.

Esta aislación se logra presentando al desarrollador una interfaz clara (contrato) donde se define una serie de métodos cuyas responsabilidades el mismo deberá implementar para facilitar al módulo gdbstub la obtención de información del sistema.
Estas rutinas son las que permiten atender y responder los mensajes recibidos por el vínculo serial,

Gdbstub implementa un *handle-exception* que toma el control cuando se detiene la ejecución del proceso, por ejemplo, en un breakpoint. En ese momento, handle-exception se comunica con GDB en la máquina host. Handle-exception actúa como un representante de GDB en la máquina target. Comienza por enviar un resumen de información del estado del proceso. Luego, continúa la ejecución, recibiendo y transmitiendo cualquier información que GDB necesita. Cuando GDB ordena resumir la ejecución normalmente, Handle-exception devuelve el control al propio código de la aplicación en la máquina target. Cada vez que handle-exception es llamada, ésta tiene la oportunidad de tomar el control. Esto puede suceder todo el tiempo, inclusive cuando se reciben caracteres por la comunicación serial. De todas formas, se puede forzar la interrupción llamando a la función *breakpoint.*

8 CONCLUSIONES

Hasta la fecha en la investigación en curso, se consiguió establecer la base inicial de una de las funcionalidades esenciales que deberá ofrecer el visualizador de S.O.D.I.U.M. en cuanto a la interacción con el estudiante. A partir del handler desarrollado, se buscará que el usuario pueda detener la ejecución del sistema operativo en forma remota desde el visualizador. Posteriormente podrá observar detalladamente el estado del sistema en ese instante. En consecuencia, como se mencionó anteriormente, se está evaluando la factibilidad de desarrollar breakpoints por software y/o hardware así como también complementar las funcionalidades del handler construido con las que ofrece el módulo de Gdbstub. De esta manera, se está construyendo una de las partes fundamentales que tendrá el visualizador de estructuras de un sistema operativo con fines educativos.

Referencias

1. Robert P. Bosch Jr., "*Using Visualization to Understand The Behavior of Computer System",* Agosto 2001.

2. Farzaneh Zareie y Mahsa Najaf-Zadeh " *OSLab: A Hand-on Educational Software for Operating System Concepts Teaching and Learning",* Research WebPub, Septiembre 2013

3. Besim Mustafa, "*Visualizing the Modern Operating System: Simulation Experiments Supporting Enhanced Learning",* Edge Hill University, SIGITE'11, Año 2011

4. Ali Alharbi, Frans Henskens, and Michael Hannaford,*"Integrated Standard Environment for the Teaching and Learning of Operating Systems Algorithms Using Visualizations",* The University of Newcastle, Australia, IEEE, Año 2010

5. Graciela De Luca, Martín Cortina, Nicanor Casas, Esteban Carnuccio, Sergio Martín, "*Mecanismos de visualización de estructuras de un sistema operativo en ejecución a través de la comunicación serial",* Universidad Nacional de La Matanza, Congreso WICC 2014, Ushuaia, Tierra del Fuego.

6. Intel, "*Intel® 64 and IA-32 Architectures Software Developer's Manual",* Mayo 2007

7." Debugging in AMD64 64-bit Mode in Theory", http://x86asm.net/articles/debugging-in-amd64-64-bit-mode-in-theory/index.html

8. ***"****How debuggers work: Part 2 – Breakpoints",* http://eli.thegreenplace.net/2011/01/27/how-debuggers-work-part-2-breakpoints/

9. *"Debuging Remote Program:",* https://sourceware.org/gdb/onlinedocs/gdb/Remote-Debugging.html#Remote-Debugging

5)Mecanismos de visualización de estructuras de un sistema operativo en ejecución a través de la comunicación serial.

Graciela De Luca, Martín Cortina, Nicanor Casas
Esteban Carnuccio, Sergio Martín.

Departamento de Ingeniería e investigaciones Tecnológicas
Universidad Nacional de La Matanza
Florencio Varela 1903 - San Justo, Buenos Aires, Argentina
Te.(54-11) 4480-8900
gdeluca@ing.unlam.edu.ar; martin.cortina@yahoo.com.ar; ncasas@ing.unlam.edu.ar
estebancarnuccio@gmail.com; smartin@ing.unlam.edu.ar

Resumen

Dentro del contexto de un proyecto en el que nos proponemos lograr el control y la visualización de las estructuras internas de un sistema operativo en tiempo de ejecución, nos encontramos con la necesidad de implementar un protocolo de comunicación viable entre el sistema estudiado y el de control. En este documento compartiremos las líneas de investigación que estamos llevando a cabo para lograrlo, teniendo como objetivos imponer un mínimo costo adicional en la comunicación y compatibilidad con herramientas de depuración remota basadas en GDB (GNU ***DeBugger***). Para ello estudiaremos el protocolo ya implementado por este depurador y su capacidad de expansión. También analizaremos la factibilidad de llevar a cabo esta comunicación tanto por puerto serie como a través de la arquitectura de plug-ins disponible en Bochs

Palabras clave: GDB, Remote Serial Protocol (RSP), gdbstub, Comunicación Serial, Instrumentación, Bochs.

Contexto

Esta Línea de Investigación es parte del proyecto “Visualización de estructuras internas de un Sistema Operativo en ejecución como herramienta didáctica”, dependiente de la Unidad Académica del Departamento de Ingeniería e Investi-gaciones Tecnológicas perteneciente al programa de Investigaciones CyTMA2 de la Universidad Nacional de La Matanza, el cual es continuación de otros proyectos, en especial del sistema operativo de características didácticas: SODIUM.

Introducción

El proyecto en el cual está inserta esta línea de investigación estará enfocado a permitir la visualización de diversas estructuras internas y estadísticas de un sistema operativo, mientras el mismo se encuentra en ejecución, primeramente del sistema operativo SODIUM desarrollado por este equipo de investigación y luego posteriormente se prevé ampliarlo a otros sistemas operativos. Se pretende poder comprobar didácticamente el funcionamiento de los algoritmos de planificación y administración de recursos utilizados, brindando de esa forma una herramienta importante a la

hora de enseñar la teoría relacionada a sistemas operativos y arquitectura de computadoras.

Se decidió la implementación de un driver de puerto serie para SODIUM debido a que es un dispositivo de fácil interacción y el costo de manejarlo en cuanto a cantidad de ciclos de CPU no es significativo. Estimamos que entre las alternativas posibles (puerto serie, placa de red, puerto USB) esta es la que menos *overhead* proporcionará al Sistema Operativo a nivel de ciclos de CPU.

En una primera etapa tenemos la intención de desarrollar un protocolo de comunicación bidireccional que nos permita alterar la ejecución de un sistema operativo y obtener los estados y estructuras internas del mismo. Dentro de este contexto, deberemos definir un set de puntos de instrumentación necesarios para identificar la situación actual de cada módulo y estructura de interés.

Implementación de puerto serie en SODIUM

Debido a que hasta el momento SODIUM carecía de un controlador de puerto serie, fue necesario desarrollarlo, para lo cual recopilamos la información necesaria para su construcción [4][5]. Teniendo en cuenta que para la comunicación serial se utilizarán *interrupciones y polling*, indistintamente, fue necesario profundizar en el funcionamiento de la UART[11] [4], específicamente de los registros para el manejo de señales mediante *interrupciones* en modo asíncrono y llevar un control que se define en el driver del puerto Serie.

El driver presenta un conjunto de funciones que permiten inicializar los puertos COM, establecer la configuración de comunicación (velocidad de transmisión, formato de envío de datos, entre otros), transmisión, recepción y tratamiento de errores, contemplando el uso de *interrupciones* y *polling*, utilizando los ***Registros de direcciones de los puertos de E/S*** (tabla 1), para definir los puertos COM y una línea de interrupciones IRQ para llamar la atención del procesador y los ***Registros utilizados para la comunicación*** (tabla 2).

Puerto	Registro	(IRQ)
COM1	3F8	4
COM2	2F8	3
COM3	3E8	4
COM4	2E8	3

Tabla1 Direcciones estándar en la mayoría de las arquitecturas en base hexadecimal

Registros E/S	Descripción
3F8/2F8	Registro de datos Emisión/Recepción
3F9/2F9	Registro de habilitación de interrupciones
3FA/2FA	Registro de identificación de interrupciones
3FB/2FB	Registro de control de la línea
3FC/2FC	Registro de control del MODEM
3FD/2FD	Registro de estado de la línea
3FE/2FE	Registro de estado del MODEM

Tabla 2 Registros para la comunicación

Se utilizará inicialmente una transmisión de 9600bps con formato 8N1 (8 bits de datos | sin paridad | 1 bit de parada). Posteriormente se evaluará incrementar la velocidad gradualmente hasta 115200bps.

El cable a utilizar es un NULL MODEM con un extremo de cable RS-232 y el otro extremo un USB adaptado, el cual permitirá la conexión con cualquier dispositivo que se use como terminal (laptop, PC sin conexión Serie, etc.).

[11] "**U**niversal **A**synchronous **R**eceiver-**T**ransmitter"

Inicio de la sesión de visualización

Durante una sesión de visualización, se utilizarán dos computadoras personales, de las cuales una de ellas puede llegar a encontrarse virtualizada. A la primera la denominaremos *Cliente* y es en donde se estará ejecutando la aplicación de visualización. Esta máquina posee el programa GDB instalado con la finalidad de depurar SODIUM remotamente. La segunda terminal, *Servidor* es donde se estará ejecutando el sistema operativo a ser estudiado. Para interconectar ambos equipos, la situación varía, por lo que detallaremos cada caso individualmente.

Computadora Física: El sistema operativo puede ejecutarse en una PC de escritorio conectada físicamente con el *cliente* a través del cable serie. Para los casos en los que la PC cliente no posee un puerto serie propio, se puede anexar un puerto serie por USB que, una vez reconocido, funcionará correctamente.

Máquina Virtual: Necesaria para el caso particular en que el usuario no cuente con disponibilidad para utilizar más de una máquina para poder probar el funcionamiento. Como consecuencia, se permitirá ejecutar este sistema operativo en un emulador como Bochs o Vmware. En la investigación en curso, se utiliza un entorno de desarrollo en el cual se puede ejecutar una imagen de Linux montada en máquina virtual Vmware, y a su vez dentro de ella es posible ejecutar el Sistema Operativo SODIUM en una VM Bochs.

Análisis y extensión del protocolo de comunicaciones utilizado por GDB.

Mediante la utilización de este puerto se prevé la posibilidad de controlar dicho sistema operativo de forma remota, permitiendo detener y reanudar su ejecución cuando el usuario lo desee. Dado que las implementaciones actuales del GDB nos permiten depurar remotamente el sistema a través del soporte de un módulo denominado **gdbstub** ya embebido en la máquina virtual BOCHS, y estando ya familiarizados con su uso y virtudes, consideramos un paso natural extender esta funcionalidad a la ejecución del SODIUM directamente sobre un equipo físico. Otro punto a favor es que los desarrolladores del GDB proporcionan implementaciones básicas de este módulo listas para adaptar a nuevas plataformas o arquitecturas de hardware.

De acuerdo a lo estudiado [1], el protocolo de comunicación remota serie utilizado por GDB (de ahora en más RSP) posee las siguientes características:

- Basado en el set de caracteres ASCII. Esto significa que todos los caracteres enviados se encuentran dentro del rango imprimible, lo que nos facilita su estudio, documentación, y depuración

- Posee caracteres de inicio y terminación de mensaje claros. La parte útil de cada mensaje comienza por un símbolo “$” y termina con un símbolo “#”

- Las operaciones se definen por un caracter alfabético sensible a las mayúsculas seguido por una lista de argumentos requeridos, separados por coma.

- Posee control de errores. Luego del símbolo terminador de cada mensaje enviado se escribe un número que es el resultado de la suma módulo 256 de la suma del peso de cada uno de los caracteres comprendidos.

- La recepción de un mensaje se confirma inmediatamente por el receptor, indicando “+” si el mismo se recibió íntegramente, “-“ si se detectó

un error en el formato del mensaje o durante la ejecución del mismo. Siguiendo a dicho caracter, si es necesario, se transcribe el resto de la respuesta, o un código y descripción de error.

- Una transacción típica consiste en la emisión de un comando por parte del depurador hacia el sistema remoto y su respectiva respuesta. La única excepción a este comportamiento se da cuando el sistema remoto emite un mensaje hacia el depurador con el objetivo de que este último lo muestre en su salida de consola. Esta acción puede llevarse a cabo en cualquier momento siempre y cuando no exista una transacción todavía en proceso.

- Los datos respondidos por el sistema depurado pueden estar comprimidos mediante la técnica RLE (Run-length encoding) o no, con lo que se puede reemplazar secuencias de más de tres caracteres consecutivos similares por la emisión de un carácter, un signo *, y un contador numérico de repeticiones.

Dentro de los comandos estándares que el RSP contempla, podemos encontrar las siguientes operaciones:
- Iniciar o detener la ejecución de un programa remoto.

- Leer o escribir uno, varios o todos los registros de la CPU a la vez.

- Leer o escribir una cantidad de bytes a partir de una dirección de memoria determinada.

- Ejecutar instrucciones de máquina de a una por vez o ejecutar a toda velocidad hasta solicitar la detención o haber alcanzado el sistema remoto un punto de excepción.

Para permitir la visualización del estado de ejecución del sistema operativo estudiado [2][3], debemos tener en cuenta una solución que nos proporcione la flexibilidad suficiente como para conectar tanto el GDB como nuestro propio programa cliente.

Para que el visualizador esté al tanto en tiempo real acerca de la creación de nuevos procesos, asignaciones de memoria, cambios de estado en el planificador, cambios de contexto, accesos a dispositivos de entrada y salida, accesos al sistema de archivos, y otros, deberemos informar al mismo cada vez que estemos transitando las regiones de código fuente de nuestro kernel asociadas a dichas acciones.
Denominamos a estas regiones “puntos de instrumentación”.
Dentro de las posibles soluciones que estamos planteando se encuentran:
- Utilización de ***Hardware Watchpoints*** implementados a través de los ***Debug Registers*** provistos por la arquitectura i386.

- Invocación a rutinas de log en los Puntos de Instrumentación.

.

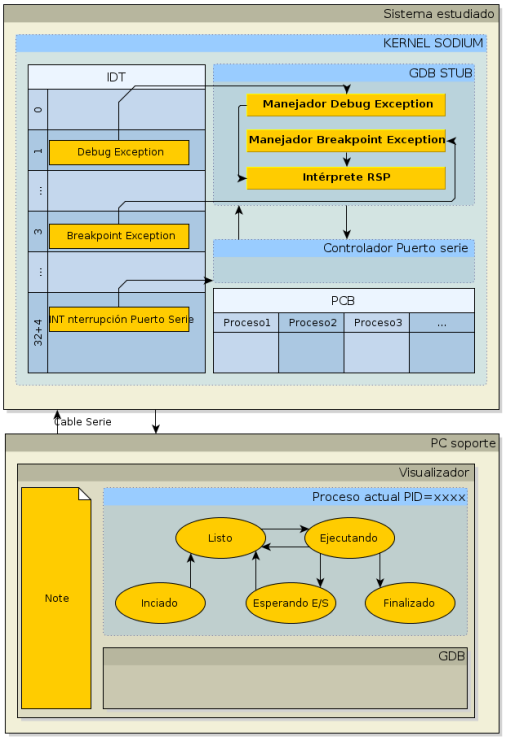


Ilustración 1 Diagrama en bloques de la implementación del gdbstub dentro del kernel de SODIUM y representación del estado de un proceso en la interfaz del visualizador [3].

Luego de tomar el control en el punto adecuado, proponemos comunicar al visualizador el cambio de estado. El mensaje deberá ser originado desde el sistema estudiado, de modo que el visualizador deberá soportar la recepción asincrónica del mismo. Para ello, podemos o bien tomar ventaja del mecanismo de salida a consola contemplado en el RSP, o desarrollar el soporte de un nuevo set de operaciones en el gdbstub y en el mismo GDB que contemple la totalidad de los puntos de instrumentación a definir.

Líneas de Investigación, Desarrollo e Innovación

- Investigar protocolos de comunicación serie existentes y adoptar o definir el que utilizaremos, para enviar datos entre la Máquina Cliente y el Servidor, para transferir los datos y las estructuras a visualizar, del sistema operativo.

- Determinar capacidad máxima sostenida de transferencia mensajes para nuestro protocolo de comunicación.

- Investigar la factibilidad de desarrollar un plug-in para máquina virtual capaz de ejecutar acciones especificadas al momento en el que el sistema operativo atraviese un punto de instrumentación.

Resultados y Objetivos

Actualmente hemos concluido el desarrollo del driver del puerto serie para el sistema operativo SODIUM.

Estamos comenzando la etapa de análisis relativa al protocolo RSP y del módulo gdbstub provisto por GDB.

A su vez, nos encontramos sentando las pautas de desarrollo del visualizador, que en una primera etapa se conectará con SODIUM, y en etapas posteriores con otros sistemas operativos de código abierto, para poder analizar el funcionamiento y utilizarlo como elemento didáctico para la enseñanza de Sistemas Operativos.

Formación de Recursos Humanos

El grupo de trabajo consta de dos investigadores de categoría IV, un investigador categoría V, un investigador con 4 años de experiencia y un investigador inicial.

Referencias

[1]Bill Gatliff “*Embedding with GNU: the GDB Remote Serial Protocol*”

revista Embedded Systems Programming, Noviembre 1999

[2]Robert P. Bosch Jr., "*Using Visualization to understand the Behavior of Computer Systems, Ph.D. dissertation, Stanford University*", August 2001.

[3]Alharbi, A.; Henskens, F.; Hannaford, M., "*Integrated Standard Environment for the Teaching and Learning of Operating Systems Algorithms Using Visualizations*," Computing in the Global Information Technology (ICCGI), 2010 Fifth International Multi-Conference on , vol., no., pp.205,208, 20-25 Sept. 2010

[4]"*Serial and UART Tutorial*", http://www.freebsd.org/doc/en/articles/serial-uart/

[5]"Puertos Serie"
http://wiki.osdev.org/Serial_Ports

ANEXO 2 CURSOS REALIZADOS

A. UNIVERSIDAD NACIONAL DE LA MATANZA
Curso: Curso para la enseñanza universitaria: “Formación para la enseñanza universitaria”
Año: 2015
Duración: 60 hs.
Profesor a Cargo: Mg. Hector Edgardo Kasem, Lic Maura Ramos.
Objetivos Generales:
Este espacio forma parte de un Programa de Desarrollo Profesional Docente que tiene a cargo la formación de docentes en ejercicio, graduados de UNLaM aspirantes a docentes y personal a cargo de las cátedras. Se desarrollan a través de cursos anuales, talleres y conferencias.
En esta caso, el curso está destinado a profesionales que ejercen su tarea de enseñar en la universidad. La concepción de desarrollo profesional de los docentes implica pensar esta formación como otro aspecto de su trayectoria, vista como un continuo; se produce cuando los expertos disciplinares construyen conocimiento relativo a la práctica, analizan su trabajo y lo relacionan con las condiciones en las que se desenvuelven la universidad y los requerimientos de cada disciplina/profesión.
Programa:
- Unidad 1 :

El aula Universitaria

- Unidad 2:

Currículo y programación de la enseñanza universitaria

- Unidad 3:

Secuenciación y evaluación de la enseñanza universitaria

B. UNIVERSIDAD NACIONAL DE LA MATANZA
Curso: Desarrollo Ágil con Tecnología ARM CORTEX-M
Evento: VI CONGRESO DE MICROELECTRÓNICA APLICADA
Año: 2015
Duración: 40 hs.
Profesor a Cargo: Ing. Nicolás Molina Vuistaz
Objetivos Generales:
En este curso se pretende dar una breve introducción a los procesadores ARM Cortex-M en el que se incluirán recomendaciones sobre las herramientas de desarrollo más adecuadas.
La propuesta continúa con ejemplos utilizando periféricos básicos, finalizando con el desarrollo de filtros digitales.

Programa:
- Módulo I

Introducción a la arquitectura ARM CORTEX-M. Conceptos básicos de desarrollo de software ARM CORTEX-M. Introducción a la placa de desarrollo STM32F4 DISCOVERY. Practicas con la placa STM32F4 DISCOVERY.

- Módulo II

Herramientas de desarrollo Ágil. Introducción Periféricos básicos. Manejo de interrupciones. Practicas con la placa STM32F4 DISCOVERY.

- Módulo III

Conceptos avanzados. Diseño de filtros digitales. Herramientas de desarrollo en forma colaborativa. Practicas con la placa STM32F4 DISCOVERY.

C. UNIVERSIDAD NACIONAL DE LA PLATA FACULTAD DE INFORMÁTICA

Calle 120 y 50 – 2do piso (1900) La Plata Pág. 1 de 2 http://postgrado.info.unlp.edu.ar TEL-FAX: (54) 221-4273235 E-Mail: postgrado@lidi.info.unlp.edu.ar

PROCESAMIENTO INTELIGENTE DE BIG DATA: TÉCNICAS, APLICACIONES Y NUEVOS RETOS

Año 2014 Duración: hs70
Profesor a Cargo: Olivas, Jose A.

PROGRAMA
SOME INTELLIGENT ALGORITHMS FOR BIG DATA:
- Data and Information Fusion
- Genetic Algorithms
- Machine Learning
- Natural Language Processing
- Dimensionality Reduction Techniques
- Multidimensional Big Data

SOME INTELLIGENT BIG DATA SEARCH & MINING METHODS:
- Data Mining
- Social Networks
- Data Science
- Web Search and Information Mining
- Scalable Search Architectures
- Cleaning Big Data (noise reduction), Acquisition & Integration
- Visualization Methods for Search
- Time Series Analysis
- Recommendation Systems
- Graph Mining and Other Similar Technologies

SOME APPLICATIONS OF BIG DATA:
- Applications in Science, Engineering, Healthcare, Visualization, Business, Education, Security, Humanities, Bioinformatics, Health , Informatics, Medicine, Finance, Law, Transportation, Retailing, Telecommunication, all Search-based applications, ...

SOME BIG DATA FUNDAMENTALS:
- Computational Science
- ComputationalIntelligence

BIBLIOGRAFÍA
Adriaans, P. W.; Zantinge, D.: Data Mining. Addison-Wesley, 1996.
Berry, M. J. A.; Linoff, G.: Data Mining Techniques. Wiley Computer Publishing.New York, 1996.
Fayyad, U.; Piatetsky-Shapiro, G.; Smyth, P.: The KDD Process for Extracting Useful Knowledge from Volumes of Data. Communications of the ACM, November 1996/ Vol 39, Nº 11, pp. 27 – 34.
Fayyad, U.; Piatetsky-Shapiro, G.; Smyth, P.; Uthurusamy, R. (Eds): Advances in Knowledge Discovery and Data Mining. AAAI/MIT Press, Cambridge MA, 1996.

Joyanes, Luis: Big Data - Análisis de grandes volúmenes de datos en organizaciones, Alfaomega, 2013.
Mayer-Schönberger, V.; Cukier, K.: Big data. La revolución de los datos masivos. Turner 2013.
Olivas, J. A.: Búsqueda eficaz de información en la Web, EDULP, 2011.
Piatetsky-Shapiro, G.; Frawley, W.: Knowledge Discovery in Databases. AAAI/MIT Press, Cambridge MA, 1991.
Siegel E.: Analítica predictiva. Predecir el futuro utilizando Big Data. Anaya Multimedia-Anaya Interactiva, 2013.

D. UNIVERSIDAD NACIONAL DE TUCUMÁN
FACULTAD DE CIENCIAS EXÁCTAS Y TECNOLOGÍAS
Horco Molle, Yerba Buena, Tucumán http://www.sase.com.ar/asociacion-civil-sistemas-embebidos/escuela/
Curso: Arquitectura y Programación de Microcontroladores de 32 bits (Parte I)
Evento: Tercera Escuela para la Enseñanza de Sistemas Embebidos
Año: 2014
Duración: 40 hs.
Profesor a Cargo: Ing. Juan Manuel Cruz
OBJETIVOS GENERALES:
Brindar un suave y sencillo acercamiento a la arquitectura, tecnología, técnicas y herramientas que faciliten la concreción de aplicaciones prácticas con micros de 32 bits, específicamente con LPC1769 (Cortex M3 de la serie LPC17XX de NXP). Para cumplir con tal objetivo se recurrirá a la presentación de temas teóricos, la presentación y aplicación de técnicas y herramientas mediante la ejercitación básica e integradora debida.
PROGRAMA
1. Introducción a Arquitectura y Programación de microcontroladores de 32 bits.
1. Diseño de Sistemas Embebidos con microcontroladores.
2. ARM Cortex M3 (Introducción, Generalidades y Fundamentos).
3. MCU NXP LPC1769.
4. Programación en C para Embebidos & CMSIS
2. Uso de modelos en la programación de microcontroladores.
1. Evolución de la Maquina de Estado al Diagrama de Estado.
2. Codificación en C de modelos
3. Herramientas de edición, verificación y validación de modelos (IAR visualSTATE)
4. Ejercitación con modelos: salidas, entradas, temporización, drivers y aplicaciones
3. Prácticas con herramientas LPCXpresso& IAR visualSTATE
1. Introducción.
2. Salidas, Entradas y Systick.
3. Drivers y Aplicaciones.

BIBLIOGRAFÍA DEL CURSO:
1. The Definitive Guide to the ARM Cortex-M3 - Joseph Yiu, 2a Edición, Newnes Elsevier Inc, 2010
2. CortexTM-M3, Revision r2p0, Technical Reference Manual – ARM
3. ARM®v7-M Architecture, Reference Manual – ARM
4. UM10360 LPC176x/5x User Manual & LPC1769/68/67/66/65/64/63 Product data sheet – NXP
5. Reference Guide & User Guide of visualState – IAR
6. El Lenguaje Unificado de Modelado, G. Booch, J. Rumbaugh, I. Jacobson, 2a Edición, Addison-Wesley, 2006

XX Congreso Argentino de Ciencias de La Computación - CACIC 2014 - Universidad Nacional de La Matanza - RED UNCI

San Justo, Buenos Aires,
http://cacic2014.ing.unlam.edu.ar/cacic2014/es/escuela/pdf/%5BC05%5DSoftware-Asociado-Misiones-Satelitales.pdf
Curso: Software Asociado a Misiones Satelitales
Evento: CACIC 2014
Año: 2014
Duración: 20 hs.
Profesor a Cargo: Carlos J. Barrientos
OBJETIVOS GENERALES:
Este curso provee una introducción general a las misiones satelitales y a las características sobresalientes del software de uso espacial. Explica las diversas etapas del ciclo de vida de una misión satelital, los diferentes segmentos o partes en que se es estructura jerárquicamente un sistema de estas características y las diversas soluciones informáticas utilizadas hasta la fecha.

ANEXO 3 Certificados

CONGRESOS - WORKSHOPS

ESCUELA DE TECNOLOGÍA
UNIVERSIDAD NACIONAL | NOROESTE BUENOS AIRES

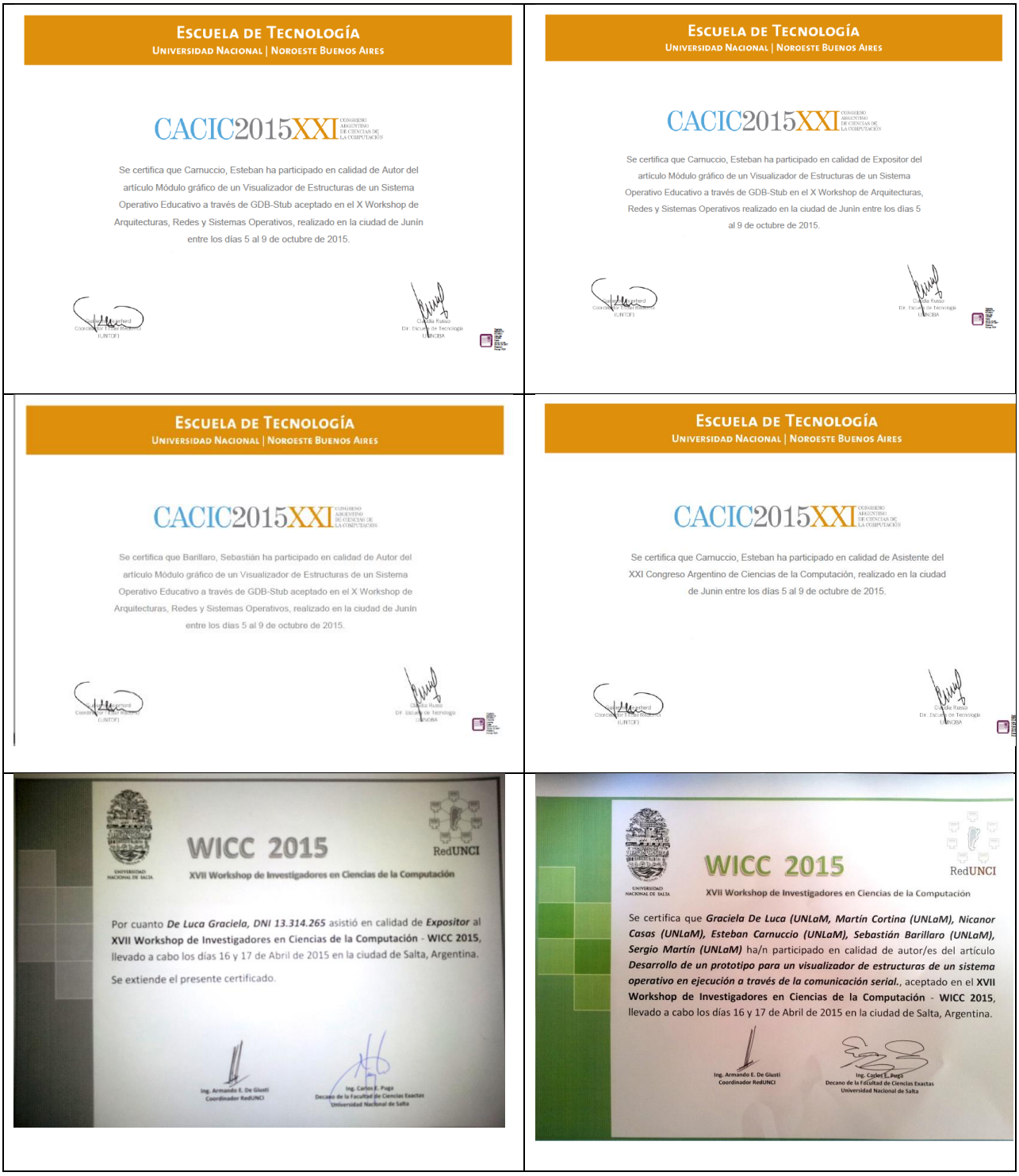
CACIC2015XXI

Se certifica que De Luca, Gaciela ha participado en calidad de Autor del artículo Módulo gráfico de un Visualizador de Estructuras de un Sistema Operativo Educativo a través de GDB-Stub aceptado en el X Workshop de Arquitecturas, Redes y Sistemas Operativos, realizado en la ciudad de Junín entre los días 5 al 9 de octubre de 2015.

ESCUELA DE TECNOLOGÍA
UNIVERSIDAD NACIONAL | NOROESTE BUENOS AIRES

CACIC2015XXI

Se certifica que De Luca, Gaciela ha participado de la Conferencia Computación Cuántica en el marco del XXI Congreso Argentino de Ciencias de la Computación, realizado en la ciudad de Junín, entre los días 5 al 9 de octubre de 2015.



ESCUELA DE TECNOLOGÍA
UNIVERSIDAD NACIONAL | NOROESTE BUENOS AIRES

CACIC2015XXI CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN

Se certifica que Carnuccio, Esteban ha participado en calidad de Autor del artículo Módulo gráfico de un Visualizador de Estructuras de un Sistema Operativo Educativo a través de GDB-Stub aceptado en el X Workshop de Arquitecturas, Redes y Sistemas Operativos, realizado en la ciudad de Junín entre los días 5 al 9 de octubre de 2015.

ESCUELA DE TECNOLOGÍA
UNIVERSIDAD NACIONAL | NOROESTE BUENOS AIRES

CACIC2015XXI CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN

Se certifica que Carnuccio, Esteban ha participado en calidad de Expositor del artículo Módulo gráfico de un Visualizador de Estructuras de un Sistema Operativo Educativo a través de GDB-Stub en el X Workshop de Arquitecturas, Redes y Sistemas Operativos realizado en la ciudad de Junín entre los días 5 al 9 de octubre de 2015.

ESCUELA DE TECNOLOGÍA
UNIVERSIDAD NACIONAL | NOROESTE BUENOS AIRES

CACIC2015XXI CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN

Se certifica que Barillaro, Sebastián ha participado en calidad de Autor del artículo Módulo gráfico de un Visualizador de Estructuras de un Sistema Operativo Educativo a través de GDB-Stub aceptado en el X Workshop de Arquitecturas, Redes y Sistemas Operativos, realizado en la ciudad de Junín entre los días 5 al 9 de octubre de 2015.

ESCUELA DE TECNOLOGÍA
UNIVERSIDAD NACIONAL | NOROESTE BUENOS AIRES

CACIC2015XXI CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN

Se certifica que Carnuccio, Esteban ha participado en calidad de Asistente del XXI Congreso Argentino de Ciencias de la Computación, realizado en la ciudad de Junín entre los días 5 al 9 de octubre de 2015.

WICC 2015
XVII Workshop de Investigadores en Ciencias de la Computación
RedUNCI

Por cuanto ***De Luca Graciela, DNI 13.314.265*** asistió en calidad de ***Expositor*** al **XVII Workshop de Investigadores en Ciencias de la Computación - WICC 2015**, llevado a cabo los días 16 y 17 de Abril de 2015 en la ciudad de Salta, Argentina.

Se extiende el presente certificado.

Ing. Armando E. De Giusti
Coordinador RedUNCI

Ing. Carlos E. Puga
Decano de la Facultad de Ciencias Exactas
Universidad Nacional de Salta

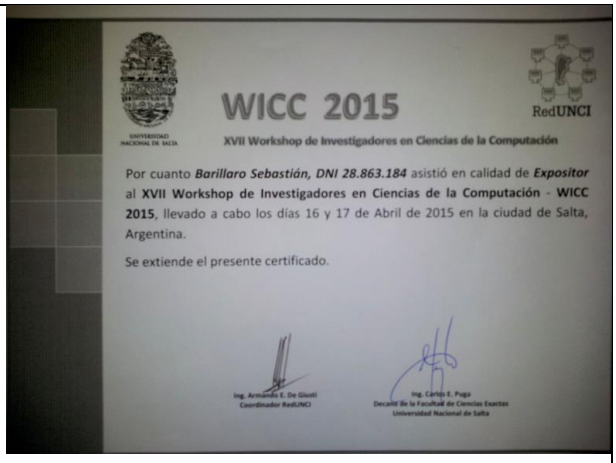
WICC 2015
XVII Workshop de Investigadores en Ciencias de la Computación
RedUNCI

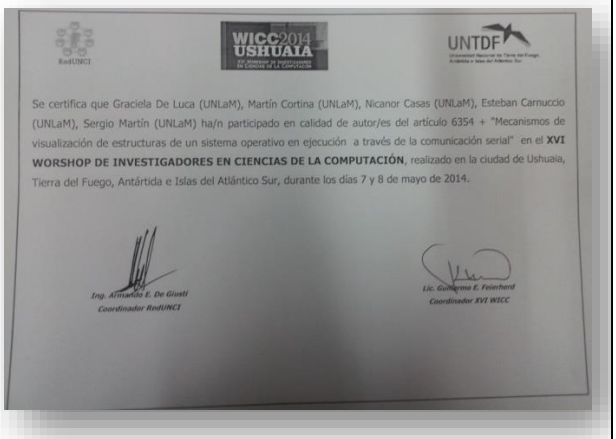
Se certifica que ***Graciela De Luca (UNLaM), Martín Cortina (UNLaM), Nicanor Casas (UNLaM), Esteban Carnuccio (UNLaM), Sebastián Barillaro (UNLaM), Sergio Martín (UNLaM)*** ha/n participado en calidad de autor/es del artículo ***Desarrollo de un prototipo para un visualizador de estructuras de un sistema operativo en ejecución a través de la comunicación serial.***, aceptado en el **XVII Workshop de Investigadores en Ciencias de la Computación - WICC 2015**, llevado a cabo los días 16 y 17 de Abril de 2015 en la ciudad de Salta, Argentina.

Ing. Armando E. De Giusti
Coordinador RedUNCI

Ing. Carlos E. Puga
Decano de la FdCultad de Ciencias Exactas
Universidad Nacional de Salta


CACIC2014
Se certifica que
Graciela De Luca
ha participado como autor del articulo
Visualizador de Estructuras de un Sistema Operativo Educativo
publicado en el XX Congreso Argentino de Ciencias de la Computación, realizado en la ciudad de San Justo, Pcia. de Buenos Aires, del 20 al 24 de Octubre de 2014.
Armando De Giusti
Coordinador Titular de RedUNCI
Osvaldo Spositto
Decano del DIIT - UNLaM
DIIT

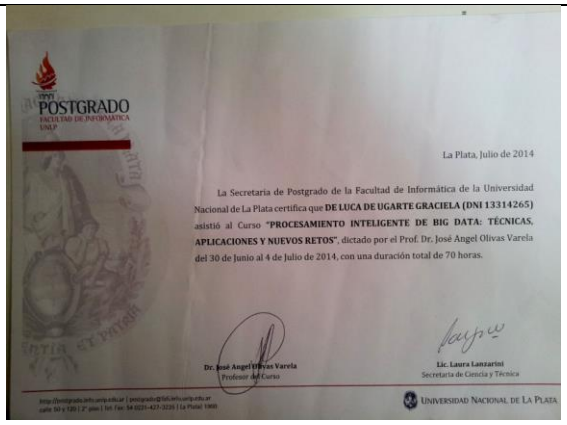

WICC 2015
RedUNCI
XVII Workshop de Investigadores en Ciencias de la Computación
Por cuanto Barillaro Sebastián, DNI 28.863.184 asistió en calidad de Expositor al XVII Workshop de Investigadores en Ciencias de la Computación - WICC 2015, llevado a cabo los días 16 y 17 de Abril de 2015 en la ciudad de Salta, Argentina.
Se extiende el presente certificado.
Ing. Armando E. De Giusti
Coordinador RedUNCI
Ing. Carlos E. Puga
Decano de la Facultad de Ciencias Exactas
Universidad Nacional de Salta


WICC2014 USHUAIA
UNTDF
Se certifica que Graciela De Luca (UNLaM), Martín Cortina (UNLaM), Nicanor Casas (UNLaM), Esteban Carnuccio (UNLaM), Sergio Martín (UNLaM) ha/n participado en calidad de autor/es del artículo 6354 + "Mecanismos de visualización de estructuras de un sistema operativo en ejecución a través de la comunicación serial" en el XVI WORSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN, realizado en la ciudad de Ushuaia, Tierra del Fuego, Antártida e Islas del Atlántico Sur, durante los días 7 y 8 de mayo de 2014.
Ing. Armando E. De Giusti
Coordinador RedUNCI
Lic. Guillermo E. Feierherd
Coordinador XVI WICC


CACIC2014
Se certifica que
Esteban Carnuccio
ha participado como Expositor del trabajo
Visualizador de Estructuras de un Sistema Operativo Educativo
en el XX Congreso Argentino de Ciencias de la Computación, realizado en la ciudad de San Justo, Pcia. de Buenos Aires, del 20 al 24 de Octubre de 2014.
Armando De Giusti
Coordinador Titular de RedUNCI
Osvaldo Spositto
Decano del DIIT - UNLaM
DIIT


CACIC2014
Se certifica que
Esteban Carnuccio
ha participado como autor del artículo
Visualizador de Estructuras de un Sistema Operativo Educativo
publicado en el XX Congreso Argentino de Ciencias de la Computación, realizado en la ciudad de San Justo, Pcia. de Buenos Aires, del 20 al 24 de Octubre de 2014.
Armando De Giusti
Coordinador Titular de RedUNCI
Osvaldo Spositto
Decano del DIIT - UNLaM
DIIT

Cursos – Jornadas - Talleres



REPUBLICA ARGENTINA

Universidad Nacional de La Matanza
Dirección de Pedagogía Universitaria

Se deja constancia que **CARNUCCIO, Esteban Andrés DNI 30.333.327**
ha aprobado el Curso del programa de Desarrollo Profesional Docente
"Formación para la enseñanza universitaria"
Con una duración de 60 horas reloj.
Por tanto, se extiende el presente Certificado.

San Justo, 17 de febrero de 2016

Lic. Jorgelina Monti — Directora
Mg. Gustavo Duch — Secretario Académico

Estrategias de Protección de Proyectos

Certificamos que
Esteban Andres Carnuccio
ha participado en la jornada *"Estrategias de Protección de Proyectos"*
realizada en Diciembre de 2015.

Cdor. Daniel Pontoriero — Secretario Administrativo y de Extensión Universitaria
Ing. Gabriel Blanco — Vicedecano

Universidad Nacional de La Matanza · DIIT · CENUNLaM

POSTGRADO
FACULTAD DE INFORMÁTICA
UNLP

La Plata, diciembre de 2014

La Secretaría de Postgrado de la Facultad de Informática de la Universidad Nacional de La Plata certifica que **CARNUCCIO ESTEBAN ANDRÉS (DNI 30.333.327)** aprobó el Curso **"PROCESAMIENTO INTELIGENTE DE BIG DATA: TÉCNICAS, APLICACIONES Y NUEVOS RETOS"**, dictado por el Prof. Dr. José Ángel Olivas Varela del 30 de Junio al 4 de Julio de 2014, con una duración total de 70 horas.
El curso otorga 3 (tres) créditos para el Doctorado en Ciencias Informáticas.

Lic. Laura Lanzarini — Secretaria de Ciencia y Técnica
Ing. Armando De Giusti — Decano Facultad de Informática

http://postgrado.info.unlp.edu.ar | postgrado@lidi.info.unlp.edu.ar
calle 50 y 120 | 3° piso | Tel. Fax: 54 0221-427-3235 | La Plata| 1900

UNIVERSIDAD NACIONAL DE LA PLATA

CACIC2014
XX Congreso Argentino de Ciencias de la Computación

Se certifica que
Graciela De Luca
DNI: 13314265
ha asistido al XVII Encuentro de Tesistas de Posgrado realizado en la ciudad de San Justo, Pcia. de Buenos Aires, el 23 de Octubre de 2014.

Daniel Giulianelli — Secretario de Investigaciones Tecnológicas
Osvaldo Spositto — Decano del DIIT - UNLaM

DIIT

CACIC2014
XX Congreso Argentino de Ciencias de la Computación

Se certifica que
Sebastián Barillaro
ha asistido al curso
Software Asociado a Misiones Satelitales
dictado en la XVIII Escuela Internacional de Informática CACIC 2014, realizada en la ciudad de San Justo, Pcia. de Buenos Aires, del 20 al 24 de Octubre de 2014.

Alicia Mon — Directora Escuela Internacional de Informática
Daniel Pontoriero — Presidente del Comité de Organización

DIIT

UNIVERSIDAD NACIONAL DE TUCUMAN
FACULTAD DE CIENCIAS EXACTAS Y TECNOLOGÍA
DEPARTAMENTO DE POSGRADO
facet

Certificamos que: **Barillaro Sebastián**
D.N.I. N° 28863184

Ha aprobado el curso:
ARQUITECTURA Y PROGRAMACIÓN DE MICROCONTROLADORES DE 32 BITS (PARTE I
Curso Aprobado Res. HCD 1011/2014 – Expte. Nro. 60956/2014 FACET
Organizado por: **FACULTAD DE CIENCIAS EXACTAS Y TECNOLOGÍA**

Lugar de realización: **Departamento de Electricidad, Electrónica y Computación**
FACULTAD DE CIENCIAS EXACTAS Y TECNOLOGÍA

Duración y fecha de inicio: **40 (cuarenta) horas, 15 de Setiembre de 2014**

San Miguel de Tucumán, 30 de Setiembre de 2014

Mag. María de los A. Gómez López — **Profesor Responsable** — **FACET - UNT**
Dra. Estela Mirta Jaén — **Directora** — **Dpto. de Posgrado FACET**
Ing. Sérgio Pagani — **Decano** — **FACET**

SASE 2015 SIMPOSIO ARGENTINO DE SISTEMAS EMBEBIDOS

Ciudad de Buenos Aires, 14 de agosto de 2015

A QUIÉN CORRESPONDA

Ref.: Certificado de disertante de Tutoriales del SASE 2015

De nuestra mayor consideración:

Por la presente nos dirigimos a ustedes a fin de dejar constancia que el **Ing. Sebastián Bartillaro** ha dictado el tutorial *Placa Intel Galileo: Primeros pasos* de 6 horas, en el Track *Linux Embebido* en el marco del Simposio Argentino de Sistemas Embebidos 2015, el día 14 de Agosto de 2015 en la Facultad de Ingeniería de la Universidad de Buenos Aires.

Sin más, agradeciendo su atención,

Los saludamos cordialmente,

Ing. Gustavo Mercado
Coordinador Tutoriales
Internet of Things
SASE 2015

Dr. Ing. Ariel Lutenberg
Coordinador General
SASE 2015

IOT INTERNET OF THINGS

SASE 2015 SIMPOSIO ARGENTINO DE SISTEMAS EMBEBIDOS

Ciudad de Buenos Aires, 14 de agosto de 2015

A QUIÉN CORRESPONDA

Ref.: Certificado de disertante de Tutoriales del SASE 2015

De nuestra mayor consideración:

Por la presente nos dirigimos a ustedes a fin de dejar constancia que el **Ing. Esteban Carnuccio** ha dictado el tutorial *Placa Intel Galileo: Primeros pasos* de 6 horas, en el Track *Linux Embebido* en el marco del Simposio Argentino de Sistemas Embebidos 2015, el día 14 de Agosto de 2015 en la Facultad de Ingeniería de la Universidad de Buenos Aires.

Sin más, agradeciendo su atención,

Los saludamos cordialmente,

Ing. Gustavo Mercado
Coordinador Tutoriales
Internet of Things
SASE 2015

Dr. Ing. Ariel Lutenberg
Coordinador General
SASE 2015

IOT INTERNET OF THINGS

SASE 2015 SIMPOSIO ARGENTINO DE SISTEMAS EMBEBIDOS

Ciudad de Buenos Aires, 14 de agosto de 2015

A QUIÉN CORRESPONDA

Ref.: Certificado de disertante de Tutoriales del SASE 2015

De nuestra mayor consideración:

Por la presente nos dirigimos a ustedes a fin de dejar constancia que el **Ing. Mariano Volker** ha dictado el tutorial *Placa Intel Galileo: Primeros pasos* de 6 horas, en el Track *Linux Embebido* en el marco del Simposio Argentino de Sistemas Embebidos 2015, el día 14 de Agosto de 2015 en la Facultad de Ingeniería de la Universidad de Buenos Aires.

Sin más, agradeciendo su atención,

Los saludamos cordialmente,

Ing. Gustavo Mercado
Coordinador Tutoriales
Internet of Things
SASE 2015

Dr. Ing. Ariel Lutenberg
Coordinador General
SASE 2015

IOT INTERNET OF THINGS

SASE 2015 SIMPOSIO ARGENTINO DE SISTEMAS EMBEBIDOS

Ciudad de Buenos Aires, 14 de agosto de 2015

A QUIÉN CORRESPONDA

Ref.: Certificado de disertante de Tutoriales del SASE 2015

De nuestra mayor consideración:

Por la presente nos dirigimos a ustedes a fin de dejar constancia que el **Ing. Martin Cortina** ha dictado el tutorial *Placa Intel Galileo: Primeros pasos* de 6 horas, en el Track *Linux Embebido* en el marco del Simposio Argentino de Sistemas Embebidos 2015, el día 14 de Agosto de 2015 en la Facultad de Ingeniería de la Universidad de Buenos Aires.

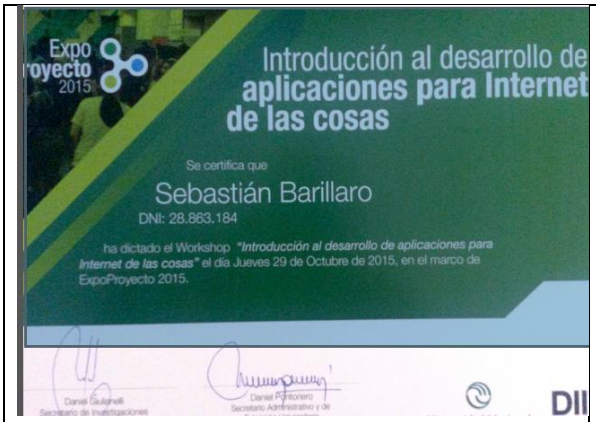
Sin más, agradeciendo su atención,

Los saludamos cordialmente,

Ing. Gustavo Mercado
Coordinador Tutoriales
Internet of Things
SASE 2015

Dr. Ing. Ariel Lutenberg
Coordinador General
SASE 2015

IOT INTERNET OF THINGS


Expo Proyecto 2015
Introducción al desarrollo de aplicaciones para Internet de las cosas
Se certifica que
Sebastián Barillaro
DNI: 28.863.184
ha dictado el Workshop "Introducción al desarrollo de aplicaciones para Internet de las cosas" el día Jueves 29 de Octubre de 2015, en el marco de ExpoProyecto 2015.


Expo Proyecto 2015
Introducción al desarrollo de aplicaciones para Internet de las cosas
Se certifica que
Graciela De Luca
DNI: 13.314.265
ha dictado el Workshop "Introducción al desarrollo de aplicaciones para Internet de las cosas" el día Jueves 29 de Octubre de 2015, en el marco de ExpoProyecto 2015.
Universidad Nacional de La Matanza
DIIT


Expo Proyecto 2015
Introducción al desarrollo de aplicaciones para Internet de las cosas
Se certifica que
Esteban Carnuccio
DNI: 30.333.327
ha dictado el Workshop "Introducción al desarrollo de aplicaciones para Internet de las cosas" el día Jueves 29 de Octubre de 2015, en el marco de ExpoProyecto 2015.
Universidad Nacional de La Matanza


Universidad Nacional de La Matanza
1989 - 2014
1ª Jornada de Innovació
Universitar
A través del presente se certifica que
Esteban Carnuccio DNI 30.333.327
ha participado en calidad de asistente a la 1ª Jornada de Innovación Universitaria de la UNL
Organizada por la Secretaría Académica, Secretaría de Ciencia y Tecnología y
Secretaría de Extensión Universitaria.
San Justo, 20 de Noviembre de
Mg. Gustavo Duek
Secretaría Académica
Mg. Ana Bidiña
Secretaria de Ciencia y Tecnología
Lic. Roberto Ayub
Secretaría de Extensión Unive

ANEXO 4: GUIAS DE INSTALACION Y CONFIGURACION

Contenido

2 SECCION A – ARMADO DEL ENTORNO DE TRABAJO

A continuación se detalla las versiones de los programas y utilitarios más antiguos con los que se sabe que se puede compilar S.O.D.I.U.M. En general, todos los programas son compatibles hacia atrás, por lo que no debería haber inconvenientes si poseen una versión posterior. La excepción del caso es el compilador (gcc) y el linkeditor (ld), que al ir evolucionando se agregan nuevas características de optimización de código, algunas de las cuales impiden la compilación o ejecución normal del S.O.D.I.U.M. De todos modos siempre dejan la opción de desactivar estas optimizaciones, que podría ser necesario de acuerdo a la compatibilidad de S.O.D.I.U.M.

Para facilitar el proceso de configuración, el script configurar.sh provisto con el código fuente ya define automáticamente los flags necesarios para compilar S.O.D.I.U.M., aunque es importante tener en cuenta que en un futuro inmediato podrían desarrollar nuevas optimizaciones, y recién una vez identificadas, se deberán agregar los parámetros que las inhabilitan al listado de CFLAGS.

Se ha registrado que la versión actual de S.O.D.I.U.M. se ha compilado satisfactoriamente utilizando las siguientes versiones de LINUX:

Ubuntu 7.04

Ubuntu 11.04

Ubuntu 12.04

Fedora Core 1, 2, 3, 4, y 5

Slackware 10

RedHat 7.3, 9

2.1. Herramientas necesarias para compilar S.O.D.I.U.M.

Las herramientas necesarias para compilar S.O.D.I.U.M. (y sus versiones que han sido probadas satisfactoriamente) son las siguientes:

Tabla 5 Herramientas necesarias

Herramienta	Versión
Kernel Linux	3.0.0-16-generic
nasm	2.09.08
mtools	3.9.1
hexdump	Cualquier Versión
gcc	4.6.1
dd	8.5
ld	2.9.1
cat	8.5
make	3.81
objcopy	2.9.1

2.2. Paquetes adicionales que son necesarios instalar

A continuación se listan paquetes adicionales que podría ser necesario instalar (dependiendo si la versión de Linux utilizada los incluye):

Tabla 6 Paquetes adicionales

build-essential	subversion	bximage	hexcat	vim-full
binutils	elfutils	glibc-doc	hexedit	samba
ctags	doxygen	console-keymaps	usbutils	xdotool
mktools	gdb(7.9)	console-utils	parted	indent
keyutils	mkisofs	elf-binutils	gpart	python(2.7)
Matplolib				

2.3. Compilación de Bochs

Bochs es un emulador de máquina virtual que puede ser descargado desde la siguiente página web:

http://Bochs.sourceforge.net/

Para probar la ejecución de S.O.D.I.U.M. sin necesidad de utilizar un dispositivo y una computadora física, se ha utilizado Bochs. Bochs permite emular una computadora de arquitectura Intel IA-32, y permite simular dispositivos de entrada (diskette, CD-ROM, disco duro) mediante el uso de imágenes de dispositivo. De esta manera, es posible probar nuevas versiones de S.O.D.I.U.M. rápidamente sin necesidad de utilizar dispositivos físicos.

Existen otros emuladores de máquinas virtuales (por ejemplo: QEMU). Sin embargo, se ha discontinuado su uso para S.O.D.I.U.M. , ya que Bochs ha demostrado ser el que mejor refleja el funcionamiento de una computadora real. Es decir, errores de configuración de bajo nivel en S.O.D.I.U.M. han fallado (correctamente) durante su ejecución en Bochs, mientras que han sido ignorados en otros emuladores. Esto da la seguridad de que si funciona en Bochs, muy probablemente funcione en una computadora real.

Nótese que Bochs es un emulador de máquina virtual, mientras que otros productos como VMware o VirtualBox utilizan directamente el hardware de la máquina huésped para ejecutar el sistema operativo a virtualizar. La diferencia entre ellos es que, en Bochs, todos los registros de CPU, BIOS, y Memoria se alojan en la RAM de la computadora huésped, y las acciones pertinentes a las instrucciones ejecutadas por los programas son emuladas sobre estos datos en RAM. Mientras que, por el contrario, VMware y VirtualBox utilizan el procesador real para ejecutar las instrucciones de sus programas.

Una de las ventajas de usar emulación (especialmente con Bochs), es que, al tener todos los registros en RAM y emular instrucciones, contamos con la posibilidad de depurar, detener, y continuar la ejecución de nuestro sistema operativo. Todas los registros del CPU son analizables y pueden ser revisados en cualquier momento.

Bochs cuenta con 3 tipos de debuggers:

- Debugger Interno (Consola)
- Debugger Interno (GUI)
- Debugger Externo (GNU Debugger)

Una vez descargado el código, se deben configurar los flags de compilación. Los comandos utilizados para compilar Bochs para depurar S.O.D.I.U.M. son los siguientes:

a. Usando Debugger Interno (Consola):

•Configurar la instalación:

./configure --enable-debugger --prefix=/usr/local/bin/Bochs --enable-debugger --enable-disasm --enable-all-optimizations --with-all-libs--enable- readline --enable-alignment-check --with-nogui --enable-show-ips --disable-plugins --

enable-pci --enable-pcidev --enable-fpu --disable-docbook --enable-pcidev --enable-usb --enable-usb-ohci --enable-usb-xhci

•Instalar Bochs:

make install

b. Usando Debugger Interno (GUI):

•Configurar la instalación:

./configure --enable-debugger --prefix=/usr/local/bin/Bochs --enable-debugger-gui --enable-disasm --enable-all-optimizations --with-all-libs --enable-readline --enable-alignment-check --with-nogui --enable-show-ips --disable-plugins --enable-pci --enable-pcidev --enable-fpu --disable-docbook --enable-pcidev --enable-usb --enable-usb-ohci --enable-usb-xhci

•Instalar Bochs:

make install

c. Usando Debugger Externo (GNU Debugger):

•Configurar la instalación:

./configure --enable-debugger --prefix=/usr/local/bin/Bochs --enable-gdb-stub --enable-disasm --enable-all-optimizations --with-all-libs --enable-readline --enable-alignment-check --with-nogui --enable-show-ips --disable-plugins --enable-pci --enable-pcidev --enable-fpu --disable-docbook --enable-pcidev --enable-usb --enable-usb-ohci --enable-usb-xhci

•Instalar Bochs:

make install

Además de lo anteriormente detallado se deberán instalar el depurador GDB versión 7.9 siguiendo los pasos explicitados en la **Sección A,** como así también la biblioteca Matplotlib siguiendo los pasos explicados en la **Sección F.**

3 SECCION B – INSTRUCCIONES DE USO DEL S.O.D.I.U.M. DEVKIT

S.O.D.I.U.M. Developer Kit v2015 está construido sobre una instalación de Ubuntu 11.10, para distribuir a los alumnos en forma de VMWare Virtual Machine. Se recomienda el uso de una máquina huésped con los siguientes aspectos técnicos: Más

de 1 GB de RAM. Procesador con 2 o más núcleos y soporte de virtualización por hardware.

3.1. – Instalación de VMWaplayer

Para ejecutar la máquina virtual que contiene al SDK, se debe instalar el VMware Player para Windows o Linux. Este programa es gratuito y permite ejecutar/crear cualquier máquina virtual de VMware. Sin embargo, no es un software de libre distribución por lo que debe ser descargado desde la página de VMware aceptando los términos de la licencia.

1-Ingresar a:
https://my.vmware.com/en/web/vmware/free#desktop_end_user_computing/vmware_workstation_player/12_0

2- Registrarse, aceptar los términos de la licencia, y continuar hacia la página de descarga.

3- Descargar el instalador adecuado para el sistema operativo huésped, e instalar el programa .Reiniciar el sistema si es requerido.

3.2. – Instalación de VMWaplayer

Copie el archivo Sodium.DevKit.rar a su disco rígido y descomprímalo en el directorio de su elección con cualquier herramienta compatible con archivos .rar (WinRAR, por ejemplo).

Una vez extraídos, los archivos resultantes se verán así:

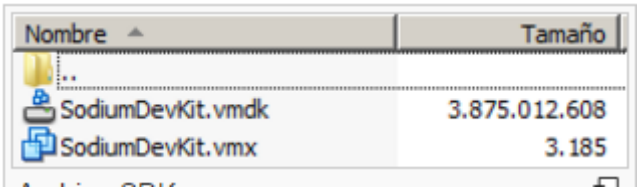


Figura 1 Archivo SDK

3.3. – Importando la máquina virtual SDK

Para importar la imagen recién extraída la imagen del SDK, es necesario importarla a la biblioteca del VMware Player. Puede hacerlo seleccionando la opción "Open a Virtual Machine", como en la siguiente imagen:

Figura 2 Abrir Máquina Virtual

3.4. – Ejecutar Suspender y Apagar el SDK

Para iniciar la máquina virtual con el SodiumDeveloper Kit instalado, basta con seleccionarla desde la biblioteca del VMware Player, y seleccionar la opción "Play Virtual Machine":

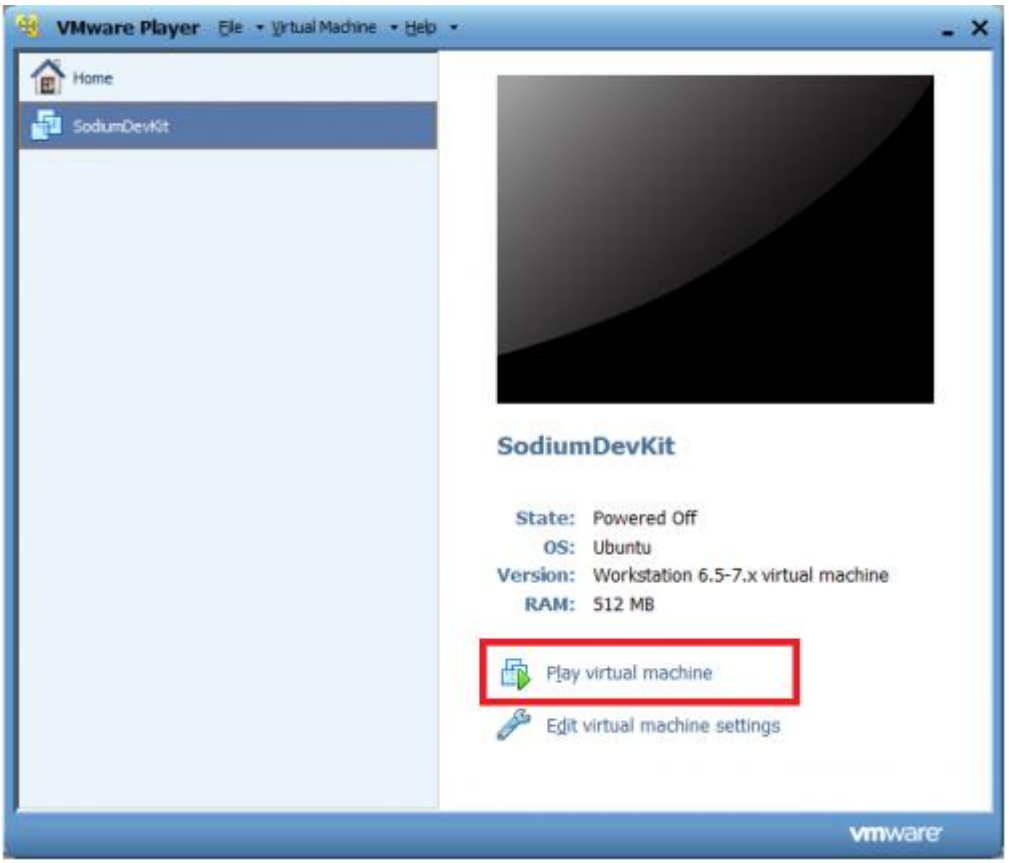


Figura 3 Ejecutar Máquina Virtual

Una vez iniciada, esperar a que bootee el sistema operativo Ubuntu y se muestre la imagen del SDK:

Figura 4 Pantalla principal del SDK

Todas las contraseñas configuradas en SDK están definidas como “S.O.D.I.U.M.”. Una vez finalizado el trabajo del día, la máquina virtual puede suspenderse (recomendado) en el estado en el que esté y ser reanudada en cualquier otro momento sin necesidad de apagarla. Para hacer esto, debe seleccionarse la opción "Suspend" del menú contextual:

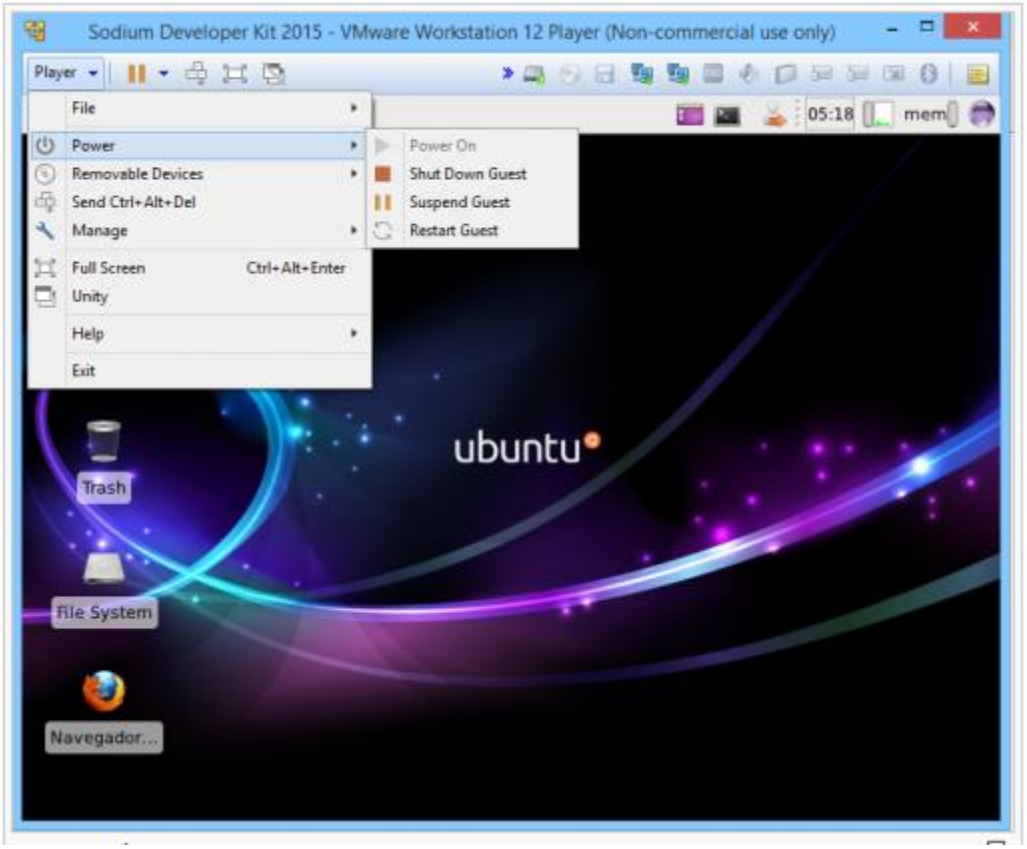


Figura 5 Suspensión de la Máquina Virtual

Si por alguna razón es necesario apagar la máquina virtual (no recomendado), puede hacerse utilizando el botón de apagado dentro de SDK

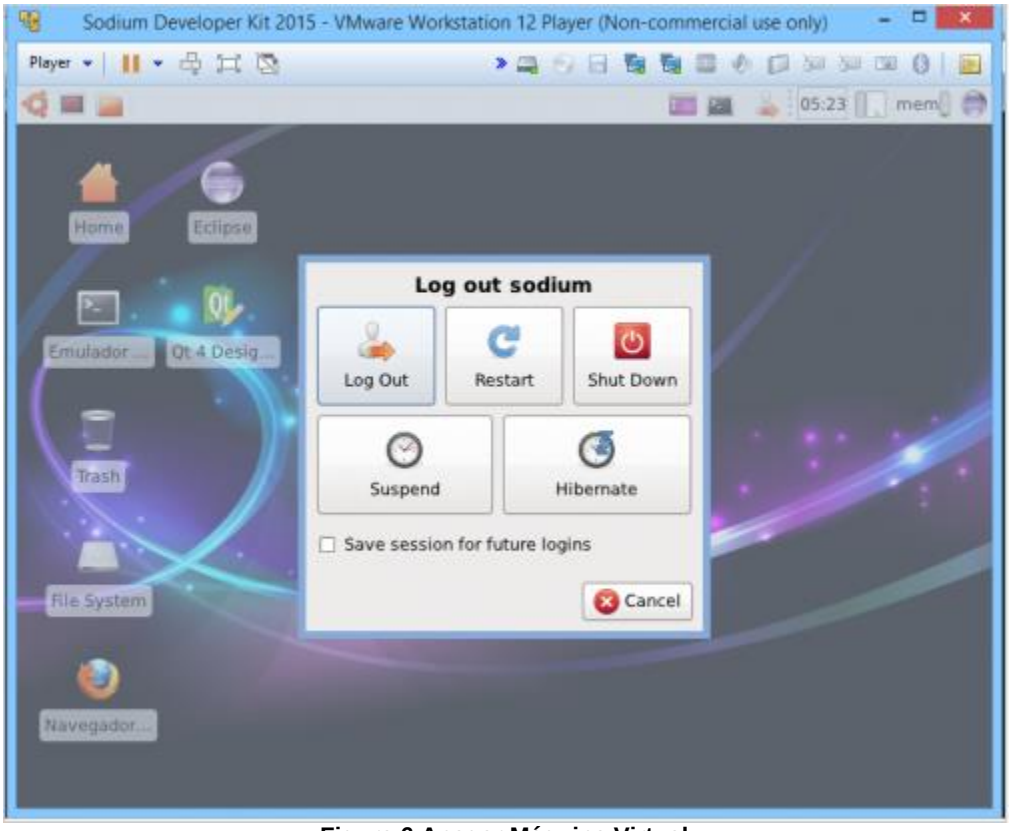


Figura 6 Apagar Máquina Virtual

3.5. – Configuración de Dispositvo Floppy

Para contar con un dispositivo de tipo floppy disk en nuestra máquina virtual, podemos crear una imagen virtual de disquete en un archivo, o utilizar la disquetera física de la computadora huésped si es que cuenta con una. Como primer paso para ambas posibilidades, hay que abrir el menú de configuración de Floppy Disk:

- Utilizar un disquete virtual:

Para utilizar un disquete virtual, se debe seleccionar la opción "Use floppy image file", y luego presionar el botón "Create". Esto creará una archivo (con nombre elegido por el usuario) sobre el cual se simulará una imagen de floppy:

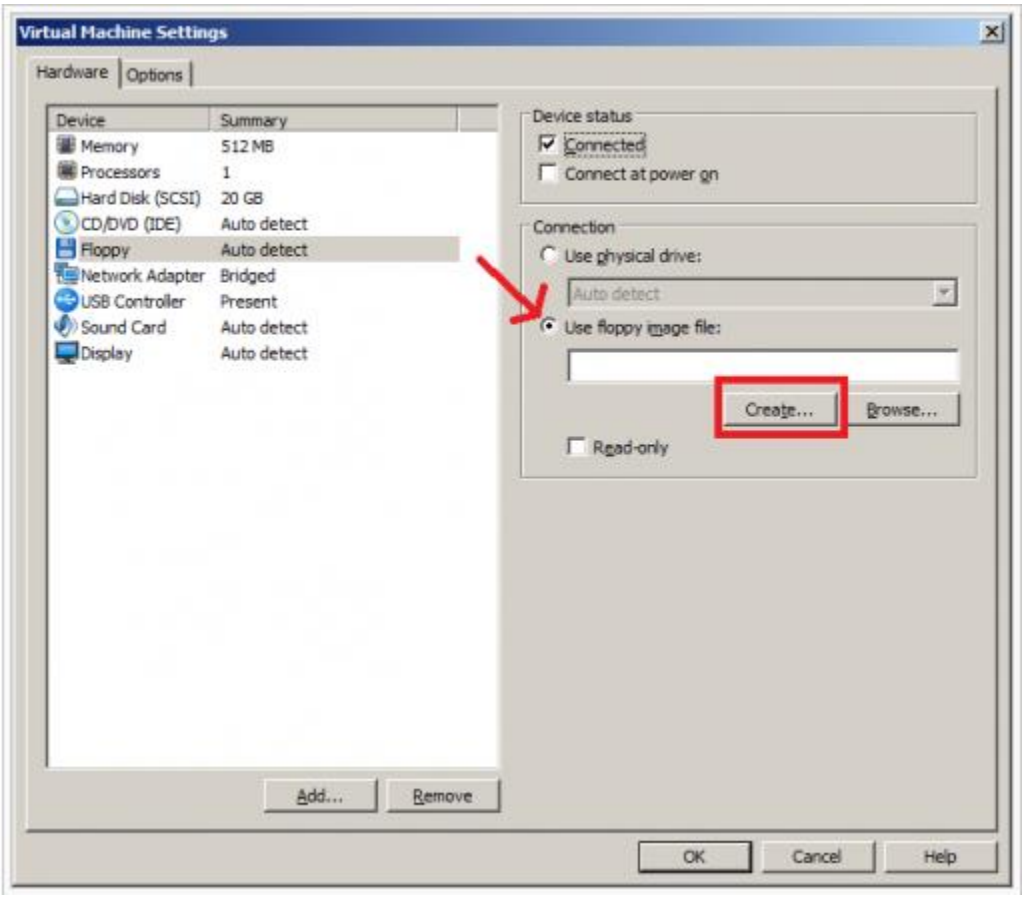


Figura 7 - Disquetera Virtual

Luego de asegurarse de que el tilde de "Connected" está marcado, se presiona OK, y ya puede utilizarse el floppy dentro de la máquina virtual.

- Utilizar la disquetera física de la computadora huésped:

Para lograr esto, debe seleccionarse la opción "Use physical drive", e indicar la unidad de disquete, o dejar "Auto Detect" para dejar a VMware elegirla automáticamente

3.6. – Configuración de Dispositvo USB

Para utilizar un dispositivo de tipo pendrive USB en la máquina virtual, basta con conectarlo a la máquina huésped. Luego de conectarlo, pueden surgir dos posibilidades:

- El pendrive se conectó mientras se estaba utilizando la máquina virtual, se conectará directamente a ésta, y no hará falta ningún otro paso.

3.7. Conectando SDK a Internet

La máquina virtual de SDK está configurada para conectarse automáticamente a internet, sólo si la máquina huésped también tiene acceso a internet. Por lo tanto, para conectar a internet desde SDK, sólo basta con tener internet en la máquina huésped.

SDK cuenta con dos dispositivos de red virtuales. Una pertenece a una red Host-Only -solo huésped- (su función se explica en el siguiente punto), y la segunda pertenece a una red Bridged -tipo puente-. El tipo de conexión bridged permite al dispositivo de red virtual conectarse al default gateway - por lo general, un router - de la red física utilizando el dispositivo de red físico de la máquina huésped. Al mismo tiempo replica el estado de conexión del dispositivo físico, por lo que si la máquina huésped está conectada a internet, SDK también estará conectado automáticamente.

3.8. Compartiendo Archivos entre SDK y la Máquina Virtual Huésped

SDK viene configurado para conectarse a una red Host-Only con la máquina huésped. En esta red virtual, solo existen dos hosts: el Ubuntu del SDK, y la máquina huésped. Por lo tanto, es posible conectarse directamente con la máquina virtual.

A su vez, también es posible acceder desde la máquina virtual hacia un directorio compartido de la máquina huésped. Este último método permite por ejemplo utilizar un IDE que requiera muchos recursos del sistema como Eclipse o Netbeans y compilar y probar S.O.D.I.U.M. desde la máquina virtual.

3.8.1. Método 1 - Accediendo a SDK desde la máquina huésped

Dado que SDK viene con un servidor Samba configurado, es posible acceder al sistema de archivos de SDK desde la máquina huésped. El siguiente es un ejemplo de cómo acceder:

1 - Abrir la Consola y ejecutar el comando "*ifconfig"* para obtener los números de IP de los dispositivos virtuales:

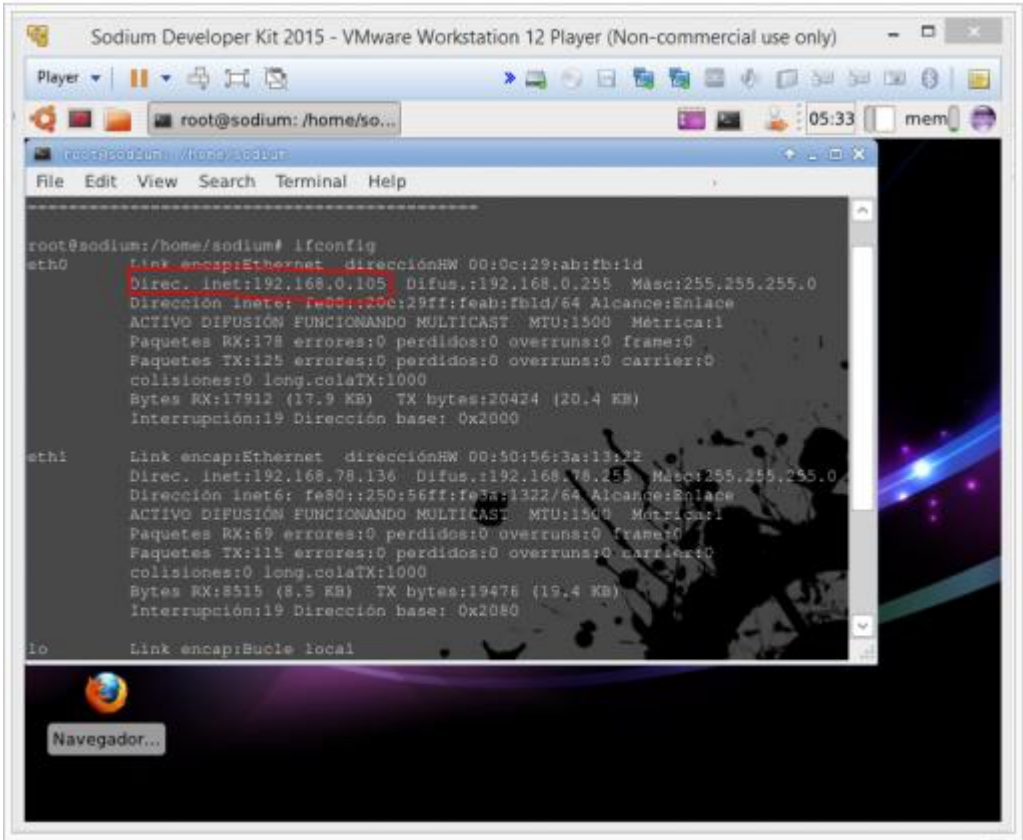


Figura 8 IP de la Máquina Virtual

2 - Seleccionar el IP del dispositivo virtual Host-Only (por defecto eth0) e ingresarlo en la barra de dirección del explorador del sistema operativo huésped:

En este punto se puede copiar, mover, eliminar, y navegar entre los archivos y directorios compartidos del SDK como si fuera cualquier otra carpeta, e incluso pueden abrirse los archivos de código utilizando algún programa de edición:

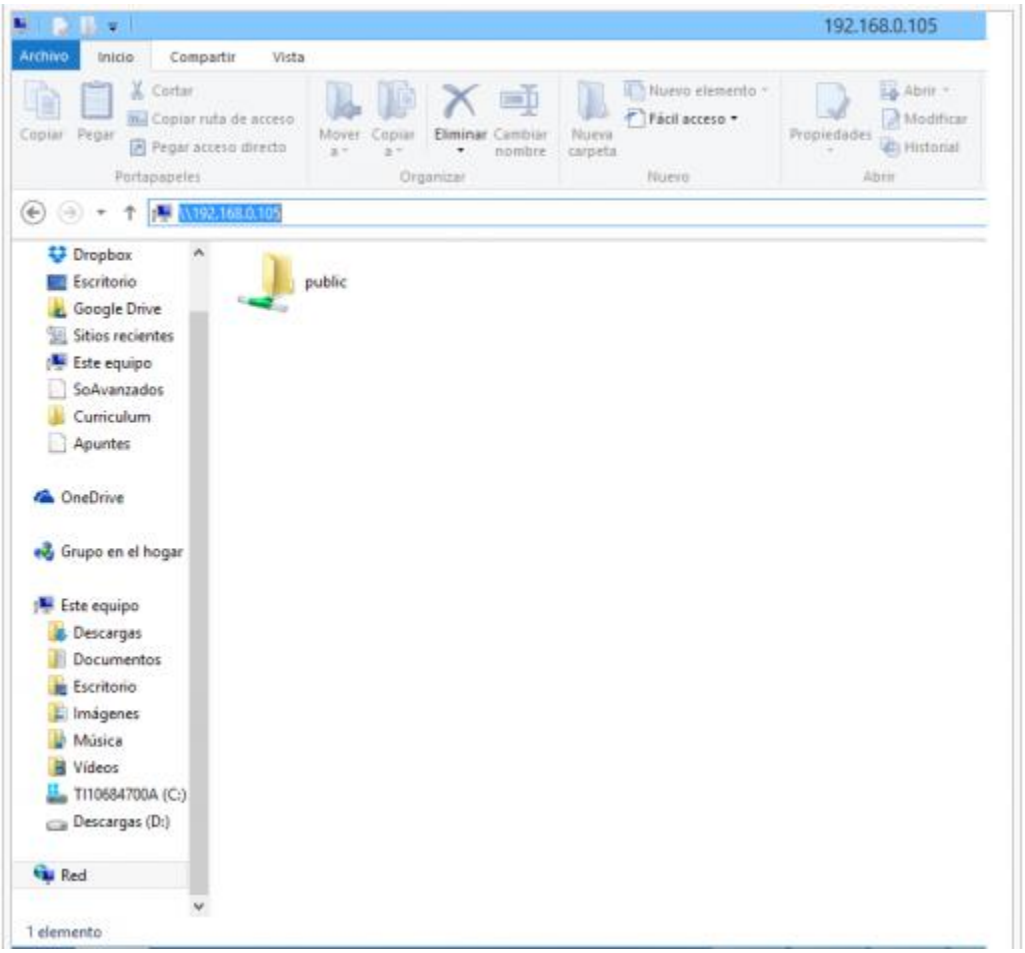


Figura 9 Acceso a Máquina Virtual por IP

3.8.2. Método 2 - Accediendo a la máquina huésped desde SDK

En caso de tener correctamente instalado el paquete vmware-tools en la máquina virtual, es posible configurar una serie de directorios de la máquina host para que sean accesibles desde la máquina virtual.

Para el caso puntual de una máquina virtual con las vmware tools de linux instaladas, estos directorios compartidos serán visibles desde el punto de montaje /mnt/hgfs.

Por ejemplo, un directorio de la máquina host windows D:\workspace se podría mapear a /mnt/hgfs/workspace en el SDK.

En el ejemplo ilustrado a continuación, se comparte el directorio:

/Informacion/Universidad/SO del host linux en el directorio /mnt/hgfs/SO de la máquina virtual

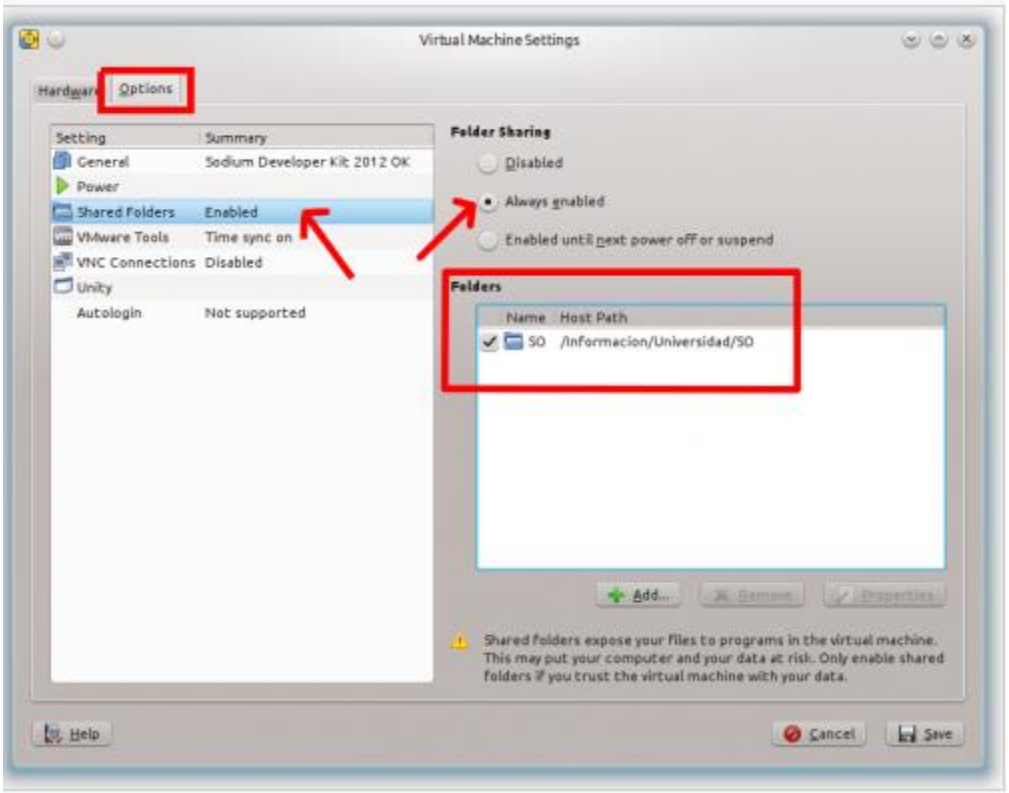


Figura 10 Compartición de directorio

Desde el SDK, se ve que este directorio se encuentra directamente montado en la estructura VFS, lo que resulta más conveniente para trabajar que si debiéramos acceder a los archivos mediante una unidad de red. Incluso aunque los archivos se almacenen remotamente, a efectos prácticos el acceso es local, permitiendo abrir un terminal y compilar S.O.D.I.U.M. sin ningún problema.

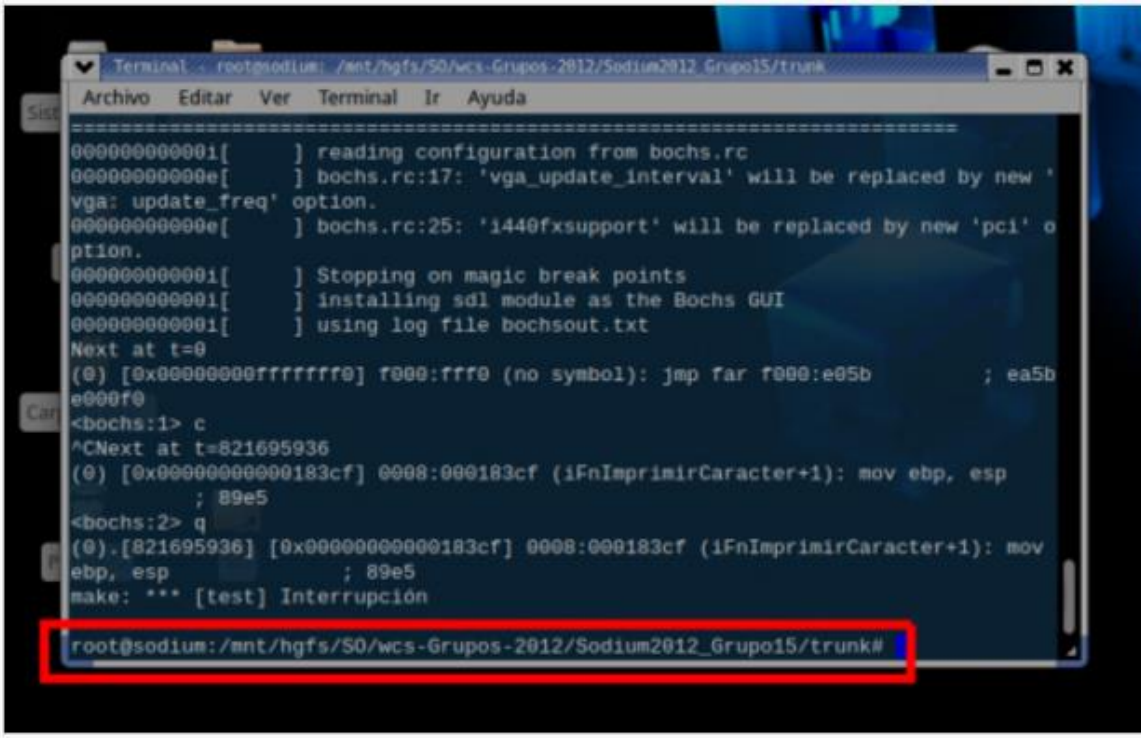


Figura 11 Acceso a compartición

Por lo tanto, mientras en el SDK podemos compilar y probar S.O.D.I.U.M., en la máquina host podemos utilizar un IDE de gran escala como Eclipse

4 SECCIÓN C – INSTRUCTIVO DE INSTALACIÓN DE GDB 7.9 EN LA IMAGEN DE LINUX DENTRO DEL S.O.D.I.U.M.-DEVKIT

I. Instalación de Python 2.7

- Instalar desde Sinaptics los ejecutables, librerías y documentación de Python 2.7

II. Descarga de GDB 7.9

- Descargar y descomprimir el código fuente de GDB 7.9

III. Instalación de GDB:

- Entrar en el directorio de gdb y ejecutar lo siguiente:

  ./configure --target=i386-elf --disable-werror --program-prefix=i386-elf- --with-python=/usr/bin/python

- Luego ejecutar:

  make

sudo make install

- En el caso de que el make diga que no encuentra Python se deberá instalar sus librerías de la siguiente forma

sudo apt-get install python2.7-dev

5 SECCIÓN D – INSTRUCTIVO DE INSTALACIÓN DE GDB 7.9 EN EL S.O WINDOWS

I. Instalación de MINGW

- Para poder utilizar el depurador en este Sistema Operativo fue necesario instalar MINGW, la implementación del compilador GCC para esta plataforma. Además se instalaron todos sus paquetes base, especialmente msys.

II. Descarga de GDB 7.9

- Descargar y descomprimir el código fuente de GDB 7.9, dentro del subdirectorio home ubicado en el directorio de instalación de Mingw.

III. Instalación de GDB:

- Ejecutar la consola de Mingw, que simula la terminal de Linux.
- Luego, dentro la terminal de Mingw., ingresar al subdirectorio donde se descomprimió el fuente de gdb e ingresar el siguiente comando: *./configure --target=i386-elf --disable-werror --program-prefix=i386-elf- --with-python=/usr/bin/Python*
    - Luego ejecutar:

make

sudo make install

6 SECCIÓN E – PASOS PARA INSTALAR PYCLEWN EN LINUX

I. Instalar Pip:

- Sudo apt-get install Python-pip

II. Actualizar Pip:

- pip install -U pip

III. Actualizar seguridad Pip:

- pip install requests[security]
- sudo apt-get install python-dev libffi-dev libssl-dev

IV. Instalar pdb-clone:

- Descargar el archivo pdb-clone-1.9.2.py2.7.tar.gz
- **Descomprimirlo**: tar xzf Plugins/pdb-clone/pdb-clone-1.9.2.py2.7.tar.gz
- **Instalarlo**: python setup.py install

V. Instalar trollius:

- **Descargar el fuente: hg clone** 'https://bitbucket.org/enovance/trollius#trollius'
- **Dentro del directorio descargado instalar trollius:**
  python setup.py install

VI. Instalar Pyclewn

- **Descargar el fuente:** pyclewn-2.1.tar.gz
- **Descomprimirlo:** tar xzf pyclewn-2.1.tar.gz
- **Dentro del directorio descargado instalar Pyclewn:** python setup.py install

- **Ingresar al directorio lib/clewn/runtime y ejecutar el siguiente comando:**
  vim -C pyclewn-2.1.vmb

- **Dentro del vim ejecutar lo siguiente** :so%
- **Salir del vim :**q! y ya estará instalado Pyclewn

VII. Ejecutar Pyclewn

- **Abrir vim y ejecutar** :Pyclewn

7 SECCION F – INSTRUCTIVO DE MATPLOTLIB EN LA IMAGEN DE LINUX DENTRO DEL S.O.D.I.U.M.-DEVKIT

I. Instalación de dependencias de Matplotlib

- Descargar todas las dependencias de Matplotlib con el siguiente comando:

sudo apt-get build-dep python-matplotlib

II. Descarga Matplotlib:

- Descargar la versión 1.3.1 de Matplotlib del siguiente sitio web: *https://github.com/cartu32/matplotlib/archive/1.3.1.tar.gz*

III. Instalación de Matplotlib:

- Descomprimir en Linux el archivo tar.gz descargado en el punto anterior con el siguiente comando:
  tar -xvfz matplotlib-1.3.1.tar.gz

- Ingresar al directorio donde descomprimió el archivo :
  cd matplotlib

- Instalar Matplotlib con el siguiente comando:
  python setup.py install

iv. **Verificación de instalación**

- Con el siguiente comando se comprueba la correcta instalación de Matplotlib:

  python2.7 -c 'import matplotlib; print matplotlib.__version__, matplotlib.__file__'

  Si luego de haber ejecutado el comando la salida por pantalla es similar a la siguiente, entonces se ha instalado correctamente Matplotlib

  ./Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/matplotlib/__init__.pyc

8 SECCION G – CONEXIÓN CON GDB DESDE S.O.D.I.U.M.-DEVKIT

III. Compilar S.O.D.I.U.M.

- Utilizar el siguiente comando desde el directorio raíz donde se encuentre el código fuente de S.O.D.I.U.M.:
  make clean test

IV. Ejecutar GDB:

- Desde la consola de Linux ejecutar el comando:

gdb

IV. Establecer la conexión S.O.D.I.U.M.:

Una vez que se abra la consola de gdb, se deberán ejecutar los siguientes comandos del depurador

- Cargar la tabla de símbolos del S.O :

file S.O.D.I.U.M./kernel/main.ld

- Establecer la velocidad de transferencia de datos utilizadas en el puerto serie:

set serial baud <velocidad en baudios>

- Establecer la conexión remota con S.O.D.I.U.M. a través del puerto 12345:

target remote <ip>:12345

9 SECCION H – CONEXIÓN CON GDB DESDE WINDOWS

Es necesario realizar los mismos pasos que en S.O.D.I.U.M.-Devkit, pero con algunas diferencias.

I. Compilar S.O.D.I.U.M.

- Utilizar el siguiente comando desde el directorio raíz donde se encuentre el código fuente de S.O.D.I.U.M.:
  make clean test

II. Copiar S.O.D.I.U.M.

- Una vez abierta la consola de Bochs, se debe copiar todo el código fuente de S.O.D.I.U.M., junto a sus archivos objetos, dentro de un directorio de Windows.

III. Ejecutar GDB:

- Luego ir al directorio home de MINGW donde está instalado gdb y ejecutarlo:
  Gdb

V. Establecer la conexión con S.O.D.I.U.M.:

Una vez que se abra la consola de gdb, ejecutar los siguientes comandos del depurador

- Cargar la tabla de símbolos del S.O, de los archivos de S.O.D.I.U.M. que fueron copiados en Windows :
  file S.O.D.I.U.M./kernel/main.ld

- Establecer la velocidad de transferencia de datos utilizadas en el puerto serie:
  set serial baud <velocidad en baudios>

- Luego con el siguiente comando, se debe indicarle a GDB cuál son los directorios de Windows en donde se encuentra el código de S.O.D.I.U.M.:
  Directory fuente/kenel

  Directory fuente/kenel/stub

  Directory fuente/common

  - Finalmente establecer la conexión remota con S.O.D.I.U.M. a través del puerto 12345
    target remote <ip>:12345

ANEXO 5: Código fuente del Visualizador del Sistema Operativo S.O.D.I.U.M.

Contenido

1. Estructura de los directorios que componen el código fuente del Sistema Visualizador

El código fuente del Sistema Operativo S.O.D.I.U.M y del Visualizador, ya se encuentra almacenado dentro de la imagen de la máquina virtual Sodium-Devkit que se puede descargar desde:

http://www.so-unlam.com.ar/wiki/index.php?title=PUBLICO:Sodium_Developer_Kit

Dentro del SDK, los archivos fuentes del proyecto se encuentran en el siguiente directorio de Linux:

/home/Sodium/Sodium-Visualizador.

El código fuente también puede ser descargado de la siguiente dirección web.

http://www.so-unlam.com.ar/wiki/index.php?title=PUBLICO:C%C3%B3digo_Fuente

El código fuente que se entrega junto con el presente documento, se encuentra organizado en distintos directorios. Cada uno de ellos cumple una funcionalidad vital en el sistema que se ha desarrollado en este proyecto. Por consiguiente, en los siguientes apartados se describen las funcionalidades de cada subdirectorio, como así también la de sus archivos más importantes.

En el directorio raíz del proyecto, se encuentran los siguientes subdirectorios:

Figura 12 Directorio raíz del Proyecto

2. Contenido del Directorio Sodium

El subdirectorio Sodium contiene la versión del código fuente de ese Sistema Operativo, configurado para funcionar con el Visualizador de estructuras desarrollado. Por otra parte dentro del subdirectorio Visualizador se encuentran los archivos Python que conforman el código del propio graficador.

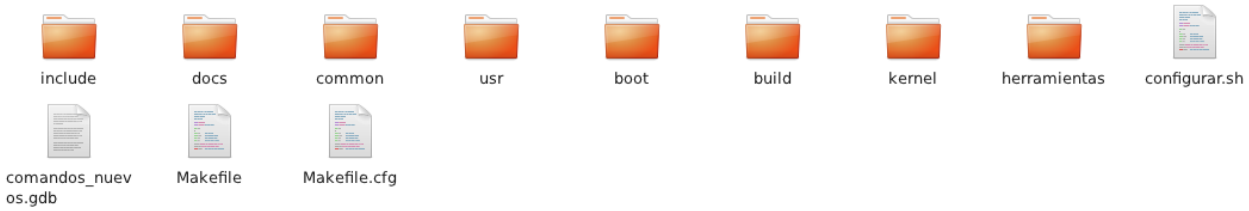


Figura 13 Contenido del subdirectorio Sodium

Como se observa en la figura anterior, el contenido del subdirectorio Sodium está compuesto por los siguientes directorios:

- **Kernel:** Contiene todos los archivos fuentes escrito en lenguaje C y Assembler pertenecientes al kernel del Sistema Operativo S.O.D.I.U.M. En este subdirectorio, se encuentran los códigos de drivers, planificadores, administradores de memoria, files system, handler de interrupciones, archivo stub, entre otros; que forman parte del núcleo del Sistema Operativo.
- **Usuario:** En este directorio se almacenan los códigos fuentes de los programas ejecutables desarrollados por los usuarios para ser utilizados desde la consola de S.O.D.I.U.M. Al ser programas de usuario, los fuentes deberán respetar los niveles de seguridad impuestos por este Sistema Operativo. Por consiguiente, muchas de estas aplicaciones deberán usar llamadas al sistema para su correcto funcionamiento.
- **include:** Contiene los archivos cabecera de los códigos fuentes pertenecientes al Kernel y a los programas de usuario.
- **common:** Almacena código que tienen en común los archivos fuentes del kernel y del usuario.
- **herramientas:** Subdirectorio que posee scripts de configuración del entorno de ejecución del Sistema Operativo. La mayoría de estos archivos son utilizados durante la compilación de S.O.D.I.U.M.
- **boot:** Su contenido utilizado únicamente en Modo Real. Contiene código Assembler, que al iniciar la PC es invocado por el BIOS para poder inicializar y ejecutar el Sistema Operativo.

Cómo se puede observar en la figura anterior, en este subdirectorio se encuentra el archivo *"comandos_nuevos.gdb",* que fue explicado en el Informe Principal del Proyecto.

3. Formato de las Estructuras IDT, GDT, TSS y PCB del S.O S.O.D.I.U.M

3.1. IDT

En S.O.D.I.U.M la estructura de la tabla IDT es la siguiente:

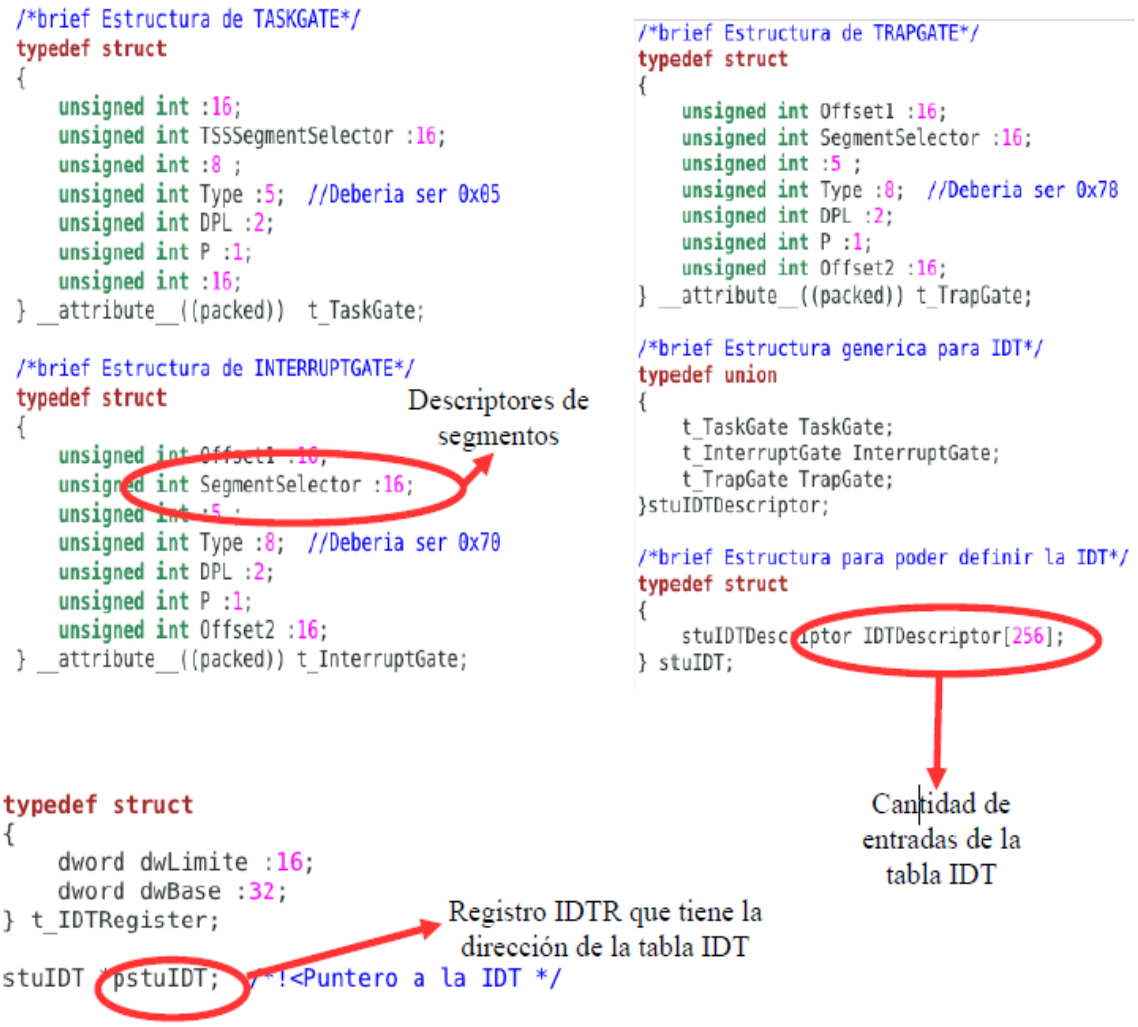


Figura 14 Formato de la estructura IDT

3.2. GDT:

La tabla tiene 8192 (máximo) entradas, cada una de 8 bytes:

```
typedef struct {
   stuGDTDescriptor stuGdtDescriptorDescs[8192]; /*!< Vector de posiciones de la GDT*/
} stuEstructuraGdt;

stuEstructuraGdt *pstuTablaGdt; /*!< puntero a la GDT */
```

Figura 15 Formato de la estrucutra GDT

La estructura tiene el siguiente formato:

```
typedef struct {
  unsigned short usLimiteBajo;    /*!< limite bajo 0..15   */
  unsigned short usBaseBajo;      /*!< base bajo 0..15     */
  unsigned char ucBaseMedio;      /*!< base media 16..23   */
        /**
                (Type es 1 byte (bits 8-11) indica si el descriptor es de codigo, de datos o de sistema.
                Bit 'S' que indica con cero, si el descriptor es de sistema o, con uno, si es de codigo
                datos.
                DPL  Define nivel de previlegio del segmento.
                Bit 'P' indica si el segmento está en memoria principal o no.
        */

  unsigned char ucAcesso;         /*! byte de acceso*/

  unsigned int bitLimiteAlto:4;    /*!< limite alto 16..19   */

        /**
                La estructura de un descriptor de segmento es:
                Bit 20: AVL, Disponible para el uso del software del sistema.
                Bit 21: deberia ser siempre cero.
                Bit 22: D/B especifica distintas opciones segun sea el segmento de codigo, de stack o de
                datos.
                Bit 23: G Bandera que indica la granularidad. Determina si el limite del
                segmento se especifica de a un Byte o de a 4 KByte.
        */

  unsigned int bitGranularidad:4;

  unsigned char usBaseAlto;        /*!< base alto24..31    */

} NOALIGN stuGDTDescriptor;
```

Figura 16-Formato de la estructura GDT

3.3. PCB:

La estructura del PCB en el S.O es la siguiente:

```
    typedef struct _stuPCB_ {
        unsigned long int ulId; /*!< id del proceso */
            unsigned int iPrivilegio;
            unsigned long ulTiempoListo; /*!< tiempo en que el proceso pasa al estado de listo */
        unsigned int uiIndiceGDT_CS; /*!< indice del descriptor de segmento de codigo de este proc en la GDT
        unsigned int uiIndiceGDT_DS; /*!< indice del descriptor de segmento de datos de este proc en la GDT
            unsigned int uiIndiceGDT_ES; /*!< indice del descriptor de segmento de codigo de la
    bibliotecadinamica en la GDT */
        unsigned int uiIndiceGDT_TSS; /*!< indice de la TSS de este proc en la GDT */
        unsigned int uiIndiceGDT_SS0; /*!< indice del descriptor del segmento de stack ss0 de este proc en l
    GDT */
        void (* vFnFuncion) (); /*!< puntero a funcion (proceso) */

        unsigned long ulParentId; /*!< id del padre */
        unsigned long ulUsuarioId; /*!< id del usuario */
        int iPrioridad; /*!< para futuro uso */
        int iEstado; /*!< ver estados mas arriba (PROC_XXX) */
        unsigned long lNHijos;
        int iExitStatus;
        unsigned long ulLugarTSS;
        char stNombre[25];
        unsigned int uiTamProc;
        struct stuTablaPagina * pstuTablaPaginacion;
        unsigned int uiDirBase,

                uiLimite;
        stuMemoriasAtachadas memoriasAtachadas[MAXSHMEMPORPROCESO];

        //agregado
        unsigned long ulTiempoEspera; /*!< Tiempos para algoritmos tipo HRN */
        unsigned long ulTiempoServicio; /*!< Tiempo de servicio del proceso */
        //fin agregado
```

```
    /*!< Estructura usada para guardar los tiempos de proceso usados para la syscall "times"*/
    struct _stuPCB_ * pstuPcbSiguiente;
    /*!< Array con los contadores utilizados como temporizadores para la SC "times"*/
    tms stuTmsTiemposProceso;
    /*!< Tiempo durante el cual el proceso tiene que permanecer "dormido" */
    itimerval timers[3];
    /*!< Puntero a donde se guardara el resto del tiempo que debia estar detenido el proceso y no l
    long lNanosleep;
    /*!< Indica si el proceso esta siendo rastreado.*/
    timespec *puRestoDelNanosleep;
    /*!< Indica si el proceso esta siendo rastreado.*/
    long lPidTracer;
    unsigned int uiTamanioTexto; /*!< Bytes de codigo ejecutable */
    unsigned int uiTamanioDatosInicializados; /*!< Bytes de datos globales inicializados (incluye E
final)*/
    unsigned int uiTamanioStack; /*!< Bytes de stack (libres+usados) */

    unsigned int uiTamanioOverhead; /*!< Bytes de Overhead (frag. interna)*/

    unsigned int uiEsperaTeclado; //Indica que el proceso está esperando una entrada de teclado.
    unsigned int uiEsperaNet; //Indica que el proceso está esperando una entrada/salida de red.
    stuHeaderELF stuCabeceraELF;
    stuEsperarMemoria stuEsperaMemoria;
    unsigned int uiBaseSs0;
    unsigned int uiPosStack;
    unsigned int uiEsperaIO; //Indica que el proceso está esperando una entrada de

        stProfile Profile;
        int iTeclaRead;
        int IOState;
        int NetState;
} stuPCB;
```

Figura 17 Formato de la estructura PCB

4. Contenido del Directorio Visualizador

Dentro del subdirectorio Visualizador se encuentran el siguiente contenido:

Figura 18 Contenido del subdirectorio Visualizador

El subdirectorio Hitos contiene todas las versiones intermedias del Visualizador que fueron obtenidas durante el desarrollo del proyecto. En cambio en el Subdirectorio Version_Entregable posee el código fuente final del Visualizador.

5. Contenido del Directorio Version_Entregable,

Este subdirectorio está estructurado de la siguiente manera:

Figura 19 Contenido del directorio Versión_Entregable

Como se observa en la figura anterior, en este subdirectorio se encuentran los archivos finales del Visualizador. Cuyo contenido es el siguiente:

- **QTgui:** Contiene los archivos.ui de las interfaces GUI que fueron desarrolladas con QtDesigner. Aquí no se guarda su código fuente, ya que únicamente en este lugar se almacenan sus diseños
- **VentanasGUI:** En este subdirectorio se almacenan los archivos con código Python, perteneciente a las interfaces GUI que fueron diseñadas con QtDesigner.
- **Clases:** Subdirectorio que posee los archivos con las clases principales del visualizador. En este sitio se encuentran los archivos GDBMI.py, GDB.py y los que contiene las acciones de cada interfaz gráfica.
- **Configuración:** Contiene los archivos XML que indican donde colocar los Puntos de Instrumentación dentro del código fuente de S.O.D.I.U.M.
- **VisorGant:** Directorio que contiene todos los archivos fuentes referidos al Diagrama Temporal del Planificador.

5.1. Archivo Main.py:

- **Descripción:** Archivo que contiene la función *Main* de todo el visualizador. Es el archivo principal del graficador, dado que es el encargado de invocar a los demás módulos que componen el Visualizador .Una vez que se ha cargado la imagen de S.O.D.I.U.M en el emulador Bochs, se ejecuta este archivo en el script Sodium.sh.
- **Ubicación del archivo:**
  /Visualizador/main.py

5.2. Archivo consolaGdb_Codigo.py:

- **Descripción:** Archivo que contiene la Clase de la Interfaz GUI encargada de recibir los comandos GDB que ingresa el usuario en forma manual
- **Clases definidas:**
  - **ConsolaGdb:** Clase que hereda de la interfaz GUI generado por QT y maneja todos los controles de dicha ventana gráfica. Posee métodos que utilizan señales para enviar los comandos a ejecutar en GDB a través de GDBMI y posteriormente recibir el resultado de dicha ejecución.
  - **Principales métodos definidos:**
    a. **generarConnect**: Método que conecta (enlaza) cada señal con el método correspondiente que la va capturar (slots). En esta clase particular, se utiliza el concepto de mapa de señales. Dado que varios elementos (botones, etc.) generaran señales que se asocian al mismo slot (enviarComandoGdbmi), son señales internas de la clase**.**

b. **enviarComandoGdbmi**: Método que se invoca por medio de una señal, para poder enviar a GDBMI comandos para que sean ejecutados por GDB. Este método recibe como parámetro el índice del elemento que emitió la señal. El cual concuerda con el índice correspondiente a la lista self.items
c. **habilitarBotonesConsola:** Método que se invoca para habilitar y deshabilitar los botones de la Interfaz GUI pertenecientes a la ConsolaGDB
d. **enviarSigintGdbmi:** Método que genera una señal a SIGINT a GDBMI para detener la ejecución de GDB. En otras palabras, se está haciendo lo mismo que cuando en la consola de GDB se presiona CTRL+C, Método encargado de mostrar por pantalla la Interfaz VerIDT
e. **capturarPtoInstrumentacion:** Método que invoca GDBMI, utilizando una señal, cuando se ejecuta un breakpoint (punto de instrumentación) definido en el archivo breakpoint.xml.
f. **generarScriptConexionSo:** Método donde se arma el comando que se utilizara para conectar el visualizador con el Sistema Operativo S.O.D.I.U.M

- **Ubicación del archivo:**
  /Visualizador/Clases/consolaGdb_Codigo.py

5.3. Archivo Gdb.py:

- **Descripción:** contiene la clase que se encarga de ejecutar en segundo plano el depurador, usando la clase ***popen.*** Gracias a que dentro de su constructor se crea un nuevo proceso que es el encargado de ejecutar el depurador en una pseudo-terminal, de la misma forma que si se estuviera realizando desde la consola de Linux.
- **Clases definidas:**
  - **Gdb:** Clase que ejecuta GDB y sus comandos en Linux a través de subprocess
  - **Principales Métodos definidos:**
    a. **signal_handler:** Handler que captura la señal SIGINT.
    b. **pushSalida**: Método que pone en la cola cada línea de la salida que emite GDB
    c. **popSalida:** Método que retorna una sola línea de la salida que emitió GDB
    d. **enviarComandoGdb:** Método para enviar a ejecutar comandos a GDB
    e. **poll:** Método pool que espera hasta que finalice la ejecución de GDB desde la consola
- **Ubicación del archivo:**
  /Visualizador/Clases/Gdb.py

5.4. Archivo Gdbmi.py:

- **Descripción:** almacena la clase Gdbmi, cuya tarea es hacer de intermediario entre la clase GDB y cada una de las interfaces GUI. Su función principal es actuar como un distribuidor de mensajes. Enviándole a la clase Gdb las peticiones de comandos para ser ejecutados en el depurador, que le solicita una determinada interfaz gráfica.
- **Clases definidas:**
  - **Gdbmi:** Clase GDBMI, que hace de intermediario entre GDB y las interfaces GUI. Su tarea principal es determinar en qué momento una determinada interfaz podrá enviar comandos a GDB. Para eso utiliza un mecanismo de posta.

    Para poder comunicar el resultado de GDB a las interfaces GUI se utilizan tres tipos de señales.

    a. **Una específica:** Que es enviada en forma de callback, cuando la interfaz GUI solicita ejecutar un comando de GDB, de forma tal que únicamente sea emitida cuando un comando especifico sea solicitado.

    b. **Una Genérica:** Que emitirá todo lo que devuelva GDB a determinadas interfaces GUI. Esta señal se genera en todo momento, por ejemplo para mostrarle al usuario lo que está haciendo el Debugger a través de la interfaz GUI de ConsolaGDB

    c. **Para Puntos de Instrumentación**: Tipo de señal que puede ser utilizada para informarle a la interfaces GUI la ocurrencia en S.O.D.I.U.M de los puntos de Instrumentación. Se genera en forma asincrónica.

  - **Principales métodos definidos:**

a. **iniciarGDB:** Con este método se ejecuta directamente el debugger gdb, generando para ello un objeto GDB que lo ejecuta a través de subprocess

b. **recibirComandoGui**: Método que es invocado por la interfaz GUI, mediante la señal senSolicitudComando, para solicitarle la ejecución de un comando de GDB a GDBMI. Además se le envía una señal Callback específica, que deberá ejecutar GDBMI para retornar el resultado de la operación a dicha interfaz (esta es la señal específica y se genera bajo demanda).

c. **enviarResultadoGui:** Con este método se retorna el resultado de haber ejecutado un comando específico de GDB la interfaz GUI. Para ello se utiliza la señal Callback que se recibió en el método recibirComandoGui (). ACA SE EMITE LA SEÑAL ESPECIFICA. SE GENERA BAJO DEMANDA

d. **enviarResultadoGenericoGUI:** Con este método se le retorna a la interfaz Consola_GDB todas las líneas de salida que emite GDB. (Acá se emite la señal genérica que se genera en todo momento, cada vez que subprocess retorna una línea de la salida de gdb)
e. **recibirSigintGui:** Método que captura la señal petición de detención de GDB emitida por la interfaz GUI y posteriormente la hace efectiva
f. **buscarBreakpoint:** Método que analiza si un breakpoint que ejecuto GDB corresponde a un punto de Instrumentación definido en el archivo breakpoint.xml. Notar que el resultado que emite gdb cuando ejecuta un breakpoint se lo transforma al utilizado por JSON, dado que es muy similar a esta forma de codificación. Esto se realiza para que después se pueda hacer un análisis y guardar dicho resultado en un diccionario, con la finalidad de posteriormente poder determinar si es o no un punto de instrumentación. (Acá se emite el tipo de señal que notifica la ocurrencia de un punto de instrumentación)
g. **borrarPtosInterrupcion:** Método que borra los puntos de interrupción del archivo breakpoint.xml
h. **run**: Método run de GDBMI

- **Ubicación del archivo:**
  /Visualizador/Clases/Gdbmi.py

5.5. Archivo parserGdb.py:

- **Descripción:** Archivo que contiene las clases encargadas de realizar el análisis (parser) de las estructuras de C devueltas por GDB, el mismo recibe el resultado de GDB en forma de un String y devuelve un Diccionario
- **Clases definidas:**
  - **parserGdb:** Clase que realiza el análisis (parser) de las salidas que genera GDB
  - **Principales métodos definidos:**

a. **convertirStringJson:** que convierte un String a JSON.
b. **convertir**: Método que convierte una captura de Qtextedit a JSON
c. **convertirlista:** Método que convierte una captura de Qtextedit a una lista
d. **crearDiccionario:** Método que convierte la cadena JSON en un diccionario

- **Ubicación del archivo:**
  /Visualizador/Clases/parserGdb.py

5.6. Archivo VentanaPrincipal_Codigo.py:

- **Descripción:** Archivo que contiene la Clase de la Interfaz GUI Correspondiente a la Ventana Principal MDI del Visualizador
- **Clases definidas:**
  - **VentanaPrincipal:** Clase de la Interfaz GUI Correspondiente a la Ventana Principal MDI del Visualizador
  - **Princiaples métodos definidos:**

a. **generarConnect**: define las acciones de los botones de menues.
b. **abrirVentanaGenerica**: Método encargado de mostrar por pantalla las Interfaces GUI como una sub-ventana MDI
c. **AbrirVentanaPCB:** Método encargado de mostrar por pantalla la Interfaz VerPCB
d. **AbrirVentanaIDT:** Método encargado de mostrar por pantalla la Interfaz VerIDT
e. **AbrirVentanaTSS:** Método encargado de mostrar por pantalla la Interfaz VerTSS
f. **AbrirVentanaGDT:** Método encargado de mostrar por pantalla la Interfaz VerGDT
g. **AbrirConsolaGDB:** Método encargado de mostrar por pantalla la Interfaz ConsolaGDB
h. **AbrirVentanaMapamemoria:** Método encargado de mostrar por pantalla la Interfaz Ver Mapamemoria
i. **AbrirVentanaDiagramaTemporal: Método** encargado de mostrar por pantalla la Interfaz Diagrama Temporal. Esta función carga los puntos de interrupciones necesarios para que el visualizador funcione antes de mostrar la ventana.

- **Ubicación del archivo:**
  /Visualizador/Clases/VentanaPrincipal_Codigo.py

5.7. Archivo visualizadorGDT_Codigo.py:

- **Descripción:** Archivo que contiene las clase de la Interfaz GUI Correspondiente a la Ventana VerGDT
- **Clases definidas:**
  - **VisualizadorGDT:** Clase de la Interfaz GUI Correspondiente a la *Ventana VerGDT*
  - **Principales Métodos definidos:**

a. **Visualizar:** Este Método recibe la señal del callback enviado y por cada línea que GDB devuelve se va concatenando a la variable que luego se enviará al Parser.
b. **Mostrar**: Este método llama al parser y muestra el árbol en la pantalla
c. **generarConnect:** Método que conecta (enlaza) cada señal con el Método correspondiente que la va capturar (slots).
d. **emitir:** Este Método es el que emite la señal correspondiente enviando el comando a GDB
e. **setupAcceso:** Configura una lista con los accesos existentes en S.O.D.I.U.M.

- **Ubicación del archivo:**
  /Visualizador/Clases/visualizadorGDT_Codigo.py

5.8. Archivo visualizadorIDT_Codigo.py:

- **Descripción:** Archivo que contiene las clase de la Interfaz GUI Correspondiente a la Ventana VerIDT
- **Clases definidas:**
  - **VisualizadorIDT:** Clase de la Interfaz GUI Correspondiente a la *Ventana VerIDT*
  - **Principales Métodos definidos:**

  a. **Visualizar:** Este Método recibe la señal del callback enviado y por cada línea que GDB devuelve se va concatenando a la variable que luego se enviará al Parser.
  b. **Mostrar**: Este Método llama al parser y muestra el árbol en la pantalla
  c. **generarConnect:** Método que conecta (enlaza) cada señal con el Método correspondiente que la va capturar (slots).
  d. **emitir:** Este Método es el que emite la señal correspondiente enviando el comando a GDB
- **Ubicación del archivo:**
  /Visualizador/Clases/visualizadorIDT_Codigo.py

5.9. Archivo visualizadorTSS_Codigo.py:

- **Descripción:** Archivo que contiene las clase de la Interfaz GUI correspondiente a la Ventana VerTSS
- **Clases definidas:**
  - **VisualizadorPCB:** Clase de la Interfaz GUI correspondiente a la *Ventana VerTSS*
  - **Principales Métodos definidos:**

  a. **Visualizar:** Este Método recibe la señal del callback enviada y por cada línea que GDB devuelve se va concatenando a la variable que luego se enviará al Parser.
  b. **Mostrar**: Este método llama al parser y muestra el árbol en la pantalla
  c. **generarConnect:** Método que conecta (enlaza) cada señal con el método correspondiente que la va capturar (slots).
  d. **emitir:** Este Método es el que emite la señal correspondiente enviando el comando a GDB
- **Ubicación del archivo:**
  /Visualizador/Clases/visualizadorTSS_Codigo.py

5.10. Archivo visualizadorPCB_Codigo.py:

- **Descripción:** Archivo que contiene las clase de la Interfaz GUI Correspondiente a la Ventana VerPCB
- **Clases definidas:**
  - **VisualizadorPCB:** Clase de la Interfaz GUI Correspondiente a la *Ventana VerPCB*
  - **Principales Métodos definidos:**

a. **Visualizar:** Este método recibe la señal del callback enviado y por cada línea que GDB devuelve, se va concatenando a la variable que luego se enviará al Parser.
b. **Mostrar**: Este Método llama al parser y muestra el árbol en la pantalla
c. **generarConnect:** Método que conecta (enlaza) cada señal con el método correspondiente que la va capturar (slots).
d. **emitir:** Este Método es el que emite la señal correspondiente enviando el comando a GDB

- **Ubicación del archivo:**
  /Visualizador/Clases/visualizadorPCB_Codigo.py

5.11. Archivo visualizadorMapamoria_Codigo.py:

- **Descripción:** Archivo que contiene las clase de la Interfaz Gui Correspondiente a la Ventana Ver Mapa memoria
- **Clases definidas:**
  - **VisualizadorMapamemoria:** Clase de muestra que hereda de la interfaz GUI generado por QT y maneja todos los controles de dicha ventana grafica para mostrar la composición de la Memoria. Posee métodos que utilizan señales para enviar los comandos a ejecutar en GDB, a través de GDBMI, y recibir el resultado de dicha ejecución GUI

    Principales Métodos definidos:

a. **generarConnect:** Método que conecta (enlaza) cada señal con el método correspondiente que la va capturar (slots).
b. **mostrarTexto**: Este Método muestra las tuplas de la composición de cada segmento en un cuadro de texto por pantalla al presionar el botón "*EjecutarComando"*
c. **emitir:** Este Método es el que emite la señal correspondiente enviando el comando a GDB
d. **dibujar:** Método encargado de comenzar a dibujar en el lienzo de la interfaz gráfica los segmentos memoria, llamando a distintos métodos, después de que el usuario haya presionado en el botón "*graficar"*
e. **fc_grafica:** Método invocado por el método *"dibujar"* para imprimir los distintos bloques de memoria en el lienzo.

  - **dibujo:** Clase que define color y nombre del bloque
  - **ventanaSecundaria:** Clase de la Interfaz GUI secundaria que se muestra las propiedades de cada bloque de memoria
  -

- **Ubicación del archivo:**
  /Visualizador/Clases/visualizadorMampamemroia_Codigo.py

5.12. Archivo visualizadorDiagTemporal_Codigo.py:

- **Descripción:** Archivo que contiene las clase de la Interfaz GUI Correspondiente a la Ventana *"Ver Planificador"*
- **Clases definidas:**
  - **VisualizadorDiagTemporal:** Clase que crea una Ventana MDI que va a incluir una ventana en donde se va a ir graficando el estado que van adquiriendo los procesos durante la ejecución de S.O.D.I.U.M.
  - **Principales Métodos definidos:**

a. **__init__:** Constructor de la clase. Crea el formulario e invoca al objeto de la clase Lienzo que es la encargada de graficar el Diagrama Temporal usando la biblioteca Matplotlib.
b. **initUI:** Método que configura los menús del formulario MDI.
c. **showTeoria:** Método que abre en un navegador una página web con una explicación teórica acerca del funcionamiento del planificador de CPU.
d. **Mostrar**: Este Método llama al parser y muestra el árbol en la pantalla.
e. **showEstadosProcesos:** Muestra una ventana con ayuda sobre los colores de los distintos estados de los procesos que se están graficando.
f. **generarConnect:** Método que conecta (enlaza) cada señal con el método correspondiente que la va capturar (slots).
g. **emitir:** Este Método es el que emite la señal correspondiente enviando el comando a GDB.

- **Ubicación del archivo:**
  /Visualizador/Clases/visualizadorDiagTemporal_Codigo.py

5.13. Archivo breakpoint.xml

- **Descripción**: Archivo XML que contiene la ubicación de los Puntos de Instrumentación que se definirán en el código de S.O.D.I.U.M
- **Ubicación del archivo:**
  /Visualizador/Configuracion/breakpoint.xml

5.14. Archivo lienzo.py:

- **Descripción:** Archivo encargado de dibujar en una interfaz gráfica los distintos estados que va adquiriendo los procesos durante la ejecución de S.O.D.I.U.M.
- **Clases definidas:**
  - **Lienzo:** Clase que va dibujando en un lienzo, usando Matplotlib, los estados de los procesos que se van ejecutando en S.O.D.I.U.M.
  - **Principales Métodos definidos:**

a. **__init__**: Inicializa la clase, configura el lienzo e inicializa el objeto de la clase temporal

b. **timerEvent :** Método que se invoca cada determinado tiempo que lee de una lista la ocurrencia de los cambios de estado de los procesos que le provee la clase temporal. Una vez que leyó el dato actualiza el lienzo mostrando gráficamente el cambio de estado.

- **Ubicación del archivo:**
  /Visualizador/VisorGant/ lienzo.py

5.15. Archivo temporal.py:

- **Descripción:** Archivo que contiene la clase que realiza el llenado de los registros para el refresco de la ventana del visualizador de procesos.
- **Clases definidas:**
  - **temporal :** Esta clase se encarga del cargado de registros para el refresco de la ventana del visualizador de procesos. Para ello, se comunica directamente con GDBMI. Su funcionamiento consiste en cargar los puntos de interrupciones necesarios para que el visualizador de procesos funcione. Una vez cargados, GDBMI informa cada ocurrencia de punto de interrupción de modo que Temporal ejecuta la macro info_plan que obtiene la información de proceso que cambia de estado.
  - **Principales métodos definidos:**

a. **Limpiar**: Limpia las listas de procesos
b. **getTiempo:** Calcula el tiempo.
c. **getCantidadRegistros:** Obtiene la cantidad de registros guardados.
d. **getRegistro:** Obtiene un registro según el índice.
e. **addRegistro:** Agrega/actualiza un registro al sistema.
f. **capturarPtoInstrumentacion:** Recibe cada ocurrencia de punto de interrupción y envía una señal para solicitar la ejecución de la macro *info_plan* a GDBMI**.**
g. **actualizarInterfaz:** Recibe los datos línea a línea de GDBMI y va guardando en un nuevo registro. Una vez que el registro este completo, se lo agrega a la Ventana
h. **generarListaPtoInterrupcion:** Genera la lista de puntos de interrupción a partir de breakpoint.xml
i. borrarListaPtoInterrupcion: Borra el diccionario con los puntos de instrumentación cuando se cierra la ventana de Diagrama temporal.

- **Ubicación del archivo:**
  /Visualizador/VisorGant/Entrada_Diagrama/ lienzo.py

6. Interrelación entre las clases que componen el Visualizador

El archivo GDB.py contiene la clase que se encarga de ejecutar en segundo plano el depurador, usando la clase ***popen.*** Gracias a que dentro de su constructor se crea un nuevo proceso que es el encargado de ejecutar el depurador en una pseudo-terminal, de la misma forma que si se estuviera realizando desde la consola de Linux. Más

adelante en otro apartado se describe la forma de ejecución del depurador en segundo plano y como son tratados los datos devueltos por el mismo dentro de este archivo.

Para eso se le envía como parámetro a su constructor el siguiente comando, indicándole a ***popen*** que argumentos debe utilizar para ejecutar a GDB como si se hiciera desde la línea de comandos

```
gdbCmd= 'gdb  -interpreter=mi -q -b 115200'
```

```
self.process = subprocess.Popen(gdbCmd.split(), stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, sh
```

Como se puede observar, con el argumento "*–interpreter=mi",* se está inicializando al depurador habilitándolo para que utilice su herramienta GDBMI. Para más detalle véase el **Apartado 8** de este documento.

En este caso en particular, al ser ejecutado utilizando ***Popen****,* el depurador siempre se mantendrá activo a la espera de entrada de comandos por parte del usuario. Además el constructor permite redireccionar el flujo de entrada, salida y error de la pseudo-terminal donde se esté ejecutando el depurador. A través de este mecanismo se le puede ordenar a GDB que ejecute ciertos comandos en su terminal y posteriormente, desde el archivo gdb.py, capturar el resultado que se retorna.

Como consecuencia de usar la herramienta GDB/MI, la salida de los resultados que devuelve el depurador son capturados de a una línea por vez en gdb.py. Con lo cual, como la ocurrencia de los resultados suceden asincrónicamente, fue necesario ir almacenando cada línea devuelta dentro de una cola. De esta forma se evitó que se produzcan pérdida de datos durante la lectura del resultado de las operaciones dentro del archivo GDBMI.py.

GDBMI.py almacena la clase Gdbmi, cuya tarea es hacer de intermediario entre la clase GDB y cada una de las interfaces GUI. Su función principal es actuar como un distribuidor de mensajes. Enviándole a la clase Gdb las peticiones de comandos para ser ejecutados en el depurador, que le solicita una determinada interfaz gráfica.

Los objetos de la clase Gdbmi son creados y ejecutados como hilos, para así trabajar independientemente del hilo principal. Por ese motivo se programó el método ***run*** de esta clase con un ciclo infinito, que es activado cada vez que se encola en la clase gdb una línea con el resultado de la salida del depurador. Por lo tanto cuando el hilo de la clase GDBMI reciba el resultado de una operación, este se lo enviará inmediatamente a la interfaz GUI correspondiente. Para ello se creó un algoritmo de selección, para que el programa pueda determinar a qué ventana gráfica le debe enviar el resultado de una operación. En el visualizador las interfaces GUI se clasificaron en tres tipos. Entonces el mecanismo utilizado para la elección va a depender del tipo de interfaz a la que se le debe entregar el resultado.

a. Interfaz que recibe siempre el resultado de cualquier operación:

Esta ventana gráfica recibirá en todo momento la salida de cualquier operación que haya ejecutado el depurador. En este caso en especial, la clase GDBMI emitirá una señal genérica para enviarle el resultado al hilo asociado a este tipo de ventana. La interfaz gráfica de la clase *Consola Gdb,* es la única de este tipo que se encuentra actualmente desarrollada en el código fuente del Visualizador. Esta ventana es la encargada de mostrarle al usuario el resultado de todos los comandos que se ejecuten el depurador.

b. Interfaz que recibe el resultado de una operación determinada:

En este caso el hilo GDBMI le devolverá únicamente el resultado de una operación a la interfaz gráfica que haya solicitado la ejecución de dicha orden. El visualizador del Sistema Operativo está configurado de manera que el depurador ejecute solamente un comando a la vez. De forma tal que el depurador no podrá ejecutar otra instrucción, hasta que el hilo GDBMI no reciba como resultado una línea que le indique que gdb completó la última operación que le solicitaron. Por ende GDBMI enviará a ejecutar un comando a GDB, únicamente cuando el depurador se encuentre inactivo. En otras palabras no esté procesando ninguna operación. Para que esto suceda, se programó el método ***run*** de la clase intermediaria para que detecte este estado cuando que reciba como resultado una línea que sea igual ***"(gdb)"***. Gracias a esto, el hilo de GDBMI puede memorizar qué interfaz GUI le solicitó la ejecución de un comando. Así podrá saber posteriormente, a qué ventana gráfica le deberá enviar el resultado de dicha operación.

Es importante mencionar, que cuando una interfaz GUI le solicita a GDBMI la ejecución de un comando, esta genera una señal tipo callback. La cual, se emite al igual que cualquier otra señal de PyQT. Pero con la particularidad de que se deberán enviar dos parámetros al slot responsable de su captura en la clase GDMI. Como primer parámetro se deberá mandar el comando de GDB que se desea ejecutar en el depurador. En el segundo se tendrá que enviar la señal que GDBMI deberá emitir, para retornar a esta interfaz el resultado de la operación que solicitó. De esta manera siempre GDBMI sabrá a qué método (slot) de la ventana gráfica le deberá retornar el resultado correspondiente. A continuación, en la siguiente figura, se representan en forma esquemática el funcionamiento del mecanismo explicado.

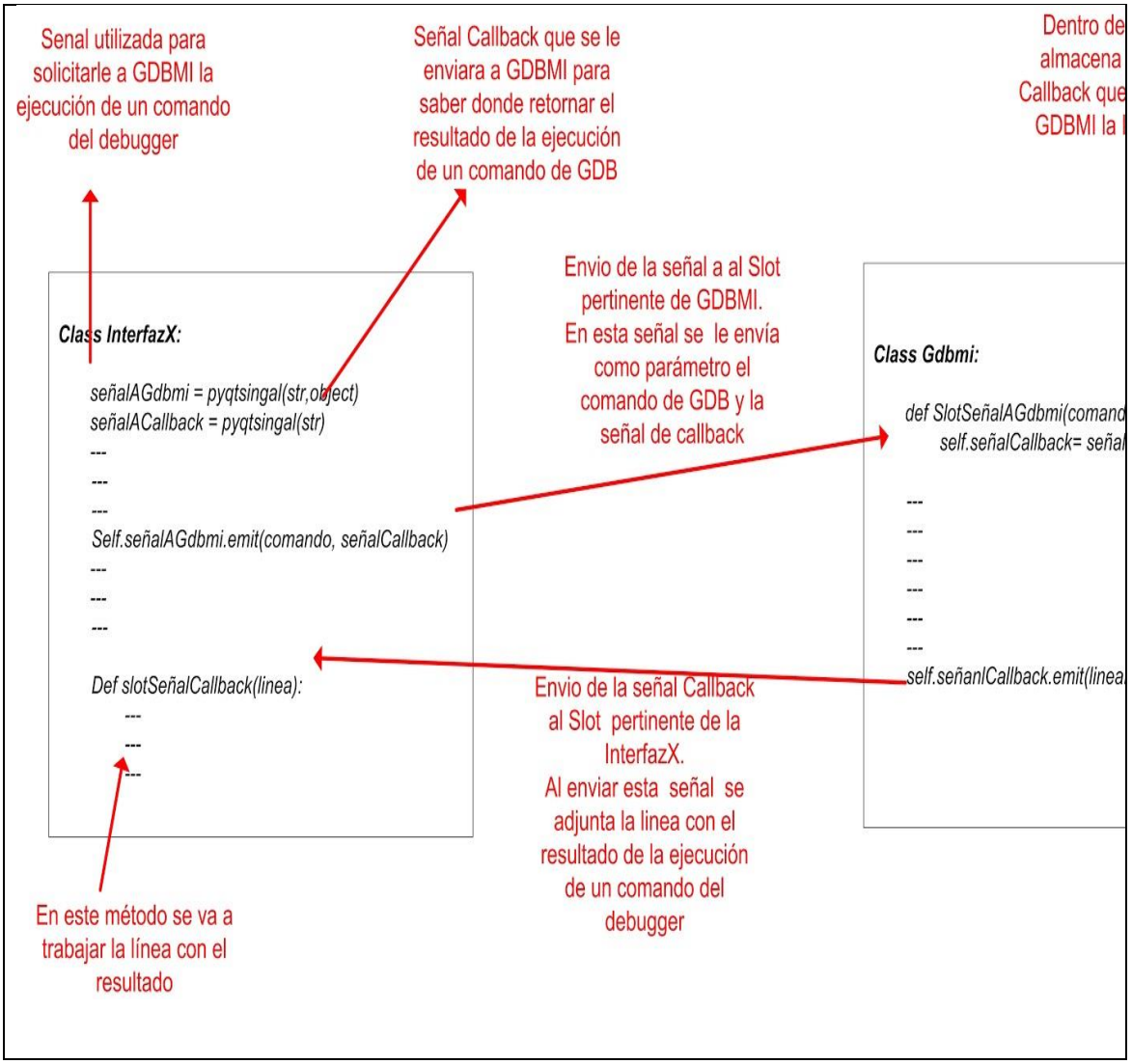


Figura 20- Funcionamiento de la señal Callback

c. Interfaz que recibe evento asincrónico:

La ocurrencia de un breakpoint durante la ejecución del sistema Operativo nos brinda mucha información sobre lo que esto ocurriendo en S.O.D.I.U.M en ese momento. Con esta premisa se diseñaron Puntos de Instrumentación, que se explicarán más en detalle en otro apartado. La ejecución de un breakpoint es considerado suceso asincrónico. Por lo tanto se ha programado al hilo GDBMI para que cuando detecte la ocurrencia de determinado breakpoint le notifique a ciertas interfaces gráficas.

7. Bloques de Código Fuente del Visualizador y el SISTEMA OPERATIVO S.O.D.I.U.M

7.1. Bloque de Código del Comando Mapamemoria

```
define mapaMemoria
    echo  ******************************************************\n
    echo Estructura\t\tInicio\tFin\n
    #Kernel Sodium
    printf "MAIN.BIN\t\t000000\t%x\n",(void*)uiTamanioKernel-1
    printf "BSS\t\t\t%x\t%x\n",uiTamanioKernel,(void*)(uiTamanioKernel+uiTamanioBSS)-1
    printf "Heap Memoria Baja\t%x\t%x\n",ptrHeapMemoriaBaja,(void*)(ptrHeapMemoriaBaja+liTamanioHeapMemoriaBaja)
    printf "Heap Memoria Alta\t%x\t%x\n",ptrHeapMemoriaAlta,(void*)(ptrHeapMemoriaAlta+liTamanioHeapMemoriaAlta)

    #Estructuras Sodium
    printf "PCB\t\t\t%x\t%x\n", pstuPCB,(void*)pstuPCB+sizeof(stuPCB)*CANTMAXPROCS-1
    printf "GDT\t\t\t%x\t%x\n", pstuTablaGdt,(void*)&(pstuTablaGdt->stuGdtDescriptorDescs[8192])-1
    printf "IDT\t\t\t%x\t%x\n", pstuIDT,(void*)pstuIDT+sizeof(stuIDT)-1
    echo ROM-BIOS\t\ta0000\tfffff\n
    busca_ES
    echo  \n******************************************************\n\n

    echo Procesos de Usuario\n

    muestraProcesosUsuario
end
```

Figura 21 - Bloque de código del comando Mapamemoria

7.2. Bloque de Código del Comando GuiGDB

```
define GuiGdb
    source Servidor/GuiGDB/MenuPrincipalGuiGdb.py
end
```

Figura 22- Bloque de código del comando GuiGDB

7.3. Bloque de Código del Comando para la conexión entre S.O.D.I.U.M y el Visualizador

Al presionar en el botón “Conectar con S.O” en la Ventana Principal del Visualizador, el objeto asociado a la Interfaz GUI *Consola GDB,*le solicitará al hilo GDBMI que ejecuten todos los comandos de conexión utilizando la siguiente macro de GDB (3).

```
script=str(
        "define ScriptConexion \n"
            "source comandos_nuevos.gdb \n"
            "symbol kernel/main.ld \n"
            "target remote localhost:12345 \n"
            "end \n"
        "ScriptConexion "
        )
```

Figura 23- macro que establece la conexión entre S.O.D.I.U.M y GDB

De esta forma se genera una macro del depurador llamada *Script Conexión con todos los comandos necesarios que debe ejecutar GDB para* poder establecer la conexión con S.O.D.I.U.M.. Los cuales fueron anteriormente explicados en el apartado *"Conexión con GDB desde Sodium-Devkit".* La particularidad que tienen los comandos que definen macros en GDB, es que permiten ejecutar varios comandos seguidos en una sola operación. Permitiendo así evitar problemas de sincronización entre el depurador y el visualizador.

7.4. Obtención de las estructuras internas de S.O.D.I.U.M a través del Visualizador

Una vez que el usuario presiona el botón Visualizar correspondiente, automáticamente el objeto de la interfaz gráfica le enviará al hilo de GDBMI una orden para ejecutar el siguiente comando en el depurador, dependiendo de la estructura que haya solicitado el usuario.

- **Ventana "Ver PCB":**
  print pstuPCB["+self.txtIndex.text()+"]
- **Ventana "Ver IDT":**
  print pstuIDT -> IDTDescriptor["+self.txtIndex.text()+"]
- **Ventana "Ver GDT":**
  print pstuTablaGdt->stuGdtDescriptorDescs["+self.txtIndex.text()+"]"
- **Ventana "Ver TSS":**
  "print stuTSSTablaTareas["+self.txtIndex.text()+"]"

De esta forma se le está solicitando a GDB que devuelva el contenido de la estructura PCB del proceso seleccionado. Como se había mencionado, el depurador siempre retorna el resultado de su ejecución en formato de texto. Con lo cual fue necesario realizar una adaptación de un formato de salida mediante un parser, realizando la conversión del texto a un diccionario Python, de manera de poder mostrar los campos en forma separada a través Treeview. La transformación no fue directa, sino que se usó un estado intermedio, aprovechando las facilidades que ofrece el formato JSON, para la conversión de texto a diccionarios. La clases encargada de realizar los parsers que se encuentran definidas dentro del archivo parserGdb.py. Luego el resultado obtenido de GDB en formato JSON en un cuadro de texto y su asociación en un Treeview

7.5. Redefinición del comando Mapamemoria

El código fuente de esta instrucción fue readaptado de la siguiente manera, para que imprima por consola la constitución de la memoria a través de tuplas Python:

```
define mapaMemoria

    printf "(MAIN.BIN,0,%X)",(void*)uiTamanioKernel-1
    printf "(BSS,%X,%X)",uiTamanioKernel,(void*)(uiTamanioKernel+uiTamanioBSS)-1
    printf "(HeapBaja,%X,%X)",ptrHeapMemoriaBaja,(void*)(ptrHeapMemoriaBaja+liTamanioHeapMemori
    printf "(HeapAlta,%X,%X)",ptrHeapMemoriaAlta,(void*)(ptrHeapMemoriaAlta+liTamanioHeapMemori

    #Estructuras Sodium
    printf "(PCB,%X,%X)", pstuPCB,(void*)pstuPCB+sizeof(stuPCB)*CANTMAXPROCS-1
    printf "(GDT,%X,%X)", pstuTablaGdt,(void*)&(pstuTablaGdt->stuGdtDescriptorDescs[8192])-1
    printf "(IDT,%X,%X)", pstuIDT,(void*)pstuIDT+sizeof(stuIDT)-1
    printf "(ROM_BIOS,A0000,FFFFF)"

    busca_BD

    muestraProcUsua
end
```

Figura 24-Redefinición del comando mapamemoria de GDB

Cada una de la tuplas impresas presenta el subsiguiente formato:

(Nombre del segmento, Dirección de Inicio, Dirección de Fin)

Este cambio se realizó para que desde el Visualizador se pueda realizar la conversión de texto a diccionario del resultado obtenido de la ejecución del comando *Mapamemoria,* utilizando para ello el formato JSON. Pudiendo así posteriormente separar el contenido de cada campo en el código de la interfaz gráfica.

7.6. Detección de los Puntos de Instrumentación

En el momento en que el depurador establece los breakpoints asociados a los Puntos de Instrumentación, cuando el usuario abre la Interfaz del Diagrama Temporal, el hilo de GDBMI empezará a evaluarla ocurrencia de dichos eventos. Es importante acotar que cualquier tipo de breakpoint sucede en forma asincrónica. Por ese motivo GDBMI deberá evaluar en todo momento si cada línea devuelta por el depurador, indica que se ha ejecutado un breakpoint en S.O.D.I.U.M. Si esto es así, se invocará al método *buscarBreakpoint* para determinar si se ejecutó un Punto de instrumentación. Para ello evaluará si el número de línea del archivo fuente donde se detuvo la ejecución de S.O.D.I.U.M, esté definido en uno de los Nodos del archivo *breakpoint.xml*. En caso de que sea afirmativo, se ha producido el cambio de estado de un proceso en el Sistema Operativo. Por ende el hilo GDBMI le solicitará al depurador información sobre el estado de ese proceso. Con lo cual, le ordenará a GDB que ejecute el siguiente comando:

info_plan

Esta instrucción pertenece a una macro de GDB, que se define automáticamente cuando se inicializa la Interfaz *ConsolaGDB*, al abrir la ventana Principal del Visualizador.

```
script=str(
            "def info_plan\n"
                "print pstuPCB[ iIndicePCBCambioEstado ].stNombre\n"
                "print pstuPCB[ iIndicePCBCambioEstado ].ulId\n"
                "print pstuPCB[ iIndicePCBCambioEstado ].iEstado\n"
                "print ulTiempo\n"
                "continue\n"
            "end\n"
            )
```

Figura 25- Definición de la macro info_plan

Como se puede observar en la figura anterior, el estado de un determinado proceso se obtiene a partir de determinados campos de su PCB:

- **stNombre:** Nombre del Proceso
- **ulId:** Pid del proceso
- **iEstado:** Si se encuentra en estado listo, ejecutando, detenido o eliminado.
- **ulTiempo:** Es el tiempo transcurrido en milisegundos, desde que se inició S.O.D.I.U.M hasta que se produjo el cambio de estado.

El hilo GDBMI, al recibir el resultado de la ejecución de la macro *info_plan,* emitirá una señal a la interfaz Diagrama Temporal con la información obtenida del S.O sobre el cambio del estado de un proceso. Por consiguiente se ejecutará el slot encargado de capturar esta señal, llamado *actualizarInterfaz,* definido dentro de la clase *Temporal.* A continuación, en dicho método, se generará un registro que almacenará los campos: **Nombre, PID, Estado, tiempo,** que fueron obtenidos a través de la macro mencionada. Esta información es utilizada por el Visualizador para representar gráficamente en el lienzo el cambio de estado de ese proceso.

Es importante mencionar que la información retornada por la macro no es representada inmediatamente en el lienzo, cuando es recibida en la Clase *Temporal, como consecuencia de* que el mecanismo para graficar en el lienzo de dicha Interfaz GUI, se comporta como una aplicación Productor-Consumidor. El código de la clase *Temporal* funciona como un Productor de información. Allí se van almacenando en una lista, los registros que contienen los datos que identifican cada cambio de estados a medida que van ocurriendo durante la ejecución del Sistema Operativo. Por otra parte el código de la Clase *Lienzo* funcionaría como Consumidor. Debido a que cada cierto determinado tiempo analizará la lista anterior para comprobar que haya información sobre nuevos cambios de estado, y en el caso de que existan representarlos gráficamente en el lienzo.

8. Sintaxis Principal de GDB/MI

En este apartado se explicarán las terminologías y notaciones que usa GDB/MI para intercambiar datos con GDB.

- **Ejecución de GDB:**

Para poder habilitar la herramienta GDB/MI, será necesario ejecutar el depurador desde la línea de comandos de Linux de la siguiente forma:

Gdb –interpreter=mi

Una vez que se haya abierto la consola de GDB e impreso su prompt por pantalla, el usuario podrá interactuar con GDB utilizando dicha interfaz. Para ello podrá ejecutar los comandos que desee del depurador mediante la siguiente sintaxis:

- ***Ejecución de comandos GDB:***

Gdb –interpreter-exec "*comando"*

- **Impresión de resultados en GDB/MI:**

GBD/MI puede retornar en distintas líneas de impresión el resultado de la ejecución de un comando en GDB. Cada una de ellas contiene información que les puede servir a los Front-Ends para tomar decisiones e informar al usuario, dependiendo del formato en que sea devuelto. Por consiguiente, a continuación se muestra las posibles representaciones de las impresiones devueltas por el depurador.

REGISTRO DE RESULTADOS:
Además de las líneas que notifican el resultado que retorna GDB, las respuestas a GDB/MI pueden incluir una de las siguientes indicaciones.

o "^done" ["," *results*]:

Si la operación sincrónica fue satisfactoria.

o "^connected":

Cuando GDB se conectó en forma remota.

o "^error" "," "msg=" *c-string* ["," "code=" *c-string*]

Cuando una operación falló.

o "^exit"

Cuando la ejecución de GDB finalizó.

REGISTRO DE FLUJOS:
GDB internamente mantiene una serie de flujos de salida, la consola, el target (aplicación que se está depurando, en este caso S.O.D.I.U.M) y el log. Las salidas provistas por cada una de ellas pueden ser canalizadas a través de la interfaz GDB/MI usando registro de flujos. Cada registro de flujo comienza con un carácter prefijado que identifica su corriente. Además el prefijo de cada registro, contiene una cadena de salida. Esto es ya sea sin formato (con una nueva línea implícita) o una nueva cadena en lenguaje C (que no contenga una nueva cadena implícita). Por lo tanto el depurador, por cada línea obtenida de GDB que comience con un único carácter específico, presentará la siguiente información:

- "~" *string-output:*

  El flujo de la consola imprimirá una respuesta textual luego de haber ejecutado un comando GDB.

- "@" *string-output*

  El flujo del target mostrará impresiones asincrónicas en la consola de GDB que son enviadas internamente desde el código de S.O.D.I.U.M durante su ejecución.

- "&" *string-output*

  El flujo de log contendrá mensajes del depurador que son producidos internamente dentro de GDB.

REGISTRO ASINCRÓNICO:
Los registros asincrónicos son usados para notificar en cualquier momento a GDB/MI que ha ocurrido algún cambio durante la ejecución de S.O.D.I.U.M. Estos cambios pueden ser consecuencia de haber ejecutado cierto comando de GDB (por ej., modificación de un breakpoint) o como resultado de una actividad (por ejemplo, detención de S.O.D.I.U.M). Por consiguiente, la siguiente es una lista de algunos de los posibles registros asincrónicos.

- *running,thread-id="*thread*"

  Indica que S.O.D.I.U.M. se encuentra ejecutándose.

- *stopped,reason="*reason*",thread-id="*id*",stopped-threads="*stopped*",core="*core*"

La ejecución de S.O.D.I.U.M se ha detenido. El campo "*reason"* indica la razón de porque ocurrió dicho evento. Los valores más importantes que puede adquirir esta variable son los siguientes:

-breakpoint-hit: Indica que ha ocurrido un breakpoint

-end-stepping-range: Un comando *next*, *step u* otro similar ha sido ejecutado por GDB.