Secretos del depurador simbólico

Allan López
Universidad de Costa Rica, Escuela de Ciencias de la Computación e Informática,
San José, Costa Rica
allan2786@gmail.com

y

Yahaira Jerez
Universidad de Costa Rica, Escuela de Ciencias de la Computación e Informática,
San José, Costa Rica
yajecer@gmail.com

Abstract. When creating software some of the main problems that programmers have to face are detecting and fixing logical failures on them, as a matter of fact, a lot of time of the software projects is spent on debugging the programs. Thanks to those needs symbolic debuggers were invented.

While most programmers in the world know how to use these tools, most of them do not know how they are implemented, or its internal architecture or structure. That's why this article will give a rapprochement to symbolic debuggers, starting with its definition and continuing with an explanation of the algorithms used to implement the most important functionalities of debuggers and a quick view over a very generic architecture of a debugger.

Keywords: Symbolic debugger, kernel, breakpoint, watchpoint.

Resumen. A la hora de crear programas computacionales unos de los principales problemas que se presentan son los de detectar y corregir errores lógicos en éstos, incluso se llega a utilizar mucho tiempo de los proyectos de *software* depurando los programas. Gracias a esas necesidades se inventaron los depuradores simbólicos.

Si bien la mayoría de los programadores del mundo saben cómo utilizar dichas herramientas, muchos no saben cómo se implementa un depurador, ni su arquitectura o estructura. Por lo que éste artículo dará un acercamiento a los depuradores simbólicos, comenzando con su definición para continuar con una explicación de los algoritmos que se utilizan para implementar las funcionalidades más importantes de los depuradores y un vistazo sobre una arquitectura bastante genérica de los depuradores.

Palabras clave: Depurador simbólico, núcleo, *breakpoint*, *watchpoint*.

1 Introducción

En la actualidad el desarrollo de aplicaciones de sistemas informáticos es muy grande, la limitación que tiene un compilador al no poder ayudar a corregir errores lógicos en los programas que se construyen es resuelta en lo posible con los depuradores simbólicos con los cuales se pueden encontrar dichos errores.

Los depuradores simbólicos, como su nombre lo indica, ayudan a encontrar y así eliminar los errores en el código fuente de los programas computacionales mientras se están elaborando. El código fuente es el código escrito en símbolos por los programadores para generar un programa específico a partir de él. Un depurador es una herramienta de *software* que permite a los programadores seguir paso a paso (instrucción por instrucción) la ejecución del programa en cuestión. Esta herramienta resulta ser muy útil ya que permite comprender de manera detallada la lógica y funcionamiento del programa, además de que permite localizar errores en el código fuente que de otra forma podrían ser muy difíciles de ubicar.

Durante el análisis del programa, por medio del depurador el programador puede verificar el estado de las variables y del programa en general, para determinar si los valores son correctos. Es así como se logran encontrar muchos errores y corregirlos.

No todos los usuarios de los depuradores saben a ciencia cierta su funcionamiento e implementación, en este artículo se presentará un breve acercamiento a éstos temas.

2 Los Depuradores Simbólicos

Si buscamos en un diccionario la palabra depurar podemos ver que es un sinónimo de limpiar o purificar. En general, la palabra depurador se refiere a cualquier aparato, sistema, o proceso que sirva para limpiar o purificar alguna sustancia o elemento. Los depuradores simbólicos, como su nombre lo indica, ayudan a encontrar y así eliminar los errores en el código fuente de los programas computacionales mientras se están elaborando. El código fuente es el código escrito en símbolos por los programadores para generar un programa específico a partir de él. En las siguientes secciones se ampliará sobre las funciones y características de los depuradores.

Definición

Un depurador es una herramienta de *software* que permite a los programadores seguir paso a paso (instrucción por instrucción) la ejecución del programa en cuestión. Esta herramienta resulta ser muy útil ya que permite comprender de manera detallada la lógica y funcionamiento del programa, además de que permite localizar errores en el código fuente que de otra forma podrían ser muy difíciles de ubicar.

Durante el análisis del programa, por medio del depurador el programador puede verificar el estado de las variables y del programa en general, para determinar si los valores son correctos. Es así como se logran encontrar muchos errores y corregirlos.

3 Funcionamiento de un depurador simbólico

La parte más importante de un depurador es su núcleo de control de ejecución. Para llevar a cabo ese control el depurador debe tomar el control sobre el programa que está siendo depurado para poder determinar su estado, establecer *breakpoints*, correr el proceso y terminarlo. En las siguientes secciones se examinarán los algoritmos que siguen los depuradores para realizar esas funciones.

3.1 Creación del proceso

Al iniciar la depuración de un programa, lo primero que debe realizar el depurador es la creación de un proceso para el programa a depurar, o adjuntarlo a un proceso ya existente. Esto se realiza por medio de los correspondientes llamados al Sistema Operativo (SO).

Después de haber hecho eso, lo siguiente es preparar al programa a depurar para poder correrlo bajo el control del depurador. Entonces el SO carga el nuevo proceso para que se ejecute su primera instrucción y esté listo para la ejecución controlada por el depurador. De ahí en adelante el depurador simplemente espera por alguna notificación del SO cuando ocurre algún evento importante que ocasiona que el programa depurado se detenga para ser examinado. Por ejemplo, la primera notificación que recibe el depurador es la que le indica cuando ya se estableció lo necesario para la depuración al inicio y que el programa a depurar está listo para ser controlado por el depurador.

Una vez que el depurador tiene el control sobre el programa, se tiene la característica especial de que las excepciones que pueda generar el programa serán notificadas primero al depurador, en lugar del proceso normal de manejo de excepciones que incluye primero un manejo por medio del programa mismo y en segundo nivel el respectivo manejo por parte del SO.

Una idea similar se maneja para los *breakpoints* que establece el usuario del depurador, donde el proceso de notificar al depurador y detener la ejecución se realiza por medio de instrucciones que están definidas en la arquitectura de los CPU.

3.2 Establecimiento de los *breakpoints*

Normalmente existen al menos dos niveles de representación para los *breakpoints*, que son el lógico y el físico. Los *breakpoints* lógicos (básicamente los puestos por el usuario) son los que están asociados con algún punto en el código fuente. Los *breakpoints* físicos (relacionados directamente con instrucciones ejecutables de la máquina) son los puntos en el espacio de texto donde realmente se escriben las instrucciones *hardware* para establecer un *breakpoint*. Es en éste nivel físico donde se debe almacenar la instrucción original que debe ser reemplazada si el

breakpoint es removido. Además el nivel lógico es el responsable de representar un *breakpoint* como resuelto (que está mapeado con una dirección física) o como no resuelto (como cuando se establece un *breakpoint* en un módulo que será cargado hasta más adelante en la ejecución). También existen tipos especiales de *breakpoints* llamados *breakpoints* condicionales, que pueden detener la ejecución o no detenerla dependiendo del valor de una condición asociada a él. Esas condiciones son almacenadas en el nivel lógico.

Además pueden existir relaciones de muchos a uno entre *breakpoints* lógicos y físicos. Ésto permite a los usuarios establecer dos *breakpoints* distintos (o con diferentes condiciones) en un mismo punto del código fuente que están mapeados con la misma ubicación física. Para llevar el control sobre este mapeo, la manera más efectiva suele ser la de mantener dos estructuras separadas para los *breakpoints* lógicos y físicos donde los nodos de cada lista mantienen una dirección que sirve de enlace entre ambas listas y entre ellas pueden ocurrir dos tipos de mapeo. Primero un mapeo hacia abajo (de lógico a fisico) ocurre cuando se establecen, eliminan o modifican los *breakpoints*, con lo que se obtiene una dirección física que servirá como *token* de búsqueda en la lista de *breakpoints* físicos. El mapeo hacia arriba ocurre cuando un *breakpoint* se dispara debido a la ejecución de una instrucción de *breakpoint* por parte del programa depurado, y resulta en una dirección física del SO que mapea de forma única a un nodo en la lista de *breakpoints* físicos. Esa dirección se usa luego para buscar en la lista de *breakpoints* lógicos para buscar todos los que se mapean con esa dirección.

A continuación se expondrá el algoritmo básico para establecer *breakpoints*, basado en las estructuras descritas antes. Antes de que inicie el algoritmo, el usuario especifica la existencia de un *breakpoint* en una línea del código fuente, luego el algoritmo comienza para mapear el *breakpoint* a una ubicación física de *breakpoint*.

Entrada Nombre del archivo y número de línea o el *offset* del archivo fuente.

Salida Ubicación física del *breakpoint* o indicación de error.

Método Mapeo del nombre del archivo mas el número de línea hacia una dirección física usando una tabla de símbolos, *breakpoint* lógico, y *breakpoint* físico.

1. Solicitar al controlador de la tabla de símbolos mapear la información del nombre de archivo y número de línea en la dirección física. Notificar si pertenece a un módulo no cargado aún.

2. Crear el objeto de *breakpoint* lógico incluyendo la información anterior.

3. Crear el objeto de *breakpoint* físico o incrementar el campo contador de referencias del objeto si ya existía.

4. El controlador de *breakpoints* físicos debe insertar la instrucción de *breakpoint* y salvar la instrucción original en esa ubicación.

Así es como los *breakpoints* quedan establecidos y listos para la ejecución del programa

3.3Ejecución del programa a depurar

Una vez que el usuario ha establecido sus *breakpoints* en el programa y desea iniciar la depuración hasta el primer *breakpoint*, seleccionará la opción de depuración y el depurador iniciará la ejecución del programa a depurar. La ejecución puede ser de dos formas: “correr el programa a máxima velocidad hasta que ocurra algún evento de depuración” o “ejecutando solo la primera instrucción y luego se genera un evento de depuración”, esto para poder ejecutar el programa instrucción por instrucción desde el inicio. Para manejar ambos casos lo que ocurre es lo siguiente: el depurador es el proceso activo e invoca al SO para que inicie el proceso del programa a depurar. El control entonces pasa al SO para atender la petición, obteniendo el nuevo proceso listo para ejecutarse y realiza un cambio de contexto con el nuevo proceso. El programa a depurar pasa a ser el proceso activo y se ejecuta de acuerdo a los algoritmos de calendarización del SO. Mientras tanto el proceso del depurador se mantendrá en estado de espera mientras el proceso del programa depurado se detiene. Al detenerse, el SO guarda el contexto del proceso cuando se detuvo y el control vuelve al depurador. A continuación el depurador busca los datos de por qué se detuvo, dónde se detuvo y en general cuál era el contexto del proceso. Mucha de esa información se encuentra en el contexto almacenado y accesible desde un *buffer* administrado por el SO, que almacena los valores de los registros cuando el programa se detuvo.

Son varios los tipos de eventos que generan los programas depurados durante su ejecución y se manejan de formas distintas, las cuales se explicarán en la siguiente sección.

3.4Eventos de depuración generados en la ejecución

Para lograr el control de la ejecución de un programa por parte del depurador, se realizan ciertas acciones al ocurrir algún evento en el programa. Esos eventos pueden ser tanto *breakpoints*, eventos de un solo paso en la ejecución (*single-step*), creación y finalización de hilos, creación y finalización de procesos, eventos de acceso a datos (*watchpoints*), la carga y descarga de módulos (DLL´s), eventos de excepciones, entre otros y la forma de atenderlos se explica a continuación.

Breakpoints* y eventos *single step

Lo primero que debe realizar un depurador al encontrarse con un evento de *breakpoint* es encontrar cuál fue específicamente el *breakpoint* que causó el evento. Eso se lleva a cabo gracias a una lista de *breakpoints* que mantiene el depurador, y gracias al contador del programa (*program counter*) que queda almacenado en el contexto del proceso al detenerse. Esa dirección de la instrucción en la que se detuvo el programa es buscada en la lista de *breakpoints* para dar con él. Sin embargo, más de un *breakpoint* puede coincidir con la dirección ya que los depuradores utilizan *breakpoints* internos como ayuda en sus algoritmos de ejecución. Una vez encontrada la dirección con la que se debe continuar, el control pasa al depurador para continuar con la ejecución del programa.

Creación y finalización de hilos

Una de las funciones más importantes de los depuradores es la facilidad que ofrecen para llevar el control sobre lo que pasa en un programa multihilos. Para esto, cuando el programa depurado crea o finaliza un hilo debe notificarse instantáneamente al depurador para que refleje en su interfaz lo ocurrido. Además el depurador mantiene ciertas estructuras de datos específicas para cada hilo y al crear o finalizar uno de los hilos, esas estructuras serán modificadas por el depurador. Así es como el depurador mantiene la información de contexto crítica de cada hilo para proveer tan importante funcionalidad. El depurador, que se encuentra en pausa, recibe entonces la notificación de la creación o eliminación del hilo y realiza la respectiva limpieza de la estructura de datos y la notificación al usuario por medio de la interfaz. Después el núcleo simplemente reinicia el programa depurado.

Si el hilo que se está eliminando es el hilo principal de ejecución, se requiere un proceso algo diferente, ya que el resto de estructuras de datos que mantiene el depurador deben reflejar el hecho de que el proceso dejara de existir.

Creación y finalización de procesos

Para manejar un programa multiprocesos es necesario que el primer proceso creado (el proceso principal) notifique al depurador de la creación de cada proceso nuevo que cree. Para poder llevar el seguimiento de cada nuevo proceso el depurador debe tener la capacidad de manejar adjuntos, y así poder solicitar que se adjunte un proceso a otro existente y seguir controlando el nuevo proceso de la misma manera que el principal. Normalmente, después de adjuntar un proceso nuevo se producen ciertas notificaciones como por ejemplo la de creación del proceso, hilos creados, y módulos cargados. Los procesos continúan normalmente hasta que surja una notificación de eliminación del proceso, permitiendo al depurador realizar una limpieza de las estructuras de datos relacionadas con el proceso eliminado. Además si el proceso a eliminar es el proceso principal del programa, esa notificación de eliminación del proceso será la última que reciba el depurador y se pierde el control sobre el proceso.

Eventos de acceso a datos (*watchpoints*)

La característica que ofrecen los depuradores de poder acceder datos del programa es también una de las funciones más importantes de los depuradores, ya que la corrupción de datos es un tipo de "pulga" muy común y difícil de aislar. Las notificaciones para este tipo de eventos suelen ser similares a las de los *breakpoints*. Para detectar la diferencia entre ambas, el depurador debe tomar el estado completo del programa que está siendo depurado y se detuvo, y revisar ciertos registros *hardware* de estado que especifican la causa de la excepción. Las notificaciones de *watchpoints* indican el acceso a una dirección o región de memoria, y el *program counter* (PC) indicará la ubicación de la instrucción transgresora o la instrucción siguiente a la transgresora. Esto podría ocasionar un detenimiento del depurador y una notificación al usuario del depurador sobre un intento de modificar una variable que él deseaba ver. Así el usuario descubre la corrupción de una posición de memoria o variable y podrá determinar dónde y cuándo ocurrió la corrupción.

Eventos de carga y descarga de módulos

Este tipo de notificaciones se llevan a cabo cada vez que el programa depurado carga o descarga una librería de carga dinámica (*Dynamic Load Library* DLL). Esta puede ser

información importante para el depurador en caso de que el usuario quiera depurar el código contenido en algunas de esas librerías. Además el usuario pudo haber establecido un *breakpoint* en el código fuente que se encuentre en una DLL que aún no ha sido cargada. En estos casos, en el momento en que ocurre la notificación de carga del DLL, el depurador debe resolver esos *breakpoints* e inmediatamente insertarlos en la reciente DLL cargada. De forma similar, cuando una DLL es descargada, la correspondiente información en la tabla de símbolos debe ser notada por el depurador y cualquier *breakpoint* establecido ahí debe marcarlo como sin resolver por si la DLL se deba cargar de nuevo más adelante. Al terminar de procesar la tabla de símbolos y los *breakpoints*, el depurador continúa con la ejecución normal del programa hasta recibir la siguiente notificación.

Eventos de excepciones

Los verdaderos eventos de excepciones son fallas en el programa como accesos ilegales a memoria. Estos son las principales pulgas que un depurador debe ayudar a encontrar y corregir. Este tipo de excepciones suelen ocasionar un detenimiento instantáneo del programa y un reporte al usuario. Normalmente son problemas tan serios que el usuario no podrá continuar con la ejecución del programa por más que lo desee. Algunas clases de excepciones no son tan serias y permiten continuar con la ejecución. Varios Sistemas Operativos permiten diferenciar entre los grados de selectividad de una excepción de depuración.

4 Arquitectura de los depuradores simbólicos

En esta sección se expone una arquitectura típica que encaja con la mayoría de depuradores de tipo simbólico, basado en interfaz gráfica de usuario (GUI), y orientados a aplicación. En la siguiente figura se muestra una representación gráfica de la arquitectura del depurador, donde podemos notar que el depurador debe estar directamente “conectado” con el Sistema Operativo en el nivel más bajo, hasta llegar a las capas externas que representan las interfaces que muestran al usuario las funcionalidades del depurador.

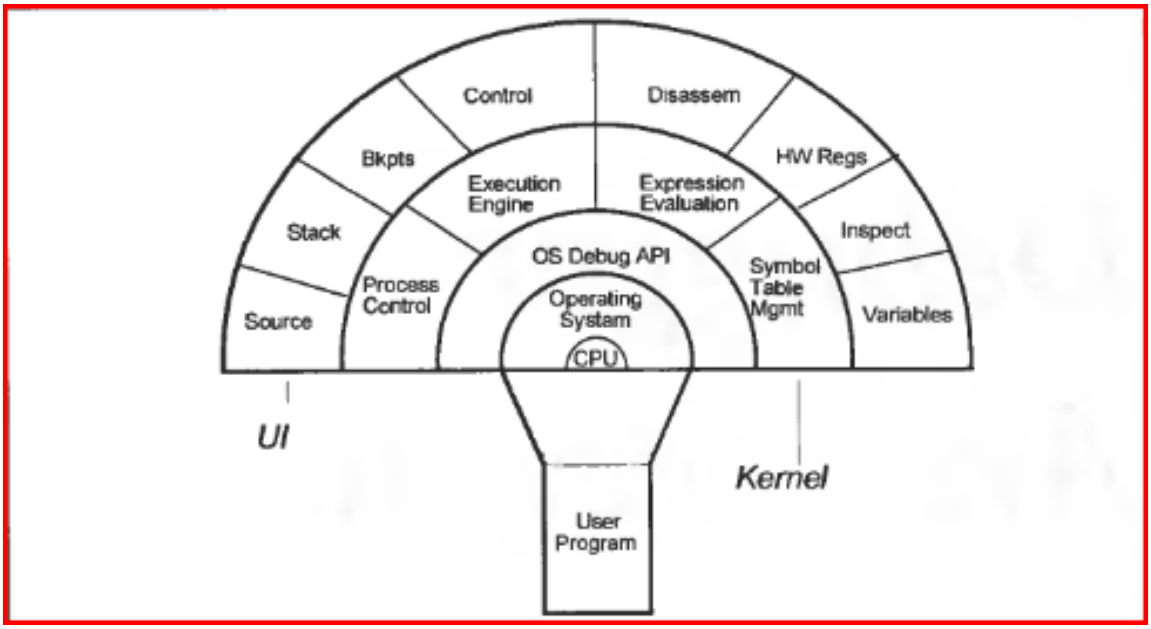

Figura 1: Arquitectura típica de un depurador. [1].

Cada bloque en la capa externa de la arquitectura representa una vista diferente que se puede presentar al usuario y estados en los que puede estar el depurador.

4.1La capa de la Interfaz de Usuario

Esta es la capa externa del depurador y es la que varía entre los distintos depuradores pero la mayoría poseen las siguientes vistas básicas:

Source View

Es una de las vistas más importantes del depurador. En ella se despliega el código fuente del programa y es donde se le da al usuario la impresión de la ejecución paso a paso de sus instrucciones. Es también el editor para el usuario.

Stack View

Vista en la que se muestra un tipo de pila de los llamados a funciones que ha realizado el programa desde que inició la depuración. Importante para entender el flujo de acciones de un programa y rastrear errores.

Breakpoints View

Presenta una descripción general de todos los *breakpoints* establecidos por el usuario en cualquier lugar de los procesos activos. Los *breakpoints* permiten al usuario controlar la

ejecución del programa y especificar cómo y dónde debe detenerse la aplicación para seguir examinando desde ese punto.

CPU View

Es una vista necesaria ya que muchas veces no es suficiente la vista del código fuente para entender el comportamiento de un programa. Esta vista entonces permite ver las correspondientes instrucciones de bajo nivel del programa como instrucciones en ensamblador, los registros del CPU y sus valores, un panel del volcado de memoria, un panel de banderas, y un panel con la pila del *hardware*.

Variables View

La vista de variables se relaciona mucho con el *browser* que a su vez está relacionado con la tabla de símbolos generada por el compilador. El *browser* permite observar funciones, tipos, variables, y clases de un programa fuente escrito en C++. Los valores de las variables del programa son críticos para el entendimiento del origen de un defecto del programa.

Inspector/Evaluator

El inspector le permite al usuario examinar la estructura de objetos complejos del programa. El evaluador es similar al inspector, pero permite el ingreso de cualquier expresión legal del lenguaje de programación, y que será evaluada por el depurador en el contexto actual del programa ejecutándose.

4.2Núcleo (*kernel*) del depurador

Ésta capa del depurador es la que sirve a todas las vistas de la capa de interfaz de usuario, y la que controla el proceso. Desde ésta capa la aplicación que deseamos depurar es para el SO un simple proceso.

Process Control

El depurador puede crear un proceso nuevo al inicio de la depuración o se le puede adjuntar a un proceso ya existente y empieza a ser el proceso a depurar. Al final una sesión de depuración el núcleo del depurador debe finalizar el proceso y desasociarlo del depurador.

Symbol Table Management

Como podemos ver el *kernel* también es responsable del acceso a la tabla de símbolos. La tabla de símbolos debe ser consultada para determinar un mapeo entre las sentencias fuente y los bytes de dirección de las instrucciones ejecutables.

Expression Evaluation

Es el proceso de usar el depurador para evaluar variables y aplicar operadores y llamados a funciones según las expresiones especificadas por el usuario, lo cual es útil para averiguar cuál sería el valor de la expresión se estuviera en el programa.

Execution Engine

Sirve para ejecutar llamados a funciones que se requieran en una evaluación de una expresión como las descritas anteriormente.

4.3Interfaz del Sistema Operativo

Cuando el *kernel* del depurador necesita acceder al programa depurado, debe usar una colección de rutinas provistas por el Sistema Operativo. Esa API (*application program interface*) es una porción del Sistema Operativo y provee las funciones básicas para crear procesos depurables, leer y escribir la memoria de ese proceso, y controlar la ejecución de ese proceso.

5 Conclusiones

Como vimos en el artículo los procesos para realizar la depuración de un programa no es algo trivial e implican mucho control, comunicación y cooperación entre el depurador y el Sistema Operativo, que se ven incrementados al proporcionar la funcionalidad de depurar programas multihilos. Además el depurador en sí está dividido en varios módulos que colaboran a realizar las tareas y a desplegar al usuario las diferentes funciones de vista de la pila, de la CPU, etc. Por lo que la implementación de un depurador puede no ser tan simple como nos pudo parecer en algún momento.

6 Bibliografía

- http://es.wikipedia.org/wiki/Depurador
- http://www.sg.com.mx/content/view/573
- [1] Rosenberg, J.B. How Debuggers Work Algorithms, Data Structures, and Arquitecture. Vol. 10, No. 2, (Winter 1992), pp.453-469.
- http://epub.ub.uni-muenchen.de/1429/
- http://www.gnu.org/software/gdb/gdb.html
- http://sourceware.org/gdb/download/onlinedocs/