Herramienta de depuración con representación de regiones de memoria

Por
Jorge Moreno Martínez

UNIVERSIDAD COMPLUTENSE
MADRID

Grado en Desarrollo de Videojuegos
Facultad de Informática

Dirigido por
Christian Tenllado van der Reijden
José Ignacio Gómez Pérez

Debugging tool with memory regions representation

Madrid, 2021–2022

Herramienta de depuración con representación de regiones de memoria

Extensión creada para la interfaz gráfica *Gede*

Memoria que se presenta para el Trabajo de Fin de Grado

Jorge Moreno Martínez

Dirigido por

Christian Tenllado van der Reijden
José Ignacio Gómez Pérez

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2022

Agradecimientos

A mi perro Sein, que ha sido quien me ha sacado de casa mientras trabajaba en este proyecto.

A mi padre, que desde el principio no ha dejado de preguntar qué tal iba el trabajo final, y a mi madre, que me hizo unos *tuppers* con guisos espectaculares y me ha regalado tantísimo tiempo para dedicarle a esto.

Resumen

Este proyecto ha consistido en crear una extensión para una interfaz gráfica de usuario (GUI, por sus siglas en inglés) de depuración, con el objetivo de añadir funcionalidades relacionadas con las regiones de memoria. El depurador escogido ha sido *Gnu Project Debugger* [1] (de aquí en adelante, **GDB**), que funciona para varios lenguajes de programación. En concreto, la extensión se ha centrado en los lenguajes C y C++, y se ha añadido a la GUI Gede [2].

La extensión permite visualizar el mapa de memoria del proceso que se está depurando, incluyendo información sobre los tamaños de cada región de memoria, sus posiciones en memoria virtual, el fichero de respaldo de cada una y los permisos de acceso del proceso.

Palabras clave

Gede, GDB, depurador, mapa de memoria, extensión de depuración, región de memoria.

Abstract

This project has consisted on creating an extension for a debugging Graphic User Interface (GUI), in order to add features related to the memory regions of a process. The debugging tool selected for this work has been Gnu Project Debugger (GDB), which works on several programming languages. This extension is particularly focused on the programming languages C and C++, and it has been developed for the GUI "Gede".

The extension allows the user to visualize the memory map of the debugging process, including information about each memory region's size, memory position, backup file and permissions.

Key words

Gede, GDB, debugger, memory map, debugging extension, memory region.

Índice general

Página

1. Introducción **2**
- 1.1. Antecedentes . . . 3
- 1.2. Objetivos . . . 4
- 1.3. Plan de trabajo . . . 4

2. GDB y herramientas gráficas de depuración **9**
- 2.1. GDB . . . 9
- 2.2. Interfaces gráficas para GDB . . . 11
  - 2.2.1. Visual Studio Code . . . 12
  - 2.2.2. Gede . . . 14

3. Trabajo desarrollado **16**
- 3.1. Comunicación con GDB . . . 18
  - 3.1.1. Clase Core . . . 18
  - 3.1.2. Clase GdbCom . . . 19
  - 3.1.3. Clase CoreMemRegion . . . 21
- 3.2. Modificaciones para la representación visual . . . 21
  - 3.2.1. Clase CustomVarCtl . . . 21
  - 3.2.2. Clase MainWindow . . . 22
- 3.3. *Pretty printers* . . . 23

4. Evaluación de la herramienta **24**

5. Conclusiones **33**

6. Bibliografía y enlaces de referencia **35**

Capítulo 1

Introducción

La depuración es el proceso de encontrar y resolver errores (*bugs*) en un programa informático. Es una parte esencial del desarrollo de programas (en muchos casos se llega a emplear más tiempo en esta parte que en escribir el propio código) y, dada su importancia, no es de extrañar que haya herramientas diseñadas para esta función concreta.

El alumnado de la rama de informática ve este proceso desde que comienza a dar sus primeros pasos en programación, observando así cómo las variables van tomando valores conforme las instrucciones lo indican. No obstante, hay elementos del software de los que programadores noveles no suelen ser conscientes.

Es el caso del espacio de direcciones de memoria virtual asociado a un proceso. Las regiones de memoria son explicadas sin un soporte visual cómodo y normalmente quedan como algo obscuro e inaccesible más allá de algunos comandos de consola poco utilizados.

La asignatura de Fundamentos de los Computadores (FC) es una de las 10 que componen el plan de estudios del primer año del Grado en Desarrollo de Videojuegos. Se cursa durante el primer cuatrimestre de la carrera, y uno de los apartados del temario es la programación en lenguaje de ensamblador para procesadores ARM.

Esos primeros ejercicios de programación en ensamblador sirven, entre otras cosas, para dar los primeros pasos en los distintos ámbitos de las variables: globales, locales y automáticas (en la sección *stack*) y dinámicas (sección *heap*), constantes que se resuelven en tiempo de compilación... En concreto, se explica dónde se almacenan en memoria cada una de ellas y su representación, y se hace explícita la relación entre una variable y su dirección de memoria. Si bien estos conocimientos conforman una base importante, pueden resultar complejos y abrumar a quienes acaban de entrar en el grado.

Posteriormente, en la asignatura de Sistemas Operativos se estudia que, cuando se ejecuta un programa, se crea su proceso correspondiente, y las secciones del archivo binario del programa se incluyen en el mapa de memoria del proceso como regiones de memoria no necesariamente consecutivas. Estas regiones pueden contener varias de las secciones del binario a la vez o formar parte de una librería dinámica externa. Cada región puede tener diferentes permisos de acceso o estar compartidas con otros procesos. Además, pueden corresponderse a las pilas (*stack*) de otras hebras creadas por el proceso, o con fragmentos

de ficheros proyectados en memoria. Por supuesto, si lo estudiado acerca de las secciones de memoria en FC no se ha asimilado correctamente, estos otros conocimientos son aún más complejos y pueden presentar una mayor dificultad.

En mi opinión, parte de la complejidad a la hora de entender el funcionamiento de las secciones y del mapa de memoria podría ser explicada por la falta de una visualización interactiva. Si bien a la hora de depurar un programa se puede observar cómo las variables van tomando diferentes valores, no ocurre lo mismo con el mapa y las secciones de memoria, que en ningún momento aparecen en un ámbito práctico donde se aplique la teoría impartida.

Las regiones de memoria de un proceso se pueden analizar utilizando comandos de GDB, pero para acceder a toda la información es necesario recurrir a llamadas al sistema o a estar en plataformas muy específicas. Combinar esta información en un programa normal es bastante laborioso, unido además a la forma de trabajar de GDB mediante línea de comandos, que requiere el esfuerzo adicional de escribir cada comando e ir comparando las salidas de los diferentes comandos según los rangos de direcciones que muestren.

La ausencia de una herramienta visual e interactiva, que aúne esta información y permita al alumnado ver qué regiones hay en el mapa de memoria de un proceso mientras lo está depurando y qué propiedades tiene, hace que estos importantes conceptos no puedan ser asimilados con facilidad. Esta carencia sido la principal motivación para crear una herramienta que ilustre los cambios que sufre el mapa de memoria de un programa durante su depuración y a dónde corresponden las secciones del fichero.

1.1. Antecedentes

GDB, del que hablaremos en profundidad más adelante, es el depurador estándar para GNU. Su función es ayudar a las personas a encontrar errores en un programa; para ello, GDB es capaz de detener el programa en ejecución, examinar su estado, e incluso cambiar valores en el programa.

GDB tiene un gran repertorio de comandos, incluyendo todos los necesarios para llevar a cabo este proyecto. Para trabajar con él se utiliza la línea de comandos; se le mandan instrucciones en forma de cadenas de texto escritas por el usuario, y devuelve los datos también en forma de cadenas de texto impresas en la consola (si procede devolver datos, por supuesto). Gracias a este formato, GDB puede ser integrado en muchos lugares tan sólo con una herramienta que sea capaz de enviar y recibir comandos por el terminal de control de GDB.

GDB puede lanzarse usando MI, una interfaz de texto orientada para facilitar el uso con máquinas. La entrada y la salida de los comandos de GDB/MI está adaptada para que sea de fácil *parseo*, es decir, que programar la lectura de la salida sea más sencillo, en comparación al formato de texto básico y orientado a usuarios humanos de GDB. Para usar GDB/MI, hay que lanzar GDB usando el parámetro *–interpreter*.

El hecho de que GDB no haga uso de una interfaz gráfica hace que, en ocasiones, trabajar

con este depurador pueda volverse incómodo. Por suerte, existen programas dedicados a ofrecer una interfaz gráfica para controlar GDB, y editores que incorporan esta funcionalidad como uno más de sus componentes o extensiones. Estas interfaces suelen presentar botones que sirven para controlar el flujo de la depuración y ventanas que se actualizan en tiempo real para ahorrarle al usuario la solicitud de los datos. No obstante, el uso de interfaces también puede ser limitante, ya que las funciones de GDB que no se hayan implementado en la interfaz quedan inaccesibles. Es el caso de la visualización del mapa de memoria, que sí está disponible en GDB como texto plano pero está ausente en las interfaces gráficas (o no se ha encontrado una que lo haga).

Entonces, para cubrir este hueco, hace falta crear una interfaz gráfica o extender una existente que muestre la visualización del mapa de memoria. Para la tarea se ha acabado tomando la decisión de extender Gede, una interfaz gráfica ligera y sencilla escrita por Johan Henriksson.

Gede es un *front end* de GDB desarrollado en C++ con el objetivo de crear la representación gráfica de las funciones más comúnmente utilizadas de GDB. Permite introducir puntos de interrupción, ver la lista completa de variables, agregar inspecciones, controlar la ejecución del programa y visualizar las hebras y los puntos de interrupción.

1.2. Objetivos

El objetivo de este proyecto es proporcionar una interfaz gráfica que ofrezca al programador una representación útil e interactiva de la información obtenida a partir de estos comandos de GDB. En concreto, la intención es crear una herramienta que:

- Muestre las propiedades más importantes de las regiones de memoria.
- Se actualice dinámicamente para que sea fácil trabajar con la información dada.
- Esté basada en GDB para que sea fácilmente portable a otros sistemas POSIX.
- No tenga una arquitectura compleja.

1.3. Plan de trabajo

En cuanto al plan de trabajo, la distribución del tiempo disponible es la siguiente:

- **Desde septiembre hasta diciembre**: selección de plataforma objetivo. Exploración de varias alternativas (Visual Studio Code [3], Eclipse CDT [4], Gede [2]...). Esto incluye estudiar las tecnologías que se usan en cada uno de estos proyectos, y valorar la complejidad de la arquitectura y la capacidad de extensión que tenemos al alcance. Adicionalmente, investigación de formas de obtener el mapa de memoria y las secciones.
- **Diciembre a enero**: estudio en profundidad de la arquitectura de la plataforma escogida. Comienzo de la memoria.
- **Febrero a abril**: desarrollo de la extensión.

- **Abril y mayo**: evaluación de la extensión y finalización de la memoria.

El resto de la memoria de este proyecto se organiza del siguiente modo: en el *Capítulo 2: GDB y herramientas gráficas de depuración* se explica en detalle qué es GDB y cuáles son las opciones de interfaces gráficas valoradas para el proyecto; en el *Capítulo 3: Trabajo desarrollado* se explica la extensión desarrollada y cómo funciona; y en el *Capítulo 4: Evaluación de la herramienta* se utilizan diferentes programas de ejemplo con mapas de memoria interesantes para demostrar el uso de la herramienta. Por último se encuentra el *Capítulo 5: Conclusiones*, con la recapitulación de los objetivos logrados y algunas sugerencias para mejorar el trabajo.

Introduction

Debugging is the process of finding and solving programming errors (bugs) in a program. It is an essential part of program development (usually, more time is invested into debugging rather than writing the code itself) and, given its importance, the existence of tools that eases this work is not surprising.

Programming students start to experience this process since the beginning of their education, watching how variables change their values with each code instruction. However, there are software elements that rookies do not know about.

Thus is the case of the memory map associated to a process. These memory regions are explained without a visual support, so understanding and interpreting them can be difficult and clumsy.

Introduction to Computers (FC, from its initials in Spanish), is one of the 10 subjects featured in the Games Development degree. One of its lessons consist on studying ARM assembly programming. These exercises serve the purpose of letting the students take their first steps on the variables' different scopes: global, local, automatic (stack section) and dynamic (heap section), constants (that are resolved on compilation)... More precisely, it is explained how the variables are stored in memory and their representation, and the connection between a variable and its memory address. Although this knowledge is fundamental, it can be overwhelming to those who have recently started their computer studies.

Later, on the Operating Systems subject, it is studied that, when a process is ran, the sections from the program's binary file are included in the memory map as regions that may or may not be consecutive. These regions can include one or more of the binary sections or be part of a dynamic library. Each region can as well have different access permissions or be shared with other processes. They can also correspond to the stacks of other threads created by the same process, or to shards of files projected on memory. Of course, if the contents about memory sections studied in FC have not been fully acknowledged, this theory turns up being even more complex and can be difficult to learn.

In my opinion, part of this complexity when understanding the functioning of the memory sections and the memory map could be explained by the absence of an adequate and interactive visualization. Although a program's variables can be checked with ease, the same does not happen with the memory map, which does not appear in the practice.

A process' memory regions can be examined by using GDB commands, but in order

to access to all the information it is mandatory to use system calls or to work in very specific platforms. Combining this information in a common program is a laborious work, that requires the additional effort of writing each command and system call and compare the outputs of each of them with the memory ranges shown.

The absence of a visual and interactive tool, that combines this information and allows the users to see which regions are displayed in the memory map of a process while it is being debugged and which properties has, makes these important concepts hard to absorb. This lack has been the main incentive to create a tool capable to show the changes that the memory map of a process experiments while it is being debugged.

Previous work

GDB, of which we will talk later, is the GNU Project Debugger. Its main purpose is to help developers find errors in a program; for that task, it is capable of stopping the running program, examine its state, or even change the values of the variables in the program.

GDB features a great collection of commands, including those needed for this project. To work with GDB, it is used a command-line interface. The user writes instructions that represent GDB commands and the interface returns the result of each command. Thanks to this format, GDB can be easily integrated in multiple platforms by using a tool that is able to send and receive strings through GDB's terminal.

GDB can be launched using MI, a line based machine oriented text interface to GDB, used to make easier the usage with machines. Input and output is adapted so parsing it is easy, compared to basic GDB. To use GDB/MI, GDB is launched specifying the option "–interpreter" with value "mi".

The fact that GDB does not use a graphic interface causes that working with this debugger is occasionally tiresome. Luckily, there are programs that offer a GDB front end, and editors that include this functionality as one of their components or extensions. These interfaces usually feature buttons or similar utilities that help control the program's execution, and windows that show the debugging information and are updated in real time to save work for the user. Nontheless, the usage of these programs can also be a limitation, because the GDB features that are not implemented in the interface are left out of reach. This is the case of memory map visualization: it is available while using a command-line interface, but is absent in all graphic interfaces, as far as the author knows.

It is then needed, to fill in this gap, a graphic interface that shows the visualization of the memory map. For this task, the final decision has been extending Gede, a lightweight graphic interface written by Johan Henriksson.

Gede is a graphical GDB front end written in C++ with the purpose of creating the graphic representation of the most common GDB utilities. Gede allows to, among other utilities: place breakpoints, watch the complete list of variables and add variable watches, control the execution of the program and visualize the threads of the process and the placed breakpoints.

Goals

The main goal of this project is to provide a Graphic User Interface that offers the programmer a useful and interactive representation of the information obtained through the previously mentioned GDB commands. In particular, the features that the tool will have are:

- Shows the most important properties of the memory regions.
- Dynamically updates so working with the given information becomes easy.
- Is based on GDB so it is easily portable to other POSIX environments.
- Does not have an excessively complex architecture.

Working plan

The development shedule is as follows:

- **From September to December**: choice of the target platform. Reconnaissance of the multiple alternatives (Visual Studio Code, Eclipse, Gede...). This includes studying the technologies used in each of them, and evaluate the complexity of the architecture and the ability to extend it. Additionally, investigation of ways to obtain the information about the memory map and memory sections.
- **December to January**: depth study of the target platform. Start of the report.
- **February to April**: development of the extension.
- **April to May**: evaluation of the extension and finishing of the report.

The rest of this report is organized as follows: the second chapter talks about GDB and the various GUIs evaluated for this project; the third chapter explains the developed extension and how it works; and the fourth chapter analyzes the usage of the tool, using for that purpose various programs with interesting memory maps. Finally, the fifth chapter summarises the achieved goals and some suggestions for future work.

Capítulo 2

GDB y herramientas gráficas de depuración

GDB [1], o GNU Debugger, es un depurador desarrollado por Richard Stallman para el compilador GNU [5]. Soporta una variedad de 12 lenguajes de programación, y su función es permitir ver lo que está ocurriendo en un programa mientras este se ejecuta, o su estado en el momento en el que ocurre un *crash*. Para ello, GDB es capaz de detener el programa en depuración, examinar su estado, e incluso cambiar valores en el programa.

Por otro lado, las interfaces gráficas son programas que permiten hacer un uso más cómodo y eficiente del depurador. Generalmente se pueden dividir en dos tipos: las interfaces desarrolladas específicamente para GDB y las interfaces que permiten la depuración con GDB mediante alguna extensión. Para este proyecto, se ha investigado y se han valorado opciones de ambos tipos.

En este capítulo primero se explicará en mayor detalle qué es y como funciona GDB, para a continuación enumerar algunas de las diferentes interfaces gráficas disponibles para GDB y explicar las opciones más importantes.

2.1. GDB

GDB es una herramienta de depuración cuya interfaz es la línea de comandos. Para utilizarlo, hay que invocar el comando *gdb* desde una consola o emulación de terminal. A continuación, GDB lee comandos de la terminal hasta que reciba el comando *quit* o se detenga su proceso. Para indicar el programa que debe depurar, se le puede especificar mismamente al ejecutarse con *gdb [nombre del programa]*. También se puede depurar un proceso que ya esté ejecutándose, indicando su ID de proceso. En este momento ya se pueden colocar puntos de interrupción, arrancar el programa, ver el contenido del mapa de memoria, etc.

Las interfaces gráficas para el depurador suelen tener varias ventanas, en las cuales muestran el código fuente de los archivos del programa y la información de GDB que desean mostrar. Además, también suelen incorporar facilidades o atajos para ejecutar los comandos de GDB. Por ejemplo, la extensión de depurador de Visual Studio Code tiene botones

para controlar el flujo de ejecución del programa, en lugar de que el usuario tenga que llamar a comandos de GDB. Además, algunas presentan la propia consola de GDB por si el usuario quiere utilizar algún comando que no ha sido representado en la interfaz gráfica.

Para la comunicación entre la interfaz y GDB, la primera crea un terminal que vincula a la entrada y salida estándar de GDB. De esta manera, la interfaz puede enviar comandos y recibir la salida normal del depurador. Para implementar la extensión, es preciso comprender la arquitectura construida para esta comunicación, es decir, cómo se produce esta vinculación a la entrada y salida. Después seremos capaces de utilizarla para enviar a GDB los comandos de interés e interpretar la salida del depurador.

De entre los comandos que incorpora, GDB cuenta con varios que sirven para la descripción del mapa de memoria y de las regiones de código. Hay dos especialmente útiles, *info proc mappings* [6] e *info files* [7], que ofrecen distinta información sobre el proceso que está siendo depurado (o sobre cualquier otro indicado, pero eso no nos interesa):

- *info proc mappings*: muestra los rangos de direcciones de memoria accesibles para un proceso. En GNU/Linux este comando nos da para cada una de las regiones: las direcciones de memoria (virtual) de comienzo y fin, su tamaño (en bytes) y su desplazamiento en el fichero de respaldo. En Solaris, FreeBSD y NetBSD este comando incluye también información sobre los permisos de acceso (lectura, escritura y ejecución) para cada rango de direcciones. Por último, en GNU/Linux, FreeBSD y NetBSD, cada rango de direcciones muestra el fichero de respaldo que está mapeado a ese rango.
- *info files*: muestra los nombres de los ejecutables y archivos de volcado que GDB tiene en uso, los archivos de los que ha leído los símbolos, y las direcciones de memoria que ocupan. Relacionando esta información con la proporcionada por *info proc mappings* podemos determinar las secciones de cada fichero que están mapeadas en cada región del espacio de direcciones del proceso. Además, podemos deducir el tamaño que tienen estas secciones con una simple resta entre la dirección de memoria final y la inicial.

Pongamos un ejemplo práctico para demostrar la información que obtenemos al combinar estos comandos.

Supongamos que estamos depurando, usando para ello GDB, un programa cuyo código fuente se encuentra en el archivo *ejemplo.c*. Al ejecutar el comando *info proc mappings*, uno de los rangos de direcciones de memoria que se nos muestran es el que hay entre las direcciones 0x58000 - 0x59000, y su programa asociado es "ejemplo". Por otro lado, al llamar a *info files* se muestran dos secciones entre los rangos 0x58000 - 0x59000: **.data**, entre 0x58000 y 0x58010, y **.bss**, entre 0x58010 y 0x58018. Así, podemos entre otras cosas saber que la sección *.data* del programa está ubicada en 0x58000 y que ocupa 16B, y también que ambas secciones se mapean sobre la misma región de memoria.

Como hemos visto, podemos obtener bastante información sólo con estos dos comandos. No obstante, los permisos de cada región del mapa de memoria sólo se pueden conseguir mediante *info proc mappings* en arquitecturas Solaris [8], FreeBSD [9] y NetBSD [10], y esa información es muy interesante para la extensión.

Como GDB no tiene un comando que muestre los permisos de cada región en cualquiera de las arquitecturas que lo soportan, debemos suplir esta carencia con una llamada al sistema. Para ello, se puede recurrir a *pmap* [11]. Este comando muestra información del mapa de memoria, y admite varias opciones. En concreto, entre las opciones *-x* y *-d* conseguimos que *pmap* muestre mucha información, de entre la cual la más relevante es la siguiente (se muestra primero el nombre que recibe cada campo según *pmap* y después se explica qué representa):

- *Address*: dirección de memoria inicial de la zona mapeada.
- *Kbytes*: tamaño de la zona mapeada en kilobytes.
- *Dirty*: páginas sucias (tanto compartidas como privadas), en kilobytes.
- *Mode*: permisos de la zona mapeada: lectura, escritura y ejecución, y si es compartida o privada.
- *Mapping*: fichero de respaldo de la zona mapeada, o "[anon]" para regiones sin respaldo en fichero ("anónimas"), o "[stack]" para la pila del programa.
- *Offset*: desplazamiento dentro del fichero de respaldo.
- *Device*: ID del dispositivo (en formato *major*:*minor*).

De toda esta información, las más interesantes para la extensión son *Mode*, para mostrar los permisos, y *Mapping*, para conseguir determinar los nombres de las regiones de memoria anónimas. Los otros elementos de interés (*Address*, *Kbytes* y *Offset*) no son necesarios porque su información ya ha sido obtenida mediante *info proc mappings*.

Es importante recalcar que *pmap* es parte del proyecto *procps-ng* (*/proc filesystem utilities*), es decir, no pertenece a GDB sino que es una utilidad propia de Linux. En las plataformas que no proporcionan *pmap*, GDB ofrece la información con *info proc mappings*.

Agrupando toda la información que aparece en estos comandos podemos acabar teniendo una lista bastante detallada, representativa del mapa de memoria del programa que estamos depurando.

2.2. Interfaces gráficas para GDB

Como ya hemos mencionado antes, existen interfaces gráficas (*GUIs*, por las siglas en inglés de *Graphic User Interface*) que facilitan el trabajo a desarrolladores a la hora de depurar. Estas interfaces pueden ser de dos tipos: *front ends* para GDB, es decir, herramientas hechas específicamente para ofrecer un soporte visual para el depurador; e IDEs (entornos de desarrollo) que incorporan, mediante extensiones o complementos, la funcionalidad de depuración.

Cuando exploramos la lista de interfaces gráficas para GDB, nos encontramos con una gran variedad de entornos. Cada uno tiene características que difieren mucho con los demás, y tomar una decisión acerca de cuál extender no ha resultado ser una tarea fácil. Por

ello, a continuación se listan las principales opciones que se barajaron para desarrollar el proyecto sobre ellas:

- **gdbgui** [12]: es un *front end* basado en tecnología web, escrito en Python y JavaScript, y que soporta C, C++ y otros lenguajes. Fue una opción muy relevante, pero descartada finalmente por la poca experiencia que se tenía en Python y JavaScript, en comparación con los lenguajes en los que están desarrolladas otras interfaces.
- **CLion** [13]: desarrollada por JetBrains, es un IDE que soporta C y C++ y es multiplataforma. El principal problema y motivo de su descarte es que su licencia no permite modificar el código fuente (ni tampoco es este accesible).
- **Eclipse CDT** [4]: basada en la plataforma Eclipse, ofrece soporte a C y C++. No obstante, está escrita en Java, lenguaje con el que tampoco se tenía demasiada soltura en el momento de elegir la GUI a extender. Por otro lado, dado su tamaño y complejidad, extender este IDE es una tarea bastante complicada.
- **Nemiver** [14]: es un *front end* para depurar C/C++ usando para ello GDB. Aunque parecía ideal por estar desarrollada en C++ (lenguaje que el autor de este trabajo maneja con gran fluidez), el repositorio del proyecto está archivado y su última actualización tuvo lugar hace dos años, lo que da a entender que su desarrollo está abandonado y que, en caso de problemas, no tendría ninguna clase de soporte a quien dirigirme.
- **CodeLite** [15]: un proyecto *open source*, y también desarrollado en C++, pero al ser un IDE presenta una gran complejidad que supera con creces el alcance de la extensión a desarrollar.
- **Visual Studio Code** [3]: es un IDE muy modular y extensible. Dada su gran popularidad actual y su adopción como entorno en muchas asignaturas de la Facultad de Informática de la UCM, este entorno se analizó en mayor detalle como primer candidato para el desarrollo. La *Subsección 2.2.1: Visual Studio Code* analiza este entorno en mayor profundidad.
- **Gede** [2]: finalmente, un proyecto desarrollado en C++, con licencia que permite su uso y modificación, y con un tamaño asumible como para trabajar con él en el contexto de un Trabajo de Fin de Grado. Se detalla más acerca de esta interfaz en la *Subsección 2.2.2: Gede*.

El resto de esta sección describe en mayor detalle las dos plataformas que se estudiaron como candidatas para el desarrollo de la herramienta: Visual Studio Code y Gede.

2.2.1. Visual Studio Code

Inicialmente se valoró desarrollar la herramienta como un *plugin* para el IDE Visual Studio Code, o como una extensión de su *plugin* de depuración. Esta plataforma parecía en un principio ideal, dada su gran modularidad y su extenso uso, además de ser de código abierto (la licencia permite su modificación y el código fuente está disponible online). Es un entorno de desarrollo muy personalizable, que permite añadir extensiones de todo tipo desarrolladas por cualquiera que quiera subir su trabajo a la *VS Code Extension Marketplace*, la plataforma oficial de donde se pueden descargar.

No obstante, esa misma modularidad viene dada por su gran complejidad, por lo que es necesario conocer bien la arquitectura completa de la extensión de depurador para poder interaccionar con él. Visual Studio Code está implementado sobre un *toolkit* para el desarrollo de navegadores web, y toda la interfaz gráfica se desarrolla utilizando tecnología web con lenguajes como TypeScript o JavaScript (tecnología desconocida para el autor de este trabajo). Además, la extensión de depuración de GDB no está documentada, y forma parte de una compleja arquitectura que se describe a continuación.

Una importante parte de esta arquitectura es el *Debug Adapter Protocol* [16]. El *Debug Adapter Protocol*, o DAP, por sus siglas, es el protocolo que establece Visual Studio Code para indicar cómo un IDE debería comunicarse con un depurador.

Para esta función, los propios desarrolladores admiten que es poco realista por su parte asumir que los depuradores ya existentes adopten el protocolo en un futuro próximo. Por eso, se crea un componente intermedio, el *Debug Adapter*, que toma el rol de adaptar un depurador ya existente al protocolo.

El protocolo especifica una serie de eventos, genéricos y necesarios para cualquier depurador. Estos eventos se utilizan desde el IDE para que este tenga la misma interfaz para todos los depuradores que se le puedan presentar, puesto que todos cumplirán el DAP. En la figura 2.1 se ilustra esta arquitectura.

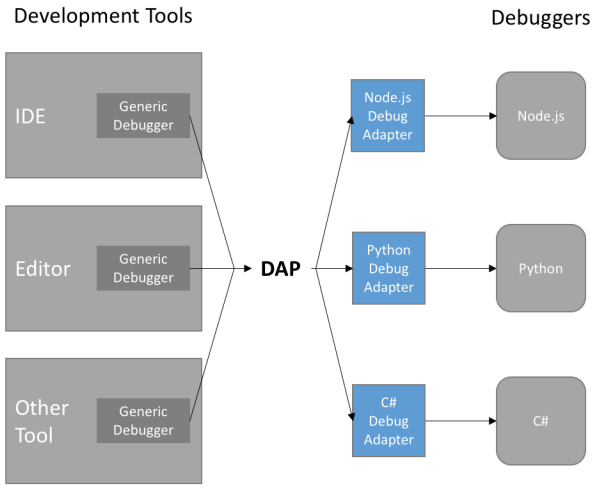


Figura 2.1: ejemplo de arquitectura resultante tras aplicar el *Debug Adapter Protocol*. Imagen obtenida de la página oficial.

El problema principal que surge llegado este punto es que la extensión que se quiere desarrollar necesita de una funcionalidad que no está soportada por muchos otros depuradores, y por ende, el soporte para obtener la información necesaria no se incluye en el API del *Debug Adapter Protocol*. Esto implica que, para crear la extensión para Visual Studio Code, no sólo hay que hacer un *Debug Adapter* basado en el de GDB, sino que hay que extender el *Debug Adapter Protocol*. Esta problemática es de una dificultad que sobrepasa el alcance del proyecto. Además, no se ha logrado encontrar el código fuente del *Debug Adapter* de GDB, ni tampoco documentación alguna sobre la implementación del *Debug Adapter Protocol* ni su código fuente.

Dado el gran tiempo empleado en investigar acerca de la arquitectura de Visual Studio Code (incluyendo pero no limitándose a la forma de desarrollar extensiones y la comunicación con depuradores mediante el DAP), y debido a no haber podido encontrar el código fuente necesario, se decidió abandonar la idea de desarrollar para Visual Studio Code y se optó por elegir un entorno de desarrollo con una arquitectura mucho más simple y centrado directamente en el depurador GDB.

2.2.2. Gede

Gede fue la interfaz gráfica escogida finalmente para llevar a cabo el proyecto. Está escrito en C++ y hace uso del *toolkit* (librería de desarrollo) de Qt [17] en su versión 5. Gracias a ello, es bastante simple y extendible sin muchos problemas a partir únicamente del código fuente. A pesar de que las "tripas" del programa no están documentadas, ni tan siquiera comentadas, son bastante autoexplicativas y entender lo que está ocurriendo es sencillo una vez que se estudia el flujo de ejecución.

Para este proyecto se ha utilizado la versión 2.18.1, del 14 de agosto de 2021.

Como se ha mencionado anteriormente, Gede hace uso de Qt5. Qt es un entorno de trabajo de desarrollo de aplicaciones para múltiples plataformas. Su producto *Qt Framework*, utilizado en Gede, es muy sencillo y a la vez muy potente, con una API bien documentada y muchos años de experiencia y de usuarios. Facilita y simplifica enormemente tareas como la creación de ventanas. Qt nos aporta una gestión gráfica tan abstraída que nos despreocupamos casi por completo del apartado visual y nos podemos centrar, en este caso, en la comunicación con GDB.

En cuanto a la arquitectura, las partes más importantes se pueden dividir en elementos para la comunicación con GDB y elementos para la ventana de la aplicación. La clase principal, llamada Core, se dedica a coordinar los eventos y transmitir la información entre ellas.

En detalle, las clases más importantes para el proyecto (y las que más modificadas han sido) son las siguientes:

- **GdbCom** se encarga de la comunicación con GDB. Sus funciones principales son inicializar la instancia de GDB, escribir los comandos para la consola del depurador y leer la salida que este le devuelve.
  GdbCom hace uso del MI para interpretar esta salida, lo que facilita mucho la tarea de *parsing*.

  La comunicación con GDB se lleva a cabo mediante un QProcess, un proceso de Qt. Este proceso es el que corresponde a la instancia del depurador, y se puede inicializar con una simple cadena de texto que represente un comando: en concreto, el que se utiliza para instanciar GDB es: "gdb –interpreter=mi2", siendo "gdb" la ruta del depurador e "–interpreter=mi2" una de las opciones, en concreto la que activa el intérprete MI para esta instancia de GDB. A partir de este momento, se

puede utilizar directamente el QProcess como una línea de comandos, ya que esta clase incluye métodos de lectura y escritura.

- **Core** es la clase principal de Gede. Se ocupa de comunicar a la ventana la información del depurador que le ofrece el GdbCom. También es capaz de mandar peticiones de comandos al GdbCom: por ejemplo, si el usuario pulsa el botón para añadir un punto de interrupción, el Core recibe la notificación de que se ha pulsado ese botón y llama al GdbCom con el comando "*-break-insert*" y la línea en la que se haya colocado el punto de interrupción. En este caso concreto, no haría falta leer la salida sino únicamente comprobar que el resultado no sea un error.
- **MainWindow** es la ventana, la clase que contiene las instancias de los *widgets* de Qt. En ella se agrupan todos los elementos visuales, y tiene métodos para reaccionar a cada evento que ocurre en el Core. Redirige los datos que recibe en los eventos a los *widgets* para que los representen visualmente.

En la figura 2.2 se ilustra cómo es el flujo de comunicación entre el usuario y GDB mediante Gede.

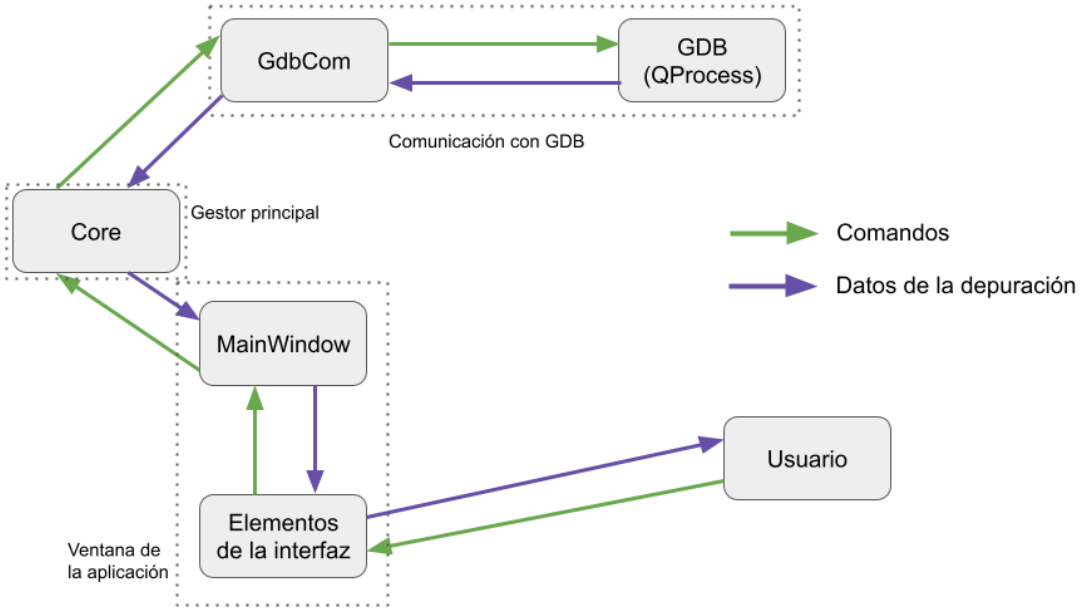


Figura 2.2: flujo de comunicación entre las diferentes secciones de Gede.

El proyecto ha consistido principalmente en modificaciones y añadidos a estas clases, en las dos primeras para aumentar la funcionalidad y la comunicación con la línea de comandos de GDB y en la última para la representación visual de la extensión.

Capítulo 3

Trabajo desarrollado

Este proyecto ha consistido, como se ha mencionado antes, en crear una extensión para la interfaz gráfica Gede, escrita en C++ y usando el entorno de desarrollo Qt. La extensión permite visualizar el espacio de direcciones de memoria virtual cada vez que se detiene el flujo de ejecución del programa que está siendo depurado.

El repositorio del proyecto está alojado en el portal GitHub, y se puede encontrar en el siguiente enlace: https://github.com/jorgmo02/TFG.

Tras modificarlo, Gede muestra el aspecto de la figura 3.1, donde la extensión de representación del mapa de memoria ha sido señalada mediante un recuadro de color rojo.

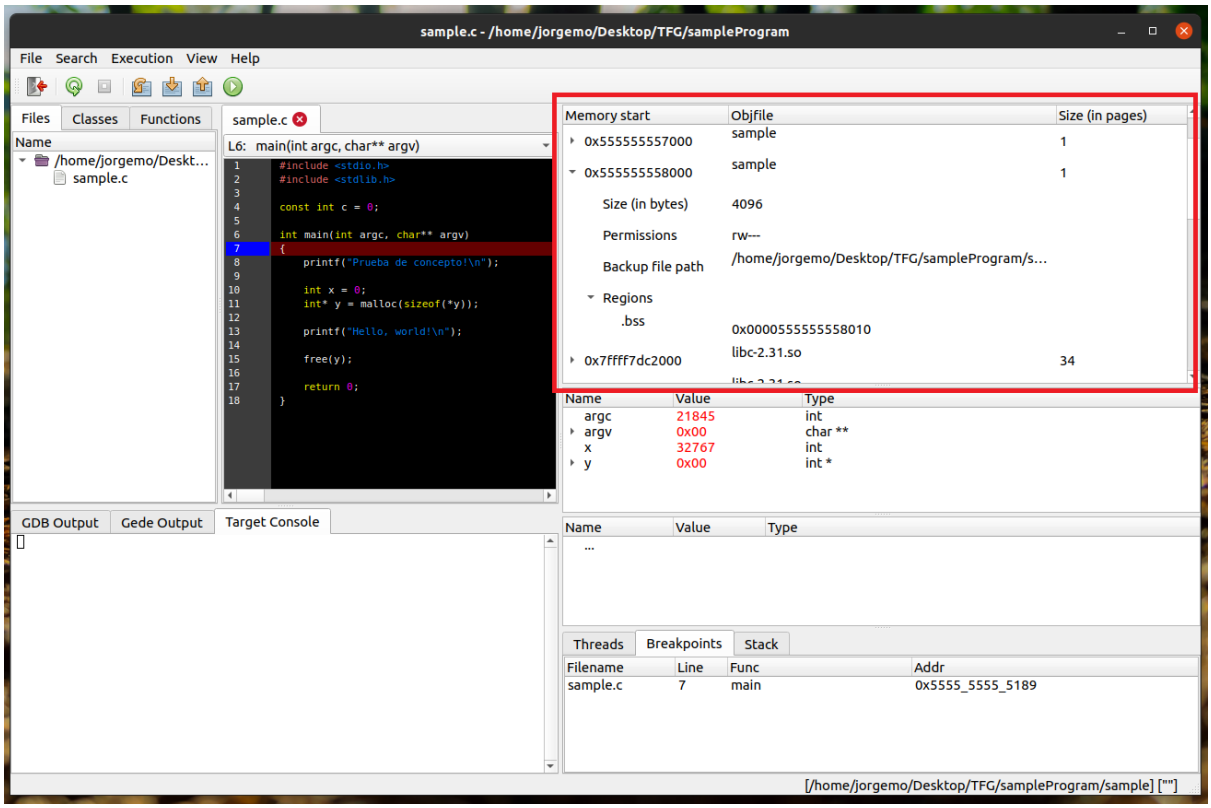


Figura 3.1: vista general del programa con la extensión habilitada.

En concreto, la extensión desarrollada es la que aparece en la esquina superior derecha, con tres columnas de nombres "Memory start", "Objfile" y "Size (in pages)". El manejo de la extensión es bastante intuitivo para quienes saben qué significa cada uno de los valores que están leyendo, y se asemeja en su organización a los comandos *info proc mappings* y *pmap*.

Veamos ahora qué significa cada valor. Para explicarlo, en la figura 3.2 se han numerado los diferentes campos y se hace referencia a cada uno de ellos según esos números.

En primer lugar, el número hexadecimal bajo la columna de nombre "Memory start" (1) señala el comienzo de la región de memoria mapeada. A su lado, "Objfile" (2) representa el nombre del fichero de respaldo. Finalmente, "Size (in pages)" (3) representa el tamaño de esa región de memoria en páginas.

Cuando presionamos el triángulo a la izquierda de la dirección inicial de memoria, se despliega la información relativa al bloque de memoria. En ese momento se pueden leer el tamaño en bytes (4), sus permisos de acceso (5), la ruta completa del fichero de respaldo (6) y otro desplegable con las regiones incluidas en ese espacio de memoria (7), vacío para este ejemplo.

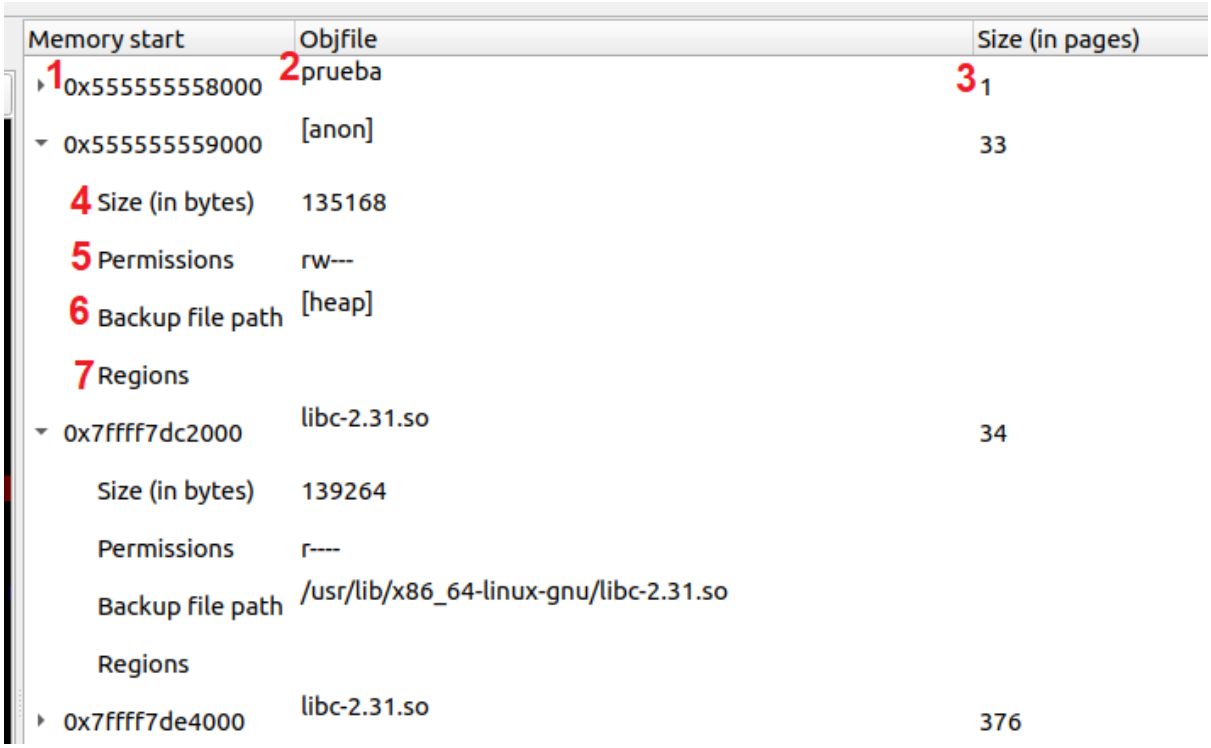


Figura 3.2: vista centrada en la extensión.

El grueso del trabajo desarrollado sobre Gede se puede dividir en dos categorías. Por un lado, están las modificaciones realizadas con el objetivo de comunicarse con el depurador o con el sistema; por otro, está el código añadido a la GUI original para crear una nueva sección para visualizar los datos.

3.1. Comunicación con GDB

Las modificaciones con el objetivo de aumentar las vías de comunicación han sido necesarias porque Gede utiliza el MI para comunicarse con GDB. Utilizar el MI es una buena decisión por parte de Gede, ya que está pensado para facilitar la interpretación de la salida de GDB. No obstante, esto presenta una complicación, ya que el MI no tiene instrucciones equivalentes a los comandos que se necesitan (*info proc mappings* e *info files*). Por tanto, hay que hacer llamadas a los comandos base de GDB. El MI permite esto, pero el analizador sintáctico (el *parser*, en inglés) de Gede no está hecho para los comandos base sino para el MI y no se puede utilizar correctamente para leer la salida de los comandos base de GDB. Por tanto, ha habido que crear una herramienta que sea capaz de interpretar la salida de un comando base de GDB usando el MI.

Los principales cambios con este objetivo han sido en las clases Core y GdbCom, además de la creación de la clase CoreMemRegion. A continuación se ahonda en las modificaciones realizadas: envío de nuevos comandos al depurador, creación de métodos que extraigan la salida sin el formato del MI, interpretación de esa salida y creación de métodos que realizan llamadas al sistema y leen sus correspondientes salidas.

3.1.1. Clase Core

Como se ha explicado en el anterior capítulo, esta clase se ocupa de mandar peticiones de comandos al GdbCom y de comunicar a la ventana la información que este le brinda.

En el Core se han creado tres métodos que se encargan de lanzar los comandos *info proc mappings*, *info files* y *pmap*. Estos métodos, que se explican a continuación, llevan a cabo las siguientes tareas:

1 Obtener la instancia del comunicador con GDB.

2 Enviarle al comunicador el comando que se quiere ejecutar y obtener el resultado.

3 Limpiar el resultado para que sólo queden los datos útiles y no aparezcan cabeceras.

4 Devolver, en forma de lista de *QString*, las líneas de texto que ha obtenido del comunicador.

El método que permite obtener la información de *info proc mappings* se llama *gdbGetMemoryMap*. Este método usa la instancia estática del GdbCom con el objetivo de mandarle un comando. Para ello, primero construye una cadena de texto con el siguiente aspecto:

```
"-interpreter-exec console \"info proc mappings [pid del proceso]\""
```

En la cadena de texto, "-interpreter-exec" es el nombre de una opción que permite indicar el intérprete a usar para el comando. En este caso, se usa *console* para evitar usar el intérprete MI en el comando y en su lugar usar GDB base. Por otro lado, "info proc mappings [pid]" es el comando que le queremos enviar a GDB.

A continuación, esta cadena de texto se envía al GdbCom mediante el método *GdbCom::commandGetOutputLines*, que toma un comando y una lista (vacía) de cadenas de

texto y rellena esa lista con la salida del comando dado. Veremos este método más en detalle en la *Subsección 3.1.2: Clase GdbCom*. Luego se limpia esta lista de caracteres innecesarios, para finalmente devolverla.

En cuanto al comando *info files*, el método que permite obtenerlo es *gdbGetMemoryNames*. Su funcionamiento es idéntico al de gdbGetMemoryMap, con las diferencias de que la información devuelta es la correspondiente a *info files* y a que la cadena de texto utilizada es:

```
"-interpreter-exec console \"info files [pid del proceso]\""
```

Por último, el método *gdbGetMemoryPermissions* sirve para hacer la llamada al sistema *pmap*. Para ello, en GdbCom se ha creado un método *readLinesFromShell* del que hablaremos en *Subsección 3.1.2: Clase GdbCom*, y que sirve para enviar llamadas al sistema y recibir las salidas de estas. Como en los anteriores, el funcionamiento es similar, y sólo varía en que se usa *readLinesFromShell* en lugar de *commandGetOutputLines* y en que el comando enviado es:

```
"pmap [pid]"
```

Como vemos, el funcionamiento de todos estos comandos incorporados a la clase Core es semejante. Además, aunque menor, otra modificación hecha en esta clase es la llamada a la actualización de la ventana cada vez que se recibe el evento de interrupción de la ejecución del programa.

3.1.2. Clase GdbCom

La clase GdbCom implementa métodos para enviar comandos y para leer la salida de esos comandos. A la hora de precisar de una comunicación que no pase por el intérprete de MI, han sido necesarios dos pasos. El primero consiste en crear una manera de mandar comandos a GDB y que el comunicador sepa que no debe intentar crear ninguna clase de estructura a partir de los datos que se reciban, porque no tienen su formato esperado. Para el segundo, ya que el comunicador no debe intentar crear ninguna estructura, hay que definir una manera de leer los datos que se reciben de GDB.

Para la primera tarea, he creado el método *commandGetOutputLines*. En esencia, este método se basa en el método preexistente *command*, cuya funcionalidad era la de enviar un comando a GDB y almacenar la salida en una estructura llamada *Tree*.

El método *commandGetOutputLines* toma como parámetros una lista de cadenas alfanuméricas llamada *resultData* (en la que almacena la salida obtenida) y una cadena alfanumérica que representa el comando a ejecutar, llamada *comm*. Es de interés que *resultData* puede tomar el valor NULL, caso que se emplearía para cuando no es importante la salida pero se quiere enviar un comando de igual manera.

En este método, en primer lugar se limpia *resultData* por si contuviera información previa. A continuación, el texto en *comm* es enviado a la línea de comandos (el QProcess del

que hemos hablado en la *Subsección 2.2.2: Gede*). Posteriormente se leen las líneas de la consola de GDB, con un método creado específicamente que explicaré a continuación. Por último, se devuelve un entero que indica si ha habido algún error. En este momento, resultData contiene una lista con todas las líneas que ha devuelto la salida de GDB.

El método de lectura de salida específico para leer la salida de un comando base de GDB ha sido necesario porque normalmente Gede hace uso de su herramienta de *parseo* para crear su estructura *Tree* que almacena los diferentes *tokens* leídos en forma de árbol. No obstante, esta lectura de *tokens* está preparada para el MI, no para comandos de GDB, como ya anticipábamos.

Para crear la lectura, he modificado la ya existente para que en lugar de tomar un *Tree*, tome la lista anteriormente descrita de cadenas de texto (*resultData*). Con esta lista dada, el método va consumiendo los *tokens* que lee del proceso línea a línea, y cada vez que termina con una de estas líneas la añade a *resultData* (si no es nula). El final de lectura llega cuando no se encuentra ninguna línea más que leer o cuando ocurre un error. Finalmente, el método devuelve un entero indicando si ha ocurrido un problema, y cuando termina *resultData* está llena con la salida de GDB línea a línea.

Finalmente, por comodidad, se ha creado un método en el comunicador que se encarga de ejecutar una llamada al sistema. En realidad, este método nada tiene que ver con GDB, pero se ha añadido al GdbCom para mantener todos los comandos en el mismo lugar.

El método que llama al sistema y devuelve las líneas leídas es bastante diferente a los anteriores, aunque de la misma manera toma por parámetros una lista de cadenas y una cadena que indica la llamada a realizar.

Este método ha recibido el nombre de *readLinesFromShell*, y lo que hace es abrir el fichero de respaldo asociado a la llamada al sistema que se quiere realizar, leer línea por línea este fichero e ir almacenándola en *resultData*, y cerrar el fichero. Devuelve un entero que indica si se ha podido abrir el archivo o no.

De esta forma, ya tenemos:

- Una manera de comunicarnos con GDB y que nos devuelva las líneas leídas, saltándonos el protocolo *Machine Interface*.
- Una manera de ejecutar una llamada al sistema y obtener su salida línea a línea.
- El resultado de los tres comandos que necesitábamos en un principio, almacenados como listas de cadenas alfanuméricas.

3.1.3. Clase CoreMemRegion

Una vez que tenemos todas las herramientas a nuestra disposición, es el momento de traducir y separar los datos, esto es, para cada cadena alfanumérica hay que separarla en varias variables individuales y con diferente tipado.

Para ello se ha creado la clase CoreMemRegion, que almacena todos estos datos y tiene varios métodos para separar las cadenas que le llegan y traducirlas a los datos que le interesan.

En concreto, las variables que usa son las siguientes:

- *quint64 address*: dirección de memoria inicial de la región, en formato entero.
- *QString backupfile*: nombre del fichero de respaldo.
- *QString backupfilePath*: ruta completa del fichero de respaldo.
- *quint64 size*: tamaño de la región (en bytes).
- *QString permissions*: permisos de la región de memoria.
- *QStringList names*: nombres de las secciones de código alojadas en esta región de memoria.
- *QStringList regionsAddr*: posiciones de memoria de las secciones de código alojadas en esta región de memoria.

La clase contiene métodos para devolver cada uno de estos valores, que son privados. Adicionalmente, también tiene métodos para conseguir la dirección de memoria en formato hexadecimal y para obtener el tamaño de la región en número de páginas virtuales.

Por otro lado, y como hemos comentado antes, la clase también tiene métodos para obtener, a partir de una cadena de texto dada, los valores de cada uno de los anteriores parámetros. Estos métodos se llaman *loadFromGdbString*, *setPermissionsFromString* y *setNamesFromString*, y su funcionalidad consiste en separar las cadenas en subcadenas (utilizando como elemento separador los espacios en blanco) y reinterpretar determinadas subcadenas de las resultantes para adaptarlas al tipo esperado.

3.2. Modificaciones para la representación visual

La mayor parte del trabajo de representación visual ha consistido en extender la clase MainWindow y en crear una clase que representa la ventana de la extensión. Esta última clase ha recibido el nombre de CustomVarCtl.

3.2.1. Clase CustomVarCtl

La clase CustomVarCtl es la encargada de contener, mediante un *QWidget*, todos los desplegables de las regiones de memoria. En un principio se consideró que su funcionamiento sería similar a las ventanas de las variables de código, y su nombre deriva de

ahí, puesto que todas estas ventanas heredan de una clase VarCtl (que a su vez hereda de QObject). No obstante, ha acabado por diferir bastante, por lo que este nombre y la propia herencia son decisiones discutibles.

Cuando se detiene la ejecución, CustomVarCtl limpia todos sus hijos (elementos de Qt de la anterior parada). A continuación, recibe n llamadas a su método *ICore_onCoreMemChanged*, siendo n el número de espacios de memoria mapeados. En cada una de estas llamadas le llega una instancia de CoreMemRegion con los datos del siguiente espacio de memoria, que aún no ha sido representado gráficamente, y CustomVarCtl se encarga de crear un elemento de Qt con tantos parámetros como atributos tiene CoreMemRegion. Algunos de estos elementos son hijos de otros, creando así los desplegables.

Borrar la lista de elementos supone que todos los que estuvieran expandidos colapsan de nuevo. Esto hace que la aproximación elegida no sea la ideal, y es por ello que en el *Capítulo 5: Conclusiones* se describe como posible trabajo futuro una mejora de esta funcionalidad que no ha llegado a ser completada.

3.2.2. Clase MainWindow

La clase MainWindow es, esencialmente, coordinadora de todas las sub-ventanas que la componen (incluyendo, claro está, la extensión de regiones de memoria). Hace uso de otra clase auxiliar llamada Ui_MainWindow, que es la encargada de contener todos los *QWidget* pertenecientes a cada elemento visible.

MainWindow es la clase a la que se llama cuando se detiene la ejecución y hay que actualizar el mapa de memoria (aunque también actualiza otros elementos, como los valores de las variables). Su método *ICore_onCoreMemChanged* se encarga de esta actualización.

En primer lugar, en *ICore_onCoreMemChanged* se obtiene la instancia del *Core* para a continuación llamar a sus métodos *gdbGetMemoryMap*, *gdbGetMemoryPermissions* y *gdbGetMemoryNames*, explicados anteriormente. Como recordatorio, estos métodos devuelven la información en forma de listas de cadenas alfanuméricas.

Ya con la información obtenida, se itera por las listas instanciando en cada iteración un nuevo *CoreMemRegion* con los datos. El *CoreMemRegion* es entonces enviado al *CustomVar*, como se ha descrito en la *Subsección 3.2.1: Clase CustomVarCtl*.

En conjunción, estas modificaciones consiguen nuestro objetivo: representar la información de las regiones mapeadas en memoria de manera cómoda e intuitiva, en una interfaz gráfica.

3.3. *Pretty printers*

Durante el desarrollo ha sido necesario depurar el propio Gede con la extensión. Esto se ha hecho desde la interfaz de consola de GDB, para lo que ha sido necesario crear lo que se denomina un *pretty printer*: un archivo de configuración, escrito en Python, que se emplea para indicarle a GDB cómo tiene que mostrar en pantalla algunos tipos de variables. En concreto, este *pretty printer* ha sido necesario para poder mostrar en pantalla de una manera legible los tipos de Qt.

Capítulo 4

Evaluación de la herramienta

En este capítulo se utilizan varios ejemplos para demostrar que la herramienta funciona correctamente, muestra el resultado esperado y es de utilidad. Estos ejemplos tratan, de manera individual, situaciones relevantes y características de los diferentes ámbitos de las variables en memoria.

Como recordatorio, el principal servicio que pretende ofrecer este trabajo es de tipo didáctico y de ayuda al profesorado a la hora de enseñar cómo funcionan las regiones de memoria. Estos ejemplos pueden constituir un buen recurso con tal objetivo.

Variables globales

Este apartado consiste en demostrar el funcionamiento de las variables globales y su diferente ubicación según su valor inicial. Para ello, usamos un programa en el que existen, entre otras, dos variables globales:

- *int global = 0*
- *int *p*

Como sabemos, *global* debe ir ubicada en la sección *.data* del binario, dado que es una variable con valor inicial, y *p* debe ir en la sección *.bss* al ser una variable sin inicializar.

Comprobemos si esto es así. Para ello, en el programa se escribe en pantalla la dirección de memoria de ambas variables, y podemos comprobar esta dirección con las de las secciones *.data* y *.bss*. En la Figura 4.1 vemos la situación.

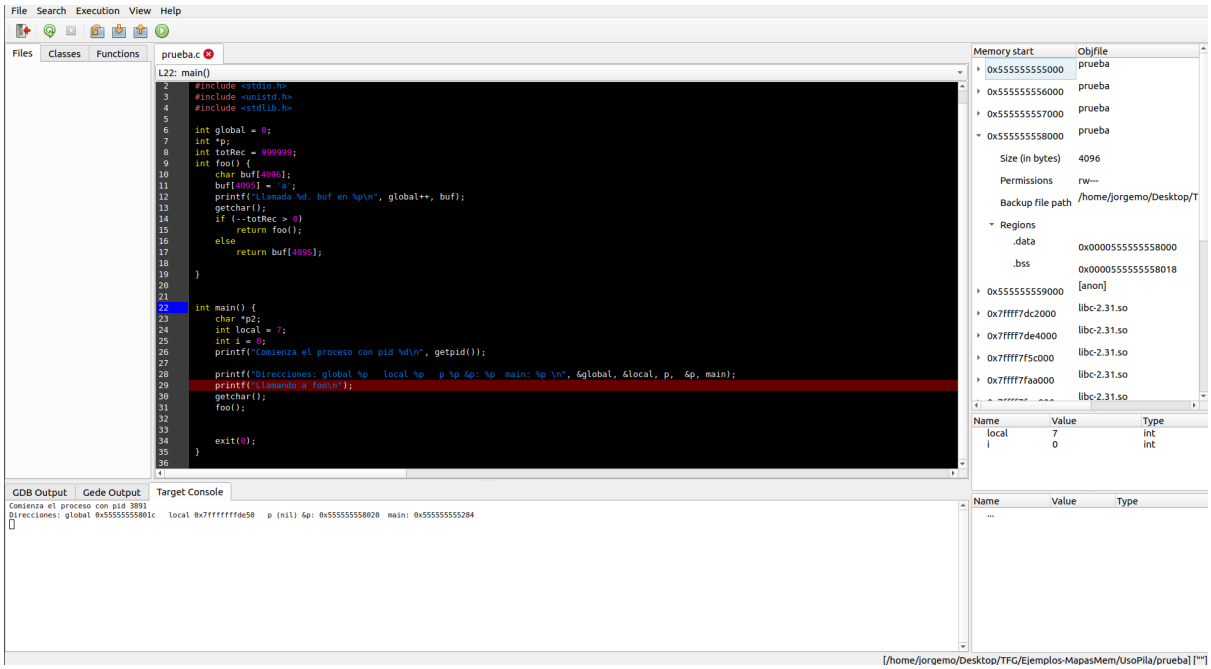


Figura 4.1: ubicación de las variables globales *global = 0* y *p*.

La variable *p* está como anticipábamos: en la sección *.bss*. Su dirección es 0x555555558020 y la de la sección es 0x555555558018, así que encaja con la explicación totalmente.

En cuanto a la variable *global*, podemos observar que el programa indica que su dirección es 0x55555555801c. La sección a la que se corresponde esta dirección es *.bss*, que comienza en 0x555555558018. Esto es contrario a nuestra hipótesis: si *global* tiene un valor inicial, debe estar colocada en la sección *.data*.

Esto puede ser explicado por el comportamiento del compilador. En la sección *.bss* no se almacena ningún valor inicial, lo que permite ahorrar espacio en el fichero compilado. Al ejecutarse el programa, todas las variables de la sección *.bss* toman por defecto como valor inicial 0 (en memoria). El compilador aquí ha detectado que el valor inicial asignado a *global* era igual a 0, por lo que a efectos de la ejecución del programa resultaba equivalente colocarlo en la sección *.bss* en lugar de en la *.data*, y además en la primera se ahorraba almacenar el valor inicial en el binario, por lo que ha escogido la primera opción.

Comprobemos esta idea. Para ello, vamos a asignar a *global* un valor diferente a 0 (1, por ejemplo), y comprobamos cómo varía la ejecución volviendo a usar para ello la extensión desarrollada. Veamos la Figura 4.2.

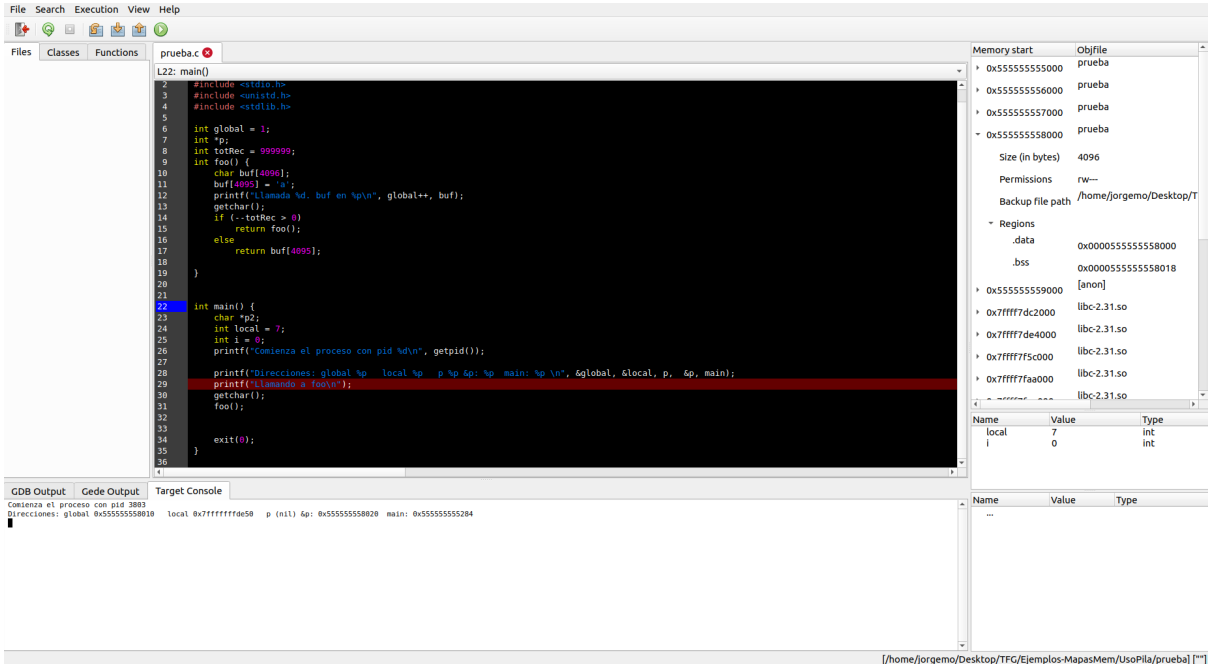


Figura 4.2: estado del programa con *global = 1*.

Ahora podemos ver que la dirección de *global* es 0x555555558010. La sección a la que se corresponde esta dirección es *.data*, que comienza en 0x555555558000. Podemos entonces comprender por qué anteriormente *global* se colocó en la sección *.bss* y ahora está en *.data*.

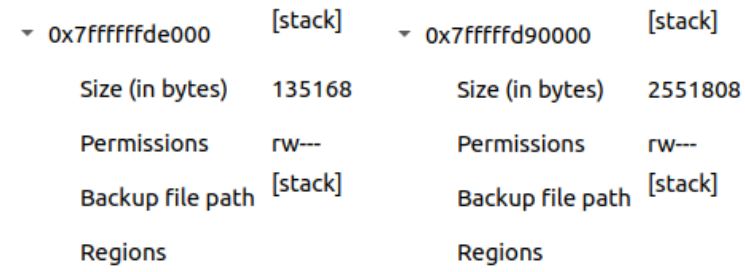
Uso de la pila

En este ejemplo vamos a demostrar parte del funcionamiento de la pila, tanto mediante variables locales como con llamadas a métodos.

Para lo primero, incluimos una variable en el método *main* y comparamos su dirección de memoria con la de la pila. La de la variable local es 0x7fffffffde50, y la de la pila, cuyo tamaño es de 135168B, es 0x7fffffde000, por lo que podemos afirmar que las variables declaradas localmente se ubican en la pila.

Por otro lado, para mostrar que en la pila se almacenan los contenidos de diferentes ámbitos, usaremos una función recursiva que crea e inicializa una variable en la pila antes de hacer su llamada recursiva. Así podremos ver cómo la pila incrementa su tamaño y se reubica cada vez que se queda sin espacio. En concreto, lo que se crea en cada llamada al método es un *array* de *char* de 4096 elementos, lo que equivale a 4096B.

Como podemos comprobar en la Figura 4.3, el tamaño de la pila es diferente en ambos casos, y la dirección de la pila cambia al tener esta que reubicarse. El número de llamadas recursivas ha sido, arbitrariamente, 615, aunque habría servido cualquier otro número que alcanzara a superar el espacio disponible de la pila.



(a) (b)

Figura 4.3: estado de la pila antes (a) y después (b) de llamar 615 veces al método recursivo.

Uso del *heap*

Para ejemplificar el funcionamiento del *heap*, en este programa se hace uso de varias llamadas a *malloc*. Primero se hace una llamada normal, y después se empieza a reservar espacio en memoria dentro de un bucle para forzar la redimensión del *heap*.

En primer lugar, se hace una llamada a *malloc* con un tamaño de 10**sizeof(int)*. Esto debería crear una nueva región anónima, correspondiente al *heap*. A la hora de comprobarlo, empezamos por el siguiente mapa de memoria (Figura 4.4):

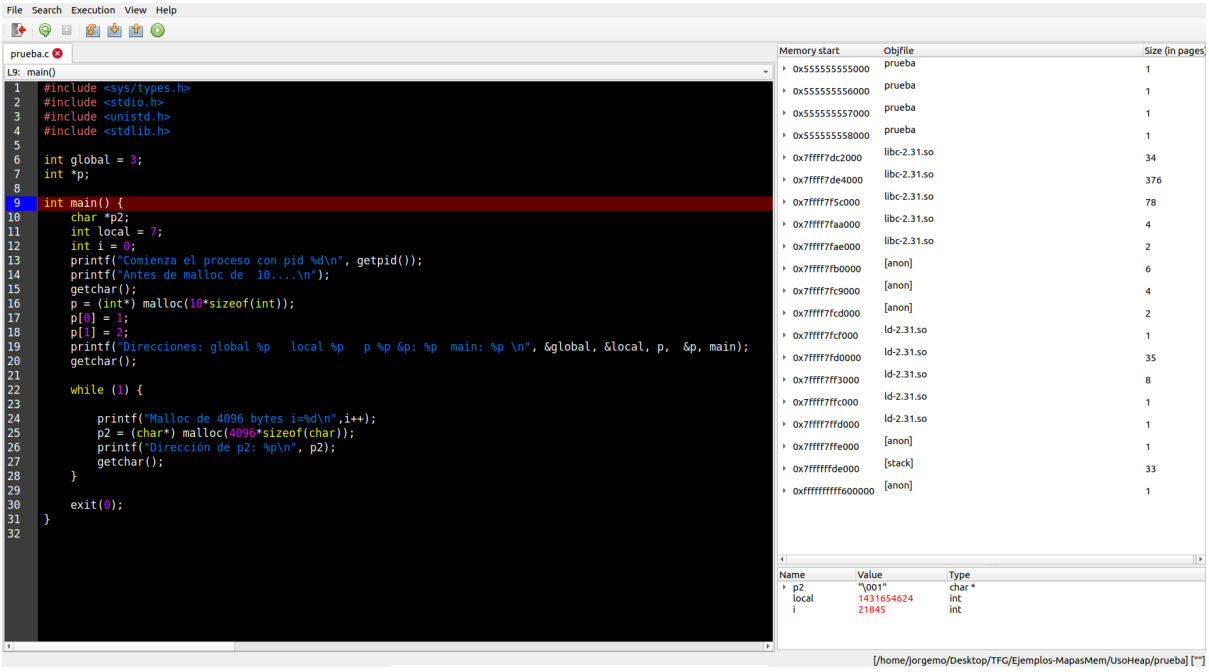


Figura 4.4: estado del programa antes de comenzar la ejecución.

Y al pasar por el primer *printf* nos encontramos con una nueva región de memoria, resaltada en el recuadro rojo de la Figura 4.5:

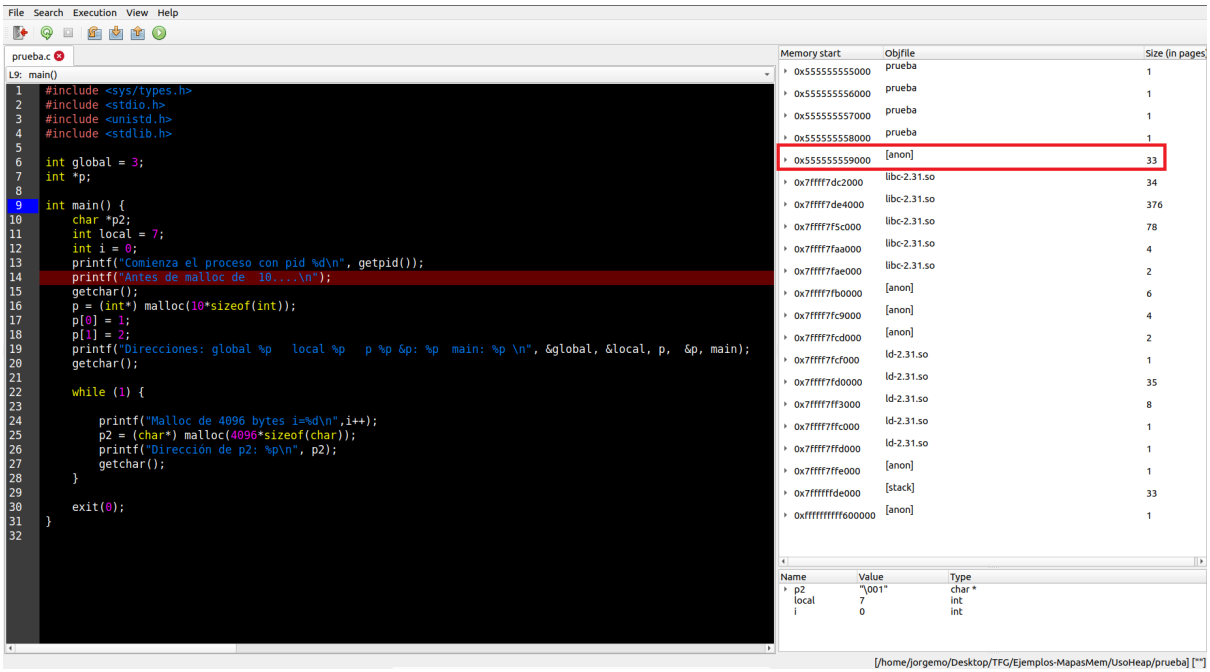


Figura 4.5: estado del programa tras pasar por el primer *printf*.

Esto podría considerarse inesperado, dado que en principio no hemos llegado a la zona de reservar espacio en memoria; no obstante, podemos asociar esta creación prematura de la región de memoria con la llamada a *printf*, que en algunas implementaciones puede reservar espacio en memoria.

Al continuar y pasar por el primer *malloc*, no sucede nada nuevo en el mapa de memoria: el *heap* ya ha sido creado previamente y no hace falta mapear más memoria. En concreto, el estado actual del *heap* es el reflejado en la Figura 4.6.

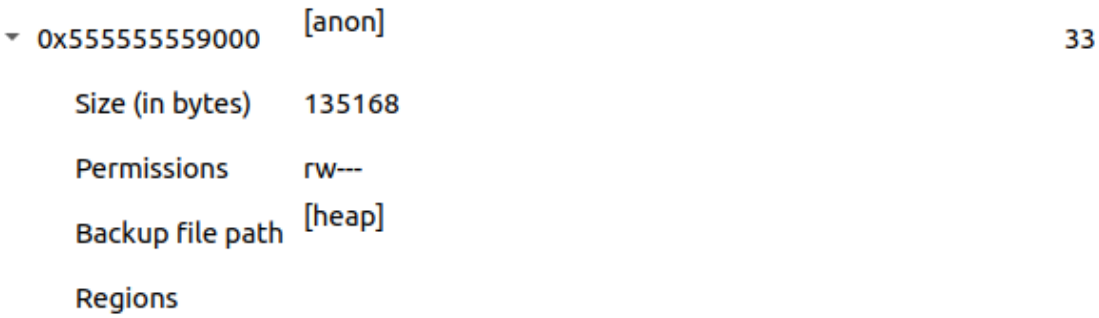


Figura 4.6: estado de la región del *heap* antes de llegar al bucle de *malloc*.

A continuación, entramos en el bucle para forzar la redimensión del *heap*. El contenido del bucle consiste en hacer una llamada a *malloc* con 4096B: por tanto, en cada llamada aumentamos el espacio ocupado en esa cantidad. Si ignoráramos el espacio que creemos que ha sido reservado por *printf* anteriormente, podemos calcular que la memoria necesaria excederá la disponible en la iteración:

$$\frac{135168B - 40B}{4096B} = 32,99$$

Es decir, en la iteración número 33 ocurrirá la redimensión. Podemos comprobar que así ocurre, como muestra la Figura 4.7.

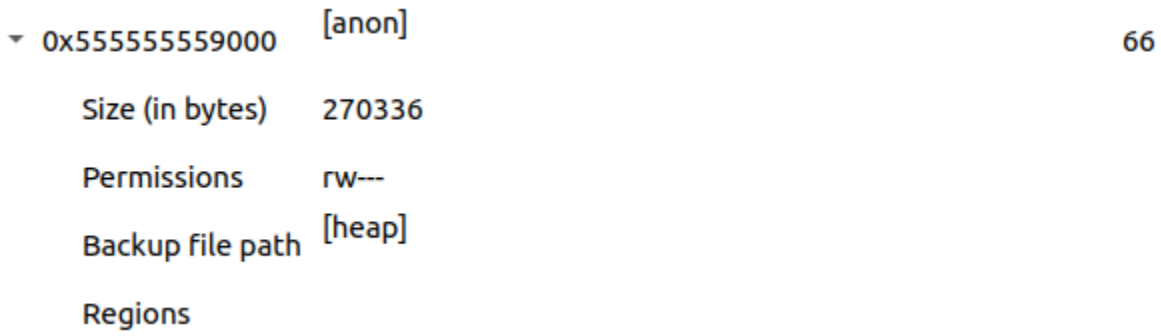


Figura 4.7: estado de la región del *heap* tras la iteración 33.

Como curiosidad, vemos que la redimensión no ha implicado necesariamente la relocalización de la región, es decir, no se ha movido de posición sino que se le ha asignado más tamaño.

Hebras

Para visualizar el mapa de memoria cuando el proceso que se está depurando es multihebra, vamos a utilizar un programa que primero crea variables que van en el *heap*, y posteriormente crea una hebra que se dedica a reservar continuamente espacio en el *heap*.

El resultado esperado es que cada hebra tenga su propia pila, pero compartan el *heap*, es decir, que la hebra que reserva continuamente espacio lo haga en la misma región que la hebra principal.

El estado del programa, antes de crear la hebra, es el de la Figura 4.8.

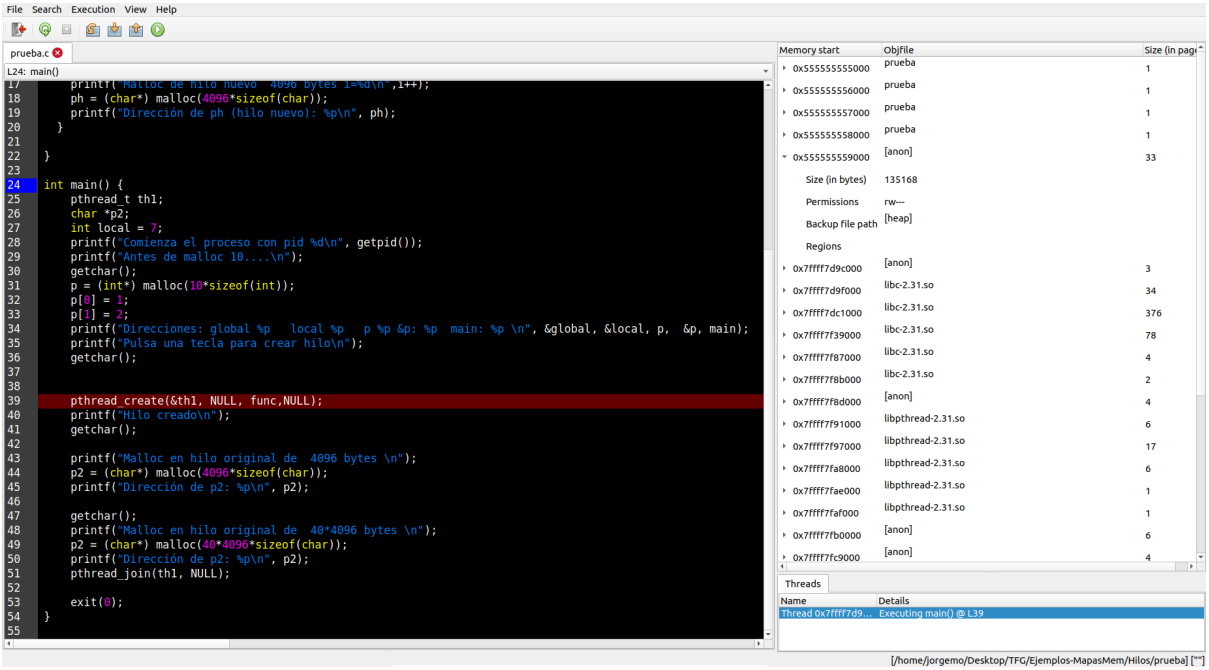


Figura 4.8: estado del mapa de memoria antes de crear la hebra secundaria.

Como podemos observar, ya existe un único *heap*, igual al que hemos visto en los ejemplos anteriores.

En el momento en el que se crea la hebra nueva, podemos ver que también se crean nuevas regiones anónimas, con permisos diferentes. Aún así, no encontramos una región llamada explícitamente *heap* (Figura 4.9), lo que concuerda con la teoría, que nos dice que las hebras comparten el *heap* y que su pila individual es necesaria para que las hebras puedan ejecutar llamadas a métodos.

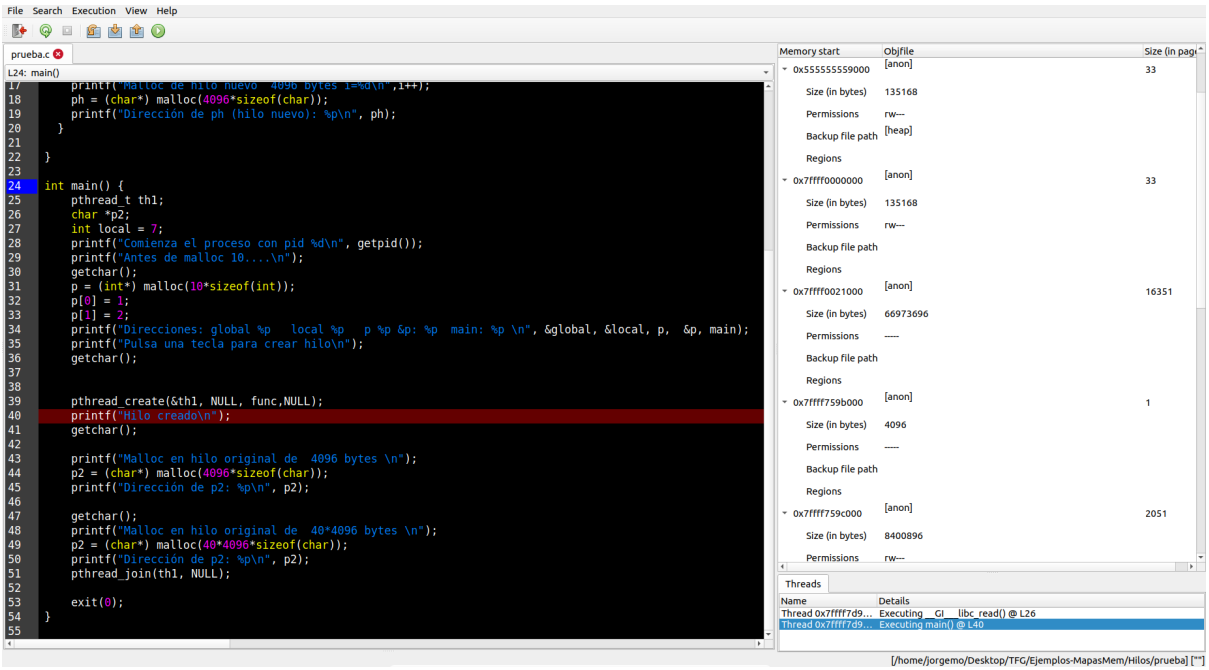


Figura 4.9: estado del mapa de memoria tras crear la hebra secundaria.

Uso de *mmap*

La función *mmap* [18] sirve para crear una nueva región en el mapa de memoria de un proceso. Esta función devuelve la dirección de memoria de la región creada como un puntero, y tiene el siguiente formato:

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Donde los diferentes parámetros significan lo siguiente:

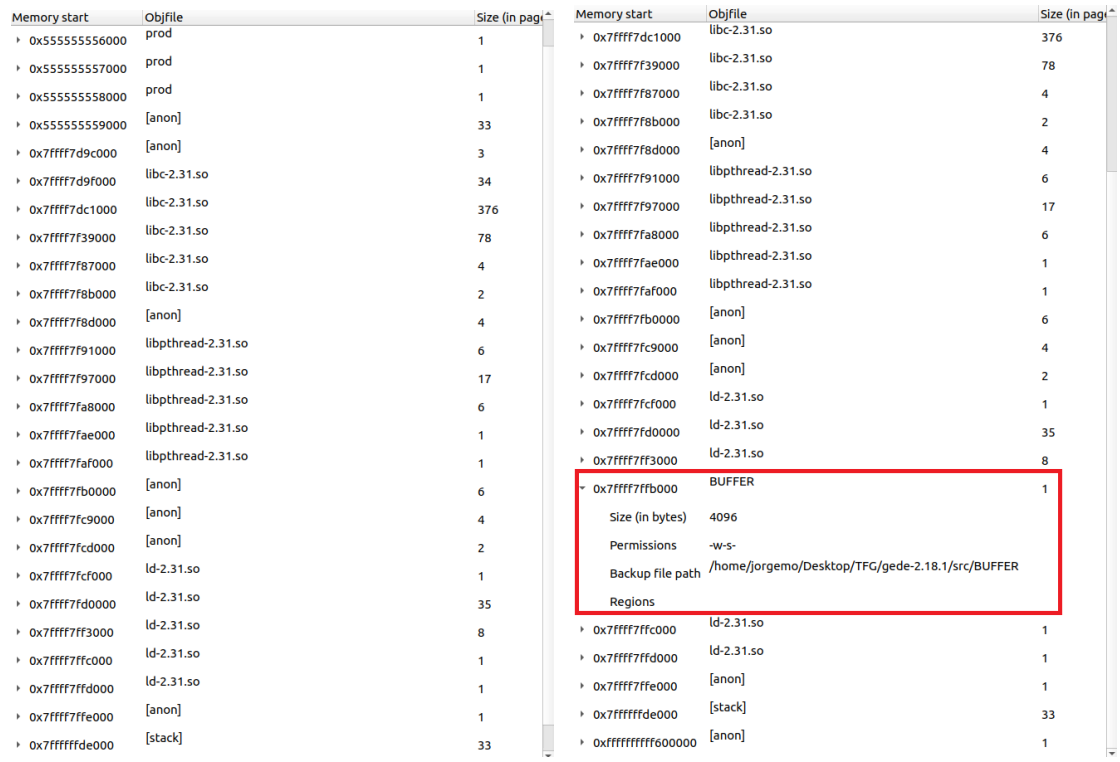
- *addr*: si este parámetro es NULL, el kernel escoge la dirección de memoria en la que crear la región. Si no, el kernel lo toma como una pista para escoger la dirección de memoria. En Linux, esto implica que intenta crearla en una página cercana.
- *length*: el tamaño de la región de memoria a crear.
- *prot*: la configuración de protección de la región de memoria. Se hace mediante uno o más *flags*.
- *flags*: se usa para indicar el modo de compartición de la región: privada o compartida, además de otros flags adicionales.
- *fd*: descriptor de fichero del fichero de respaldo.
- *offset*: desplazamiento con respecto al inicio del fichero de respaldo.

Se ha comprobado el funcionamiento de *mmap* mediante dos programas que se ejecutan a la vez: un ”escritor” y un ”lector” (en referencia a su interacción con el fichero de respaldo). Es importante recalcar que ambos han sido depurados simultáneamente porque la región que se muestra es compartida.

El escritor crea y abre para escritura un fichero llamado ”BUFFER”, para después generar, mediante *mmap*, una nueva región de 4096B en el mapa de memoria, usando ese fichero como respaldo. Esta nueva región es compartida y de escritura (especificado mediante los parámetros explicados anteriormente).

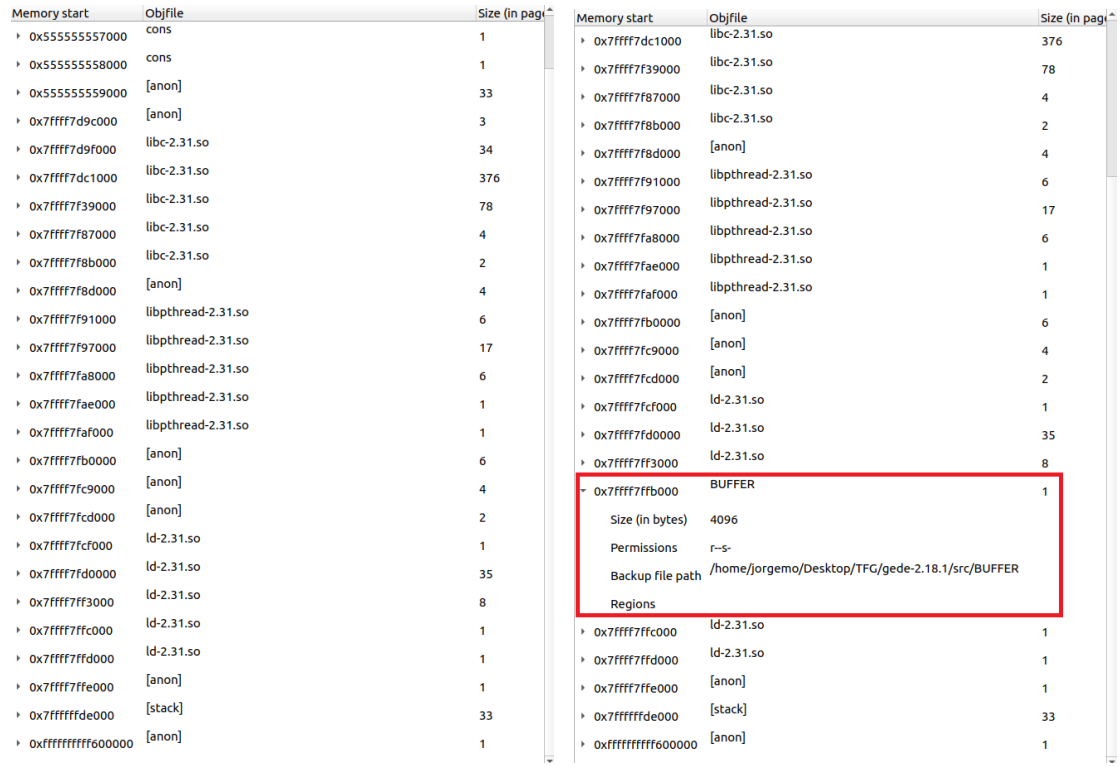
A continuación vemos el estado del mapa de memoria del proceso escritor antes de hacer *mmap* y después (Figura 4.10). Podemos apreciar que en la segunda imagen aparece la nueva región (resaltada mediante un recuadro rojo), que representa justo el comportamiento esperado de *mmap*: se crea una nueva región, de tamaño 4096B, con fichero de respaldo ”BUFFER”, compartida y con permisos de escritura.

El lector abre para lectura el fichero ”BUFFER” y después intenta crear la nueva región del mapa de memoria con *mmap*. Esta región, de las mismas características que la anterior (exceptuando permisos), en realidad ya ha sido creada por el escritor: es una región compartida. Podemos observar que esto se cumple en la Figura 4.11: la región que aparece tras hacer *mmap* tiene la misma dirección de memoria que la que hemos visto en el proceso del escritor, pero en este caso no tiene permisos de escritura y sí los tiene de lectura.



(a) (b)

Figura 4.10: mapa de memoria del escritor antes (a) y después (b) de ejecutar *mmap*.



(a) (b)

Figura 4.11: mapa de memoria del lector antes (a) y después (b) de ejecutar *mmap*.

Capítulo 5

Conclusiones

El proyecto cumple la función prometida: es una herramienta gráfica de depuración que permite visualizar las regiones de memoria. Si bien no ha acabado siendo desarrollada para Visual Studio Code, es fácilmente utilizable y bastante ligera. En su estado actual, el prototipo desarrollado en Gede tiene que ser utilizado como una herramienta independiente. Sin embargo, en un futuro podría portarse a entornos más completos y con uso más extendido, como es el caso de Visual Studio Code.

El trabajo admite bastantes mejoras que podrían ser objeto de modificaciones en el futuro. Algunas sugerencias de mejoras son:

- No limpiar por completo la lista de regiones cada vez que se detiene la ejecución, sino iterar a través de ella eliminando regiones que ya no existan, modificando las que hayan sufrido cambios y añadiendo las regiones que no aparecieran anteriormente.
- Colorear de rojo (u otro tono representativo) las regiones que hayan sufrido cambios o que se hayan añadido en la última parada. Esto permitiría encontrar de un vistazo las regiones más importantes en cada momento.
- Añadir a cada elemento de la lista las variables de código que están incluidas (si las hubiera). Esto se podría lograr obteniendo las direcciones de memoria de las variables y comparándolas con las direcciones que aparecen en las regiones de memoria. Para el caso específico de las variables de tipo puntero, se podrían añadir a la lista tanto la dirección en la que se encuentra el puntero como la dirección de la variable a la que apunta (el valor del puntero).

Por otro lado, también podemos mantener abierta la posibilidad de crear una extensión para Visual Studio Code. Si bien la idea fue abandonada porque los conocimientos requeridos y la investigación a realizar eran demasiado amplios, en este proyecto se dan algunas pistas acerca de cómo empezar a crear la extensión de depuración. Entre varios miembros, o con más tiempo de desarrollo, se podría lograr crear la extensión para Visual Studio Code, lo que presumiblemente aumentaría el éxito y el uso de la extensión. Como recordatorio, y aunque ya hablamos de ello en el *Capítulo 2: GDB y herramientas gráficas de depuración*, se trataría de crear un *Debug Adapter* basado en el actual para GDB, además de una extensión de Visual Studio Code (es decir, la ventana que represente la información).

Conclusions

The project fulfills the desired purpose: it is a graphic debugging tool that displays the memory regions. Although it has not been developed as an extension for Visual Studio Code, it is light and easy to use. In its current state, the prototype developed over Gede has to be used as a stand-alone application. However, in future iterations it could be ported to more complex environments with a wider use, such as the case of Visual Studio Code.

The project admits several improvements. Some of them would be:

- Take in account the list of memory regions that are already displayed and don't need to be updated, instead of clearing the whole list each time that the execution is stopped.
- Color in red (or any other representative color) the memory regions that have suffered changes or have been recently added. This would allow to find in a quick sight the most important regions at each time.
- Add the code variables to each element at the list that contains them. This could be done by obtaining the memory addresses of each variable and comparing them to the ones in the memory regions. For the special case of pointer variables, both the address of the variable and the variable that is pointed could be added to the list.

On the other hand, the possibility of creating a similar extension for Visual Studio Code is still open. Although the idea was initially abandoned because of the skill and time barrier, this project includes some hints in order to start a debugging extension for that IDE. With the help of a group of developers, or with more development time, the extension could be created, presumably increasing the amount of users. As a reminder, this work would consist on creating a Debug Adapter based on the current GDB one, and an extension that represents graphically the information received from the Debug Adapter.

Bibliografía

[1] Copyright Free Software Foundation, Inc., "GDB: The GNU Project Debugger." https://www.sourceware.org/gdb/, 2022. Último acceso el 26 de mayo de 2022.

[2] J. Henriksson, "Gede : Start." https://gede.dexar.se/, 2022. Último acceso el 26 de mayo de 2022.

[3] Microsoft Corporation, "Visual Studio Code - Code Editing. Redefined." https://code.visualstudio.com/, 2022. Último acceso el 26 de mayo de 2022.

[4] Eclipse Foundation, "Eclipse CDT | The Eclipse Foundation." https://www.eclipse.org/cdt/, 2022. Último acceso el 26 de mayo de 2022.

[5] Copyright Free Software Foundation, Inc., "GCC, the GNU Compiler Collection - GNU Project." https://gcc.gnu.org, 2022. Último acceso el 26 de mayo de 2022.

[6] Copyright Free Software Foundation, Inc., "Process Information (Debugging with GDB)." https://sourceware.org/gdb/current/onlinedocs/gdb/Process-Information.html, 2022. Último acceso el 27 de mayo de 2022.

[7] Copyright Free Software Foundation, Inc., "Files (Debugging with GDB)." https://sourceware.org/gdb/onlinedocs/gdb/Files.html, 2022. Último acceso el 27 de mayo de 2022.

[8] Oracle, "Oracle Solaris 11 | Oracle." https://www.oracle.com/solaris/solaris11/, 2022. Último acceso el 27 de mayo de 2022.

[9] The FreeBSD Foundation, "The FreeBSD Project." https://www.freebsd.org/, 2022. Último acceso el 27 de mayo de 2022.

[10] The NetBSD Foundation, "The NetBSD Project." https://www.netbsd.org/, 2022. Último acceso el 27 de mayo de 2022.

[11] M. Kerrisk, "pmap(1) - Linux manual page." https://man7.org/linux/man-pages/man1/pmap.1.html, 2021. Último acceso el 27 de mayo de 2022.

[12] C. Smith and Richard, más conocido como Bob, "gdbgui." https://www.gdbgui.com/, 2022. Último acceso el 28 de mayo de 2022.

[13] JetBrains s.r.o., "CLion: A Cross-Platform IDE for C and C++ by JetBrains." https://www.jetbrains.com/clion/, 2022. Último acceso el 28 de mayo de 2022.

[14] D. Seketeli and J. Jongsma, "Apps/Nemiver - GNOME Wiki!." https://wiki.gnome.org/Apps/Nemiver, 2019. Último acceso el 28 de mayo de 2022.

[15] E. Ifrah, "CodeLite • A free, Open Source, Cross Platform C,C++,PHP and Node.js IDE." https://codelite.org/, 2022. Último acceso el 28 de mayo de 2022.

[16] Microsoft Corporation, "Official page for Debug Adapter Protocol." https://microsoft.github.io/debug-adapter-protocol/, 2021. Último acceso el 28 de mayo de 2022.

[17] The Qt Company, "Qt | Cross-platform software development for embedded desktop." https://www.qt.io/, 2022. Último acceso el 28 de mayo de 2022.

[18] M. Kerrisk, "mmap(2) - Linux manual page." https://man7.org/linux/man-pages/man2/mmap.2.html, 2021. Último acceso el 27 de mayo de 2022.

[19] Copyright Free Software Foundation, Inc., "GDB/MI (Debugging with GDB)," 2022. Último acceso el 26 de mayo de 2022.

Jorge Moreno Martínez
28 de mayo de 2022
Ult. actualización 28 de mayo de 2022
LaTeX lic. LPPL & powered by TEFLON CC-ZERO