# pDI-Tools

## Mecanismo de interposición dinámica de código

**Gerardo García Peña**

**Jesús Labarta**

**Judit Giménez**

**pDI-ToolsMecanismo de interposición dinámica de código**

por Gerardo García Peña, Jesús Labarta, y Judit Giménez

# Tabla de contenidos

# Lista de tablas

# I. Introducción

# Prefacio

La instrumentación de código es una tecnología que permite analizar o variar el comportamiento de un programa durante su ejecución. Mediante este análisis se pueden descubrir muchas cosas del programa objetivo o incluso del sistema operativo que se está usando.

Hay muchísimas técnicas de instrumentación de código, desde las más sencillas como sería instrumentar estáticamente el mismo código fuente y luego compilar, hasta complejas técnicas de instrumentación en tiempo de ejecución (técnicas conocidas como instrumentación dinámica).

Cuando un programa se carga en memoria este suele requerir la cooperación de otras librerías para poder realizar su trabajo. Al conjunto del programa junto a las librerías se les conoce como objetos compartidos dinámicamente (DSO). Cuando el sistema operativo ejecuta un programa carga todos estos DSO en memoria y, mediante unos mecanismos definidos por el estándar ELF, establece como se han de llamar estos entre si.

En el siguiente documento presentamos el proyecto pDI-Tools. pDI-Tools es un mecanismo creado para poder realizar sesiones de instrumentación sobre un binario, incluso sin información de depuración del mismo, dinámicamente. Este programa interpone el código mediante mecanismos basados en interceptar las llamadas entre objetos compartidos dinámicamente. Estos mecanismos se basan en explotar la información de ELF y así interponer código creando un entorno apto para desarrollar herramientas de instrumentación.

pDI-Tools fue diseñado basándose en los mecanismos de interposición de DITools, herramienta desarrollada por el CEPBA para Irix sobre MIPS®. pDI-Tools, a pesar de inspirarse en ella, se basa en un nuevo diseño orientado a obtener una gran portabilidad sin apenas sacrificar rendimiento y eficiencia.

# 1. Introducción a la instrumentación

Aunque el programador haga un gran esfuerzo por obtener un código óptimo y correcto, hay muchos factores externos e internos que dificultan esta labor: ordenadores complejos, recursos cuyo acceso no es uniforme, sistemas operativos de tiempo compartido, sistemas de comunicación con singularidades, miles de opciones de configuración (y de configuraciones posibles) y un largo etcétera, que añadido a la complejidad del software y/o de los datos con los que trabaja hacen que el comportamiento de un programa sea algo muy difícil de predecir. Y nos podemos encontrar en alguna de estas situaciones:

- El rendimiento del software desarrollado no se corresponde en absoluto con el rendimiento teórico calculado para la máquina.
- Cuesta demasiado imaginar la ejecución del programa y por lo tanto determinar dónde están exactamente los verdaderos cuellos de botella.
- Ocurren cosas raras en un determinado sistema y en otros no.
- Sabemos que los algoritmos que usamos ya son óptimos, pero aún así creemos que se puede mejorar la velocidad, ¿cómo averiguar dónde debemos optimizar?
- Aún peor: no se dispone del código fuente, sólo de una caja negra que está dando un rendimiento pésimo debido a alguna característica del sistema que no se consigue identificar.
- O simplemente: no se sabe que está pasando ahí dentro, pero pasa algo inesperado.

La instrumentación de código es una herramienta ideal en estas situaciones: nos permite analizar la ejecución del programa en los puntos clave sin llegar al extremo nivel de detalle e interferencia de las herramientas de depuración, sin sufrir la rigidez de las herramientas de “profiling”. Además la

instrumentación de código, al poder variar fácilmente la ejecución del programa, no se limita tan sólo al análisis de software: sus aplicaciones son virtualmente infinitas.

No obstante la flexibilidad ofrecida varía mucho según el tipo de instrumentación usada. A grandes rasgos hay los siguientes tipos:

- **Instrumentación estática.** Se puede realizar de dos maneras:
  - **Sobre el código fuente.** Consiste en insertar el código de instrumentación en el código fuente. Es necesario compilar de nuevo para obtener una versión instrumentada de la aplicación.
  - **Sobre el binario.** Consiste en usar una herramienta especial que inserta el código de instrumentación en el binario ya compilado. No es necesario recompilar, pero si puede ser necesario disponer de información de depuración o similar.
- **Instrumentación dinámica.** Consiste en alterar el código del ejecutable (o al menos las tablas de símbolos) en tiempo de ejecución para redirigir y controlar la ejecución del programa. No es necesaria la información de depuración aunque si se dispone de ella es posible aprovecharla.

Instrumentar estáticamente el código fuente es la técnica más potente ya que permite realizar cualquier tipo de estudio sobre el programa; en contrapartida, al introducir cambios en los fuentes, con la recompilación, a veces, se obtiene binarios muy distintos del original, obteniendo mediciones bastante trastocadas al instrumentar[1].

La instrumentación estática sobre el binario permite instrumentar con menos potencia que con la técnica anterior pero a cambio no altera tanto el resultado de la compilación e introduce menos ruido en la instrumentación. No obstante sigue alterando el ejecutable final (lo cual es un problema en caso de que el ejecutable vaya firmado y/o tenga algún mecanismo de protección). Otro problema es que esta técnica suele requerir información de la que no se suele disponer (código fuente, información de depuración, tablas de símbolos, ...).

La instrumentación dinámica se realiza en tiempo de ejecución y sin modificar el binario. Esta instrumentación se puede realizar de diversas formas: modificando el código en memoria (este caso es similar a la instrumentación estática pero en tiempo de ejecución) o trabajando directamente en las tablas de enlace dinámico del programa. La primera solución es algo lenta aunque permite instrumentar con mucha potencia el código, sin embargo necesita información detallada sobre el ejecutable. La segunda es el tipo de instrumentación más rápida, menos intrusiva y más ágil pero se limita tan sólo a las llamadas entre objetos compartidos (entre librerías y desde el ejecutable a librerías), pero a cambio no requiere ningún tipo de información de depuración.

# 2. DITools

Como ya se ha comentado, DITools es un software desarrollado por el CEPBA que implementa un mecanismo de interposición dinámica de código orientado a redirigir las llamadas a funciones entre *Dynamic Shared Objects* o DSO's.

Se presenta como un conjunto de objetos ELF que se deben cargar, justo antes de comenzar la ejecución del programa principal, en el espacio de ejecución de la aplicación que se desea instrumentar.

Una vez instalado en el entorno de ejecución de la aplicación, DITools toma el control e instala un conjunto de interposiciones contenidas en unos ficheros que contienen "comandos de interposición de código".

DITools no decide el código interpuesto, de eso se encarga la aplicación de instrumentación construida sobre él. Por lo tanto, una vez instaladas estas interposiciones, DITools no vuelve a tomar el control

hasta la finalización del programa, momento en que cierra el programa de instrumentación, desinstala las interposiciones y finaliza la ejecución.

El objetivo original de este proyecto era portar DITools a GNU/Linux™ sobre INTEL® 386, y posteriormente decidir si portarlo a otras arquitecturas. Debido a que DITools está muy ligado a Irix sobre MIPS® y su diseño ya no puede da mucho más de sí sin reestructurarlo seriamente, se decidió reescribir este programa de nuevo.

Aún así se estudió a fondo este programa para extraer de él toda la experiencia posible y no repetir las limitaciones y errores de DITools. Se discute más a fondo sobre este programa en la sección *Descripción y estructura de DITools* del capítulo *Introducción al ELF*.

# 3. Estado del arte

Antes de comenzar el proyecto se hizo una pequeña investigación para averiguar el estado del arte en lo que es tratamiento de las estructuras ELF y/o interposición dinámica de código. Esto sirvió para enriquecer nuestros conocimientos sobre interposición de código y ELF, y conocer otras opciones y soluciones a parte de DITools.

Al parecer este campo no es muy comercial y hay muy poca oferta de “software” que sirva de base para crear programas de instrumentación de código, y mucho menos “software” libre.

El ejemplo “comercial” más notable es Dyninst (http://www.dyninst.org/) de la Universidad de Maryland. Dyninst es un “software” que ofrece un API que facilita la creación de un “software” de instrumentación dinámica.

De Dyninst derivan la mayoría de productos de instrumentación dinámica de código:

- DPCL
- Open SpeedShop™ for Linux™
- Dynamic Kappa-Pi
- Dynaprof
- Metasim
- OCM: OMIS Compliant Monitoring system
- Stride - Stream Identification During Execution
- TAU
- ToolGear
- Tracetool

Lo malo es que a pesar de ser un producto “open source” (está disponible su código fuente) se entrega bajo una licencia de “software” privativa que sólo permite su uso gratuito en caso de investigación. En otros casos se ha de pedir permiso de uso.

Esto hizo que automáticamente se dejase de profundizar en él. Inspirarse en él hubiera podido llevar a plagiar accidentalmente partes del mismo. Por ello se ha evitado a toda costa investigar más sobre Dyninst y así poder entregar un producto libre de las restricciones de Dyninst (tanto a nivel de copyright como de posibles patentes).

Si la oferta en el mundo comercial es reducida (se reduce casi a una opción), en el mundo del “software” libre no encontramos ninguna herramienta de instrumentación. El “software” libre ha sido desa-

rrollado en su mayor parte por voluntarios que generalmente no tienen en casa un supercomputador: no hay apenas “software” relacionado de instrumentación de código.

El primer producto relacionado que encontramos no es exactamente un programa de instrumentación dinámica, pero dispone de todos los mecanismos y técnicas para poder implementar uno: GNU Debugger. El depurador de GNU está íntimamente ligado a como el sistema operativo organiza los objetos de una aplicación. Por ello en más de una ocasión hemos recurrido a su código fuente para poder ver como se resuelve un problema u otro.

La segunda oferta que he encontrado de la cual sacar ideas o información es libelf. Es una librería que permite manipular ficheros ELF de casi cualquier arquitectura. No permite manipular estructuras ELF en tiempo de ejecución. Se puede encontrar información sobre el mismo en la página oficial de libelf (http://directory.fsf.org/libs/misc/libelf.html).

Lo que mas se ajusta a nuestra búsqueda es ELFsh. Es un programa específico para manipular estructuras ELF en tiempo de ejecución. Puede usarse perfectamente para crear “software” de instrumentación dinámica, aunque no está pensado específicamente para ello. En realidad está orientado a depurar “software” y a experimentar con las aplicaciones. Aporta un API bastante potente y con ella es posible llegar a crear un programa de interposición dinámica de código.

También se encontró información y fragmentos de código en los “ezines” de “hackers”. Las técnicas de “PLT poisoning” (hablaremos sobre ellas más adelante) se usan bastante para realizar ingeniería inversa, saltar protecciones y para desarrollar virus.

# 4. Objetivos

El objetivo original era portar DITools a GNU/Linux™. Al final resulto más viable implementarlo desde cero, pero basándose en los mecanismos originales. La herramienta cumple los siguientes puntos y propiedades:

- **No requiere nada más que un ejecutable para comenzar a trabajar.** La herramienta permite instrumentar un programa del cual tan sólo se tiene el binario.
- **Es muy veloz y ligera.** El principal “target” de esta herramienta es la instrumentación de aplicaciones de cálculo intensivo. El impacto que debe tener sobre la velocidad de ejecución ha de ser mínimo. Durante la ejecución del programa que se instrumenta el código de la librería suele tener un coste constante (O(1)); y lo más breve posible.
- **Basada en estándares.** La herramienta trabaja y respeta los estándares al máximo posible. De esta forma se asegura su robustez frente a nuevas versiones del sistema operativo “host” y frente a diferentes lenguajes y compiladores. Esta aplicación esta basada en el estándar ELF debido a que está pensada fundamentalmente para entornos Unix™ y similares.
- **Altamente portable.** En principio, la teoría dice que ciñéndose y respetando estrechamente los estándares al máximo, se puede obtener un software que funcione en cualquier plataforma realizando una serie de cambios mínimos. De antemano se sabe que esto es muy difícil, pero aún así el esfuerzo siempre facilita el “porting” a otras plataformas.
- **Fácil de usar.** Ofrece un control de errores adecuado y una interfaz sencilla de usar, apartando lo máximo posible al usuario de las particularidades del SO y del formato ELF. Además es resistente a errores, ya sean por parte del usuario o por sucesos inesperados provocados por la implementación de ELF del SO.
- **Ofrece una API sencilla y potente.** La idea consiste en escribir un mecanismo de instrumentación mínimo, pero completo, para que el desarrollador goce de la máxima flexibilidad a la hora de crear

una herramienta de instrumentación.

- **Una herramienta libre realizada con software libre.** Esta herramienta ha sido escrita desde cero con herramientas libres (GNU C Compiler, GNU Make, GNU/Linux™, DocBook, ...) y como respuesta se entregará como una aplicación libre. Debido a que en esencia es un DSO y se enlaza en tiempo de ejecución con la aplicación que se instrumenta se entregará bajo una licencia LGPL[2] la documentación bajo una licencia FDL[3].
- **Se introducen cambios, pero en esencia el funcionamiento es muy similar al del DITools original.** Se ha de tener en cuenta que OMPItrace de SGI® confía en gran medida en el funcionamiento actual de DITools.

El proyecto se entrega implementado al 100% sólo para GNU/Linux™ sobre la plataforma IA32. Inicialmente se da sólo soporte a las versiones del kernel 2.4.x y 2.6.x con GNU C Library 2.2.x o 2.3.x.

El resto de versiones para otras plataformas son completamente funcionales a excepción de que no implementan el mecanismo de interposición por “callback”.

En total se da soporte a todas estas plataformas:

- GNU/Linux™ (kernel 2.4.x o 2.6.x) sobre INTEL® 386, PowerPC™ y PowerPC™ 64.
- Solaris 8, 9 y 10 sobre SPARC para 32 y 64 bits.
- Irix sobre MIPS® para 32 y 64 bits.

El objetivo de dar soporte a todas estas plataformas es demostrar que el código es altamente reciclable y muy fácil de adaptar a nuevos escenarios.

# 5. Desarrollo del proyecto

El proyecto propuesto por Jesús Labarta y Judit Giménez fue portar DITools a GNU/Linux™ sobre INTEL® 386.

El desarrollo de pDI-Tools se ha dividido en cuatro partes básicas: evaluación del proyecto, formación e investigación, desarrollo de pDI-Tools y creación de la documentación. Seguidamente describimos cada una de ellas.

## 5.1. Evaluación del proyecto

El primer paso fue evaluar la viabilidad del proyecto. Para ello solicite una copia del código fuente de DITools y toda la documentación posible sobre el mismo. Resultó que no se pudo encontrar documentación sobre DITools, a parte de los artículos [DITools] y [DITools2]. Estos artículos son descriptivos y no dan información detallada sobre el funcionamiento interno de DITools.

Así que se empezó a leer directamente el código e intentar conocer en detalle el funcionamiento de DITools. Por el estado del código fuente de DITools se deduce que esta aplicación se desarrolló según las necesidades que aparecían y se ha modificado por diferentes personas. El código está vagamente documentado y es en ocasiones bastante inconsistente y caótico. Esto significó un gran esfuerzo de ingeniería inversa para poder deducir el funcionamiento del código.

A la vez que se estudiaba DITools se empezó a estudiar el funcionamiento de ELF y como este se aplica sobre Irix/MIPS®.

De contrastar ambos estudios se llegó, en dos semanas, a concluir que la tarea de portar DITools directamente a GNU/Linux™ sería extremadamente compleja e ineficiente. Además, visto como se

resolvían ciertas cosas en DITools, se prefirió comenzar desde cero para tener un programa con mejores prestaciones y más sencillo de mantener y portar.

## 5.2. Formación e investigación

Se retomó el estudio de DITools para extraer de él toda la experiencia e ideas posibles. Era necesario conocer bien su funcionamiento para que la nueva aplicación ofreciese, no sólo nuevas funcionalidades, sino una importante compatibilidad con él.

Fue complicado por la falta de comentarios y el degenerado estado del código debido, principalmente, a que distintos programadores introdujeron modificaciones sin reciclar código y sin evaluar el impacto que tendrían éstas en el resto del programa. El resultado es un código difícil de seguir con mucho código desactivado, apaños, condiciones extrañas y notas escritas por los programadores justificando su último "hack".

El esfuerzo de ingeniería inversa se combinó con un esfuerzo de reestructuración y depuración del código. Modificando y mejorando el código a la vez que se estudiaba permitió acelerar el proceso de aprendizaje. Tomar una posición activa facilita la comprensión y es más interesante que una simple posición pasiva donde simplemente se lee.

Se trabajó sobre DITools intensamente durante unos tres meses a la vez que se estudiaba el estándar ELF. Es aconsejable tener un conocimiento de ELF para poder entender como funciona DITools, aunque no muy profundo. Esto se debe a dos motivos: el primero es que DITools se apoya fuertemente sobre las API específicas de rld(5) (el "Runtime Linker" de Irix). El segundo motivo es que la implementación de ELF en Irix sobre MIPS® es bastante particular.

Por ello se dedicó una gran cantidad de tiempo a analizar "Runtime Linker" de otros sistemas operativos. El formato ELF es un estándar que se ve muy afectado por la arquitectura, y además se implementa de forma irregular.

Esto se debe a que un formato de enlace dinámico debe ser muy eficiente. No tiene sentido un enlace dinámico lento: nadie lo usaría y sería rechazado de inmediato. Por eso mismo siempre que se puede se explotan al máximo las características del "hardware" subyacente y por lo tanto se introducen serias diferencias de una plataforma a otra.

También es debido a que el enlace dinámico sólo se suele gestionar por herramientas específicas del sistema operativo y rara vez las aplicaciones acceden directamente a sus estructuras. Y cuando no es así, en la mayor parte de los casos intermedia la librería de C, que al igual que el "Runtime Linker", está muy ligada al sistema operativo.

Por ello, a pesar de las incompatibilidades, apenas hay problemas a la hora de portar aplicaciones normales. Sólo en algunos pocos casos, en los que las aplicaciones acceden directamente a estas estructuras, puede haber problemas. Pero estas aplicaciones son pocas ya que suelen ser muy especiales: un depurador de "software" (por ejemplo GNU Debugger), una herramienta de "profiling", o el mismo pDI-Tools. Por eso no hay un importante esfuerzo por seguir estrictamente los estándares.

Se descubrió que apenas hay documentación y "software" relacionado con ELF y el enlace dinámico. Se recurrió a todo tipo de documentos, artículos, notas, código y "mails" que pudiesen aclarar ELF, los "Runtime Linker", técnicas de reenlace, particularidades de ELF, etc.

Esta recopilación de información junto el conocimiento obtenido de DITools se usó para valorar que requisitos debía cumplir la nueva aplicación que desarrollaríamos. Se decidió que características de DITools se debían mantener, cuales evitar y como evitar futuros problemas de portabilidad.

## 5.3. Desarrollo de pDI-Tools

Con la información y experiencia reunida se comenzó un nuevo programa que recibe el nombre de pDI-Tools. Este introduce importantes cambios estructurales (respecto DITools) orientados a mejorar la portabilidad, velocidad, estabilidad y eficiencia. Aún así se intentaría imitar el comportamiento de DITools para facilitar el "porting" de OMPItrace (programa de instrumentación de programas paralelos desarrollado por el CEPBA) a pDI-Tools.

Durante todos estos estudios se fueron realizando pruebas para comprobar si era posible implementar los mecanismos de reenlace y gestión de "backends" en GNU/Linux™ sobre INTEL® 386. Estas pruebas eran pequeños programas específicos, parte en ensamblador y parte en C que introducían a fuerza bruta interposiciones, comprobaban si las estructuras ELF eran siempre coherentes, evaluaban casos extremos y demás. Estos programas, que denominamos "pruebas de concepto", sirvieron para demostrar que el proyecto era totalmente viable.

También se hicieron pruebas de concepto sobre otras arquitecturas como SPARC e Itanium®. Estas pruebas mostraron hasta que punto podemos confiar en el estándar para conseguir nuestros propósitos.

Las pruebas de concepto no se conservan en su mayor parte, ya sea porque fueron borradas al acabar de usarse o recicladas en otras pruebas. Tampoco hubiera aportado mucho conservarlas ya que dependían de demasiados parámetros (como el compilador, las librerías, el programa de reenlace, etc.) y sólo servían para probar un determinado fenómeno.

Cuando se tuvo suficiente experiencia y estaba seguro sobre como se implementarían los mecanismos comencé a escribir DITools comenzando por la parte independiente de la arquitectura. Esto incluye:

- Un mecanismo de depuración que comprueba el estado de las estructuras ELF en memoria. Útil para detectar anomalías lo antes posible y así evitar situaciones inesperadas durante la ejecución del programa. Además es muy práctico a la hora de portar pDI-Tools a una nueva versión del sistema operativo ya que detecta automáticamente las diferencias con el anterior "Runtime Linker". Evidentemente este código, por eficiencia, está desactivado en la configuración por defecto.
- El nuevo sistema de configuración de pDI-Tools.
- Los mecanismos de lectura e interpretación de los comandos de reenlace.
- Las estructuras de inventario de interposiciones y objetos y el API necesaria para administrarlas.
- El mecanismo necesario para manipular los "backends", el cual es siempre portable ya que está basado en la interfaz estándar del "Runtime Linker": dlopen(3), dlclose(3) y dlsym(3)
- Establecer la API necesaria para crear reenlaces, redefiniciones e interposiciones basadas en "callback".

Mientras se desarrollaba todo esto se escribió en paralelo un "Elf Backend" ("driver" encargado de gestionar las estructuras ELF específicas para un sistema operativo y arquitectura) para GNU/Linux™ sobre INTEL® 386.

El desarrollo de este "Elf Backend" a su vez redirigió la escritura del código de la parte no dependiente: la experiencia lograda con las pruebas de concepto sumado a esta puesta en práctica ayudaba a discernir que era lo bastante genérico como para ir a la parte independiente de la arquitectura y que era específico para Linux™/INTEL® 386.

Cuando se obtuvo un primer prototipo aparentemente portable se empezó a desarrollar en paralelo una versión para Solaris sobre SPARC a la vez que se terminaba la versión para GNU/Linux™/INTEL® 386.

Se escogió esta arquitectura SPARC debido a que es una arquitectura frecuente en laboratorios de cálculo. Otro buen motivo fue que generalmente todo lo relativo a ella y en ella suele estar muy

pulido y claro. Se encontró una documentación muy buena, el ensamblador es muy claro y elegante, y además la implementación suele ser muy fiel a las especificaciones.

Para investigar y aprender más sobre esta plataforma adquirí mi propia máquina Sun Microsystems™, una UltraSPARC™ II, por un módico precio en el mercado de segunda mano.

Este "porting" desveló que la parte independiente de la arquitectura no era tan independiente. Se encontraron diferencias entre las librerías de C de Solaris, nuevas particularidades del "Runtime Linker" y nuevas necesidades a la hora de gestionar los mecanismos de interposición.

Pero la novedad más impactante fue la introducción de los 64 bits. Esto afecto a la totalidad de pDI-Tools, desde los "scripts" de configuración hasta la organización del código. A pesar de tratarse la misma CPU, cada modelo implica un formato ELF distinto y tipos de datos algo distintos. Se destaparon y resolvieron problemas de portabilidad dados por los tipos de datos y obligó a crear una nueva capa de abstracción en los "drivers" de sistema operativo para poder manejar diferentes CPU's.

También obligó a escribir un código aún más genérico en la parte no dependiente que nos permitió ver que código era redundante en ambos "Elf Backend" y así detectar código portable que a priori pensábamos que era específico.

Los resultados fueron satisfactorios ya que en poco más de un mes ya habían tres versiones nuevas plenamente funcionales de pDI-Tools: GNU/Linux™ sobre INTEL® 386 (sólo 32 bits), Solaris sobre SPARC en 32 bits, Solaris sobre SPARC en 64 bits.

La única diferencia entre la versión INTEL® 386 y las versiones SPARC son las interposiciones mediante el mecanismo de "callback". De hecho este tipo de interposiciones sólo está disponible en pDI-Tools para GNU/Linux™ sobre INTEL® 386. Debido a su complejidad y coste se decidió implementar únicamente sobre INTEL® 386 únicamente para demostrar que son posibles. La programación de este tipo de interposición llevo más un mes de trabajo, y la mayor parte del código es totalmente dependiente de la arquitectura INTEL® 386.

Una vez terminadas de programar estas versiones se comenzó a analizar la plataforma PowerPC™ 32 bits, muy similar a SPARC 32 bits. Esta tarea ayudó a que el API tuviese unas estructuras de datos más firmes y flexibles ante nuevas plataformas.

Crear esta versión de pDI-Tools para Linux™/PowerPC™ 32 bits permitió ver como se gestiona la portabilidad dentro de un mismo "Elf Backend". Recordemos que el código se divide en sistemas operativos, y la gestión de los "drivers" de CPU corren a cuenta del "Elf Backend". Luego se hizo un gran esfuerzo por marcar un esquema en los "Elf Backend" que aislará toda la parte dependiente de la CPU en localizados puntos.

Con todo esto el prototipo actual ya presentaba un gran nivel de portabilidad. Ahora sólo quedaba completar el trabajo sobre PowerPC™ e implementar la versión de 64 bits de pDI-Tools.

Aparentemente, teniendo hecha la versión de PowerPC™ 32, portar pDI-Tools a 64 bits tendría que ser algo fácil. Pero sus respectivos ABI son muy distintos, y esto hizo que fuese en realidad la versión más difícil de implementar.

La especificación ELF se divide en un documento general para todas las plataformas y luego en documentos específicos para cada plataforma que reciben el nombre de suplementos. El formato ELF para GNU/Linux™ sobre PowerPC™ 64 está todavía en desarrollo y no hay todavía un verdadero suplemento ELF, sino una especie de indicaciones de como se debería implementar. Estas indicaciones están basadas en la especificación PowerOpen™ del formato COFF. Como IBM AIX™ implementaba inicialmente el ABI PowerOpen™, sus desarrolladores lo disimularon mediante un disfraz que ofrece un pequeño grado de compatibilidad con el formato ELF. Este disfraz no permite trabajar correctamente con casi ninguna estructura ELF y de hecho no cumple el estándar ELF.

En resumen esta arquitectura no dispone de buena documentación y la implementación es muy discordante respecto a la poca documentación existente.

Todo ello llevo a una nueva etapa de investigación bastante dura ya que se tuvo que desensamblar mucho código para ver si era posible o no implementar pDI-Tools sobre GNU/Linux™ para PowerPC™ 64.

Por suerte todo fue bien y en el plazo de casi 3 semanas se consiguió una implementación funcional y correcta de pDI-Tools para PowerPC™ 64.

Para evaluar el nivel de portabilidad logrado, se probó a portar a Irix sobre MIPS®. El objetivo era ver el tiempo necesario para implementar un nuevo "Elf Backend" desde cero. Los resultados fueron sorprendentes: en una semana pDI-Tools estaba funcionando en Irix sobre MIPS®, tanto para 32 como 64 bits. Además no se tuvo que realizar ningún cambio en la parte no dependiente de la arquitectura. Con esto quedo probada la gran portabilidad de pDI-Tools.

La ventaja de haber desarrollado sobre todas estas arquitecturas, y en algunas ocasiones en paralelo, es haber constrastado las diferencias más importantes entre arquitecturas. Con ello se ha ganado experiencia y ayuda a ver donde la implementación es susceptible a presentar incompatibilidades. Además ha obligado a adoptar una serie de hábitos y métodos que ayudarán en el futuro a portar pDI-Tools.

Finalizada la versión de Irix/MIPS® todos los requerimientos de la especificación ya estaban cumplidos. Ahora sólo quedaba mejorar el rendimiento del programa, buscar posibles errores y/o limitaciones y arreglarlas y dar ciertas garantías de que el código es correcto y portable.

## 5.4. Creación de la documentación

Durante los últimos siete meses se había empezado ya el entorno de documentación y se dedicaba una parte importante del tiempo a escribir esta documentación.

La documentación se empezó con la creación del entorno de proceso de documentos XML en formato DocBook. DocBook/XSL es un lenguaje XML que permite escribir documentos grandes, insertando marcas semánticas, sin preocuparse de su presentación. Es similar a LaTeX pero en XML. Se escogió este sistema de documentación en lugar de Texinfo, el sistema sugerido por la FSF para el software GNU, debido a que es más potente, flexible y especialmente por su auge en importantes proyectos como Gnome™ o el kernel de GNU/Linux™.

Para editar documentos DocBook/XSL basta con un editor de textos cualquiera. El problema surge cuando se intenta traducir el documento a un formato imprimible o legible, es decir, que ya no contenga marcas semánticas y en su lugar se hayan aplicado unos estilos de presentación.

No hay "software" libre en el mercado que permita realizar fácilmente esta operación. Sólo se dispone es del programa OpenJade y un conjunto de hojas de estilo DSSSL publicadas por Norman Walsh que permiten convertir un documento DocBook en formato PDF, PostScript®, RTF y HTML. También existe FOP[4], pero se desestimó su uso por su dependencia en Java™, una herramienta no libre.

Fue necesario crear un entorno compuesto de ficheros `makefile`, hojas de estilo XSLT y CSS, "scripts", etc. que facilitasen y automatizasen al máximo el proceso de generación de la documentación formateada.

También se han creado varios programas en Perl encargados de realizar ciertas tareas sencillas. Por ejemplo, se usan estos "scripts" para generar las marcas de índice que se usarán en el documento DocBook/XSL para generar el índice conceptual de la versión impresa. Estos "scripts" generan ficheros XML que se combinarán con los ficheros XML de la documentación.

La documentación se divide en una multitud de pequeños ficheros XML que se combinan, junto a los XML generados por los "scripts", en un documento final. Este fichero se construye con hojas de transformación XSLT y el programa xsltproc.

Una vez ya se tiene el documento DocBook este se pasa a un conjunto de “scripts” encargados de aplicar las hojas de transformación de Norman Walsh con OpenJade. Esto resultará en una renderización del documento. Aprovechando que OpenJade es en realidad un procesador de documentos SGML y hojas de estilo DSSSL, con unas hojas de estilo escritas por mi se adecua el aspecto del documento.

Se necesitó alrededor de dos semanas para escribir lo más básico del entorno de desarrollo. A partir de ahí se fue mejorando poco a poco y se le fueron añadiendo más funcionalidades.

En cuanto pudo generar documentos se uso para escribir el documento de presentación del proyecto: *pDI-Tools: Descripción del proyecto*.

Durante la escritura de ese documento se fue perfeccionando el entorno DocBook/XSL. Cuando este documento estuvo acabado, el entorno estaba lo suficiente evolucionado como para poder usarlo en la escritura de la presente memoria.

# Notas

1. Aun así existen técnicas basadas en el uso de macros y librerías compartidas para intentar insertar código alterando el binario resultante lo menos posible.
2. Acrónimo para “The GNU Lesser General Public License (http://www.gnu.org/licenses/lgpl.txt)”.
3. Acrónimo para “The GNU Free Documentation License (http://www.gnu.org/licenses/fdl.txt)”.
4. Es un formateador de documentos XML en formato XSL-FO: FOP (http://xml.apache.org/fop/).

# II. pDI-Tools: documentación del proyecto

# Capítulo 1. Fundamentos

## 1.1. Introducción al ELF

En esta sección se da una pequeña introducción sobre como se enlaza en tiempo de ejecución objetos en formato ELF. Esta sección no pretende ser un manual detallado de ELF. Es simplemente una descripción de sus mecanismos y estructuras para poder entender más o menos como funciona. Si se desea tener un conocimiento detallado de ELF se recomienda leer la especificación [Sys5ABI]. En ella se encontrará una descripción general y no ligada a la plataforma del estándar. Para terminar de entender la implementación de ELF sobre plataformas MIPS® léase la especificación [Sys5mips], especialmente los capítulos 4 y 5.

### 1.1.1. Descripción del formato

El formato ELF, o *Executable and Linking Format* fue originalmente desarrollado y publicado por UNIX System Laboratories como parte de la *Application Binary Interface* (ABI). Más tarde el *Tool Interface Standards committee* (TIS) eligió el formato ELF como el formato portable de ficheros objeto en máquinas INTEL® de 32 bits para una amplia variedad de sistemas operativos.

El formato es muy extensible, y no sólo limitado a 32 bits para máquinas INTEL® 386. Actualmente se usa en una amplia variedad de máquinas, desde 32 bits hasta 64 bits pasando por todos los modelos concebibles de microprocesadores. No sólo el formato es ampliamente flexible, sino que además incorpora una serie de estructuras capaces de almacenar la información necesaria para saber sobre que arquitectura trabaja un determinado binario. Por ejemplo, en Itanium® es posible tener hasta cuatro ABI distintos funcionando en un mismo sistema operativo: 32 bit little endian, 32 bit big endian, 64 bit little endian y 64 bit big endian. El formato ELF permite al sistema operativo distinguir de que tipo es un determinado objeto, saber si puede usarlo o no y como usarlo.

La especificación de ELF se divide en dos partes: *Los ficheros objeto* y *Carga del programa y enlace dinámico*. En la primera parte se explica la estructura de los ficheros objeto, el formato binario en disco, y como se deben codificar e inicializar las estructuras. La segunda se refiere al formato ELF en tiempo de ejecución, con las estructuras ya descodificadas y con valores válidos, a parte de nuevas estructuras e información generadas por el “Runtime Linker”. Casi todas las explicaciones que se dan a continuación giran alrededor de esta parte de la especificación, a no ser que se indique lo contrario. Es importante hacer esta distinción ya que el contenido de algunas secciones varía según este en disco o en memoria.

El motivo de que la primera parte de la especificación no nos interese mucho es porque DITools y pDI-Tools sólo trabajan con estructuras ELF una vez hecha la carga del programa y nunca manipulan ficheros en disco.

Hay muchos tipos de ficheros ELF: “Relocatable files” (ficheros reubicables), ficheros ejecutables, “Shared Objects”, “core files”, etc.

No obstante sólo dos de ellos intervienen en la ejecución de una aplicación: el fichero ejecutable y los “Shared Objects” (librerías). De hecho el primero contiene el programa y una lista de las librerías que se usan. Se usa esta lista para construir la lista de objetos compartidos que deben cargarse. Debe tenerse en cuenta que los objetos compartidos a su vez pueden requerir otras librerías y de esta forma generar un árbol de dependencias entre ellos.

### 1.1.2. Secciones y “Dynamic Tags”

Los ficheros ELF, ya sea en disco o en memoria, están divididos en secciones. Virtualmente una sección puede contener cualquier cosa: código, información de depuración, un icono, un MP3, una nota de texto con el copyright de la aplicación, etc. Las secciones son segmentos del fichero ELF que en principio no deberían solaparse (aunque nada se lo impide), con un nombre. Existen unas cuantas secciones predefinidas por el estándar, pero se pueden crear tantas como se quiera.

El responsable de la carga de los DSO (objetos) en memoria, de resolver los enlaces dinámicamente y poner en marcha la aplicación es el “Runtime Linker”. Se explicará con detalle más adelante en esta sección. De momento sólo se explican los requerimientos genéricos que debe resolver cualquier “Runtime Linker”.

Una vez se han cargado los objetos en memoria, el “Runtime Linker” recorre las diferentes secciones y estructuras de datos construyendo una cadena de estructuras dinámicas. Estas estructuras reciben el nombre de “Dynamic Tags” y contienen un identificador numérico que identifica el tipo de la estructura y, dependiendo de este tipo, la estructura puede contener un puntero (tipo ElfW(Addr)) o un valor (tipo ElfW(Word)).

Las estructuras dinámicas que contienen un valor del tipo ElfW(Word) suelen dar información sobre otras secciones. Por ejemplo la sección dinámica `DT_STRSZ` indica el tamaño en memoria de la sección “String Table”.

Las estructuras que incorporan un puntero indican la ubicación en memoria de una sección ELF. Por ejemplo la estructura `DT_STRTAB` nos indica la posición en memoria de la sección “String Table”.

Se ha de tener en cuenta que la cadena de “Dynamic Tags” no es sólo una manera de agrupar la información para simplificar el acceso a ella. Estas estructuras dinámicas sirven para seleccionar la información relevante para el “Runtime Linker” y además pueden aportar información nueva en tiempo de ejecución. Por ejemplo la sección dinámica `DT_DEBUG` está generada en tiempo de carga y suele apuntar a información de depuración específica según el sistema operativo.

Como se ha dicho, la cadena de “Dynamic Tags” sólo agrupa información de interés para el “Runtime Linker”, ignorando toda sección que el éste no use. Esto no perjudica al funcionamiento de DITools o pDI-Tools ya que estos nunca usan estructuras que use el “Runtime Linker”.

Según la especificación [Sys5ABI], en tiempo de ejecución sólo se debería usar un conjunto de estructuras dinámicas predefinidas, aunque es posible no usarlas todas o añadir nuevas. Algunas arquitecturas aportan nuevos “Dynamic Tags”, ignoran o incluso varían el comportamiento de algunos existentes.

La especificación hace un especial hincapié en que se respete la obligatoriedad y estructura de las secciones. Pero en la realidad ningún fabricante hace caso de la especificación. Así que a la hora de interpretar la especificación debemos ser bastante flexibles y apoyarnos sobre los “System V Application Binary Interface Supplements” específicos para las diferentes arquitecturas.

El problema radica en que ELF es un esfuerzo por definir una interfaz binaria común entre todos los Unix™. Evidentemente ELF entró en juego cuando ya había una gran diversidad de ABI’s, y aunque la mayoría derivaron de COFF, las diferencias entre algunas eran realmente radicales. Evidentemente los fabricantes se negaron a reimplementar las ABI desde cero: esto hubiera roto la compatibilidad hacia atrás con sus antiguos productos. Lo que se hizo fue camuflar sus antiguos mecanismos de carga con una interfaz ELF, y sólo en algunos raros casos reimplementarla desde cero. De esta forma los binarios antiguos y ELF podían coexistir en un mismo sistema operativo. Algunos fabricantes lo hicieron implementaciones notablemente excelentes, como es el caso de INTEL® con su [Sys5i386] o Sun Microsystems™ con su [Sys5Sparc] y [Sys5Sparc64]. En cambio encontramos otros fabricantes que realizaron implementaciones que incumplen el estándar en numerosos puntos. Dos claros ejemplos son las especificaciones [Sys5mips] de SGI® y [PPC64ELF] de IBM®[1].

Reuniendo un conocimiento general de gran parte de las arquitecturas conseguimos tener una visión global de las implementaciones de ELF que nos permite ver que es lo que realmente se usa de la especificación, y lo más importante: que es más o menos común entre muchas arquitecturas.

De este estudio realizado remarcamos como importantes, y bastante comunes, un conjunto de estructuras dinámicas. No obstante no siempre están presentes o su contenido es muy especial en determinadas arquitecturas. Seguidamente las enumeramos y comentamos superficialmente:

- **`DT_NULL`.** No aporta ninguna información, pero su presencia marca el final de la lista de "Dynamic Tags". Su presencia, evidentemente, es muy importante para poder iterar sobre esta lista.
- **`DT_PLTGOT`.** Es de todas la sección más dependiente de la arquitectura. Según el estándar debe estar siempre presente. En algunas plataformas, aún así, puede no llegar a estar presente.

  Según el estándar puede contener punteros (una "Global Offset Table" o GOT) y/o una serie de procedimientos de enlace ("Procedure Linkage Table" o PLT). La presencia de uno, u otro, o ambos, depende de la implementación.
- **`DT_SYMTAB`.** Apunta a la "Symbol Table". Esta tabla sólo contiene estructuras con información de un símbolo. Estas estructuras básicamente contienen un puntero al nombre de la función en la sección "String Table", información sobre el tipo de símbolo y otra información complementaria.
- **`DT_STRTAB`.** Este elemento apunta a la "String Table". Contiene el nombre de los símbolos, el nombre de las librerías, y otras cadenas que residen en esta tabla.
- **`DT_HASH`.** Esta sección se ha encontrado en todas las plataformas y es esencial para poder hacer búsquedas eficientes en la tabla de símbolos.

  Como el estándar es muy explicito en su implementación (incluso llegando a dar porciones de código que se deben usar), todas las plataformas probadas han demostrado ser idénticas en su gestión y creación.

  Su función es acelerar al máximo las búsqueda de una estructura de información de símbolo mediante su identificador. Se describen más adelante estas estructuras en la sección dinámica `DT_SYMTAB`.
- **`DT_REL` o `DT_RELA`.** Estos elementos guardan la dirección de su tabla de reubicación. En una arquitectura determinada sólo se usa una de las dos, aunque técnicamente sería posible usar ambas al mismo tiempo. A partir de ahora nos referiremos sólo a `DT_REL`, aunque todo lo que digamos se aplica también a `DT_RELA`.

  La diferencia entre una sección y la otra es el tipo de estructura usada. En el primero, `DT_REL`, se usa una estructura de reubicación ("relocation") absoluta, es decir, cuyo puntero se puede usar tal cual. En el segundo se usa una estructura de reubicación con un "addend" explícito que se ha de aplicar al puntero antes de usarlo.

  Estas estructuras relacionan el *índice* de una estructura de información símbolo en la tabla de símbolos con un puntero. Generalmente el puntero, o bien es la dirección del símbolo en sí, o bien una referencia dentro de `DT_PLTGOT`.
- **`DT_RELSZ` o `DT_RELENT`.** Dan información sobre el tamaño de la `DT_REL` y el tamaño de la estructura usada en la ella. En el caso de usarse `DT_RELA` encontraremos también las estructuras dinámicas `DT_RELASZ` o `DT_RELAENT`.
- **`DT_PLTREL` y `DT_PLTRELSZ`.** El primero indica el tipo de estructura de reubicación usada, mientras que el segundo indica el tamaño de dicha estructura. El primero puede tomar los valores `DT_REL` o `DT_RELA`.

Dejando de momento de lado las interrelaciones entre estas secciones, se hace un inciso en la nomenclatura usada para los tipos de datos de ELF. En ELF todos los tipos van precedidos por el prefijo

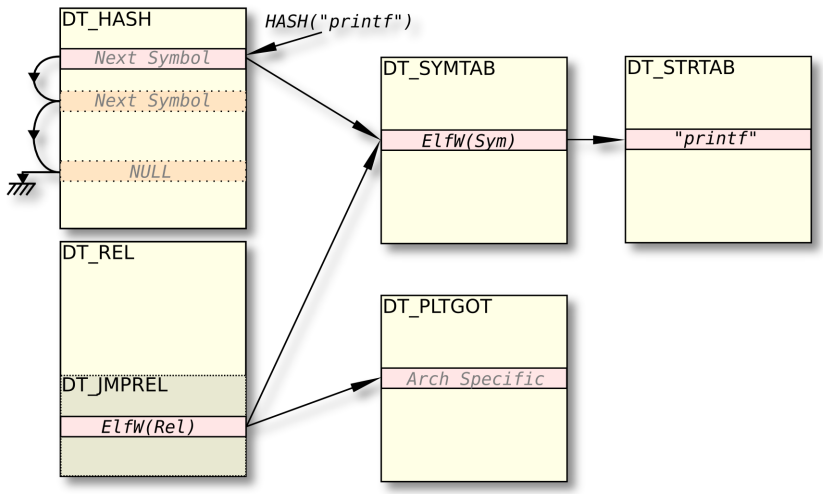
`Elf32_` o `Elf64_`, según estemos en una plataforma de 32 o 64 bits, respectivamente. Así tenemos Elf32_Rel, Elf64_Sym, etc. Como la elección del tipo de estructura se hace en tiempo de compilación, y no depende de nosotros, sino del ABI escogido, accedemos a ellas mediante la macro `ElfW(x)`. Esta macro nos permite usar por ejemplo el tipo ElfW(Sym) que se traducirá posteriormente, en tiempo de compilación, en el tipo adecuado (Elf32_Sym o Elf64_Sym).

Igualmente, las macros de ELF llevan todas el prefijo `ELF32_` o `ELF64_`. De la misma forma usamos otra macro para nombrarlas: `ELF(x)`. Así tenemos las macros `ELFW(R_SYM)`, `ELFW(R_TYPE)`, etc.

## 1.1.3. Relaciones entre las diferentes secciones

En esta sección describimos las interrelaciones que se dan entre las diferentes secciones en tiempo de ejecución. No obstante esta explicación la daremos mediante la descripción de las operaciones de consulta más usuales que se hacen.

Antes de comenzar observemos el siguiente diagrama. Este pequeño mapa nos facilitará tener un imagen global de como se relacionan las estructuras dinámicas más importantes.




Este diagrama muestra como el símbolo read(2) se declara en las estructuras ELF de algún DSO que lo use y/o implemente.

### 1.1.3.1. Resolución de un símbolo

La primera operación que se plantea es la resolución de un símbolo a partir de su nombre. En esta explicación se simplifica no teniendo en cuenta modificadores como `DT_SYMBOLIC`[2] ni los símbolos "weak" (que se explican más adelante).

La búsqueda de un símbolo por su identificador se realiza con ayuda de la tabla "hash" contenida en la sección `DT_HASH`. Esta nos permite acceder rápidamente a la información de un símbolo sin tener que realizar una búsqueda secuencial por toda la tabla de símbolos. La implementación y mecanismos de la tabla "hash" se describen hacia el final de esta sección.

La sección `DT_SYMTAB` contiene una serie de estructuras de ElfW(Sym). Esta estructura contiene varios campos, de los cuales los más interesantes son la posición del nombre del símbolo en `DT_STRTAB`, una dirección, el tamaño del símbolo (si es aplicable), y un campo con información especial sobre el símbolo.

En el campo de información especial se indica la visibilidad y el tipo de símbolo. La visibilidad asigna en realidad un nivel de precedencia al símbolo por si hay colisiones con otros símbolos de otros objetos compartidos. Los niveles de precedencia son los siguientes:

- **`STB_LOCAL`.** El símbolo sólo es visible desde este mismo objeto y no debe exportarse. Los símbolos locales con el mismo nombre pueden existir en varios ficheros sin interferir entre ellos.
- **`STB_GLOBAL`.** Los símbolos globales son visibles a todos los objetos en memoria. Una definición global de un símbolo satisfará cualquier referencia no resuelta al mismo símbolo global.
- **`STB_WEAK`.** Un símbolo débil ("weak") es equivalente a un símbolo global. La diferencia estriba en que si existe un símbolo global con el mismo nombre, este toma precedencia y el símbolo "weak" es ignorado.

El tipo de símbolo es un identificador numérico que identifica el tipo de objeto que la estructura declara. Las herramientas DITools y pDI-Tools sólo trabajan con símbolos del tipo `STT_FUNC`.

En el tipo de símbolo `STT_FUNC` (sólo en los casos de visibilidad `STB_GLOBAL` y `STB_WEAK`) el puntero de la estructura puede tomar dos valores: 0 en caso de que el símbolo sea referido pero no resida en el objeto, o un puntero real al símbolo dentro del objeto.

Con las operaciones hasta aquí descritas se puede escribir una rutina capaz de buscar un símbolo, averiguar de que tipo es, y en el caso de pDI-Tools o DITools, saber si es de nuestro interés y que operaciones podemos hacer sobre él.

### 1.1.3.2. Enlazar dinámicamente un símbolo

La siguiente operación más común es realizar un enlace en tiempo de ejecución de un objeto con una definición de función en otro objeto.

Esta operación requiere que entre en juego tres nuevas secciones: `DT_REL`, `DT_JMPREL`[3] y `DT_PLTGOT`.

La sección `DT_REL` contiene estructuras ElfW(Rel). Cada una de estas estructuras relaciona un símbolo con su información de reubicación. La estructura contiene un puntero, el tipo de estructura de reubicación y el índice del símbolo con el que se relaciona.

El uso que se debe hacer del puntero depende del tipo de reubicación que estemos tratando. Estos tipos son dependientes de la arquitectura y se definen en los suplementos de [Sys5ABI].

Pero esta tabla generalmente no la usamos ya que en casi todas las plataformas se separan las estructuras de reubicación relacionadas con funciones en una sección aparte: `DT_JMPREL`. Esta sección suele contener sólo reubicaciones de un tipo, que indican la posición del código/datos de reubicación dentro de la sección `DT_PLTGOT`.

La operación comienza por buscar el símbolo a reenlazar en la tabla de símbolos según se ha explicado más arriba. Una vez encontrado, nos aseguramos de que el puntero del símbolo sea cero (o en otras palabras que el símbolo no pertenezca al objeto). Una vez conocemos el índice del símbolo dentro de la tabla de símbolos, buscamos en `DT_JMPREL` una estructura de reubicación con el índice de este símbolo.

En la mayor parte de las arquitecturas esta búsqueda se puede realizar de forma dicotómica debido a que estas estructuras están ordenadas según el índice del símbolo.

Con la estructura de reubicación encontramos el lugar de `DT_PLTGOT` que debemos alterar para ejecutar el reenlace. El contenido de esta sección es muy variable, incluso en la misma plataforma según se use un ABI de 32 o 64 bits. Podemos encontrar desde porciones de código escrito en tiempo de ejecución hasta una simple tabla de punteros (y a veces incluso mezclas de los dos).

Ponemos como ejemplo de sección `DT_PLTGOT` el caso de la arquitectura INTEL® 386 donde cada entrada de la sección DT_JMPREL se corresponde con una entrada del PLT y contiene varios “stub” que realizan el salto desde un DSO a otro DSO. Por ejemplo, si un objeto A desea llamar a una función de B, A simplemente ejecuta una llamada al “stub” en su sección `DT_JMPREL`. Este “stub” a su vez consulta otra sección dinámica, el GOT, donde encontrará la dirección donde está la función de B, saltando de inmediato allí.

### 1.1.4. El “Runtime Linker”

La función del “Runtime Linker” es ejecutar operaciones similares a las anteriores para resolver las llamadas entre objetos. Estas resoluciones las puede hacer en tiempo de carga, es decir todas de golpe, o bajo demanda, es decir cada vez que se llama a un símbolo por primera vez.

El primer caso se explica por si mismo. En cambio la resolución dinámica de símbolos, o “Lazy Binding”, es algo más compleja de entender. El “Lazy Binding” consiste en no resolver ninguna referencia en tiempo de carga. En su lugar se dejan referencias a un punto de entrada especial del “Runtime Linker”. Este código, cuando toma el control, averigua a que símbolo se estaba intentando acceder. Con esta información resuelve una dirección al verdadero símbolo (en otro DSO), que luego usa para sustituir su entrada en el `DT_PLTGOT`. Una vez ha terminado de modificar el `DT_PLTGOT` restaura el estado de la aplicación y salta a la función que se estaba intentando llamar. A partir de este momento los futuros accesos a este símbolo resolverán directamente en el símbolo sin hacer falta la intervención del “Runtime Linker”.

Evidentemente este modelo presenta dos problemas. El primer problema radica en la posibilidad de que se intente resolver un símbolo que no existe en ningún DSO en memoria. Si ocurriese esto la aplicación terminaría bruscamente. El segundo problema surge de la lentitud de la resolución de un símbolo: aunque el resto de las llamadas tarden un tiempo razonable y constante, la primera vez que se resuelve el símbolo es difícil de predecir.

Normalmente no se realiza la resolución en tiempo de carga ya que este tipo de resolución hace más lento el proceso de carga de una aplicación. No obstante en algunos casos un software no puede tolerar la impredecibilidad de la resolución de símbolos en tiempo de ejecución. En este caso se puede declarar la variable de entorno LD_BIND_NOW que garantiza que todos los símbolos se resolverán en tiempo de carga.

Hasta ahora nos hemos referido en muchas ocasiones al “Runtime Linker”, pero no hemos definido todavía exactamente lo que es. El “Runtime Linker” es un objeto compartido, que se presenta en forma de librería, con algunas características especiales. El estándar ELF recomienda que el “Runtime Linker” sea la misma librería de C `libc.so` quien lo implemente. No obstante las implementaciones no hacen mucho caso de esta indicación del estándar y cada una usa un “Runtime Linker” distinto. Por ejemplo GNU/Linux™ usa `ld-linux.so` como “Runtime Linker”, mientras que Solaris usa `ld.so.1`.

Se ha dicho que el “Runtime Linker” es un DSO especial ya que será usado directamente por el sistema operativo para cargar una aplicación. Cuando el sistema operativo desea cargar un ejecutable en memoria, lo único que hace es mapear el ejecutable en memoria (con un mmap(2)), mapear el “Runtime Linker”, y transferir el control a él en modo usuario. Este se encargará de completar la carga del resto de objetos y de darle el control al programa.

Desde el punto de vista de la ejecución, al ser el “Runtime Linker” un objeto compartido más, es posible acceder a sus estructuras internas, a pesar de ser algo muy específico y poco portable entre diferentes sistemas operativos.

No obstante es necesario acceder a ellas para resolver unos cuantos problemas básicos. El estándar ELF sólo da mecanismos para poder acceder a las estructuras ELF de un objeto desde el mismo objeto. Es decir, un objeto puede consultar sus estructuras ELF, pero no tiene manera de navegar a través de las estructuras de los otros objetos.

Evidentemente el "Runtime Linker" suele guardar un inventario de los objetos que tiene en memoria, con referencias a sus estructuras. Como las herramientas DITools y pDI-Tools deben poder inspeccionar cualquier objeto en memoria requieren hacer uso de estas estructuras.

El acceso al inventario se hace de formas distintas según el sistema operativo: directamente, mediante protocolos de depuración o incluso mediante una API.

A veces se ofrecen mecanismos verdaderamente potentes y flexibles como es el caso de Irix, o parcos y escuetos como el de GNU/Linux™ y Solaris. En el primer caso no sólo se ofrecen mecanismos para acceder a las estructuras ELF, sino que además es posible manipular objetos en memoria, funciones para resolver símbolos, funciones de búsqueda, etc.

No obstante no se dispone siempre de una API tan potente como la de Irix ni tampoco está contemplado por el estándar. Por ese motivo es necesario poder iterar sobre los objetos para poder hacer las búsquedas nosotros mismos. El mecanismo usado para buscar un símbolo en un objeto se basa en usar una tabla "hash" nombrada antes más arriba.

El estándar ELF obliga a disponer de la sección `DT_HASH`, que contiene una tabla "hash" que permite hacer búsquedas de símbolos en tiempo casi constante, junto a la información necesaria para poder decodificarla. Esta sección se divide en dos partes, que aparecen en este orden y sin separación alguna:

- **"bucket".** Contiene la tabla "hash". Cada entrada contiene un puntero a una entrada de la siguiente sección o `NULL`.
- **"chain".** Es una tabla donde se contienen todas las colisiones de la tabla "hash". Cada entrada de esta tabla contiene un puntero a la próxima colisión de símbolo o `NULL` en caso de no haber más colisiones.

La sección `DT_HASH` está organizada de tal manera que cada entrada se corresponde directamente con su entrada en la tabla de símbolos. De esta forma, una vez encontrado el primer símbolo sólo tenemos que ir siguiendo la cadena hasta encontrar el símbolo que resuelve nuestra búsqueda o fracasar al agotar la cadena de búsqueda.

La función usada para calcular el "hash" de un símbolo está definida por el estándar y se respeta en todos los sistemas. Esta función devuelve un número de 32 bits. El valor final de "hash" lo obtenemos el resto de dividir el resultado de la función "hash" entre el número de entradas en el "bucket". Este valor es un índice del "bucket" y posiblemente la primera entrada de una cadena de colisiones.

Esta tabla nos permite reducir notablemente el tiempo empleado en las búsquedas y se debería usar (en el caso de pDI-Tools siempre) en todas las búsquedas por nombre de símbolo.

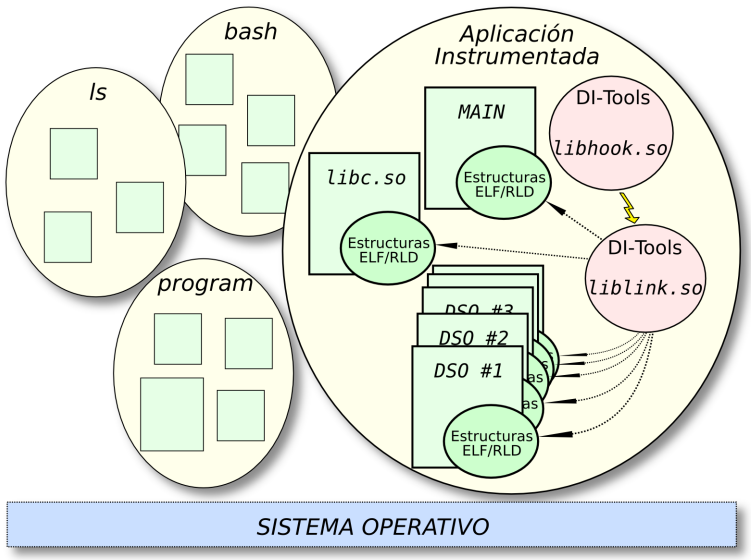
## 1.2. Descripción y estructura de DITools

DITools es un software desarrollado por el CEPBA que implementa un mecanismo de interposición dinámica de código orientado a redirigir las llamadas a funciones entre *Dynamic Shared Objects* o DSO's.

DITools opera cargando, en el espacio memoria de la aplicación instrumentada, un conjunto de objetos compartidos llamados "backend". Un "backend" es un DSO normal que contiene un conjunto de funciones que serán el destino de las interposiciones hechas con DITools.

DITools se presenta como unas librerías que cooperan entre ellas. En concreto son dos librerías: `libhook.so` y `liblink.so`.

El esquema general de funcionamiento dentro del sistema se puede observar en la imagen siguiente. DITools opera exclusivamente en el espacio de trabajo de una aplicación, en modo usuario, y por lo tanto de forma aislada a otras aplicaciones. Como se puede observar en el esquema, casi toda la acción la lleva `liblink.so`. Es el encargado de implementar los mecanismos de interposición y de manipular las estructuras de ELF y de rld(5).




En este esquema observamos varias aplicaciones, más una instrumentada por DITools. Los objetos compartidos (DSO) de la aplicación se representan como rectángulos. En cambio, los DSO de DITools se muestran como dos elipses.

Seguidamente explicamos los detalles sobre el funcionamiento interno de DITools y observaciones sobre la tarea de ingeniería inversa realizado sobre él.

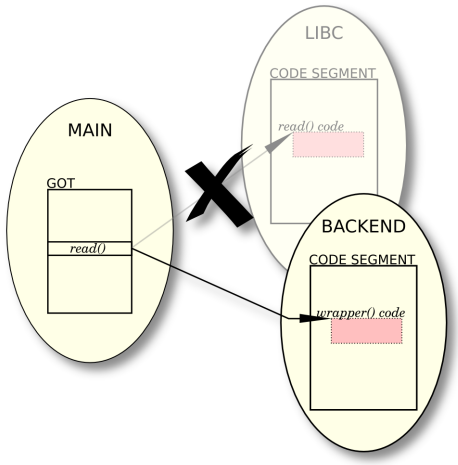
## 1.2.1. Mecanismos usados para interponer código

Todas las acciones de DITools tienen como objetivo final interponer código.

El mecanismo se ha implementado sólo sobre el sistema operativo Irix para MIPS®, tanto en 32 como 64 bits. De hecho, y como curiosidad, el mecanismo es el mismo en ambos modos de funcionamiento, cosa que no es para nada común en otras arquitecturas, donde el ABI de 32 bits es muy distinto al de 64 bits.

En Irix, la única diferencia realmente importante entre el ABI de 32 y el de 64 bits es el tamaño de los punteros y algunos tipos de variable como long int. En el resto ambos modos operan casi de la misma forma.

El mecanismo se basa fundamentalmente en alterar el contenido de la tabla GOT (“Global Offset Table”). Cada DSO tiene su propia tabla de punteros GOT. Esta tabla le indica al DSO la posición absoluta de los símbolos a los que accede y pertenecen a otros DSO. Así, alterando convenientemente estos punteros podemos redirigir la ejecución hacía código de los “backend”, tal como se puede ver en el siguiente diagrama:




En este diagrama se puede observar como interponer código en las llamadas a la función read(2) desde el programa principal simplemente alterando una entrada del GOT del ejecutable.

Estas redirecciones y la carga de estos "backend" se configura mediante unos ficheros de texto llamados "ficheros de comandos de reenlace".

Para poder entender completamente el funcionamiento de DITools es necesario conocer bien ELF para MIPS® ya que explotaremos sus estructuras de datos para realizar las interposiciones.

No obstante DITools no suele aprovechar todas las interrelaciones que ofrecen las estructuras ELF. En su lugar, para averiguar datos, usa la interfaz de rld(5), herramienta exclusiva del sistema operativo Irix. rld(5) es un "Runtime Linker" y se encarga de cargar los DSO y resolver los enlaces dinámicos entre objetos en tiempo de ejecución, al igual que hace ld en tiempo de compilación.

rld(5) ofrece un API para manipular los objetos en memoria, consultar estructuras de ELF, variar su propio comportamiento y/o inspeccionar los objetos en memoria. Esta API se usa numerosas veces en el código, relegando las operaciones de consulta a rld(5), lo cual liga seriamente DITools a esta aplicación.

### 1.2.2. La librería `libhook.so`

Esta librería en realidad es un pequeño DSO que contiene una única función que se instala en la sección de código de inicialización `.init`.

Cuando el "Runtime Linker" carga un ejecutable y todas sus librerías en memoria, antes de entregar el control al programa principal, lanza el código de inicialización de los DSO. Cada DSO puede aportar su sección `.init`. El orden de ejecución de estas secciones depende de dos cosas: de las dependencias entre los DSO y su orden de aparición. Si un DSO B depende de otro A, el código de inicialización de B se ejecutará después de A. Si tenemos dos DSO que dependen de otro, aunque en principio el orden de ejecución sea indistinto, el "Runtime Linker" ejecutará primero el código de inicialización del DSO que haya aparecido primero en la cadena de carga.

Antes de cargar una aplicación es posible forzar a rld(5) a añadir uno o más DSO a la cadena de carga. Esto se realiza mediante la variable de entorno _RLD_LIST. Por ejemplo, ejecutando la línea **`_RLD_LIST="mylib.so:DEFAULT" ./myprogram`** obligamos a que `mylib.so` se cargue antes que las librerías necesarias para ejecutar el programa `myprogram`.

El recurso descrito en el párrafo anterior se explota para introducir a `libhook.so` en la lista de librerías necesarias de una aplicación.

Una vez cargada en memoria, la librería `libhook.so`, mediante un mecanismo específico de rld(5), inserta la librería `liblink.so` exactamente después del ejecutable en la cadena de objetos de la aplicación. Esta cadena de objetos se genera en tiempo de carga por rld(5) y recibe el nombre `_RLD_LIST` (rld(5)).

Normalmente, cuando se inicia un programa, se ejecuta el código de inicialización del binario que se corresponde con el código de inicialización de C Library. Posteriormente se ejecutan, en el orden de aparición de `_RLD_LIST`, los bloques de inicialización de cada objeto. Añadiendo `liblink.so` justo después del programa principal se garantiza que su inicialización será antes que el resto de librerías, pero con el programa ya inicializado (aún así el programa instrumentado no ha comenzado a ejecutar código "propio" todavía).

De una forma no explicita, una vez que `libhook.so` termina su ejecución, el "Runtime Linker" pasa el control al código de inicialización de `liblink.so`.

Todavía no se ha averiguado el motivo por el cual es necesario usar `libhook.so` para cargar `liblink.so`. En principio no es esencial y debería funcionar correctamente cargando `liblink.so` ejecutando simplemente **`_RLD_LIST="liblink.so:DEFAULT" ./myprogram`**. Se supone que `libhook.so` debe ser un "hack" para rodear algún tipo de limitación o "bug" de rld(5).

### 1.2.3. La librería `liblink.so`

Donde está toda la maquinaria de gestión de interposiciones de DITools es en `liblink.so`. Esta librería es la encargada de procesar los ficheros de comandos, inicializar los "backends", instalar las interposiciones y, al finaliza el programa principal, deshacer todos estos cambios.

El proceso de inicialización de la librería `liblink.so` sigue el siguiente curso:

- **Lectura de los ficheros de comandos.** Terminadas las inicializaciones básicas, DITools comienza la lectura de los ficheros de comandos indicados en variables de entorno. En concreto son las variables DI_RUNTIME_FILE y DI_CFG_FILE. La primera contiene la ruta al fichero de comandos de sistema llamado "Runtime Config". Este fichero contiene los "backends" e interposiciones que se instalan primero y que deben instalarse en toda sesión de instrumentación. La otra variable, DI_CFG_FILE, indica el fichero de comandos de reenlace (el cual declara también sus "backends" e interposiciones) creado por el usuario, y específico para sus instrumentaciones.

  Al ser una lectura totalmente secuencial y sin comprobación de dependencias entre "backends", se va construyendo la lista de "backends" y la configuración final de las interposiciones, sin instalarlas todavía.

  Y aunque técnicamente sería posible y bastante sencillo, no se comprueba en ningún momento si se producen conflictos entre los comandos de interposición.

- **Carga e inicialización de los "backend".** En este paso se repite por cada "backend" la siguiente operación: carga del "backend" en memoria con ayuda de la API de rld(5) y ejecución de la función de inicialización `di_init_backend()` del "backend". Es importante tener en cuenta que no es lo mismo este código de inicialización que la sección `.init` de ELF.

  El orden de carga e inicialización de los "backends" es el orden de la primera aparición de cada uno de ellos en los ficheros de comandos.

- **Instalación de las interposiciones.** La instalación de las interposiciones se realizan siguiendo la configuración final resultante del primer paso. Es durante ese recorrido cuando se comprueba si los DSO usados están presentes o no en memoria, e igualmente, si las funciones referenciadas existen.

  Hay tres tipos de interposiciones: reenlaces, redefiniciones e interposiciones mediante "callback".

Más adelante en esta sección se explican con más detalle las interposiciones.

Terminadas todas estas operaciones, DITools devuelve el control al “Runtime Linker” y se continua con la inicialización del resto de las librerías. Posteriormente comienza la ejecución del programa y hasta su finalización DITools no vuelve a coger el control. Tan sólo se ejecutará el programa y el código que se ha interpuesto.

Con la finalización del programa DITools vuelve a activarse aprovechando el código de la sección de finalización de ELF (`.fini`).

Como el “Runtime Linker” ejecuta el código de finalización de los DSO en el orden inverso en que ejecuto el código de inicialización resulta que DITools será el último DSO en ejecutar su sección de finalización (`.fini`), y por lo tanto tenemos garantizado que hasta ese punto se habrá instrumentado enteramente la aplicación.

Nótese que si los “backend” tuvieran una sección de finalización `.fini`, ésta se ejecutaría. Por este motivo no es aconsejable definir las secciones `.init` y `.fini` en los “backend”: es difícil saber que interacción tendrán con DITools.

El código de finalización de DITools recorre la configuración aplicada en el primer paso y deshace todas las interposiciones de código en el orden en que las creo. Deshechas las interposiciones se puede ejecutar de forma segura las funciones de finalización de los “backend” (que no tiene nada que ver, si existiese, con la sección `.fini` de los mismos), es decir, sin que se vean afectados por ellas. Esta finalización se realiza en orden inverso a la inicialización.

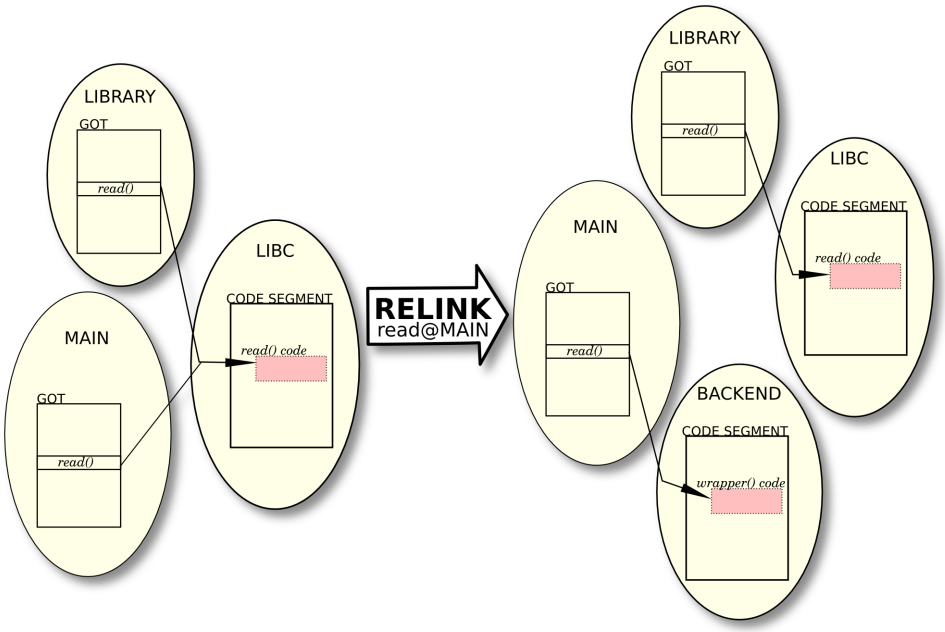
Finalmente, y más por elegancia que por necesidad, se descargan de memoria los “backend”. Hecho esto termina la ejecución de DITools.

### 1.2.4. Tipos de interposiciones

Seguidamente se explica, en orden de complejidad creciente, las interposiciones que implementa DITools.

El reenlace, consiste en redirigir una llamada a una función de otra librería (que en realidad es otro DSO) hacia una función de un “backend”. Es un cambio puntual que sólo afecta a un determinado objeto.

Su implementación consiste, *a grosso modo*, en sustituir el puntero del GOT (de dicho DSO) que apunta a la verdadera función por un puntero al código (función) que se interpone.

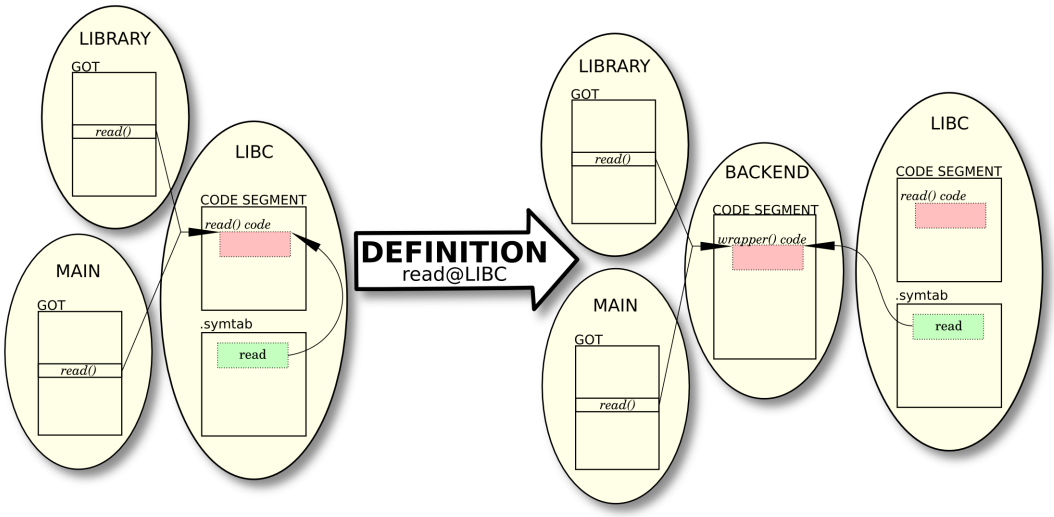



En este esquema vemos como se instala un reenlace en la aplicación, exactamente sobre el programa principal, sin afectar al resto de objetos en memoria.

El segundo tipo, la definición, en cambio afecta a nivel global (a todos los DSO de la aplicación). Consiste en sustituir una función pública de un DSO por otra función perteneciente a un “backend”. Esta interposición además tiene la ventaja de que es heredada también por los DSO cargados en tiempo de ejecución.

Su implementación consiste en modificar la información del símbolo que interceptamos. En el DSO que contiene ese símbolo existe una entrada en la tabla de símbolos de ELF que lo describe. En esa entrada hay un puntero que indica la posición de la función en el DSO. Reajustando ese puntero, haciéndolo apuntar hacia el código interpuesto, conseguimos que cada vez que el “Runtime Linker” resuelva un símbolo, sea él mismo quien haga el reenlace.

No obstante tiene un grave problema: no funciona cuando se redefine un símbolo que ya ha sido usado[4], ni cuando la variable de entorno LD_BIND_NOW está definida y con un valor que no sea 0. Esta variable de entorno, la cual ha de ser reconocida siempre por el “Runtime Linker” según el estándar ELF, indica que se desea que todas las referencias sean resueltas en tiempo de carga, antes de que comenzar a ejecutar el código del programa. Evidentemente, en este caso el “Runtime Linker” nunca más vuelve a consultar las tablas de símbolos ni a resolver llamadas, inutilizando totalmente las definiciones.




En este esquema vemos como se sustituye la función read(2) de `libc.so` por nuestra versión de la función. Como se puede ver, esta sustitución afecta a todos los DSO de la aplicación.

El tercer tipo, que no parece funcionar correctamente, recibe el nombre de “callback”. Los “callback” espían las llamadas que realiza un determinado DSO a otros DSO de un modo automático y programable.

Los “callback” se instalan como reenlaces o redefiniciones, según diga el usuario. En caso de reenlace se monitorizan todas las funciones de un determinado DSO. En el caso de las redefiniciones el código es muy caótico y no parece ser muy coherente. Deduzco que el resultado de este comando sería monitorizar todas las llamadas que se hacen a un DSO.

Los “callback” se implementan en tres funciones que cooperan con DITools para gestionar las intercepciones. En concreto son las funciones `di_callback_required()`, `di_pre_event_callback()` y `di_post_event_callback()`.

La mecánica es la siguiente: cuando se intercepta la llamada a la función instrumentada DITools pasa el control a la función `di_callback_required()`. Esta función recibe el nombre de la función que se ha interceptado y con él debe decidir si desea instrumentar o no esta llamada. En caso de no quererla interceptar simplemente tiene que devolver un cero. En caso de quererla interceptar debe devolver un número distinto de cero que luego será pasado a las otras dos funciones. Este número suele ser un identificador numérico que permite identificar que función se está interceptando.

Si se decide no interceptar la llamada, DITools transfiere el control al programa y la función se ejecuta normalmente.

Si en cambio se intercepta la llamada, DITools transfiere el control a la función `di_pre_event_callback()` si esta existe. Cuando esta termina se ejecuta la función normalmente y, al terminar esta, se transfiere llama a la función `di_post_event_callback()` (si esta existe).

No obstante, la implementación que analicé solamente llamaba a `di_pre_event_callback()`, ejecutaba la función y terminaba. El código encargado de la segunda parte estaba desactivado, sin comentar y aparentemente inacabado.

## 1.3. Limitaciones de DITools

El estudio hecho al código de DITools sirvió para identificar una serie de limitaciones y problemas

que a continuación se enumeran:

- **Se pueden usar sólo dos ficheros de comandos.** Recordemos que se usarán sólo dos ficheros de comandos de reenlace, y estos dos son los declarados en las variables de entorno DI_RUNTIME_FILE y DI_CONFIG_FILE. Tener sólo dos ficheros de configuración obliga a modificar con frecuencia fichero que pueden llegar a ser muy grandes. A la larga esto puede llevar al usuario a errores y confusión.
- **Carece de mecanismos de protección.** No se incorpora ningún mecanismo que contraste la información entre los ficheros de comandos y/o que deduzca si hay algún tipo de contradicción o interacción indeseable entre ambos ficheros. Esto puede llevar errores difíciles de detectar.
- **Alta dependencia respecto a rld(5) e Irix.** Al basarse tan fuertemente en mecanismos internos de Irix es fácil que se produzcan incompatibilidades serias de una versión a otra del sistema operativo. Además, al disponer tan sólo de una parte de la información (la parte que nos suministra rld(5)), es difícil implementar chequeos que comprueben la consistencia de los estructuras en memoria.
- **Las redefiniciones no son muy fiables.** Como se ha comentado antes, si el "Runtime Linker" ha resuelto alguna referencia a un símbolo, antes de haber instalado la redefinición en él, esta definición no tendrá efectos sobre algún o algunos DSO. De la misma forma la presencia de la variable de entorno LD_BIND_NOW inutilizará todas las definiciones que se instalen.
- **Los "callback" parecen no funcionar correctamente.** Los "callback" se presentan con un código confuso, poco documentado, sin pistas de como usarlos y al parecer es un mecanismo no resistente a programas multi hilo.

  Además se intentó, infructuosamente, realizar algunas pruebas para analizar su comportamiento. Al parecer el mecanismo no funciona del todo correctamente, especialmente con las llamadas al código de post-tratamiento de la interposición mediante "callback".
- **Poca documentación y código poco comentado.** No se ha encontrado ninguna documentación que explique el uso e implementación de DITools. El código está muy poco comentado y suele ser confuso en muchas partes. Esto ha llevado a que DITools, a pesar de ser una herramienta en uso durante años en el CEPBA, sea una caja negra que nadie se haya atrevido a tocar.

# Notas

1. En realidad nunca se ha llegado a considerar un suplemento de [Sys5ABI] ya que incumple muchos requisitos mínimos. Al parecer, esta implementación para PowerPC™ 64 del ABI es en realidad un burdo disfraz de ELF para la especificación [POPEN32].
2. La presencia de este elemento altera el algoritmo de resolución del enlazador dinámico para referencias desde este DSO. En lugar de comenzar la búsqueda de la referencia en el ejecutable, la búsqueda comienza en el mismo objeto. En caso de que este DSO no aporte una definición del símbolo referenciado, el "Runtime Linker" comenzará de nuevo la búsqueda por el ejecutable y seguirá con el resto de objetos como suele ser habitual.
3. En algunas plataformas como MIPS® esta sección no existe, aunque por regla general existe en todas las plataformas.
4. DITools dispone de una API que pueden usar los "backends" para realizar interposiciones en tiempo de ejecución. Sería en este caso cuando nos encontraríamos este problema.

# Capítulo 2. Trabajo realizado

En este capítulo documentamos el funcionamiento, decisiones tomadas y tareas realizadas sobre pDI-Tools a lo largo del proyecto.

El capítulo comienza por una descripción global de pDI-Tools que permite construir una imagen global sobre su funcionamiento.

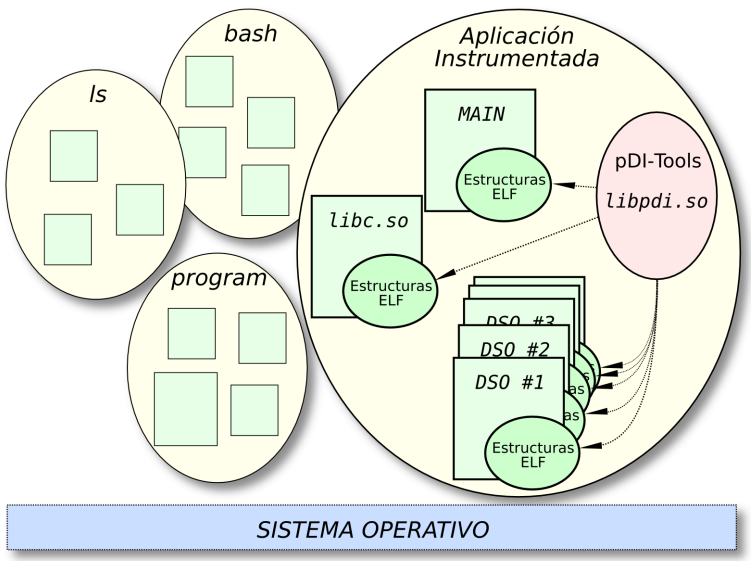
Seguidamente describimos las decisiones de implementación tomadas, hablando a grandes rasgos sobre el flujo de ejecución que seguirá pDI-Tools hasta los mínimos detalles como los nombres de variables.

Todo ello finaliza en una última parte donde se explica la implementación de los mecanismos de interposición. Partiendo de los conceptos más básicos para luego ir aumentando la complejidad hasta llegar a explicar con detalle los “callback” sobre INTEL® 386.

## 2.1. Funcionamiento

El funcionamiento de pDI-Tools, observado superficialmente, es muy similar al de DITools. La primera diferencia más evidente es que pDI-Tools está implementado todo en una única librería `libpdi.so`. Los mecanismos de interposición son esencialmente idénticos, a excepción del “callback”, el cual se implementa de un modo distinto, un poco más robusto y más eficiente.

El esquema de funcionamiento es muy similar al de DITools como se puede observar en este diagrama que muestra la interacción de pDI-Tools con los objetos de la aplicación instrumentada.




En este esquema observamos varias aplicaciones, más una instrumentada por pDI-Tools. Los objetos compartidos (DSO) de la aplicación se representan como rectángulos. En cambio, los DSO de DITools se muestran como dos elipses.

Al igual que su antecesor para usarlo debemos aprovechar la variable de entorno LD_PRELOAD (`_RLD_LIST` (rld(5)) en caso de Irix). Esta variable de entorno nos permite añadir un nuevo objeto compartido en el espacio de trabajo de una aplicación que instrumentamos. Durante su inicialización, que siempre es antes de comenzar la ejecución de la aplicación, se realizan todas las tareas de inicialización, configuración e interposición de código.

Seguidamente explicamos el proceso que sigue habitualmente pDI-Tools, desde su inicialización hasta que empieza a ejecutarse el programa instrumentado.

La inicialización de pDI-Tools comienza por recoger una serie de punteros *reales* a muchas funciones de las librerías de C. Cada puntero es absoluto a la definición de una función, libre de todos los mecanismos de enlace de ELF. Estos punteros permitirán a pDI-Tools acceder a las funciones de la librería de C sin verse afectado por interposiciones accidentales.

Acto seguido se recoge la configuración de pDI-Tools. pDI-Tools permite variar notablemente su comportamiento y funcionalidades sin ser necesaria una recompilación. pDI-Tools incorpora un sistema más potente de configuración que DITools. Este nuevo mecanismo de configuración se basa en ficheros de configuración y, por compatibilidad con DITools, acepta también variables de entorno. Los ficheros de configuración son sencillos ficheros de texto divididos en secciones en las cuales encontramos asignaciones a parámetros.

La configuración a su vez determina el siguiente paso de ejecución: la carga e interpretación de los comandos de interposición de código. Durante la configuración se indica que ficheros de comandos se usarán, que a diferencia de DITools estos pueden ser más de dos.

Pero permitir más de dos ficheros de comandos de interposición de código aumenta la complejidad a la hora de combinar los "backends" y los comandos. Se deben resolver posibles dependencias entre "backends". La resolución de estas posibles dependencias resultará en un orden de carga e inicialización de los "backends" junto a un guión final de comandos de interposición.

Una vez completado el guión final se realizan muchas comprobaciones. Lo primero es buscar los "backend" que intervendrán en esta sesión. Luego se comprueba si los objetos en memoria se corresponden con los indicados en los ficheros de comandos. También se comprueba que no haya interacción entre los diferentes comandos.

En estos momentos pDI-Tools está en condiciones de ejecutar los comandos. Al igual que en el antiguo DITools primero se cargan e inicializan los "backend" y posteriormente se instalan las interposiciones de código. La carga de los "backend" y su inicialización se realiza en el orden calculado anteriormente. Es en este punto donde se ejecuta la mayor parte del código dependiente de la arquitectura.

Llegados a este punto sólo queda realizar algunas tareas de limpieza y liberación de memoria y se transfiere el control al programa principal. A partir de aquí no se vuelve a retomar el control a no ser que la aplicación lo entregue, ya sea mediante su propia terminación o mediante una llamada explicita a la API ofrecida.

Generalmente durante la ejecución de la aplicación no se ejecuta código de pDI-Tools a excepción de los "stubs" de interposición, si estos existen. Es en este momento cuando se ejecuta el código interpuesto por los "backend".

Una vez el programa termina su ejecución (normalmente mediante una llamada a la función exit(3) o por la finalización de la función `main`) el código de finalización de pDI-Tools vuelve a retomar el control.

La finalización comienza por deshacer todas las interposiciones realizadas antes y durante la ejecución del programa. Esto nos permitirá ejecutar con tranquilidad y sin interferencias el código de finalización de los "backend".

Posteriormente se ejecuta el código de finalización de los diferentes "backend". Las rutinas de finalización de los "backends" se ejecutan en el orden inverso al de las rutinas de inicialización.

Acto seguido se descargan explícitamente los "backend" de memoria. Esto se realiza por cuestiones de corrección, ya que de todas formas esta operación la hace el sistema operativo al finalizar la ejecución de una aplicación.

Finalmente se imprime el último mensaje de "log", se cierran todos los ficheros abiertos y se libera toda la memoria usada.

## 2.2. Estructura y decisiones de diseño

En esta sección se discuten decisiones que se tuvieron que tomar antes y durante el desarrollo del software. Desde el sistema de compilación, pasando por el lenguaje, hasta las convenciones de programación usadas.

Todas las decisiones se hacen entorno a unas cuantas premisas y/o objetivos marcados que ya se han comentado en el capítulo de introducción. Aún así recordemos que los cuatro objetivos principales son:

- Ha de ser una aplicación muy veloz y ligera.
- Altamente portable manteniéndose simple.
- Sencillo de compilar, configurar e instalar.
- Debe ofrecer una API pero que esta no implique trabajar sólo con un determinado lenguaje.

En las siguientes subsecciones se discuten las principales decisiones tomadas para lograr estas metas.

### 2.2.1. Lenguaje y sistema de compilación

Debido a que se decidió no reusar el código de DITools en el desarrollo de pDI-Tools la primera decisión que se tuvo que tomar fue que lenguaje usar, y consecuentemente, el paradigma de programación adecuado.

El lenguaje escogido fue el C con un diseño de programación descendente. Los lenguajes C y C++ son idóneos para programar a bajo nivel. No obstante la programación orientada a objetos presenta varios inconvenientes en un proyecto de esta índole. Los motivos para desestimar el lenguaje C++ y la programación orientada a objetos se enumeran a continuación:

- **Excesivo código "overhead".** El binario de un programa compilado por C++ incluye mucho código necesario para implementar las características del lenguaje. Esto implica un importante "overhead" que se traduce en una ejecución más lenta. Esta aplicación va destinada principalmente a proyectos de instrumentación. La instrumentación de por sí introduce un importante "overhead". Si encima añadimos el "overhead" que implica usar C++ la instrumentación se ralentiza y podemos introducir demasiado ruido en los resultados de la instrumentación.
- **La complejidad del programa y sus datos no requieren la potencia de la OOP.** Es posible que el programa hubiese podido ser un poco más fácil de leer si hubiera sido escrito en C++, pero en realidad hubiera sido muy poco más. La cantidad de estructuras de datos que hay es muy reducida. Además estas estructuras son muy sencillas e inconexas entre ellas, lo cual nos hace prescindir de muchas de las ventajas de la orientación a objetos como la herencia, la sobrecarga de operadores o el polimorfismo.

  También es verdad que es posible implementar un programa orientado a objetos en lenguaje C puro[1]. No obstante esto hubiera aumentado la complejidad del programa, hubiera consumido más memoria y además hubiera hecho todo un poco más difícil de entender.

- **Obliga a los "backend" a usar C++.** Si se hubiese programado pDI-Tools con C++ esto obligaría a migrar todo el software existente para DITools a C++. Además nos obligaría a usar C++ en futuros proyectos. En cambio, al estar escrito en C, pDI-Tools puede usarse tanto con "backend" escritos en C como en C++ o cualquier otro lenguaje ya que nos libramos de toda la complejidad de interacción con C++.
- **Diferencias entre implementaciones de C++.** Las diferencias entre compiladores de C++ son mucho más acentuada que las que hay entre compiladores de C. A veces incluso encontramos serias diferencias entre versiones distintas de un mismo compilador para diferentes plataformas. Usar C++ hubiera ligado excesivamente pDI-Tools a un determinado compilador. Usando el lenguaje ANSI-C garantizamos una mayor portabilidad (tanto entre compiladores como entre plataformas) y a la vez evitamos desagradables sorpresas debidas a dichas diferencias.
- **C++ no siempre está disponible.** En entornos de desarrollo antiguos, o incluso algunos entornos de desarrollo modernos, no disponemos de C++. Por ejemplo el lenguaje C++ en un entorno GNU/Linux™ con GNU C Compiler, a pesar de ser algo instalado normalmente, no deja de ser opcional.

  Usando sólo C ANSI garantizamos que la aplicación se podrá compilar en casi cualquier sistema.

También se usan las aplicaciones Autoconf y Automake para gestionar la compilación del software. La herramienta Autoconf permite despreocuparse de las peculiaridades del compilador, de las librerías y de la arquitectura ya que es él quien se encarga de detectarlas y, en muchas ocasiones, ocultarlas. Automake automatiza la creación de los ficheros `makefile`, generándolos específicos para la versión de make instalada.

El motivo de escoger el par de aplicaciones Autoconf y Automake en lugar de otro equivalente como podría ser Imake es debido a su gran flexibilidad. Autoconf frecuentemente ha funcionado en sistemas en los que nunca se había ejecutado antes, lo cual es directamente imposible en un sistema de base de datos como Imake. Esto hace que este par funcione sobre una gran variedad de sistemas POSIX. Además, una vez construido el "script" `configure.in`, sólo es necesario disponer de una "shell" sh(1), un compilador de C y la herramienta make para poder compilar el programa. Imake, por el contrario, necesita estar instalado para poder realizar la compilación. Por último es mucho más difícil extender las funcionalidades de Imake debido a su naturaleza basada en una gigantesca base de datos: no es posible distribuir sólo los cambios y usarlos sin instalarlos en la base de datos de Imake.

## 2.2.2. Organización del código

Todas las funciones y símbolos públicos de pDI-Tools van precedidos del prefijo `_pdi_`. Esto reduce la posibilidad de que haya una colisión de nombres con los del programa a instrumentar.

En determinadas ocasiones se añade un infijo para agrupar determinadas funciones de tal forma que nos permita, sólo mediante el nombre, deducir a que parte del programa pertenece. Así tenemos el prefijo `_pdi_bcfg_` para las funciones encargadas de manejar los ficheros de comandos, el prefijo `_pdi_linux_` para todo los símbolos específicos para GNU/Linux™ y así sucesivamente.

Los prefijos registrados actualmente son los siguientes:

**Tabla 2-1. Prefijos de los símbolos de pDI-Tools**

| Prefijo | Descripción |
|---|---|
| `_pdi_` | Prefijo genérico. Todos los símbolos públicos de pDI-Tools *deben* estar precedidos por este prefijo. |

| Prefijo | Descripción |
|---|---|
| `_pdi_becfg_` | Este prefijo agrupa todas las funciones relacionadas directamente con la gestión de los ficheros de comandos. |
| `_pdi_ebe_` | El infijo “ebe” se traduce como “Elf Backend”. Todas las funciones con este infijo acceden directa o indirectamente a las estructuras de información ELF tanto para consultas como para realizar modificaciones. No obstante su implementación es muy genérica o totalmente desligada de la arquitectura subyacente, así que se trata de un código altamente portable. Estas funciones logran esta independencia mediante una capa de abstracción específica para la plataforma. |
| `_pdi_arch_` | Todas las funciones precedidas por este prefijo son totalmente dependientes de la arquitectura y deben ser implementadas por el código específico para la plataforma. En realidad nunca encontraremos símbolos exportados con este prefijo. Este prefijo en realidad es una macro que se traducirá al prefijo específico para la plataforma: así si llamamos a la función `_pdi_arch_init` en realidad estamos llamando a `_pdi_linux_init` en GNU/Linux™, o a `_pdi_irix_init` en Irix, etc. |
| `_pdi_safe_` | Este grupo de símbolos son en realidad punteros a funciones. Estos punteros apuntan a funciones de la librería de C. Así por ejemplo `_pdi_safe_fputc` es un puntero a la función `fputc`. Estos punteros nos permiten llamar a funciones de C sin vernos afectados por instrumentaciones agresivas. |
| `_pdi_irix_` | Funciones y símbolos específicos para la implementación de pDI-Tools para Irix. |
| `_pdi_linux_` | Funciones y símbolos específicos para la implementación de pDI-Tools para GNU/Linux™. |
| `_pdi_solaris_` | Funciones y símbolos específicos para la implementación de pDI-Tools para Solaris. |

La convención usada para los tipos, constantes y variables es la siguiente:

- Las variables y funciones globales se distinguen mediante el prefijo `_pdi_`. La variable debe estar escrita toda en minúsculas, a no ser que este formada por varias palabras. En ese caso la primera letra de cada palabra, a excepción de la primera, estará en mayúsculas. Por ejemplo:

```
PDI_ELF_OBJ *_pdi_objList;
int _pdi_init(void);
int _pdi_initSafeFuncs(void);
```

- Las variables locales y funciones locales no deben ir nunca precedidas del prefijo `_pdi_`. No obstante su identificador debe comenzar siempre por minúsculas, y si es un identificador formado por múltiples palabras la primera letra de cada palabra estará en mayúsculas a excepción de la primera, o se separarán las palabras mediante un signo de subrayado. Por ejemplo:

```
int myLocalVar;
int my_var;
int refreshObjects(void);
```

- Las constantes del preprocesador y los tipos de datos siempre están en mayúsculas. Además siempre llevan el prefijo `PDI_`. Si además la constante o tipo está ligado a la arquitectura se le añade un infijo que la identifique como tal. Así las constantes y tipos específicos de GNU/Linux™ llevan el prefijo `PDI_LINUX_`, mientras que los de Irix se identifican por el prefijo `PDI_IRIX_`. El uso del signo de subrayado para aumentar la legibilidad del identificador es arbitrario, es decir, no hay una regla fija.
- Nunca se nombra directamente los tipos de datos ligados a la plataforma (por ejemplo los declarados por el fichero `elf.h`). En su lugar se accede a ellos mediante macros adecuadas. Por ejemplo, si se usa tal cual el tipo Elf32_Addr se comete un error ya que el código queda totalmente ligado a plataformas de 32 bits. La forma correcta de referirse a estos tipos es mediante la macro `ElfW(`*`X`*`)`, donde *`X`* es el tipo que queremos usar. En el ejemplo anterior usaríamos el tipo ElfW(Addr). De esta forma es el programa `configure.in` quien decide si el tipo debe traducirse a la versión de 32 o 64 bits.
- Se evitan en la medida de lo posible las construcciones `#if ... #else ... #endif` complejas, ya que estas construcciones, sin ningún tipo de control, llevan a códigos ilegibles y muy propensos a errores.

## 2.2.3. Distribución de los ficheros

pDI-Tools se distribuye en un fichero `.tar.gz` con un nombre de la forma `pditools-`*`VERSIÓN`*`.tar.gz` donde *`VERSIÓN`* indica la versión que contiene de pDI-Tools.

La estructura de directorios en la distribución de pDI-Tools obedece a las indicaciones hechas por GNU. En el directorio principal se encuentra el "script" de configuración `configure.in` y ficheros de texto describiendo el contenido del paquete. También encontramos los siguientes subdirectorios:

### Directorios contenidos en la distribución de pDI-Tools

`./doc/`

En este directorio se encuentran todos los "script" y ficheros fuente XML DocBook necesarios para construir la documentación. Además se incluye la documentación ya generada en diversos formatos (entre otros HTML, PDF y PostScript®).

`./etc/`

Aquí se encuentran los ficheros de configuración independientes de la plataforma.

`./include/`

Separamos en este directorio todos los ficheros de cabecera necesarios para poder programar con pDI-Tools. Son útiles para la programación de objetos "backend" o aplicaciones que interaccionen con pDI-Tools.

Dentro de este directorio encontramos más subdirectorios donde cada uno contiene los ficheros de cabecera específicos para una plataforma. Así por ejemplo encontramos los directorios `./include/irix/`, `./include/linux/` y `./include/solaris/`.

`./src/`

Dentro de este directorio encontramos otros subdirectorios con el código fuente necesario para construir pDI-Tools, sus utilidades y programas de "test". Y es en el directorio `./src/link/`

donde está el código fuente que termina generando el núcleo de pDI-Tools: `libpdi.so`.

Dentro del directorio `./src/link/`, al igual que pasaba en el directorio `./include/`, encontramos varios subdirectorios donde cada uno contiene el código específico para una determinada plataforma.

# 2.3. Tipos de interposiciones

En este apartado explicamos los fundamentos y estrategias usadas para implementar las interposiciones de código. Además estos mecanismos han derivado en tres posibles formas de interponer código. Cada una de ellas explota una característica distinta del estándar ELF para llegar al mismo objetivo.

## 2.3.1. Conceptos básicos para crear una interposición de código

Esta explicación es un recordatorio que enumera y explica superficialmente las principales características de ELF usadas para crear las interposiciones de código. Todos estos conceptos están explicados con mucho más detalle en la sección *Introducción al ELF*.

Los protagonistas de esta parte son las secciones dinámicas `DT_PLTGOT` y `DT_SYMTAB` de ELF, con una importante colaboración por parte de la sección `DT_JMPREL` (o `DT_REL` en su defecto). Explotaremos la modificación de los datos contenidos en estas secciones para interponer código en los DSO.

Cada uno de los DSO en memoria disponen de estas tres secciones. La sección `DT_JMPREL` nos proporciona una estructura ElfW(Rel) por cada función que contiene o es referenciada desde el objeto a la que pertenece. Así pueden haber tanto estructuras ElfW(Rel) con punteros a funciones del DSO como otras que permiten resolver direcciones de funciones en otros DSO que se usan en este DSO.

Las estructuras ElfW(Rel) de funciones externas son interesantes ya que indican donde hacer modificaciones para hacer que el programa use una determinada función u otra. Estas estructuras relacionan la tabla de símbolos (`DT_SYMTAB`) con las entradas (totalmente dependientes de la arquitectura) de la sección `DT_PLTGOT`.

De hecho la sección `DT_JMPREL` se crea a partir de la tabla de símbolos (`DT_SYMTAB`) a la vez que la sección `DT_PLTGOT` por el "Runtime Linker" al cargar los objetos en memoria. La sección `DT_SYMTAB` contiene una relación de todos los símbolos que trae o usa el DSO. De ellos un subconjunto son las funciones. Las estructuras de información que describen las funciones indican varias cosas, que de ellas interesan principalmente dos: si es una función aportada por el DSO (con un puntero a la misma) o si es una función externa que usa el DSO.

El contenido y distribución de la sección `DT_PLTGOT` es algo totalmente específico para cada arquitectura. No obstante se puede clasificar su contenido en dos clases o construcciones comunes:

- **Global Offset Table.** Se trata de una tabla de punteros que se consulta en tiempo de ejecución para resolver los enlaces externos al DSO. Esta tabla la construye el "Runtime Linker" y suele abreviarse como GOT. Este es el mecanismo que se usa en la arquitectura MIPS® (tanto en 32 como 64 bits).
- **Procedure Linkage Table.** Se trata de una serie (o tabla) de subrutinas escritas en ensamblador encargadas de transferir el control a una función externa al DSO. Estas rutinas se escriben en tiempo de ejecución por el "Runtime Linker" y son simples "stubs" que transfieren el control a la función

enlazada o al “Runtime Linker”, según la referencia esté resuelta o no. Este es el mecanismo que se usa en la arquitectura SPARC (tanto en 32 como 64 bits).

El resto de casos suelen ser variantes o híbridos de estos dos casos. Por ejemplo, en la arquitectura INTEL® 386 encontramos ambas tablas en la sección `DT_PLTGOT`.

El “Runtime Linker” usa la información contenida la tabla de símbolos y la tabla de reubicación para resolver los enlaces en tiempo de ejecución. El “Runtime Linker” sólo suele recibir ElfW(Rel) como toda la información necesaria para resolver un símbolo en tiempo de ejecución: en dicha estructura encuentra el índice del símbolo que describe el objeto que debe resolver. De la información de símbolo extrae el nombre y comienza una búsqueda por todos los objetos recopilando una lista de funciones candidatas. La dirección de estas funciones las encuentra en la información de símbolo de cada una de ellas. Por último se aplica, si es necesario, un esquema de precedencias para elegir la información de símbolo correcta. Una vez conoce la dirección de la función que debe resolver, vuelve a consultar el contenido de la estructura ElfW(Rel) y de ella extrae la posición de la información de enlace en la sección `DT_PLTGOT`. Y altera el contenido de dicha porción del `DT_PLTGOT` con la información necesaria (o código necesario) para que en próximas ejecuciones de la función no se tenga que volver a resolver el símbolo.

Por último debemos explicar que existe una tabla “hash” ideada para acelerar las búsquedas de símbolos por su nombre. Esta tabla la encontramos con ayuda del “Dynamic Tag” `DT_HASH`. Ésta permite hacer la búsqueda de un símbolo en un tiempo mínimo, reduciendo notablemente el tiempo empleado en las búsquedas, y se usa en todas las búsquedas por nombre de símbolo.

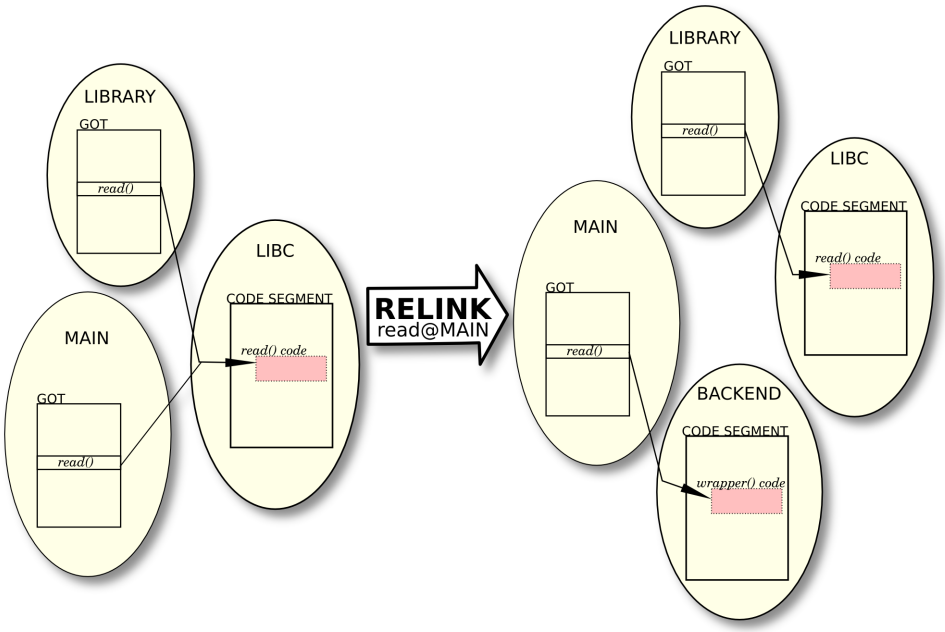
### 2.3.2. Mecanismo de reenlace

Un reenlace consiste en modificar las estructuras ELF de un determinado objeto para que cuando intente llamar a una función externa en realidad llame a una función suministrada por un “backend”.

A favor de su simplicidad tenemos que es la interposición más sencilla de usar y más controlable ya que sólo afecta a un determinado objeto.

El reenlace consiste en una modificación puntual de la sección `DT_PLTGOT`. Para encontrar la entrada que queremos modificar se usa la información de la estructura ElfW(Rel) del símbolo que queremos reenlazar. Esta estructura contiene un puntero a dicha entrada.

Retocando esta entrada adecuadamente se provoca que, cada vez que un determinado DSO llame a una determinada función, el control se transfiera en realidad a una función de un “backend”.




En este esquema vemos como se instala un reenlace en la aplicación, exactamente sobre el programa principal, sin afectar al resto de objetos en memoria. Por claridad en cada DSO solamente se muestran las partes que intervienen (GOT o el "Code Segment").

Seguidamente añadimos un ejemplo de fichero de comandos de interposición que muestra como instalar unos cuantos reenlaces:

```
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Ejemplos de reenlaces
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

; Definimos los alias a las librerías y los backends que usaremos
#backend BACKEND        ./testbe.so
#define  SOME_LIB       libtest.so

; Comandos de reenlace
# commands

R MAIN fputc BACKEND fputc_wrapper
R SOME_LIB write BACKEND write_wrapper
R * read BACKEND read_wrapper
```

En este ejemplo primero definimos los objetos que usaremos. Obsérvese que `BACKEND` es nuestro "backend" y al declararse se usa la directiva `#define`. El identificador `SOME_LIB` es un objeto dinámico que usa nuestro ejecutable, cuyo DSO recibe el nombre `MAIN`.

Seguidamente, separados por la directiva `#commands`, se listan los comandos de reenlace.

El primero provocará que todas las llamadas a fputc(3) desde `MAIN` sean redirigidas al "wrapper" (función) `fputc_wrapper` del "backend".

El siguiente comando redirige las llamadas a la función write(2) realizadas desde la librería `libtest.so` (`SOME_LIB`) al "wrapper" `write_wrapper` del "backend".

Por último se redirigen, en todos los objetos instrumentables (ver en el manual de usuario las opciones de configuración `donttouch_backends` y `donttouch_pdi`), la llamada a read(2) hacia al "wrapper" `read_wrapper`.

### 2.3.3. Mecanismo de redefinición

Las redefiniciones son totalmente opuestas al mecanismo de reenlace. Un reenlace sólo afecta a un único objeto, mientras que una redefinición afecta a *todos* los objetos instrumentables (ver nota).

Debido a esto la redefinición necesita una cierta persistencia ante nuevos objetos cargados en memoria en tiempo de ejecución. Es decir, que cuando un nuevo objeto se cargue en memoria, este se vea afectado también por la redefinición automáticamente sin que sea necesario instalar el reenlace a mano.

Para lograr tal objetivo la estrategia usada consiste en modificar la información de símbolo en el objeto que exporta la función.

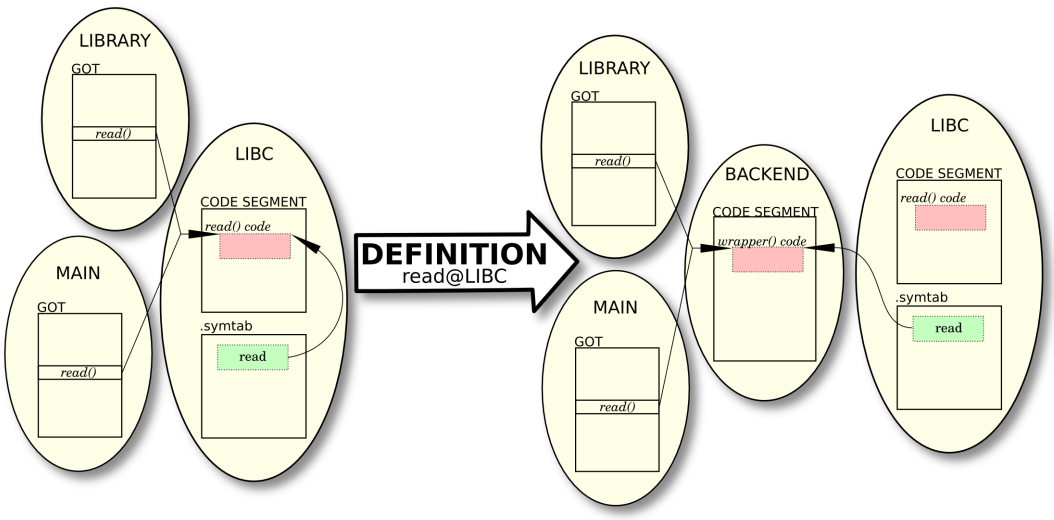
Como se ha explicado anteriormente el "Runtime Linker", cuando debe resolver un símbolo, busca primero de todo el objeto que exporta la función a resolver. En él busca la información del símbolo que resuelve ya que esta contiene un puntero al código que la implementa. Después copia/aplica esta dirección en la sección `DT_PLTGOT`.

Modificando dicha información de símbolo se provoca que el "Runtime Linker" siempre resuelva en nuestra nueva definición.

No obstante no se puede confiar en que el símbolo no haya sido resuelto todavía, así que no sólo basta con alterar esa entrada. Por ejemplo consideremos que está definida la variable de entorno LD_BIND_NOW: todos los enlaces ya han sido resueltos en tiempo de carga, y por lo tanto nunca se aplicará nuestra redefinición (las redefiniciones se instalan en tiempo de ejecución).

Esto nos obliga a realizar, antes de terminar, un reenlace incondicional sobre todos los objetos en memoria para garantizar que se antepone siempre nuestra nueva definición.

El resultado de aplicar la redefinición se puede ver en este gráfico:




En este esquema vemos como se sustituye la función read(2) de `libc.so` por nuestra versión de la función. Como se puede ver, esta sustitución afecta a todos los DSO de la aplicación.

Si el parámetro de configuración *`donttouch_backends`* está activado (por defecto sí), la redefinición no afectará a los "backend". En caso de activarse esta opción se debe trabajar con mucho cuidado ya que es fácil que se generen intercepciones recursivas. Por último, pDI-Tools sólo se verá afectado por las redefiniciones en caso de desactivarse el parámetro *`donttouch_pdi`*.

Seguidamente añadimos un ejemplo de fichero de comandos de interposición que muestra como instalar una redefinición:

```
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Ejemplo de redefinición
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

; Definimos el libc de GNU/Linux
#define LIBC            /lib/libc.so.6
#backend BACKEND        ./testbe.so

; Comandos de reenlace
# commands

D LIBC read BACKEND read_wrapper
D LIBC write BACKEND write_wrapper
```

Primero de todo definimos un alias a la librería de C. De esta forma declaramos que deseamos usarla y además le asignamos un alias. Los alias son muy prácticos por ejemplo en caso de que cambiar la ruta de la librería.

Seguidamente se instalan dos redefiniciones sobre la librería de C. En estos comandos indicamos que queremos sustituir la funciones read(2) y write(2) de la librería de C por nuestras propias versiones, `read_wrapper` y `write_wrapper` en el "backend".

A partir de ahora cualquier llamada desde cualquier DSO a read(2) y/o write(2), con excepción de C Library (no se verá afectado ya que las llamadas dentro de un mismo objeto, por regla general, se realiza mediante saltos relativos) y los objetos no instrumentables, se transferirá en realidad a nuestras funciones "wrapper".

### 2.3.4. Mecanismo de "callback"

Los "callback" es un tipo de interposición de código muy distinta al resto. Son aptos para realizar una interposición masiva sobre todas las llamadas de un DSO a cualquier otro DSO. Mientras que los reenlaces y las redefiniciones, desde el punto de vista de un DSO, sólo actúan sobre una única función objeto, un "callback" intercepta todas las llamadas que realiza un DSO a otros DSO.

Además con los reenlaces o las redefiniciones, durante la ejecución del programa instrumentado, sólo se ejecuta código de los "backend" ya que pDI-Tools se ha encargado de redirigir el control hacia ellos. En el caso del "callback" interviene tanto código del "backend" como de pDI-Tools y estos cooperan para realizar la interposición. En estas interposiciones quien recibe el control primero es el "callback handler" (también llamado en ocasiones "generic wrapper"), un código que forma parte de pDI-Tools encargado de gestionar la intercepción. Este código exige al "backend" tener implementadas dos o tres funciones:

```
int PDI_BE_FUNC_CB_REQ(char *func_name);
```

Esta función debe decidir si la intercepción que ha realizado pDI-Tools interesa o no. Por ello recibe una cadena con el nombre de la función interceptada. Si interesa interceptar esta función debe devolver un valor distinto de cero. Este valor se pasará a las otras funciones por lo que suele ser un identificador numérico que identifica a la función o un grupo de funciones. Si no interesa interceptarla se devuelve el valor 0.

```
void PDI_BE_FUNC_PRE_CB(int virtual_processor, int event_id, ...);
```

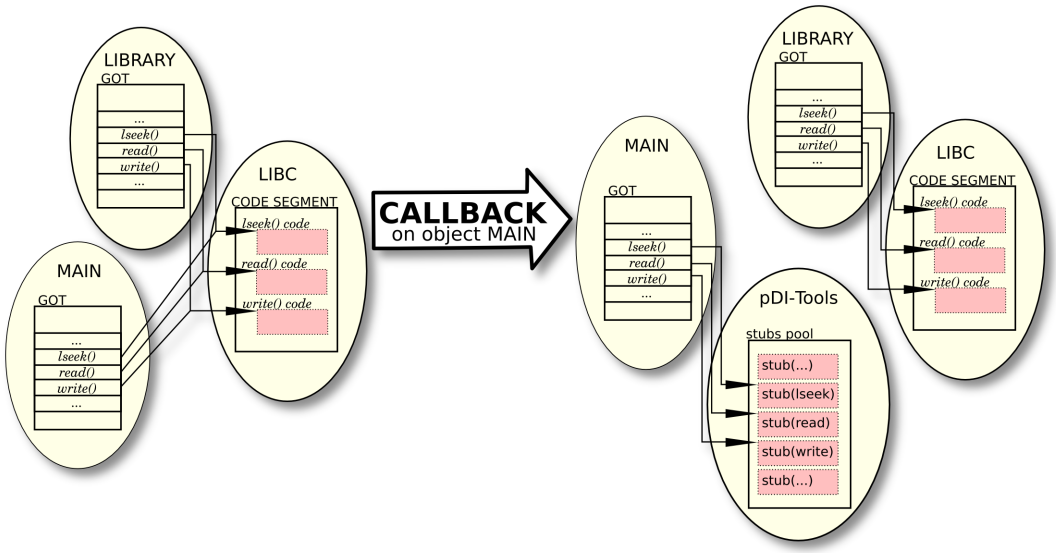
Esta función recibe un identificador de "thread" en el que se ha interceptado la llamada (para saber más sobre como se calcula este valor ver Capítulo 8), un identificador de evento que es el valor devuelto por `PDI_BE_FUNC_CB_REQ` y una lista de parámetros variable que son los parámetros de la llamada interceptada.

Esta función se ejecuta justo antes de que se ejecute la función interceptada. Se ha de tener cuidado al modificar los parámetros de llamada ya que, dependiendo de la arquitectura, pueden llegar alterados a la función instrumentada.

```
void PDI_BE_FUNC_POST_CB(int virtual_processor, int event_id, int retval);
```

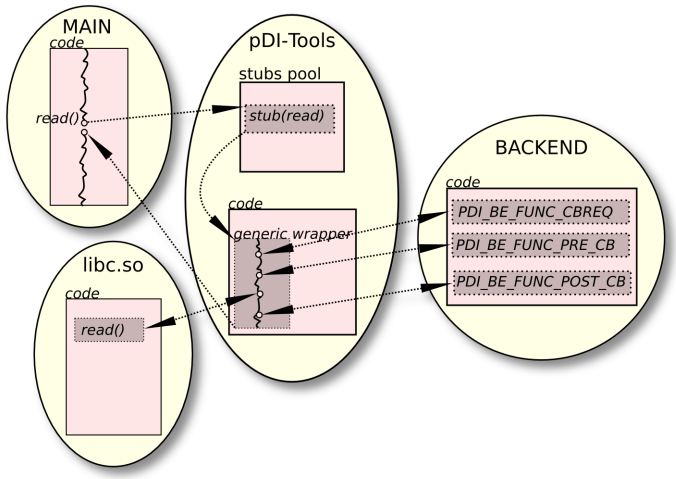
Esta función se ejecuta al finalizar la función interceptada. Recibe un identificador de "thread" en el que se ha interceptado la llamada, un identificador de evento y el valor devuelto por la función interceptada.

El método usado para interponer el código en esta ocasión es totalmente distinto del usado para los reenlaces y definiciones. En lugar de enlazar el "wrapper" directamente, en este método se enlazan unos "stubs" creados dinámicamente encargados de llamar al código que gestiona los "callback" de los "backends".



Podemos observar como el mecanismo de “callback” consiste en reenlazar cada función del objeto instrumentado con unos “stubs” creados por DITools. Cada “stub” se encarga de llamar al “callback handler” con parámetros distintos. Por motivos de espacio y claridad sólo se muestran las llamadas que se hacen a C Library. Podría haber llamadas a otros DSO y se redirigirían también al “callback handler”.

El resultado de uno de estos reenlaces es una sucesión de llamadas que se plasma en el siguiente diagrama para el caso de la función read(2):




En este diagrama se observa como al llamar a la función read(2) desde `MAIN` en realidad se transfiere el control a un “stub” que indica al “generic wrapper” que función estamos instrumentando. Esta función se encarga de transferir el control a las funciones de control de “callback” del “backend” y al verdadero código de read(2).

Los “stubs” se crean en tiempo de ejecución por pDI-Tools. Un “stub” es una pequeña porción de código que deja, ya sea en registros o en la pila, información sobre la llamada interceptada. Acto seguido salta al “callback handler”.

El “callback handler” es una función, también llamada “generic wrapper”, escrita en ensamblador. Esta función se encarga de llamar a la función interceptada y a las funciones del “backend” encargadas de gestionar el “callback”: `PDI_BE_FUNC_CB_REQ`, `PDI_BE_FUNC_PRE_CB` y `PDI_BE_FUNC_POST_CB`.

La función “callback handler” está escrita en ensamblador ya que debe interponer código sin alterar el contenido de la pila ni los registros desde el punto de vista de la llamada a la función interceptada. En INTEL® 386 los parámetros se pasan por la pila y no hay manera de determinar su número ni tipo de una forma genérica. El “callback handler” debe ingeniárselas para ejecutar el código de la función sin que se note en la pila el paso por el “stub” y por el propio código del “callback handler”. Además debe ejecutar el código de las funciones `PDI_BE_FUNC_PRE_CB` y `PDI_BE_FUNC_POST_CB`.

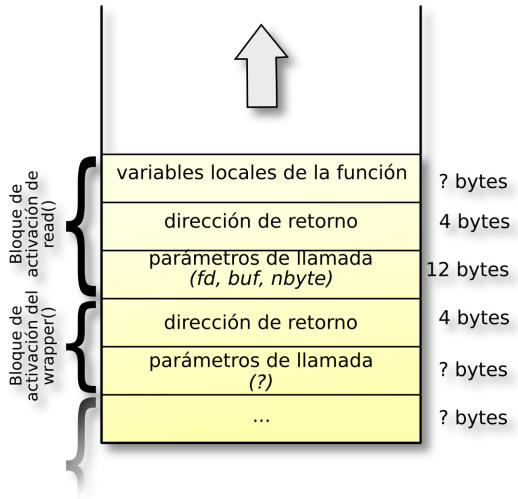
Esto implica a bajo nivel una gran diferencia respecto a los reenlaces y redefiniciones. Los “wrapper”, para interceptar y ejecutar una llamada, deben pasar de nuevo los parámetros a la función. Con los “callback” esto no es necesario, y además, no sólo se conserva casi todo el aspecto que debería tener la pila si no hubiese sido instrumentada, sino que además se conservan los valores de los registros.

Para entender esto bien, seguidamente detallamos las diferencias entre reenlace y “callback”.

La interposición de código en los reenlaces y redefiniciones se consigue mediante la creación de funciones que copian los parámetros y transfieren el control a la función interceptada. Por ejemplo:

```
/* read() wrapper */
ssize_t read_wrapper(int fd, void *buf, size_t nbyte)
{
  ++read_no;
  return read(fd, buf, nbyte);
}
```

Obsérvese que este “wrapper”, una vez incrementado el contador `read_no`, copia los parámetros de entrada y transfiere el control a la función read(2). Una vez ésta acaba, devuelve un valor que el “wrapper” devuelve a su vez. Si parásemos la ejecución justo al comienzo de la función interceptada, tendríamos el siguiente contenido en la pila (nos referimos al caso del INTEL® 386, la única arquitectura actualmente que soporta el mecanismo de “callback”) donde el top de la pila es la posición más alta:




Como podemos ver en el gráfico sólo se conocen los tamaños de los elementos apilados por el “wrapper”. No hay una manera única y simple de averiguar cuantos parámetros habrá apilado el llamador. De igual forma, read(2) no podrá consultar bien los parámetros y pila del llamador.

De aquí se desprenden varios problemas. El primer problema es que la función a interceptar (y su llamador) no respete el ABI de la arquitectura e intente consultar el contenido de la pila que no le corresponde encontrando en su lugar un fragmento de pila del “wrapper” y no el de la función que le llamaba. Otro posible problema es que espere parámetros por registro, los cuales habrán sido machacados anteriormente por el “wrapper”. El primer caso, aunque improbable, es posible encontrarlo. El segundo caso en cambio es común en rutinas programadas en ensamblador.

El “callback handler” es una rutina programada a bajo nivel capaz de solventar dichos problemas. Es una rutina compleja y habilidosa ya que salva el estado de ejecución (pila y registros) y así interpone código sin que se vea afectada la llamada interceptada.

El “callback handler” una vez toma el control lo primero que hace es decodificar la información proporcionada por el “stub” y apilarla en una pila secundaria e interna. Esta pila la usa el “callback handler” para guardar su estado en caso de llamadas reentrantes y para guardar parte del estado de la pila y los registros en el momento de la intercepción.

El siguiente paso es llamar a la función `PDI_BE_FUNC_CB_REQ` del “backend”. Esta función debe decidir si se desea procesar, o no, la función interceptada. Esta función devuelve, en caso de quererse interceptar la función, un valor distinto de cero que luego se entrega a las funciones

`PDI_BE_FUNC_PRE_CB` y `PDI_BE_FUNC_PRE_CB` en concepto de identificador de evento. En caso de devolver cero se restaura el estado del programa y se transfiere el control a la función interceptada.

La ejecución continua con la llamada de la función `PDI_BE_FUNC_PRE_CB` (si esta existe) pasándole el identificador asignado anteriormente. Una vez terminada la ejecución de `PDI_BE_FUNC_PRE_CB` se ejecuta la función original. Cuando esta termina de ejecutarse se transfiere el control a `PDI_BE_FUNC_POST_CB` (si esta existe). Y por último se restaura la ejecución normal del programa.

A pesar de su potencia, los "callback" tienen una limitación: no pueden alterar la ejecución del programa. Debido a que el mecanismo salva el estado de ejecución antes y después de la llamada a la función interceptada, las funciones `PDI_BE_FUNC_PRE_CB` y `PDI_BE_FUNC_PRE_CB` no pueden alterar ni los parámetros ni el valor devuelto por ella.

Si se instala un "callback" sobre un DSO luego ya no se puede instalar ningún otro tipo de interposición sobre cualquier función del mismo DSO ya que ambas entrarían en conflicto.

Para finalizar mostramos un fichero de comandos que instala una interposición mediante "callback" con pDI-Tools:

```
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Ejemplo de redefinición
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

; Definimos el libc de GNU/Linux
#backend BACKEND        ./testcb.so

; Comandos de reenlace
# commands

C MAIN * BACKEND
```

Con este comando, cualquier llamada a otros objetos que realice el programa principal será redirigida hacía las rutinas de gestión del "callback" en el "backend" `testcb.so`.

## 2.4. Mejoras y extensiones

En esta sección comentamos limitaciones observadas en pDI-Tools y las evaluamos una por una. En esta evaluación sopesamos el tipo de limitación, el interés del usuario por que este implementada, la viabilidad de su resolución y como resolverla.

De esta discusión sacaremos como conclusión que mejoras propuestas se implementarán en el futuro y que otras no. Entre otros motivos para desestimar una mejora, tenemos que no sea de gran relevancia para el usuario, o bien que la solución implique demasiado coste de cálculo y por lo tanto lentitud, o bien implique una solución demasiado compleja que posiblemente nunca llegue a funcionar correctamente.

### Se mantiene un buen nivel de compatibilidad con DITools

No es exactamente una limitación ya que se ha hecho intencionadamente y se podría considerar una "feature". No obstante mantener esta compatibilidad tiene un coste que se manifiesta en forma de ciertas limitaciones en pDI-Tools.

Con el tiempo se irá rompiendo poco a poco esta compatibilidad y así poder dar nuevas funcionalidades más potentes y sencillas de usar.

### No se pueden interceptar llamadas internas de un DSO

La limitación más importante de pDI-Tools viene dada por la estrategia usada para interponer código. No se pueden instrumentar llamadas estáticas dentro de un mismo DSO. Debido a que las llamadas dentro de un mismo DSO no hacen uso de las estructuras ELF, ya que son saltos relativos dentro del mismo objeto, pDI-Tools en principio no puede interponer código.

Se está contemplando, para el futuro, la posibilidad de interponer código en estos casos usando la información que suministra ELF sobre los símbolos estáticos junto a la información de depuración dada por el estándar DWARF (es la información de depuración ofrecida por el compilador). No obstante, si se implementa, este tipo de interposiciones sólo se podrán realizar cuando se disponga de la información de depuración (la información de depuración sólo se incluye si se compila el programa con el "flag" `-g` del compilador de C).

Además resolver este problema introduce problemas de portabilidad, ya no sólo entre sistemas operativos y arquitecturas, sino entre compiladores y herramientas de generación de binarios.

### Es difícil interceptar llamadas con parámetros variables

Otra importante limitación es la falta de un mecanismo similar al reenlace (o redefinición) que permita instrumentar funciones con una cantidad variable de parámetros.

Por ejemplo, supongamos que instrumentamos la función printf(2). Esta función recibe una cadena de formato y un número variable de parámetros. La cantidad de parámetros que se procesarán depende del contenido de la cadena de formato. Cuando un "wrapper" recibe el control no puede saber fácilmente cuantos parámetros se le han pasado: de hecho ni el mismo printf(2) lo sabe. Lo que hace una función con una cantidad de parámetros variables es procesarlos como una lista y parar cuando se cumple una cierta condición. Construir un mecanismo que copie una serie de parámetros no es una tarea sencilla ya que sería algo ligado a la arquitectura y totalmente específico para la función interceptada.

No obstante, si se dispone de una versión de la función que use una lista de argumentos variable (`stdarg.h`), se puede hacer un apaño. Por ejemplo, un posible "wrapper" de funciones printf(2) usando vprintf(2) sería:

```
static int printf_count = 0;

int printf_wrapper(char *format, ...)
{
  va_list args;
  int ret;

  printf_count++;

  va_start(args, format);
  ret = vfprintf(format, args);
  va_end(args);

  return ret;
```

```
}
```

Esto tiene dos problemas: no instrumentamos printf(2), en realidad instrumentamos algo parecido. Y el peor: no siempre tenemos una versión de la función preparada para aceptar parámetros vía va_list.

En este último caso el único mecanismo que puede gestionar una función de parámetros variables es el “callback”, debido a que nunca copia los parámetros, sino que entrega el control a la función interceptada como si virtualmente no se hubiese interpuesto ningún código.

### No se puede acotar el radio de acción de un “callback”

Un problema al usar el mecanismo de “callback” es que es muy radical: o intercepta todas las llamadas de un objeto o no intercepta ninguna. Instalar un “callback” sólo para interceptar una función es algo un poco drástico. Hasta el momento no se ha implementado ningún mecanismo que permita reducir el radio de acción de un “callback” sólo a determinadas funciones.

La solución a esto no es técnicamente compleja y no sería difícil de implementar. No obstante esta es una limitación resultante de mantener la compatibilidad con DITools. En el futuro se añadirá esta funcionalidad o en todo caso una nueva interposición derivada del “callback” que actúe sólo sobre una colección de funciones.

### Los “callback” consumen más memoria que el resto de interposiciones

Otro problema del “callback” es su alto consumo de memoria. Al igual que los reenlaces y en las redefiniciones gran parte del consumo de memoria se centra en salvar la información necesaria para poder desinstalarse posteriormente. Esto en un “callback” tiene una gran repercusión, ya que se instala sobre todas las funciones del DSO, y en caso de que estas sean muchas, representa guardar una importante tabla de punteros.

Además, por cada función interceptada del DSO, pDI-Tools debe crear un “stub” (pequeño bloque de código generado dinámicamente) que identifique la función y pase el control al “callback handler”.

Todo junto representa un importante consumo de memoria a pesar de que se ha intentado reducir al máximo. Por ejemplo, pongamos que tenemos una librería para Linux™/INTEL® 386 con 1000 funciones e instalamos sobre ella un “callback”. Se deberán salvar 1000 entradas del GOT (punteros) y generar 1000 “stubs”. Teniendo en cuenta que cada “stub” ocupa 20 bytes y cada puntero 4 bytes se consumirán unos 24 kilobytes de memoria, que es relativamente poco, pero notable si se compara con un reenlace que con suerte consume unos 30 bytes de memoria.

### No hay modo de excluir determinados objetos de una redefinición

Igualmente las redefiniciones actúan sobre todos los objetos instrumentables sin excepción. Es decir, no hay manera de hacer que uno o más objetos no sean afectados por una redefinición.

Evitar esto, al contrario que en el caso anterior, es algo complejo de resolver ya que requiere que pDI-Tools monitorice más intensamente la aplicación.

Instalar la redefinición y los reenlaces iniciales adecuadamente no es ningún problema. El problema surge cuando se carga, en tiempo de ejecución, un objeto que en teoría queremos excluir: el “Runtime Linker” al cargarlo intentará resolver las referencias aplicando, en contra de nuestra voluntad, la redefiniciones. Para evitar esto tendríamos que monitorizar todos los mecanismos de carga de objetos, y dependiendo de lo que estos hagan aplicar los reenlaces adecuados a mano.

Debido a que acotar el radio de acción de las redefiniciones sería muy costoso, tanto en complejidad como en lentitud, esta es una limitación que seguramente nunca se resolverá. Además no es habitual querer acotar una redefinición, así que la compensación obtenida no justificaría el esfuerzo.

## Comparación de punteros de funciones

En C es posible (y necesario en algunas ocasiones) obtener el puntero de una función. Según el ABI de muchas arquitecturas un puntero a una función sólo tiene sentido dentro del objeto donde trabajamos. Por ejemplo, supongamos que tenemos un programa formado por dos DSO. El primer DSO recoge un puntero a printf(2) y lo imprime, y el segundo hace exactamente lo mismo. El resultado de la ejecución sobre un SPARC (32 bits) podría ser:

```
MAIN: printf() = 0x8b001243
LIB:  printf() = 0x8f223744
* las funciones NO coinciden *
```

¿Nos encontramos que hay dos funciones printf(2)? Bueno, en realidad no, en realidad sólo hay una. En SPARC el reenlace se implementa mediante “stubs” escritos en tiempo de ejecución. Lo que tenemos en realidad son punteros a dos “stubs” encargados de redirigir el control a la verdadera implementación de printf(2).

Pero esto no es siempre así. En algunas arquitecturas como PowerPC™ en modo 64 bits el resultado hubiese sido dos números idénticos:

```
MAIN: printf() = 0x00fa002b8b001243
LIB:  printf() = 0x00fa002b8b001243
* las funciones COINCIDEN *
```

El motivo está en que esta arquitectura (y algunas más) distinguen el concepto de puntero a función de llamada a función. Es decir, si se consulta la dirección de una función, se devuelve la ubicación del código de la función. En cambio si llamamos a la función, en realidad se transfiere el control a otra parte (generalmente a un “stub” de reenlace). Ahora veamos que pasa si reenlazamos la función printf(2) en el objeto `LIB`. El resultado del programa sería:

```
MAIN: printf() = 0x00fa002b8b001243
LIB:  printf() = 0x0bbb0c786ac766a7
* las funciones NO coinciden *
```

Tenemos dos valores distintos debido a que `LIB` en realidad está usando la versión de un “backend” de printf(2), y no la versión de C Library.

Esto implica que cualquier programa que confiase en comparaciones de funciones pasaría a funcionar de forma distinta cuando lo instrumentásemos. Este problema es también insoluble, aunque sólo afecta a los reenlaces y “callback” (las redefiniciones afectan a todas las referencias). Por suerte y de todas formas no es habitual encontrar programas que realicen comparaciones entre punteros a funciones, además que es algo muy poco portable.

## pDI-Tools no puede finalizar correctamente su ejecución si la aplicación termina abruptamente

Un problema existente, y de muy difícil solución, es que dependiendo de como finalice su ejecución el programa principal pDI-Tools podrá finalizar correctamente o no. pDI-Tools sólo finalizará su ejecución correctamente cuando el programa termine con un exit(3) o debido a que termina el código de `main`.

En el caso de que el programa termine abruptamente (mediante un abort(3) o por generar una excepción grave), pDI-Tools no podrá ejecutar su código de finalización y por lo tanto la instrumentación quedará truncada y sin finalizar.

Esto implica que no se cierren correctamente los ficheros abiertos y se pierda la información todavía contenida en los "buffer" de escritura, o simplemente, se dejen los ficheros o bases de datos en un estado incoherente.

Debido a que pDI-Tools se ejecuta en modo usuario dentro de la misma aplicación, este problema no se puede evitar fácilmente. Siempre cabe la posibilidad de instalar "signal handlers" para cubrir una cantidad de casos, pero haciendo esto es posible entrar en conflicto con la aplicación instrumentada si esta también instala "signal handlers".

### No puede haber colisión entre interposiciones

La última limitación impuesta es que no se permite que las interposiciones colisionen entre si. Es posible que el usuario piense que lo lógico fuese que si una interposición colisiona con otra, la última en aplicarse fuera la vigente.

No obstante este comportamiento no es aceptable por dos motivos: hay interposiciones de código como las redefiniciones que no se pueden deshacer fácilmente si entran en conflicto con otras interposiciones. Por otro lado esto implicaría unas estructuras de datos más complejas, una gestión más complicada y lenta y un mayor consumo de memoria, tres cosas inaceptables en un programa donde lo que prima es la velocidad.

La solución de DITools a este problema es simplemente no hacer nada. Es responsabilidad del programador que no hayan colisiones, y si las hay, el resultado es impredecible.

## 2.5. Validación

Para demostrar que pDI-Tools se ejecuta correctamente se hace una serie de pruebas que a continuación comentaremos. Estas pruebas garantizan la correcta ejecución de las interposiciones de código, una interpretación adecuada de los ficheros de comandos de interposición, que se gestionan correctamente las estructuras de memoria, etc. Las pruebas se dividen en dos tipos: automáticas y manuales.

Las pruebas automáticas están dedicadas especialmente al código no dependiente de la arquitectura, aunque algunas de ellas comprueban por encima el funcionamiento de las interposiciones. Normalmente éstas ponen a prueba el código de gestión de las estructuras de pDI-Tools.

Las pruebas manuales se realizaban sobre pDI-Tools cada vez que se portaba a una nueva arquitectura. Están orientadas a comprobar que los mecanismos de interposición y gestión del "Elf Backend" funcionan correctamente.

### 2.5.1. Pruebas automáticas

Las pruebas automáticas forman parte de la distribución y utilizan el banco de pruebas de Autoconf y Automake para realizarse. Por ello, si se desean realizarse las pruebas tan sólo hay que llamar a make desde el directorio de la distribución de pDI-Tools con la orden **`make check`**. Esto hará que se ejecuten, una tras otra, las pruebas que describimos a continuación.

Este chequeo no sólo sirve para comprobar la corrección de pDI-Tools cuando está en desarrollo, sino que también es útil para un administrador que va a instalar el "software" y desea comprobar que las librerías se han generado correctamente.

Cada prueba es en realidad un pequeño “shell script” que genera automáticamente ficheros de configuración, de comandos de reenlace junto a pequeños programas en C que, o bien importan parte del código de pDI-Tools, o bien lo usan directamente como un objeto dinámico. Por ello es necesario haber compilado pDI-Tools antes de ejecutar los “tests”. Estos ficheros generados serán usados por el “shell script” para realizar uno o varios chequeos.

Se realizan diversas comprobaciones, pero la mayor parte van dirigidas a comprobar el correcto funcionamiento de la parte no dependiente de la arquitectura de pDI-Tools.

El primer “script” comprueba el sistema de recogida de parámetros de configuración. Para ello crea unos programas en C, los ficheros de configuración necesarios y establece las variables de entorno necesarias. Las pruebas que realiza son las siguientes:

- El primer “test” que realiza comprueba la recogida de parámetros desde distintos ficheros de configuración. En este chequeo se intenta poner en juego todos los elementos de los ficheros de configuración de pDI-Tools: asignaciones, comandos, acciones, etc.
- El segundo “test” prueba a configurar pDI-Tools sólo mediante variables de entorno.
- Por último hace pruebas donde combina variables de entorno con ficheros de configuración para ver si se aplica bien las reasignaciones y las preferencias.

En el siguiente “shell script” se prueba el algoritmo de la resolución de dependencias entre “backends”. En esta ocasión se pone a prueba el algoritmo de resolución de dependencias y los algoritmos de combinación de ficheros de comandos de interposición:

- Primero de todo se prueba un conjunto de ficheros de configuración en los cuales no hay dependencias entre los diferentes “backends”. Esto ha de generar una lista de carga de “backends” por orden de aparición, al igual que sus interposiciones.
- Lo siguiente es cargar en diferentes ordenes ficheros de configuración que generan una dependencia lineal entre sus “backends”. El resultado tiene que ser siempre el mismo a excepción de los comandos de interposición, que debe ser distinto en cada ocasión.
- La prueba que se realiza en esta ocasión es cargar, también en varios ordenes, ficheros de configuración que terminan generando un árbol de dependencias complejo. Al igual que antes se comprueba que el orden de las interposiciones varíe correctamente.
- Por último se prueban varios ficheros de configuración que generan ciclos y que por lo tanto deberían fallar.

Por último se realizan unas pruebas básicas con los reenlaces y redefiniciones, y en caso de estar disponibles en esta arquitectura, las interposiciones mediante “callback”. Debido a que sería muy complejo hacer un “software” que chequease los casos especiales, según la arquitectura, estas pruebas se realizan manualmente tal como se explica en la siguiente sección.

### 2.5.2. Testeo y comprobación manual

Cada vez que se acababa un “porting” de pDI-Tools a una nueva plataforma se sometían los mecanismos de interposición de código a una serie de pruebas que dan una cierta garantía de que la implementación es correcta y da exactamente el resultado esperado.

Para realizar estos chequeos se escribieron algunas librerías dinámicas, dos “backends” y un pequeño programa de “test”. También se entrega un fichero de comandos para cada arquitectura, pero al no ser una prueba automática, debemos editarlo a mano para probar las diferentes interposiciones.

Este código se entrega con la distribución de pDI-Tools, pero no se compila en la configuración por defecto. Si se desea trastear con él debe indicar el parámetro `--enable-debug` al llamar a `configure`. El código fuente reside en el directorio `PREFIX/src/link`, junto al código fuente de `libpdi.so`. Se eligió esta ubicación por comodidad: es el directorio donde se genera el binario y permite ejecutar en un mismo comando la reconstrucción de pDI-Tools y el programa de “test”.

El programa de “test” simplemente ejecuta unas cuantas funciones sencillas como printf(2), fputc(3), puts(3), etc. y termina. Además requiere las librerías `libtest.so` y `libdyn.so`. La primera es una dependencia de la aplicación, y que por tanto carga automáticamente el “Runtime Linker”, que contiene una función que imprime cadenas con las funciones printf(2) y fputc(3). La segunda se carga dinámicamente por el programa principal y contiene una función muy parecida a la de la otra librería.

Se entrega otro programa, llamada `testth`, pensado para testear los “callback” en una aplicación paralela.

Los “backends” son `testbe.so` y `testcb.so`. El primero es un “backend” muy completo con “wrappers” y un gestor de “callback”. El segundo es un caso sencillo de “backend” sólo orientado a interposiciones mediante “callback”.

Por último se entrega un pequeño `runtime`, `runtime.so`, de ejemplo que instala un “thread id resolver” para las librerías pthread. Es esencial para que el programa `testth` funcione correctamente.

Seguidamente describimos que pruebas se han realizado con cada interposición.

#### 2.5.2.1. Reenlaces

Para probar los reenlaces se comienza por probar a interceptar la función fputc(3) en el ejecutable (`MAIN`). Este es reenlace más sencillo ya que la función sólo recibe dos parámetros y devuelve un valor fácilmente predecible en todas las arquitecturas: el carácter que imprime. De esta forma se prueba el envío de parámetros y la entrega del valor de retorno.

La siguiente prueba consiste en interceptar la misma función, pero esta vez en los objetos compartidos (`libtest.so` y `libdyn.so`). Se realiza esta prueba ya que los ejecutables se enlazan siempre estáticamente y se instalan siempre en una determinada dirección base de memoria. En cambio los objetos compartidos trabajan con punteros relativos y ello implica, en algunas plataformas, que se tenga que hacer un pequeño tratamiento previo antes de instalar la interposición.

De nuevo trabajando sobre el objeto `libtest.so` probamos a interceptar una llamada desde él hacía el ejecutable `MAIN`. Con esto comprobamos que se realiza también correctamente el caso inverso al anterior. En algunas plataformas como PowerPC™ 64 estas dos últimas pruebas son importantes.

Antes de acabar se intercepta en cada objeto en memoria llamadas a printf(2). Debido a la existencia de funciones como esta, que pueden recibir un número de parámetros variable, se ha de comprobar que los “wrapper” puedan usar la interfaz `stdarg.h` de C. Con esta prueba se garantiza que se deja la pila en un estado coherente, sin afectar para nada al paso de parámetros y al valor de retorno de la función.

Para finalizar se prueban reenlaces según las limitaciones o características de la plataforma. Por ejemplo en PowerPC™ se probaron reenlaces a funciones cercanas y lejanas (ver más sobre esto en la sección *PowerPC sobre GNU/Linux* del capítulo *Portabilidad e implementación en las diferentes plataformas*.

#### 2.5.2.2. Redefiniciones

Las redefiniciones, al reutilizar código usado por el mecanismo de reenlace, se suelen probar una vez que los reenlaces funcionan correctamente.

Para probar las redefiniciones el programa redefine la función fputc(3) de la librería de C. Esto afecta a todas las llamadas realizadas a fputc(3) desde cualquier DSO de la aplicación. Comprobamos que esto es así, pero nos centramos especialmente en la llamada a fputc(3) desde la librería `libdyn.so`. Esta librería se carga dinámicamente y por lo tanto cuando acceda a fputc(3) el "Runtime Linker" tendrá que resolver la función, en tiempo de ejecución, con las estructuras de datos alteradas por pDI-Tools (en concreto la información de símbolo de fputc(3) en C Library).

Se hace también una prueba de redefinición con una librería que no sea C Library ya que este objeto, en la mayoría de plataformas, es especial y toma como valor base la dirección 0x00000000. Por ello redefinimos la función `lib_function` del objeto `libtest.so` de un modo incompleto: forzamos (comentando código) a pDI-Tools para que no realice un reenlace masivo de esta función en todos los objetos. De este modo comprobamos si el "Runtime Linker" resuelve correctamente símbolos de otros objetos.

#### 2.5.2.3. Interposiciones mediante "callback"

Este mecanismo también requiere que funcionen bien los reenlaces, ya que se basa totalmente en este mecanismo para realizar su cometido.

Estas pruebas comienzan por instalar un "backend" sencillo (`testcb.so`) sobre el ejecutable. Este "backend" simplemente intercepta un pequeño conjunto de funciones. Cada vez que intercepta una función escribe un mensaje antes de ejecución, con su nombre, y un mensaje al finalizar la función. De esta forma comprobamos que lo más básico del mecanismo de "callback", que es la invocación de las funciones de control y la función interceptada, funciona correctamente.

En la siguiente prueba se usaba el "backend" `testbe.so`. Este "backend" es aún más complejo que el anterior y no sólo intercepta las llamadas a las funciones, sino que opera con sus parámetros y comprueba su valor de retorno. Con este "test" comprobamos que el "generic wrapper" deja la pila en un estado correcto que permita usar la interfaz `stdarg.h` para recoger una cantidad variable de parámetros.

Por último se ejecutó el programa `testth`, que ejecuta muchos "threads" en paralelo que escriben un mensaje por pantalla y al cabo de un tiempo aleatorio mueren. Para ejecutar esta prueba se implementa un "thread id resolver" en el "backend" `runtime.so`. Con este chequeo comprobamos la robustez y buen funcionamiento del mecanismo de "callback".

Todas estas pruebas sólo se han realizado sobre pDI-Tools para Linux™/INTEL® 386 ya que es la única implementación que presenta este mecanismo.

# Notas

1. Es posible con algunos "hacks" basados en declarar punteros de funciones en las estructuras, el uso de `union` y algunas macros

# Capítulo 3. Portabilidad e implementación en las diferentes plataformas

En este capítulo discutimos las decisiones tomadas y mecanismos usados para obtener un código altamente portable.

Se comienza por exponer las diferencias y similitudes entre distintas plataformas, tanto por sistema operativo como por arquitectura. Se exponen las plataformas soportadas actualmente: GNU/Linux™ sobre INTEL® 386, PowerPC™ y PowerPC™ 64; Solaris sobre SPARC; Irix sobre MIPS®.

De exponer estas plataformas, escribimos una sección dedicada a ver cuales son las diferencias y similitudes entre ellas. Explicaremos, muy por encima, las funcionalidades básicas que debe ofrecer la parte dependiente de la arquitectura para que pDI-Tools pueda desarrollar sus funciones. Esta capa recibirá el nombre de “Elf Backend” y ofrecerá un API a pDI-Tools que le permitirá desempeñar sus funciones sin preocuparse de que tipo de arquitectura o sistema tiene debajo.

En la siguiente sección se explican las decisiones tomadas para hacer el código portable. Buscamos portabilidad tanto a nivel de código como de herramientas de programación. Un programa fácilmente portable entre diferentes sistemas se consigue creando capas de abstracción, organizando bien el código, procurando no usar muchas estructuras e interfaces específicas de la arquitectura. Pero también se tiene que escribir el programa de tal forma que sea fácilmente compilable con otras herramientas.

De hecho el primer problema que se afronta al portar un programa es la disponibilidad de las herramientas con que se creo. Por ejemplo, a la hora de portar un programa desde Mac OS™ X a Solaris el primer problema es la inexistencia de las herramientas y librerías necesarias para SPARC. Esto significa un trabajo extra para crear el entorno de trabajo, los “scripts” de construcción del programa y encontrar herramientas equivalentes a las usadas en el sistema operativo de Apple Computer, Inc..

En la sección *Políticas seguidas para obtener un código portable* se discute que decisiones se han tomado para llegar a obtener un código portable tanto a nivel de compilador como de arquitectura.

En la sección *Organización del código según la plataforma* se discute desde como se organiza el código hasta que funcionalidades se implementan.

Finalmente, en las sección *Herramientas* explicamos que herramientas se usaron para desarrollar pDI-Tools, y en especial, en que contribuyen a que sea más portable.

## 3.1. El formato ELF en diferentes sistemas

pDI-Tools no trabaja directamente con el “hardware”, aunque si es dependiente de tres componentes: el formato y suplemento ELF específicos de la arquitectura, la CPU para los fragmentos en ensamblador de pDI-Tools y como organiza la información del “Runtime Linker” el sistema operativo.

Por ello, en las siguientes secciones describiremos las principales características y diferencias de las arquitecturas sobre las que funciona pDI-Tools.

Veremos que hay básicamente tres tipos de mecanismos de enlace dinámico (todos ellos se tratarán con más detenimiento en las siguientes secciones):

- Enlaces basados en una tabla GOT (“Global Offset Table”). Esta tabla contiene las direcciones de los símbolos externos usados por el DSO. Este es el caso de MIPS®.
- Enlaces basados en una tabla PLT (“Procedure Linkage Table”). Esta tabla contiene “stubs” que ejecutan en enlace. Este es el caso de SPARC y PowerPC™ 32.

- Por último existen combinaciones de ambos, como el caso de INTEL® 386.

Se mostrará que el formato ELF, a pesar de establecer con más o menos exactitud que estructuras deben usarse para organizar la información de un objeto, no explica como implementar el enlace dinámico en cada arquitectura. Además deja cabos en el aire que deben resolverse según las necesidades del fabricante. Esto es debido a que la especificación ELF no indica ninguna interfaz genérica para los "Runtime Linker" ni se ha llegado a ningún tipo de acuerdo entre fabricantes para producirla.

Con estas explicaciones reuniremos la suficiente información para poder entender, en la siguiente sección, como se estructuró la capa dependiente de la arquitectura de pDI-Tools.

### 3.1.1. GNU/Linux™ sobre INTEL® 386

Linux™ es un "kernel" libre de licencia GPL, creado por Linus Torvalds y una amplia comunidad de usuarios, que ha evolucionado mucho a lo largo de su vida. Linux™ fue inicialmente escrito exclusivamente para la plataforma INTEL® 386. Durante sus primeros años de vida básicamente creció hasta casi ofrecer las mismas funcionalidades que un "kernel" comercial de la época. Fue en la rama 2.2.x cuando se hizo un gran esfuerzo para crear un "kernel" fácilmente portable capaz de competir con otros sistemas estilo Unix™. Con esta versión aparecieron implementaciones de Linux™ para múltiples arquitecturas. Esto provocó que se empezasen a depurar mucho las interfaces del sistema operativo, cumpliendo cada día más los estándares. La versión 2.4.x se considera un "kernel" muy depurado, eficiente y apto para entornos de trabajo intenso, además de funcionar en un gran número de arquitecturas. Y ahora en la versión 2.6.x se está empezando a usar bastante en grandes computadoras y cálculo intensivo.

Linux™ además está íntimamente ligado a la librería de C GNU C Library (por ello suele recibir el nombre de GNU/Linux™). Ésta, al igual que el "kernel", ha cambiado mucho a lo largo de su historia, soportando cada día mejor los estándares y mejorando su calidad y portabilidad.

El resultado, especialmente por los orígenes INTEL® 386 de Linux™, es una de las implementaciones de ELF más cuidadas y respetuosas con la especificación [Sys5ABI].

Sólo ha sido ahora con la versión 2.6.x del "kernel" y la versión 2.3.x de GNU C Library cuando se han introducido algunos cambios que rompen un poco la especificación ELF. No obstante los cambios no son muy bruscos y se resuelven con unas pocas condiciones.

El "Runtime Linker" de GNU/Linux™ ofrece un protocolo de depuración que permite acceder a la lista de objetos que él maneja. Esta lista es accesible desde la estructura global `_r_debug`. En esta estructura encontramos una lista encadenada de objetos, donde el primero es el binario de la aplicación y el resto las librerías y objetos dinámicos que ésta usa. Esta estructura y lista no forman parte del estándar y es el mecanismo que `ld-linux.so` (el "Runtime Linker") pone a nuestra diposición para acceder a la información sobre los DSO en memoria.

Cada elemento de la lista es una estructura `_r_linkmap` que nos permite llegar a la información ELF de un DSO. Examinando las estructuras ELF la primera sorpresa que tenemos es que en la información del ejecutable no tenemos el nombre de la aplicación. Por este motivo en GNU/Linux™ sobre INTEL® 386 pDI-Tools siempre se refiere al programa principal con el alias `MAIN`.

El suplemento del ABI para INTEL® 386 [Sys5i386] indica que en esta arquitectura siempre se ha de usar estructuras de reubicación con un "addend" implícito, o lo que es lo mismo, estructuras ElfW(Rel).

Esto implica que encontraremos seguro las secciones `DT_REL`, `DT_RELSZ` y `DT_RELENT`.

No obstante no las usaremos ya que GNU/Linux™ separa todas las reubicaciones de función en una tabla a parte: `DT_JMPREL`. Esta tabla está ordenada ascendentemente por el índice de símbolo

de cada reubicación. Esto es muy práctico ya que nos permite hacer las búsquedas en esta tabla dicotómicamente.

Todas las entradas en la sección `DT_JMPREL` son del tipo `R_386_JMP_SLOT`. Este tipo de reubicación apunta a una entrada del GOT de la sección `DT_PLTGOT` del objeto. Esta entrada contiene la dirección real del objeto en memoria, o en su defecto un puntero a un "stub" encargado de llamar al "Runtime Linker" para que la resuelva.

Esta tabla se usa por unos "stubs" de reenlace generados en tiempo de compilación. Un "stub" se divide en dos partes: una que trata de transferir el control a la verdadera definición de la función, y otra encargada de transferir el control al "Runtime Linker" si la parte anterior no tuvo éxito.

Estos "stubs" están también contenidos en la sección `DT_PLTGOT`. Esta parte de la sección recibe el nombre de "Procedure Linkage Table". El aspecto del PLT es el siguiente:

```
.PLT0:  pushl   got_plus_4
        jmp     *got_plus_8
        nop ; nop
        nop ; nop
.PLT1:  jmp     *name1_in_got
        pushl   $offset
        jmp     .PLT0
.PLT2:  jmp     *name2_in_got
        pushl   $offset
        jmp     .PLT0
        ...
```

De este código podemos deducir que las tres primeras entradas del GOT están reservadas y su contenido es especial. La primera entrada, que no se usa en este código, contiene la dirección de la estructura dinámica `_DYNAMIC`. La segunda entrada sirve para dar información de identificación al "Runtime Linker", mientras que la tercera es un puntero a una función del "Runtime Linker" encargada de realizar las resoluciones en tiempo de ejecución.

Cuando el programa arranca la tabla GOT está inicializada de tal forma que la entrada `name1_in_got` contiene la dirección de la siguiente instrucción a `.PLT1`, la entrada `name2_in_got` contiene la dirección de la siguiente instrucción a `.PLT2`, y así sucesivamente.

Esto hace que la primera vez que usamos una función, el "stub" `.PLT`*n* provoque que se termine saltando a `.PLT0` y de este al "Runtime Linker". El "Runtime Linker" alterará la entrada *n* del GOT con la verdadera dirección de la función para saltar directamente a ella en próximas llamadas.

El PLT se presenta en INTEL® 386 de dos formas: con direcciones estáticas para objetos no reubicables, como es el caso del binario ejecutable, o con "offsets" relativos, que es el caso de la mayor parte de las librerías compartidas.

Una última diferencia respecto al estándar, pero que no nos afecta muy seriamente, es el "Runtime Linker". La especificación recomienda que sea la misma librería de C el "Runtime Linker". Sin embargo, GNU/Linux™ usa el objeto compartido `ld-linux.so` como "Runtime Linker".

Realmente no es técnicamente más complejo darle el rol de "Runtime Linker" a GNU C Library. Los motivos que han llevado a separar esta funcionalidad a parte no están claros, aunque si es útil esta separación en algunos casos especiales. En concreto se nos ocurren dos posibles motivos:

- Supongamos que escribimos una aplicación íntegramente en ensamblador y totalmente autosuficiente (no recurre a otras librerías, funciona exclusivamente mediante llamadas al sistema). Si el "Runtime Linker" está integrado en la librería de C obligamos a estas aplicaciones supercompactas a cargar esta macro librería, aumentando su consumo de memoria y tiempo de carga.

- En el caso de los binarios enlazados estáticamente no es necesario usar la librería de C del sistema (ver [Teensy]). De hecho aplicaciones como versiones antiguas de Acrobat Reader vienen enlazadas estáticamente para evitar usar las librerías de sistema y así funcionar sobre casi cualquier distribución de GNU/Linux™ sin ser necesaria una recompilación. Estos binarios de por sí suelen ser muy grandes y consumen mucha memoria al incluir toda la librería de C, y además pueden dejar de funcionar correctamente si las llamadas al sistema operativo cambian.

## 3.1.2. Solaris sobre SPARC

El sistema operativo Solaris sobre SPARC tiene una implementación muy correcta del estándar ELF al igual que GNU/Linux™ sobre INTEL® 386.

Solaris es un sistema operativo desarrollado por Sun Microsystems™ a partir de la rama BSD, pero con muchas influencias de “System V” (de hecho ELF es parte de “System V”). Al contrario que GNU/Linux™, Solaris es un sistema operativo Unix™ y no un clon.

La arquitectura SPARC es el buque insignia de Solaris. Sun Microsystems™ ha usado mucho y aún usa este tipo de CPU RISC. Se caracteriza por ser una arquitectura eficiente, clara, con un código ensamblador elegante y fácil de mantener. De hecho gracias a esto las SPARC modernas ofrecen total compatibilidad hacia atrás con CPU’s más antiguas.

La arquitectura SPARC prácticamente solo ha sido usada por Sun Microsystems™ para sus estaciones de trabajo y servidores. Debido a esto el suplemento para SPARC de la ABI ha sido escrito casi por entero por Sun Microsystems™, a la vez que la implementación en Solaris. El resultado es una implementación bastante rigurosa de la misma.

A pesar de ello, en Solaris hemos encontrado algunos pequeños detalles que incumplen la especificación ELF. No parecen casuales ya que tienen la pinta de ser pequeñas optimizaciones. Por ejemplo, al igual que en GNU/Linux™, encontramos que el “Runtime Linker” es una librería a parte llamada `ld.so.1`.

Para poder acceder a los datos de los objetos en memoria el “Runtime Linker” ofrece un protocolo de depuración basado en una estructura muy similar a `_r_debug` de GNU/Linux™. Esta estructura recibe el nombre de `r_debug`.

No obstante esta estructura no está exportada globalmente y se ha de buscar en la sección dinámica `DT_DEBUG` de algún objeto. Esto nos lleva a una recurrencia: necesitamos consultar la sección `DT_DEBUG` pero no tenemos como llegar a las estructuras ELF de un objeto.

La especificación ELF, previsora, obliga a exportar la variable global `_DYNAMIC` a todos los objetos para resolver estas situaciones. Esta variable global es un puntero a la lista de secciones dinámicas del ejecutable. En esta lista precisamente encontraremos la sección `DT_DEBUG` que nos indicará la posición de la estructura `r_debug` que buscamos.

Esta estructura, al igual que su homóloga en GNU/Linux™, contiene un puntero a una lista encadenada de objetos. Dentro de cada elemento de la lista encontraremos un puntero a la lista de estructuras dinámicas.

También encontramos que en algunos objetos faltan secciones como `DT_RELA`. Esto sucede en objetos especiales que no exportan ni usan símbolos externos. Estos objetos se encuentran en Solaris y en las nuevas versiones de GNU/Linux™, y se desconoce su papel en el sistema, aunque tampoco afectan a nuestros propósitos. Estos objetos son marcados por pDI-Tools como Broken Objects (objetos rotos) para no ser manipulados.

Otra característica de esta arquitectura es que usa reubicaciones relativas, o lo que es lo mismo, del tipo ElfW(Rela). Estas estructuras se diferencian de ElfW(Rel) en que tienen un “addend” explícito. Este valor debe usarse con el puntero contenido en la estructura según el tipo de reubicación que sea.

Al usar estructuras ElfW(Rela) se deben usar las secciones `DT_RELA`, `DT_RELASZ` y `DT_RELAENT`. Estas, al igual que en INTEL® 386, contienen punteros al interior de la sección `DT_PLTGOT`. En concreto nos interesan las estructuras del tipo `R_SPARC_JMP_SLOT`, que son las encargadas de reubicar las funciones externas. No obstante esta vez los punteros de las estructuras ElfW(Rela) no apuntan a otros punteros, sino directamente a código.

La especificación de la arquitectura SPARC no usa en ningún momento tablas GOT, sino que reescribe en tiempo de ejecución el código de la tabla PLT.

Cada entrada del PLT es un "stub" encargado de realizar un salto a la verdadera definición de la función, o al "Runtime Linker" en caso de no conocerse todavía. Un "stub", debido a las limitaciones que impone el juego de instrucciones, se compone de unas cuantas instrucciones aritméticas que generan la dirección donde saltar y una instrucción de salto sin retorno.

Seguidamente copiamos un código de ejemplo de la tabla PLT ya inicializado por el "Runtime Linker":

```
.PLT0:   save    %sp, -64, %sp
         call    runtime_linker
         nop
.PLT1:   .word   obj_identification
         unimp
         unimp
         ...

.PLT101: sethi   %hi(name1), %g1
         jmpl    %g1+%lo(name1), %g0
         nop

.PLT102: sethi   (.-.PLT0), %g1
         ba,a    .PLT0
         nop

.PLT103: sethi   %hi(name3), %g1
         jmpl    %g1+%lo(name3), %g0
         nop

.PLTN:   sethi   %hi(nameN), %g1
         jmpl    %g1+%lo(nameN), %g0
         nop
```

El código en `.PLT0` es el encargado de resolver un símbolo en tiempo de ejecución. Inicialmente los "stubs" tienen el aspecto del "stub" `.PLT102`. Este transfiere el control a `.PLT0` junto a su posición en `%g1`. `.PLT0` a su vez transferirá el control al "Runtime Linker". Éste se encargará de resolver la función, alterar el código en la posición `.PLT102` y transferir el control a dicha función.

A partir de entonces los "stubs" toman el aspecto de los ejemplos `.PLT101` y `.PLT103`. Estos "stubs" simplemente realizan un salto a la verdadera definición de la función.

En el caso de SPARC 64 bits, los "stubs" son distintos, además de más grandes. No obstante es prácticamente el mismo mecanismo.

### 3.1.3. Irix sobre MIPS®

Irix es el sistema operativo de SGI®. Está presente en muchas plataformas, entre ellas Itanium®. No obstante la más extendida y usada es para MIPS®. MIPS® es una CPU RISC de alto rendimiento, para 32 y 64 bits.

Este sistema operativo tiene una larga historia y soporta bastantes ABI. Actualmente sólo se usa, o se recomienda usar, el ABI de ELF en sus formas `N32` (32 bits ELF) o `64` (64 bits ELF).

Por compatibilidad hacia atrás el "Runtime Linker", al contrario que SPARC o INTEL® 386, ofrece varios métodos para acceder a los objetos en memoria.

El que nos interesa es el acceso mediante la información de depuración en la estructura ElfW(Obj_Info). Esta estructura, a pesar de tener un nombre del estilo de los tipos de ELF, no forma parte del estándar ELF y es totalmente exclusiva de Irix. Esta estructura ofrece mucha información de depuración junto una lista encadenada con información sobre los objetos en memoria.

Y es en los elementos de esa lista donde encontramos una primera particularidad de este sistema operativo: los campos *`oi_ehdr`* y *`oi_orig_ehdr`*. Estos campos indican la dirección donde espera cargarse el objeto compartido, y la dirección donde realmente ha sido ubicado. La diferencia entre ambos campos se llama *desplazamiento delta* y se usa para recalcular los "offsets" indicados en *algunas* estructuras ELF.

De hecho el desplazamiento delta sólo se debe usar en las estructuras ELF que se han leído directamente de disco sin alterarse, como la tabla de símbolos o las tablas de reubicación. En otras estructuras generadas por el "Runtime Linker" como sería la tabla GOT se usan punteros absolutos.

Estos valores no se reajustan debido a que los ficheros binarios se mapean en memoria. De esta forma el programa no se copia entero a memoria sino que se consulta de disco de una forma transparente y sin consumir apenas memoria. Esto permite ahorrar bastante memoria, y además ya que, si están funcionando varias copias, todas comparten páginas cacheadas y por lo tanto se gana también rendimiento. No obstante es posible escribir en dichas páginas ya que por defecto están marcadas como páginas COW ("Copy-On-Write"), o lo que es lo mismo, cuando se modifican se reserva una página de memoria virtual anónima donde copiar el contenido y la modificación.

Por ello se debe detectar y tener en cuenta en que estructuras debemos usar el desplazamiento delta o no para interpretar sus valores.

La arquitectura MIPS® usa reubicaciones con "addends" implícitos, o lo que es lo mismo la estructura ElfW(Rel) y las secciones dinámicas `DT_REL`, `DT_RELSZ` y `DT_RELENT`.

Otra gran diferencia es la ausencia de una sección `DT_JMPREL` que acelere las búsquedas de reenlaces, pero como comentamos seguidamente, no es necesaria por la disposición que tienen las reubicaciones en la sección `DT_REL`.

De hecho en esta arquitectura pDI-Tools nunca llega a usar la tabla `DT_REL`. El suplemento ELF de MIPS® introduce muchos detalles sobre como debe organizarse la tabla de símbolos y la tabla GOT. Además añade nuevas secciones dinámicas con información para poder explotar dicha organización. Las estructuras dinámicas nuevas más importantes:

- **`DT_MIPS_RLD_VERSION`.** Este elemento contiene la versión de la API del "Runtime Linker". Puede ser necesaria en el futuro para distinguir versiones de Irix que presenten alguna incompatibilidad con la versión actual.
- **`DT_MIPS_LOCAL_GOTNO`.** Este elemento nos indica cuantas entradas del GOT pertenecen a símbolos locales. Es importante ya que nos permite dividir el GOT en dos partes claras, de las cuales sólo una debemos manipular.

- **`DT_MIPS_GOTSYM`.** Por último este elemento nos da el índice del primer símbolo de la tabla de símbolos que tiene una entrada en la "Global Offset Table".

Las sección `DT_PLTGOT` se divide en dos partes lógicas: *local* y *externa*. Las entradas locales residen en la primera parte de la tabla. Estas entradas sólo requieren ser reubicadas si ocurren en un objeto compartido y la dirección donde se ha cargado dicho objeto no es donde esperaba ser cargado.

La sección de entradas externas reside en la segunda parte de la tabla GOT. La estructura dinámica `DT_MIPS_LOCAL_GOTNO` indica cuantos elementos contiene la tabla GOT local, y por lo tanto, donde comienza la tabla GOT externa.

Cada entrada en la parte externa del GOT se corresponde directamente con una entrada de una parte especial de la tabla de símbolos. Esta parte de la tabla de símbolos la llamamos `.dynsym`. La tabla de símbolos, de forma similar al GOT, se divide en dos partes: símbolos que no tienen una entrada en el GOT y símbolos que sí. Con ayuda de la información que nos da `DT_MIPS_GOTSYM` podemos conocer cual es la primera entrada de la tabla de símbolos que forma parte de `.dynsym`.

Toda esta información nos lleva a una conclusión: podemos conocer la posición en el GOT de una función tan sólo sabiendo la información de símbolo sin ser necesario consultar las estructuras `DT_REL`. Además el acceso será prácticamente directo debido al orden que hay entre `DT_SYMTAB` y `DT_PLTGOT`.

Por ejemplo, supongamos que queremos modificar la entrada del GOT del símbolo externo con índice *`symndx`*. Aplicando la siguiente operación aritmética obtenemos el índice de la entrada del GOT que debemos modificar:

*`gotndx`* = val(`DT_MIPS_LOCAL_GOTNO`) + *`symndx`* - val(`DT_MIPS_GOTSYM`)

Esto sin duda es mucho más rápido que una búsqueda en la tabla de reubicación, aunque ésta sea dicotómica.

Además el "Runtime Linker", que en Irix es una aplicación llamada rld(5), ofrece una completa API para manipular los objetos y símbolos en memoria. No obstante en pDI-Tools no se ha aprovechado esta API ya que parece muy dependiente de la versión de Irix y puede ser que no este presente en próximas versiones del sistema operativo. Manipulándolos con las funciones dlopen(3), dlsym(3) y dlclose(3) garantizamos que esta parte del código sea altamente portable sin perder apenas eficiencia (se supone que la implementación de estas funciones debe usar la API descrita).

Particularidades como esta hicieron tomar la decisión de separar el código para cada sistema operativo y arquitectura. No aprovechar algunas ayudas que brinda la plataforma Irix sobre MIPS® sería un error ya que obtendríamos soluciones menos eficientes.

Con esto se han explicado las diferencias más importantes de esta plataforma. Como se ha podido ver Irix sobre MIPS® es una plataforma con una implementación de ELF muy peculiar, tanto por sus optimizaciones como por las funcionalidades ofrecidas por el "Runtime Linker".

## 3.1.4. PowerPC™ sobre GNU/Linux™

Esta explicación la dividimos en dos partes, la primera para PowerPC™ y la segunda para PowerPC™ 64 bits, debido no sólo a que son dos ABI completamente distintos, sino que además el segundo no forma parte del estándar ELF aunque sigue sus indicaciones.

### 3.1.4.1. PowerPC™

PowerPC™ es una de las principales plataformas sobre las que corre GNU/Linux™. Tiene una larga historia junto a una gran comunidad encargada de soportarla y mejorarla.

Debido a la buena portabilidad del sistema GNU/Linux™, las diferencias entre el "Runtime Linker" de INTEL® 386 y PowerPC™ no son muchas, y casi todas las operaciones y funcionalidades se mantienen intactas de una implementación a la otra. Las estructuras de depuración `_r_debug` son casi idénticas y no presentan diferencias a la hora de manejarlas.

Encontramos también algunas discrepancias comunes respecto al estándar en ambas versiones (INTEL® 386 y PowerPC™). Por ejemplo, el objeto compartido `ld-linux.so` en ambas versiones asume el rol de "Runtime Linker", en lugar de la librería de C tal como aconseja el estándar.

Respecto a la implementación del ABI ELF para PowerPC™ podemos decir que está muy cuidada y es respetuosa con la especificación, a excepción de los aspectos que comentaremos seguidamente.

En esta arquitectura también nos encontramos con Broken Objects (ver GNU/Linux™/INTEL® 386 o Solaris). Estos objetos tampoco son tratados en pDI-Tools y pasan desapercibidos durante la instrumentación.

El suplemento ELF de esta arquitectura es parecido a SPARC, pero con características de INTEL® 386. Se basa, al igual que INTEL® 386 en tener una a la vez una tabla PLT y una tabla GOT, pero se modifica el código en tiempo de ejecución al igual que SPARC. No obstante GNU/Linux™ no se mantiene fiel a la especificación ELF en este aspecto, e implementa un mecanismo de enlace algo distinto, aunque coherente con la especificación.

La implementación de los enlaces es casi idéntica a la implementación de SPARC en 32 bits. Sólo hay dos diferencias significativas: el nuevo juego de instrucciones de PowerPC™ y que las estructuras de reubicación no usan un "addend" explicito, es decir, se usan estructuras ElfW(Rel).

También encontramos que la sección `DT_JMPREL` también está presente, al igual que en GNU/Linux™ para INTEL® 386. En esta sección encontramos solamente estructuras del tipo `R_PPC_JMP_SLOT`. Estas estructuras son las encargadas de reubicar las funciones que usa o implementa un objeto.

Al igual que en SPARC, no usa en ningún momento tablas GOT, sino que reescribe en tiempo de ejecución el código de la tabla PLT.

Cada entrada del PLT es un "stub" encargado de realizar un salto a la verdadera definición de la función, o al "Runtime Linker" en caso de no conocerse todavía. Seguidamente mostramos el aspecto que puede tener el PLT de `MAIN` sólo iniciarse el programa en GNU/Linux™ sobre PowerPC™:

```
.PLTcall:
        addis   r11, r11, .PLTtable@ha
        lwz     r11, .PLTtable@l(r11)
        mtctr   r11
        bctr
.PLTresolve:
        addis   r12, r0, dynamic_linker@ha
        addi    r12, r12, dynamic_linker@l
        mtctr   r12
        addis   r12, r0, symtab_addr@ha
        addi    r12, r12, symtab_addr@l
        bctr
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
.PLT1:
```

```
        addi    r11, r0, 4*0
        b       .PLTresolve
.PLT2:
        addi    r11, r0, 4*1
        b       .PLTresolve
.PLT3:
        addi    r11, r0, 4*2
        b       .PLTresolve

        ...

.PLTN:
        addi    r11, r0, 4*(N-1)
        b       .PLTresolve
```

Cuando llamamos a una función, el control se transfiere a un "stub" como `.PLT1`, `.PLT2`... Estos "stubs" escriben su índice en el registro `r11` y transfieren el control a `.PLTresolve`. Esta rutina llama a "Runtime Linker", pasándole el índice de la función en la tabla de símbolos y un puntero a dicha tabla. Con esta información el "Runtime Linker" resolverá la dirección del símbolo, y cambiará consecuentemente el contenido de la tabla PLT y GOT. Acto seguido transfiere el control a la función resuelta.

Pero GNU/Linux™ en esta ocasión no se mantiene fiel a la especificación y e introduce aquí una optimización copiada de la plataforma SPARC. La rutina `.PLTcall` está implementada en esta plataforma al principio de la sección `DT_PLTGOT`, al contrarió del estándar, que indica que tendría que ser la segunda rutina. Además, si se puede, el código que se copia es distinto del expuesto: transfiere el control directamente a la función sin usar la tabla GOT. Seguidamente se explica cuando se usa este código o el otro.

Si la diferencia entre el punto de entrada de la función desde la posición y `.PLT`*`i`* está en un rango de 0x03ffffff bytes se realiza un salto relativo a la función, y por lo tanto se escribe esta entrada en el PLT:

```
PLTi:
        b       (@target - .PLTi)
```

Si la dirección de la función está demasiado lejos, es decir, no entra en el rango antes descrito, se realiza el enlace mediante una consulta al GOT y un salto a a función `.PLTcall`, tal como indica el estándar:

```
PLTi:
        addi    r11, r0, 4*(i-1)
        b       .PLTcall
```

En próximas llamadas, el "stub" transferirá el control directamente si es un salto cercano, o mediante la rutina `.PLTcall` a la dirección de memoria almacenada en la posición `i-1` de la tabla GOT.

Con esta optimización se consigue acelerar notablemente la velocidad del enlace en muchísimos casos sin romper la compatibilidad con el estándar.

#### 3.1.4.2. PowerPC™ 64

GNU/Linux™ sobre PowerPC™ 64 es un "porting" relativamente reciente. Está dirigido considerablemente por IBM® e implementa soluciones similares a las de IBM AIX™ en el "kernel". Debido a que este formato no se considera parte de ELF no se entrará en mucho detalle sobre él.

IBM AIX™ originalmente usaba un ABI distinto de ELF llamado PowerOpen™. El formato de los ejecutables PowerOpen™ era una variación del formato COFF y adaptado a PowerPC™. Posteriormente se adaptó el "Runtime Linker" de IBM AIX™ para que pudiera usar binarios ELF.

De esto resultó unas indicaciones, que no se consideran parte del estándar ELF, sobre como implementar algo parecido a ELF siguiendo un ABI similar al de PowerOpen™ y por lo tanto un esquema de enlace también similar.

El resultado final es una especificación muy distinta a todas las anteriores, y a la vez muy vaga e inexacta.

Con el desarrollo del sistema operativo GNU/Linux™ se ha hecho un esfuerzo por retomar este formato de enlace e intentar convertirlo en algo parecido a ELF. No obstante la implementación no sigue ni la especificación resultante ni tampoco las antiguas especificaciones ELF de IBM AIX™.

De hecho en esta arquitectura, a pesar de que existe una sección `DT_PLTGOT`, su contenido no es útil. Los enlaces se realizan todos en tiempo de carga mediante la construcción de descriptores de función.

Un descriptor de función es una estructura de esta forma:

```
typedef struct function_descriptor {
        void *addr;
        unsigned long toc;
        unsigned long env;
} f_desc_t;
```

El campo *`addr`* contiene la dirección del punto de entrada de la función. El campo *`toc`* contiene la dirección base del TOC para esta función. El campo *`env`* contiene el puntero de entorno para lenguajes como Pascal o PL/1.

Estos descriptores tendrían que estar contenidos en una tabla llamada TOC contenida en la sección `DT_PLTGOT`.

Cuando un programa quiere llamar a una función, este consulta el descriptor de la función, carga en el registro `r2` (o registro TOC) el valor correcto del TOC para esta función, consulta su dirección en el descriptor y salta a ella.

Aprovechando esto se construyen los reenlaces: para reenlazar una función a otra simplemente tenemos que copiar su descriptor encima del descriptor de la función interpuesta.

Este mecanismo es el único posible ya que la mayoría de estructuras ELF están corruptas en esta arquitectura. Con esto quiero decir que las tablas de símbolos están vacías o faltan símbolos, punteros que no apuntan a ninguna parte e incoherencias.

Respecto al "Runtime Linker", éste tiene el mismo comportamiento que en PowerPC™ 32 bits. Por ello el "driver" de sistema apenas se ve afectado por esta nueva arquitectura.

### 3.1.5. Organización del "Elf Backend"

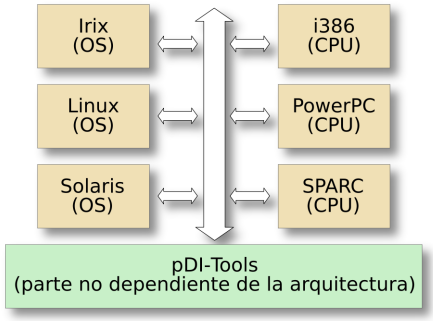
Las descripciones anteriores nos muestran que las diferencias generalmente se suelen dar a nivel del "Runtime Linker". Respecto a las implementaciones de ELF, con mayores o menores innovaciones,

éstas se suelen respetar bastante la especificación ELF: tabla de símbolos, tabla de cadenas, formato de las estructuras, etc.

Rara poder manejar todas estas diferencias se debe decidir como se organizarán los “Elf Backend” para poder obtener una portabilidad óptima sin sacrificar la legibilidad del programa.

La idea más inmediata es dividir los “Elf Backend” en unidades, que llamaremos “drivers”. Es fácil pensar en dos tipos de “drivers”: los dedicados a interaccionar con el “Runtime Linker” y por lo tanto dependientes del sistema operativo, y los dedicados a realizar los enlaces en una cierta arquitectura (dependientes de la CPU).

A priori la organización más sencilla sería una comunicación directa entre todos los “drivers” y la parte independiente de la arquitectura, como se puede ver en el siguiente diagrama:



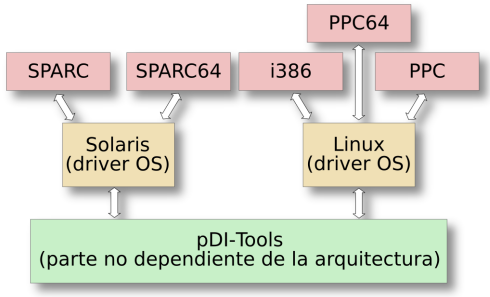

En este diagrama mostramos como todas las comunicaciones serían a un mismo nivel. Es decir, pDI-Tools usaría un “driver” del sistema operativo para interaccionar con el “Runtime Linker”, y otro de CPU para realizar las interposiciones.

Pero a pesar de que los mecanismos de enlace (dependientes de la CPU) parecen a priori muy portables entre sistemas operativos, hay ligeras diferencias impuestas por los sistemas operativos que hacen que sean difíciles de reutilizar en su totalidad.

Además apostar por fijar un API demasiado estricta para ELF haría que ciertos casos como PowerPC™ 64 fuesen muy difíciles de resolver, además de en el futuro vetar el soporte a otros ABI que no sean ELF.

Por ello, aprovechando que los sistemas operativos suelen ir muy ligados a las arquitecturas sobre las que funcionan, se hará una implementación de estos “drivers” menos portable, pero muchísimo más compacta. Y por ello estos deberán replicarse para cada sistema operativo, aunque por su reducido tamaño (unas 200 líneas de código en el caso de GNU/Linux™) no implicará un gran problema.

De este modo la comunicación se realizará de la siguiente forma: la parte independiente solicitará operaciones al “Elf Backend”, que en realidad será directamente un “driver” para el sistema operativo. Este llamará posteriormente al “driver” de CPU adecuado para ejecutar las interposiciones:



En este diagrama podemos ver como del “driver” de un sistema operativo penden sus “drivers” específicos para CPU.

De esta forma podemos camuflar el tratamiento de ABI’s algo distintas de ELF como sería el caso de PowerPC™ 64.

Además se reduce notablemente el API que expondrá el “Elf Backend” a la parte no dependiente de la arquitectura y a los “backend” de usuario. Esta fuerte separación reducirá la cantidad de condicionales en el código, haciéndolo más compacto, legible y eficiente.

A la hora de implementar el “driver” de sistema operativo nos encontramos con diferencias entre sistemas, pero no demasiado fuertes, ya que se puede seguir un hilo general. Por ello se traza un esquema que se seguirá en todos los “drivers” de los diferentes sistemas operativos.

Hacer esto facilita el trabajo de crear un “driver” para un nuevo sistema operativo a partir de otro ya existente. El “driver” de sistema operativo debe facilitar a pDI-Tools la gestión de los objetos en memoria, y además aislar los mecanismos de reenlace usando los servicios del “driver” de CPU.

Vistas las secciones anteriores observamos que la primera tarea que debe resolverse es como acceder a las estructuras ELF. Para ello el “Elf Backend” debe ser responsable de construir una lista de objetos, para que la parte independiente pueda trabajar con ellos sin preocuparse de como se organiza el “Runtime Linker”.

Cada nodo de la lista de objetos de pDI-Tools contendrá información sobre el objeto. Esta información será recogida de los “Dynamic Tags” de los objetos en memoria, tratada si es necesario y precalculada en algunos casos. Como veremos más adelante, la información sobre cada objeto en memoria, irá contenida en una estructura PDI_ELFOBJ, donde cada estructura contiene un enlace a la información de objeto anterior y posterior, según el orden de la lista de objetos del “Runtime Linker”.

Parte de esta información se presentará al código no dependiente de la plataforma, mientras que el resto será para uso interno del “Elf Backend”.

Esta información no sólo acelerará y simplificará la obtención de un cierto dato durante la ejecución, sino que además centrará el proceso de los “Dynamic Tags” a un sólo punto del programa.

Como hemos visto, por la presencia o ausencia de los “Dynamic Tags” cuando se dan diferencias incluso entre diferentes versiones de un mismo sistema operativo. Por ejemplo, en GNU/Linux™ 2.6.x con GNU C Library 2.3.x, nos encontraremos con objetos que carecen del “Dynamic Tag” `DT_REL` y `DT_RELA`, mientras que con GNU C Library 2.2.x siempre está presente. En estos casos se adjunta en la información de objeto una marca, que llamaremos “BrokenObject”, que indica que el objeto no se puede instrumentar.

Tambien encontramos otras diferencias que no quebrantan el estándar pero implican muchas distinciones posteriores. Por ejemplo en Irix la la sección `DT_JMPREL` no existe. Esta sección es opcional, según el estándar, aunque por regla general está presente en el resto de sistemas operativos. Dejando los mecanismos de búsqueda dentro del “Elf Backend” no sólo aislamos la parte independiente un poco más de ELF, sino que obtenemos un código más simple, más específico y más rápido.

El resto de diferencias afecta directamente al contenido de las secciones ELF. Algunos “Runtime Linker” a veces modifican el contenido de las secciones (respecto a su aspecto en disco) por motivos de optimización. Esto implica que el código específico para un determinado sistema operativo, aunque no sea excesivamente diferente al de otro sistema operativo, esté lleno de pequeños ajustes para poder utilizar dichos datos.

Por último encontramos que algunas implementaciones proporcionan nuevos “flags” o datos que pueden facilitar y acelerar algunos mecanismos y que por lo tanto vale la pena aprovechar.

PDI_ELFOBJ. Estas estructuras contienen mucha información precalculada y con ella se consigue acelerar notablemente la ejecución de pDI-Tools ya que reduce la cantidad de código replicado y

cálculo.

Ejemplos de esta información son punteros a las principales secciones ELF, valores extraídos de las mismas, nombre del DSO, su alias, punto de carga del DSO, etc.

Pero no en todas las plataformas nos interesa la misma información. Por ello esta estructura de información se divide en una parte común para todas las plataformas (no dependiente de la arquitectura) y una parte específica, establecida en tiempo de compilación, totalmente dependiente de la arquitectura.

Por último explicamos que labores debe cumplir la capa encargada de gestionar la CPU.

El "driver" de CPU puede realizar tres posibles operaciones: establecer los reenlaces, establecer las redefiniciones y establecer las interposiciones mediante "callback". De estas tres sólo son frecuentes las dos primeras.

No se establece un API común para estas operaciones, y se deja libre frente a cualquier necesidad del "Elf Backend". No obstante, hasta la fecha se sigue el mismo esquema para todas las implementaciones.

En la siguiente sección describimos a grandes rasgos el API que ofrece el "Elf Backend" para realizar todas las operaciones antes citadas.

#### 3.1.5.1. API de un "Elf Backend"

En esta sección enumeramos una por una las funciones que debe implementar cara a pDI-Tools el API de un "Elf Backend". Antes de comenzar a enumerarlas, damos unas pinceladas generales para entender mejor este API.

Se pretende mantener un API lo más reducida y simple posible. Se observa que cada punto de entrada tiene un objetivo claramente definido aunque en ningún momento se marca el formato del ABI que se trata por debajo.

Pero se hace una excepción con la gestión de los "callback". Debido a la complejidad de los "callback" es muy difícil decidir una única estructura que deban seguir todas las arquitecturas. De hecho puede ser hasta contraproducente ya que implicaría mayores niveles de abstracción, más complejidad y consiguientemente más código y lentitud. Por lo tanto se deja en manos del programador de la arquitectura como se implementará el mecanismo de "callback" y todas las estructuras en memoria que este requiera.

Respecto al resto el funcionamiento es bastante claro, y como veremos el "Elf Backend" implementa funcionalidades básicas como su propia inicialización y finalización, resolución de símbolos, aplicar y deshacer las interposiciones de código y gestión de los "callback".

```
int _pdi_arch_init(void);
int _pdi_arch_fini(void);
```

**Descripción.** Estas funciones se encargan de inicializar y finalizar el "Elf Backend", respectivamente. La primera se llama durante la inicialización de pDI-Tools y la segunda durante su finalización.

Generalmente un "Elf Backend" no suele requerir una inicialización previa. Sin embargo hay casos como Solaris donde se deben realizar unas cuantas operaciones para poder obtener información clave para poder trabajar con las estructuras ELF. O el caso de GNU/Linux™, que debemos averiguar antes que versión de GNU C Library se está ejecutando ya que en la versión 2.3.x se

han introducido cambios importantes en el “Runtime Linker”. Para inicializar y finalizar el “Elf Backend” se deben implementar las funciones `_pdi_arch_init()` y `_pdi_arch_fini()`.

**Parámetros.** Estas funciones no reciben parámetros de entrada.

**Valor devuelto.** 0 si todo fue bien, -1 en caso de error.

```
int _pdi_arch_initSafeFuncs(void);
```

**Descripción.** Esta función se encarga de obtener punteros reales a las funciones que se usarán para evitar que pDI-Tools sea instrumentado accidentalmente.

**Parámetros.** Esta función no recibe parámetros de entrada.

**Valor devuelto.** 0 si todo fue bien, -1 en caso de error.

```
int _pdi_arch_initObjectList(void);
int _pdi_arch_refreshObjectList(void);
```

**Descripción.** La función `_pdi_arch_initObjectList()` crea una copia independiente de la plataforma de la lista de objetos del “Runtime Linker”. La función `_pdi_arch_refreshObjectList()` nos permitirá resincronizar esta copia con la lista actual del “Runtime Linker”.

Con estas funciones podemos implementar casi toda la lógica de la aplicación en la parte no dependiente de la arquitectura y reducir notablemente la cantidad de código en los “Elf Backends”. La mayor parte de pDI-Tools es código de gestión de las estructuras en memoria y comprobaciones simples como si está o no cargado un DSO en memoria, resolver referencias a objetos tanto por nombre como por alias, comprobar que las interposiciones no se interfieren, etc.

**Parámetros.** Estas funciones no reciben parámetros de entrada.

**Valor devuelto.** 0 si todo fue bien, -1 en caso de error.

```
void _pdi_arch_finiObjectList(void);
```

**Descripción.** Esta función se encarga de liberar la memoria de la lista de objetos independiente de la plataforma. Se le llama durante la finalización de pDI-Tools.

**Parámetros.** Esta función no recibe parámetros de entrada.

**Valor devuelto.** Esta función no devuelve nada.

```
void *_pdi_arch_resolveSymbol(char *name);
void *_pdi_arch_resolveSymbolInObject(PDI_ELFOBJ *obj, char *name);
```

**Descripción.** Para resolver un símbolo el "Elf Backend" debe ofrecer las funciones `_pdi_arch_resolveSymbol()` y `_pdi_arch_resolveSymbolInObject()`. La primera función busca entre todos los objetos en memoria un puntero al símbolo en el objeto que lo exporta. La segunda busca la información de un símbolo en un determinado objeto. No debe interpretarse como "buscar un símbolo en el espacio de nombres de un determinado objeto": simplemente busca la implementación del símbolo en el objeto, y si fracasa devuelve `NULL`.

**Parámetros.**

- ***obj*.** Indica en que objeto debemos buscar el símbolo indicado por el parámetro *name*.
- ***name*.** Es el nombre del símbolo que pretendemos resolver.

**Valor devuelto.** Se devuelve un puntero al punto de entrada de la función en caso de éxito, o `NULL` en caso de no poder resolverse.

```
int _pdi_arch_symbolIsUsedOrReferenced(PDI_ELFOBJ *obj, char *name);
```

**Descripción.** Esta función indica si el símbolo de nombre *name* es referenciado o implementado por el objeto *obj* (o ninguno de los dos).

**Parámetros.**

- ***obj*.** Indica en que objeto debemos buscar la información del símbolo indicado por el parámetro *name*.
- ***name*.** Es el nombre del símbolo que pretendemos examinar.

**Valor devuelto.** Devuelve 0 si el símbolo *name* no se usa ni se implementa por *obj*. Devuelve un valor positivo si está implementado por el objeto, o un valor negativo en caso de sólo usarse por él.

```
int _pdi_arch_initCallback(void);
int _pdi_arch_finiCallback(void);
```

**Descripción.** Estas funciones se encargan de inicializar y finalizar el sistema de gestión de interposiciones por "callback" dependiente de la arquitectura.

**Parámetros.** Esta función no recibe parámetros de entrada.

**Valor devuelto.** Esta función devuelve un número natural.

```
int _pdi_arch_newInterposition(PDI_ELFOBJ *eo, char *wname, PDI_INTERCEPT
*i);
```

**Descripción.** Esta función da de alta la información dependiente de la arquitectura en una interposición. Sólo se actualizan y preparan las estructuras de datos, no se instala la interposición. Para instalar la interposición se deberá llamar posteriormente a la función `_pdi_arch_callback()`, `_pdi_arch_redefine()` o `_pdi_arch_relink()`.

**Parámetros.**

- ***eo*.** Objeto en el que damos de alta la interposición.
- ***wname*.** Nombre de la función “wrapper”, o lo que es lo mismo, nombre de la función que interponemos.
- ***i*.** Interposición donde añadimos la información dependiente de la arquitectura.

**Valor devuelto.** Devuelve 0 en caso de éxito, o -1 en caso de algún error.

```
int _pdi_arch_callback(PDI_ELFOBJ *eo, PDI_INTERCEPT *i);
```

**Descripción.** Esta función realiza la acción de instalar una interposición de código mediante el mecanismo de “callback” el objeto *eo*.

**Parámetros.**

- ***eo*.** Objeto sobre el que estableceremos las interposiciones por “callback”.
- ***i*.** Estructura que define la interposición que se va a instalar, es decir, se indica el tipo de interposición y sus parámetros.

**Valor devuelto.** 0 en caso de éxito, -1 en caso de error.

```
int _pdi_arch_redefine(PDI_ELFOBJ *eo, PDI_INTERCEPT *i);
```

**Descripción.** Estas función instala una redefinición sobre la función implementada en el objeto *eo*.

**Parámetros.**

- ***eo*.** Objeto que contiene la implementación de la función a redefinir.
- ***i*.** Estructura que define la interposición que se va a instalar, es decir, se indica el tipo de interposición y sus parámetros.

**Valor devuelto.** 0 en caso de éxito, -1 en caso de error.

```
int _pdi_arch_relink(PDI_ELFOBJ *eo, PDI_INTERCEPT *i);
```

**Descripción.** Esta función se encarga de instalar un reenlace sobre las llamadas a una función desde el objeto *eo*.

**Parámetros.**

- ***eo*.** Objeto sobre el que instalaremos el reenlace.
- ***i*.** Estructura que define la interposición que se va a instalar, es decir, se indica el tipo de interposición y sus parámetros.

**Valor devuelto.** 0 en caso de éxito, -1 en caso de error.

```
int _pdi_arch_undoCallback(PDI_ELFOBJ *eo, PDI_INTERCEPT *i);
int _pdi_arch_undoRedefine(PDI_ELFOBJ *eo, PDI_INTERCEPT *i);
int _pdi_arch_undoRelink(PDI_ELFOBJ *eo, PDI_INTERCEPT *i);
```

**Descripción.** Estas funciones sirven para desinstalar los diferentes tipos de interposiciones de código.

**Parámetros.**

- ***eo*.** Objeto al que pertenece esta interposición.
- ***i*.** Interposición a desinstalar.

**Valor devuelto.** 0 en caso de éxito, -1 en caso de error.

```
int _pdi_arch_deactivateRedefinition(PDI_INTERCEPT *i, PDI_ELFOBJ *eo);
```

**Descripción.** Desactiva la redefinición *i* en el objeto *eo*. Esta función se usa cuando se carga un "backend" en memoria en tiempo de ejecución para que no se vea afectado por las redefiniciones instaladas.

**Parámetros.**

- ***i*.** Redefinición que desactivamos.
- ***eo*.** Objeto en el que desactivamos la redefinición.

**Valor devuelto.** 0 en caso de éxito, -1 en caso de error.

```
void _pdi_arch_freeInterposition(PDI_ELFOBJ *eo, PDI_INTERCEPT *i);
```

**Descripción.** Esta función libera la memoria usada por la información dependiente de la arquitectura de la interposición *i* instalada en el objeto *eo*. No desinstala la interposición, para ello ha

de usarse una de las funciones `_pdi_arch_undoCallback()` `_pdi_arch_undoRedefine()` o `_pdi_arch_undoRelink()`.

**Parámetros.**

- ***eo*.** Objeto en el que está dada de alta la interposición.
- ***i*.** Interposición en la que quitaremos la información dependiente de la arquitectura, liberando memoria.

**Valor devuelto.** Devuelve 0 en caso de éxito, o -1 en caso de algún error.

```
int _pdi_arch_defaultMaxStubs(void);
```

**Descripción.** Esta función aconseja el tamaño del "pool" de "stubs" por defecto para esta plataforma. Generalmente la máxima cantidad de "stubs" que quepan en una página de memoria.

**Parámetros.** Esta función no recibe parámetros de entrada.

**Valor devuelto.** Esta función devuelve un número natural.

```
int _pdi_arch_newElfObj(PDI_ELFOBJ *eo, int objpool_index);
```

**Descripción.** Esta función reserva una estructura de información especifica de la arquitectura sobre un objeto (una estructura PDI_ARCH_ELFOBJ) y la asigna al objeto *eo*.

pDI-Tools no reserva memoria dinámicamente para las estructuras de información sobre los objetos independiente de la arquitectura. En su lugar, al iniciarse, reserva un "pool" de objetos estático. Por ello, y en caso del que "Elf Backend" quisiese organizar de la misma manera, se le pasa el índice de la información no dependiente en el "pool" de objetos.

**Parámetros.**

- ***eo*.** Información no dependiente de la arquitectura del objeto que se le quiere añadir la información dependiente.
- ***objpool_index*.** Índice en el "pool" de objetos del objeto *eo*.

**Valor devuelto.** Devuelve 0 en caso de éxito, o -1 en caso de algún error.

```
void _pdi_arch_freeElfObj(void);
```

**Descripción.** Libera la memoria ocupada por la parte dependiente de la arquitectura en una estructura PDI_ARCH_ELFOBJ.

**Parámetros.**

- ***eo*.** Objeto en el que liberamos memoria.

- ***`objpool_index`.*** Índice en el "pool" de objetos del objeto *`eo`*.

**Valor devuelto.** Devuelve 0 en caso de éxito, o -1 en caso de algún error.

# 3.2. Políticas seguidas para obtener un código portable

En esta sección se explica que decisiones se han tomado para maximizar la portabilidad del código fuente de pDI-Tools.

Se trata desde el dialecto de C usado hasta las funciones de librería usadas. Esto delimita el tipo construcciones de C usadas, el uso del preprocesador, los tipos de datos disponibles y otros detalles.

Se explica el impacto que han tenido estas decisiones y consideraciones en el código, y como se ha hecho para se mantenga simple y mantenible.

Por último se explica la estrategia usada para incorporar información dependiente de la arquitectura de forma que sea transparente y usable por la parte portable.

## 3.2.1. Variante de C usada

Para programar pDI-Tools determine usar un C lo más estándar posible. Para ello se ha escogido el dialecto ISO C90 de C. El ISO C90 es un dialecto de C entendido por todos los compiladores actuales y rara vez genera incompatibilidades.

Para verificar que pDI-Tools cumple este requerimiento se realizaban cada cierto tiempo compilaciones con los "flags" `-ansi` y `-pedantic`. Estos "flags" hacen que compilador GNU C Compiler muestre avisos y errores en caso de usarse construcciones específicas del compilador y/o llamadas a funciones exclusivas de GNU C Library.

Lo malo es que el ISO C90 en cierto modo es un poco incompleto, y en ocasiones queda a manos del implementador del compilador tomar algunas decisiones. Un ejemplo sería indicar que una determinada función sea el segmento de código de inicialización o finalización de un DSO (secciones `.init` y `.fini`). No obstante se procura recurrir a estas características lo mínimo posible.

Todo esto significa a renunciar a muchas extensiones, que aunque muy comunes, nadie nos garantiza que estén presentes en todos los compiladores. Por lo tanto nunca se usan comentarios C++, ni uniones anónimas, ni espacios de nombres (de C++) ni declaración de variables en medio del cuerpo de una función.

El resultado es un código bastante claro, clásico y sencillo en el aspecto de que no explota explícitamente características avanzadas del compilador. Esto no significa que algunas veces se escriba el código con un poco de picardía para poder reconducir un poco la generación del binario y así explotar ciertas optimizaciones comunes entre muchos compiladores modernos.

## 3.2.2. El preprocesador

El preprocesador (cpp(1)), a pesar de ir ligado a un compilador de C, es una aplicación que implementa un lenguaje de macros muy potente. De hecho es muy normal encontrar proyectos en otros lenguajes que hacen uso del preprocesador de C.

El problema es que el preprocesador a veces varía mucho de un compilador a otro. No todos ofrecen la misma potencia, flexibilidades o incluso algunos interpretan de forma diferente algunas estructuras. Por ejemplo GNU C Compiler permite declarar macros con número de parámetros variable, cosa nada habitual en otros compiladores.

El preprocesador, es esencial para establecer buenos entornos de depuración o para crear programas multiplataforma. Es ideal para tomar opciones de configuración en tiempo de compilación, para crear alias a estructuras, tipos y funciones y para activar/desactivar código.

El preprocesador decide que se compila finalmente, y por lo tanto, el código que intervendrá en la generación del ejecutable. Esto nos permite activar y desactivar código sin penalizaciones en el tiempo de ejecución de la versión final. Además nos permite escoger el código que se usará en una plataforma u otra, reduciendo el tamaño del binario final.

No obstante sólo se usan unas pocas características del preprocesador para evitar ligarnos accidentalmente a un compilador determinado:

- **El carácter # siempre se escribe en la columna cero.** Todas las directivas del preprocesador comienzan por el carácter # (`#define`, `#if`, `#error`, ...). En algunos preprocesadores antiguos no se acepta que antes del carácter # hayan espacios. En cambio si que se acepta en todos los preprocesadores que hayan espacios entre el carácter # y la directiva del preprocesador.

  Esta limitación deriva del C original de Kernighan y Richie y ha sido mantenida en muchos compiladores hasta hace poco.

  Así las directivas se indentan de la siguiente forma:

  ```
  #if HAVE_STRING_H
  #  include<string.h>
  #else
  #  if HAVE_STRINGS_H
  #    include<string.h>
  #  endif
  #endif
  ```

- **No se declaran nunca constantes vacías.** Es normal encontrarse construcciones como esta en programas en C:

  ```
  #define DEBUG

  void myfunc() {
  #ifdef DEBUG
    debug_msg("Se ejecuta la función 'myfunc()'");
  #endif

  /* ... */
  }
  ```

  El problema de usar constantes vacías es que es fácil escribir a veces por accidente `#if DEBUG` en lugar de `#ifdef DEBUG`. El resultado depende totalmente de la interpretación del compilador y por lo tanto es difícil saber si se usará el código en el interior del `#if`.

  Por ello toda constante debe tener asignado un valor, y en caso de querer usarse como "flag" de compilación esta debe recibir exactamente el valor 1. Así se interpretará siempre de la misma forma en las condiciones `#ifdef` y en las condiciones `#if`.

- **Se evita el uso de condicionales siempre que se pueda.** La mejor manera de usar el preprocesador es de forma que no se note. Si se usan intensivamente las directivas condicionales del preprocesador se llega a un código ilegible, difícil de depurar y comprender.

  Un programa portable en muchas ocasiones debe presentar diferentes versiones de código para diferentes arquitecturas. La manera más inmediata de hacerlo es mediante condiciones del preprocesador. Esta manera es adecuada cuando las porciones de código son pequeñas y representan una pequeña parte del total de una función.

  El problema aparece cuando estas directivas engloban muchas funciones o trozos de código grande. En estos casos es preferible aislar el código para las distintas plataformas en ficheros diferentes. Así, si se pretende leer el código de una implementación, no se ve código de otras implementaciones que pueda inducir a confusión.

- **Sólo se usan macros en determinadas ocasiones.** Se ha intentado reducir el número de macros al máximo posible, siempre y cuando la legibilidad no se vea afectada por ello.

  Las macros se usan principalmente para crear cadenas de texto con el nombre de funciones y para construir los tipos ELF adecuados según la plataforma. Para ello el preprocesador debe implementar dos operadores pertenecientes a la especificación ISO C90: “ISO stringification operator #” y “token paste operator ##”.

  El primero, “ISO stringification operator #”, permite construir una cadena de texto a partir de un “token”. De esta forma podemos referirnos a una función y su nombre mediante macros:

```
#define STRINGIZE(x)    #x
#define MI_FUNCION      hello
#define MI_FUNCION_NAME STRINGIZE(hello)

int MI_FUNCION() {
  printf("La función ’%s’ dice ’hola mundo!’\n", MI_FUNCION_NAME);
}
```

  El segundo permite concatenar dos parámetros o “tokens”. Esto nos permite usar unas constantes o macros del preprocesador según la arquitectura. Por ejemplo todos los tipos ELF llevan el prefijo `Elf32_` o `Elf64_` según la arquitectura objeto sea de 32 o 64 bits. Por ello se usa la macro `ElfW(x)` que escoge por nosotros el tipo adecuado. La implementación de esta macro hace uso del operador de concatenación de “tokens”:

```
/* NOTA: __ELF_NATIVE_CLASS puede valer 32 o 64. Depende de la arquitectura */
#ifndef ElfW
#define ElfW(type)      _ElfW (Elf, __ELF_NATIVE_CLASS, type)
#define _ElfW(e,w,t)    _ElfW_1 (e, w, _##t)
#define _ElfW_1(e,w,t)  e##w##t
#endif

ElfW(Sym) *search_sym(char *sym)
{
  ElfW(Rel) *rel;
  ElfW(Addr) tmp;

  /* ... */
}
```

  El resto de macros por regla general suelen ser muy sencillas y se limitan tan sólo a operaciones aritméticas. Se evita usar macros complejas, de gran tamaño, o que representen bloques de código.

### 3.2.3. Tipos de datos

Al escribir una aplicación multiplataforma se ha de ser cuidadoso con los tipos de datos, especialmente cuando se apunta tanto a plataformas de 32 como de 64 bits.

El tamaño de los tipos de datos en un modelo de programación y otro a veces varían, y por lo tanto también varía el rango de los números que puede manejar.

Además hay tipos que son exclusivos de un compilador o de un determinado estándar. Por ejemplo ISO C99 aporta el tipo long long int que fuerza a usar un entero de 64 bits.

Usaremos siempre tipos sencillos y fácilmente predecibles en todas las arquitecturas. Por ello nos ceñiremos sólo a tipos, operaciones y asignaciones entre tipos declaradas por el estándar ISO C90.

Sólo se quebrantará esta regla en el caso de operaciones con punteros a funciones, que por características del programa, son inevitables. Por suerte esto es algo soportado por todos los compiladores del mercado.

Respecto a la construcción de estructuras de datos y uniones se será respetuoso al máximo con el estándar. Esto significa que no existirán estructuras o uniones anónimas, ni se usarán extensiones del lenguaje para poder modelarlas mejor.

Los tipos de datos basados en estructuras se crearán usando el operador `typedef`. De esta forma nos referiremos a las estructuras normalmente por su nombre de tipo y no por el nombre de la estructura. El nombre de la estructura será el nombre del tipo pero con el prefijo `_tag_`. Por ejemplo:

```
typedef struct _tag_BECFG_GRAPH_NODE {
  BECFG_OBJ_INFO        *gn_object;

  int                   gn_weight;
  int                   gn_hasPar;

  int                   gn_nto;
  struct _tag_BECFG_GRAPH_NODE **gn_to;
} BECFG_GRAPH_NODE
```

Por último se usarán, siempre que se pueda, los tipos de datos indicados por el ISO C90 para las funciones de las librerías de C. Así por ejemplo siempre se preferirá usar el tipo ssize_t a int. También se usará siempre que se pueda tipos ELF para evitar hacer conversiones innecesarias.

### 3.2.4. Datos no-dependientes y dependientes de la arquitectura

Los datos dependientes de la arquitectura siempre van encapsulados en los datos independientes de la arquitectura. Pondremos por ejemplo la estructura de información de una interposición. Esta estructura, a parte de indicar el tipo de interposición, el símbolo sobre el que actúa, el “backend” y función que reciben el control, debe guardar también como deshacerse más tarde así misma. Esta información depende mucho de la arquitectura y sistema operativo.

Generalmente toda la información que maneja pDI-Tools suele ir agrupada en estructuras. Así tenemos estructuras de información sobre las interposiciones, estructuras de información sobre los objetos en memoria, etc.

La información dependiente de la arquitectura podría ir mezclada en la misma estructura con la información independiente de la arquitectura. Luego, mediante construcciones `#if ... #endif` se podría activar y desactivar de una plataforma a otra. El problema con esta solución es que lleva a un

código difícil de mantener, y lo peor de todo, no marca una separación clara entre la información dependiente de la independiente de la arquitectura.

Para resolver estos problemas se decidió que la información dependiente de la arquitectura debe encapsularse en una estructura que reciba el nombre “arch”. Volviendo al ejemplo anterior, tendríamos:

```
typedef struct _tag_PDI_INTERCEPT {
  /* Información no-dependiente sobre la interposición */
  int                     i_type;     /* tipo de interposición                */
  char                    *i_symbol;  /* símbolo al que afectará              */
  struct _tag_PDI_ELFOBJ  *i_backend; /* backend que ejecuta esta interposición */
  void                    *i_wrapper; /* función encargada de interponerse    */

  /* datos específicos según arquitectura */
  PDI_ARCH_INTERCEPT      *i_arch;

  /* Siguiente intercepción en la lista */
  struct _tag_PDI_INTERCEPT *i_prev;
  struct _tag_PDI_INTERCEPT *i_next;
} PDI_INTERCEPT;
```

Donde la estructura *`i_arch`* es proporcionada por el “Elf Backend” específico de la arquitectura. Nótese que en realidad es un puntero a la información dependiente de la arquitectura y no la información en sí. Los motivos son los siguientes:

- Si el “Elf Backend” usa estas estructuras desde ensamblador, le facilitaremos el trabajo si en las estructuras independientes de la arquitectura usamos punteros. Al usar punteros a la información dependiente hacemos que sea más fácil predecir el tamaño de la estructura no independiente y por lo tanto facilitar el acceso a los campos de la estructura desde ensamblador.
- Como se deja la gestión de memoria de la información independiente al “Elf Backend” este puede ejecutar sus propias optimizaciones.
- No es habitual, pero sí posible, que en un determinado caso no haga falta información independiente de la arquitectura. En ese caso ahorraremos espacio en memoria siguiendo esta estrategia.

## 3.3. Organización del código según la plataforma

En la sección *Distribución de los ficheros* se ha explicado por encima los directorios y ficheros de pDI-Tools. En esta sección entraremos en detalles sobre la organización de directorios del código fuente y sus ficheros.

Aunque la distribución del código fuente de pDI-Tools se ve claramente influenciada por las herramientas Autoconf y Automake, en realidad esta distribución está marcada por el objetivo de la portabilidad.

La primera decisión tomada es sobre la longitud de los nombres de ficheros. Esta aplicación no está pensada para sistemas MS-DOS, por lo tanto no se establece la dura limitación de nombres de fichero de 8 carácteres. No obstante no todos los sistemas operativos Unix™ son tremendamente generosos en este aspecto. Aún podemos encontrar sistemas donde exista una limitación de 32 carácteres en el nombre de archivo. Por ello siempre se procura usar nombres de ficheros de menos de 16 carácteres para garantizar que no habrán problemas

Por último siempre se escriben los nombres de archivo en minúscula. Sólo existirá la excepción de los ficheros con la extensión `.S` (fichero en ensamblador que debe ser procesado por el preprocesador de C).

El código fuente se distribuye en dos directorios principales: el directorio `./include/` y el directorio `./src/`.

En las dos siguientes secciones comentamos cada uno de estos directorios. El primero, `./include/` se comenta en la sección "Ficheros de cabecera de pDI-Tools" y el otro, `./src/`, en la sección "Código fuente de pDI-Tools".

## 3.3.1. Ficheros de cabecera de pDI-Tools

El directorio `./include/` contiene todos los archivos de cabecera que se instalarán en el sistema para poder desarrollar "backends" y aplicaciones que usen la API ofrecida por pDI-Tools.

Este directorio a su vez contiene otros directorios. Estos directorios contienen ficheros de cabecera específicos para determinados sistemas operativos. Por ejemplo tendremos el directorio `./include/linux/`, `./include/solaris/`, etc.

Los ficheros específicos de una arquitectura nunca deben usarse directamente, sino a través de ficheros de cabecera no dependientes de la arquitectura. De hecho los ficheros de estos subdirectorios están para complementar los ficheros de cabecera en el directorio `./include/`.

Los principales ficheros del directorio `./include/` son los siguientes:

- **`backend.h`.** Este fichero incluye toda la información necesaria para poder programar un "backend" para pDI-Tools.
- **`config.h`.** Este fichero debe incluirse en todos los ficheros de código de pDI-Tools. Incluye todos los ficheros de cabecera necesarios para compilar pDI-Tools además de incluir la configuración en tiempo de compilación generada por Autoconf (en el fichero `autoconfig.h`).
- **`ebeif.h`.** Este fichero lo construye Autoconf a partir del fichero `ebeif.h.in`. Se encarga de declarar los principales tipos de datos, mostrar la mayor parte del API y de incluir los ficheros `exports.h` y `types.h` específicos de una arquitectura.

  Sólo es útil usar este fichero en caso de querer trabajar directamente con el API de pDI-Tools o para acceder a sus estructuras internas.
- **`log.h`.** Este fichero exporta las funciones de depuración de pDI-Tools. Este sistema es útil para depurar los "backends". No sólo los mensajes de depuración se verán afectados por la verbosidad configurada por el usuario para pDI-Tools, sino que además es seguro usarlas durante instrumentaciones agresivas.
- **`threadid.h`.** Este fichero nos permite configurar la función encargada de asignar identificadores a los "threads" de la aplicación.

  Es necesario definir un "thread id resolver" en caso de que estemos instrumentando con "callbacks" una aplicación multihilo.

  En teoría el módulo encargo de instalar el "thread id resolver" es el "backend" `runtime`. Este, en su código de inicialización debe inicializar ciertos aspectos dependientes de la arquitectura, como es este caso.
- **`pdiconfig.h`.** Permite acceder a la configuración de pDI-Tools. Es útil cuando se desea que los "backends" sean sensibles a ella.

Dentro de un directorio específico a una arquitectura encontramos fundamentalmente dos ficheros imprescindibles: `types.h` y `exports.h`.

El primero, `types.h`, declara todas las constantes y estructuras de datos específicas para la arquitectura.

El segundo, `exports.h`, exporta todas las funciones dependientes de la arquitectura que pDI-Tools requiere para completar su cometido. Además le asigna a cada función un alias para poder referirnos a ellas de forma única desde el código portable. Por ejemplo, a la función de GNU/Linux™ `_pdi_linux_initObjectList()` y a la función de Irix `_pdi_irix_initObjectList()` se les asigna el mismo alias: `_pdi_arch_initObjectList()`.

Como los ficheros contenidos en el directorio `./include/` son esenciales para poder seguir desarrollando una vez instalada la aplicación, los ficheros contenidos en él también son instalados.

No obstante, respecto a los ficheros de cabecera sólo se instalarán los ficheros específicos para el sistema operativo que hemos compilado. Es decir, si estamos instalando una versión de pDI-Tools para Solaris no se instalarán los ficheros de cabecera para GNU/Linux™.

### 3.3.2. Código fuente de pDI-Tools

El directorio `./src/` contiene tanto ficheros de código como de cabecera que son necesarios para construir los binarios de pDI-Tools. No obstante este código sólo es necesario para construir los binarios de pDI-Tools, no para programar luego sobre el mismo.

Así aislamos en `./src/` todo el código fuente que no intervendrá en la instalación.

En este directorio nos encontramos un directorio por cada binario que se vaya a crear, a parte de un directorio especial `tests/` donde encontramos programas que testean el buen funcionamiento de los componentes de pDI-Tools.

El directorio más importante es `./src/link/` que genera la librería `libpdi.so` que implementa el mecanismo de interposición de código.

Dentro de `./src/link/` encontramos fundamentalmente dos tipos de elementos: ficheros de código C y subdirectorios con más código en C y ensamblador.

En el primer nivel (directamente dentro de `./src/link/` encontramos todo el código independiente de la arquitectura, y por lo tanto común para todos los sistemas.

En cambio los subdirectorios reciben el nombre de la plataforma a la cual va destinado el código que contienen. Así encontramos directorios como `linux/`, `solaris/`, etc.

Los ficheros de código de estos directorios van precedidos todos por un prefijo de dos letras y un guión. En GNU/Linux™ el prefijo es `lx-`, en Irix es `ix-`, etc.

Este prefijo nos permite ver rápidamente a que arquitectura pertenece un fichero (si depende de la arquitectura) y además evita colisiones de nombres con ficheros de otras arquitecturas. Esto simplifica el control de versiones y les da un nombre único a cada fichero, al menos dentro de la misma aplicación. Esto último muy práctico a la hora de imprimir mensajes de log y/o depuración.

## 3.4. Herramientas

Seguidamente explicamos el entorno de trabajo que se ha establecido para desarrollar pDI-Tools.

Habitualmente el entorno del programador y el entorno usado para compilar la aplicación suele ser el mismo. Es decir, si un usuario desea compilar una aplicación generalmente hace uso del mismo entorno de desarrollo que el programador. Con las herramientas Autoconf y Automake no es exactamente así.

Las herramientas Autoconf y Automake separan el desarrollo del entorno de programación y distribución. De hecho, estas herramientas motivan a que se distribuya el código y que la compilación forme parte de la distribución del programa.

Esto es algo muy propio del software libre y rara vez visto en software comercial. El software comercial suele venir empaquetado, cerrado y preparado para instalar. No te da oportunidad de ajuste más que las opciones de configuración que se hayan decidido poner en él. El software libre viene directamente con el código fuente a la vista, invitando a que se compruebe lo que hace, que se modifique y se ajuste tanto como se desee a las necesidades del usuario.

Las herramientas Autoconf y Automake se encargan de simplificar el proceso de compilación de la aplicación, automatizando la toma de decisiones para poder compilar el programa sobre la plataforma que se pretende ejecutar.

Estas herramientas se encargan de escribir una serie de “scripts” para la “shell” capaces de construir y configurar la aplicación en muchos sistemas distintos. Al ser “scripts” sencillos en una “shell” muy común, sh(1), garantizamos que funcionarán en una amplia variedad de sistemas.

Estos “scripts” se generan por las aplicaciones Autoconf y Automake. Son dos programas escritos en el lenguaje de macros m4(1). De la ejecución de estos programas saldrán un montón de “scripts” para sh(1) capaces de configurar la aplicación y generar los ficheros make necesarios.

Una vez generados estos ficheros no volverá ser necesario disponer de un intérprete del lenguaje m4(1), y por lo tanto la aplicación se podrá compilar en casi cualquier sistema.

Además estas herramientas son capaces de dejar la construcción del programa a medias para poder distribuir luego parte de los ficheros generados y así eliminar programas necesarios a la hora de compilar. Por ejemplo, supongamos que nuestra aplicación usa un parser generado por bison(1). En principio esto obligaría a todo usuario que quisiera compilar la aplicación a instalar bison(1). Pero Autoconf permite indicar que se genere y distribuya el fichero `.c` creado por bison(1). De esta forma el usuario podrá compilar el programa sin disponer de bison(1).

Todo esto sumado hace que sea muy fácil portar pDI-Tools a otras plataformas: si se es cuidadoso no es necesario ni disponer siquiera de todas las herramientas usadas para desarrollarlo. En principio es suficiente con la “shell” sh(1), un compilador de C, un make y, en caso de necesitarse, un ensamblador.

Respecto a la herramienta make sólo diremos que nos sirve cualquiera. Aunque make es una herramienta que presenta una gran compatibilidad entre diferentes versiones de fabricantes, Automake genera ficheros `makefile` tan simples que rara vez dan problemas. Y si una determinada versión de make da problemas, Autoconf lo detecta y avisa a Automake para que genere un `makefile` específico.

El compilador escogido para desarrollar pDI-Tools ha sido GNU C Compiler. GNU C Compiler es el compilador que soporta mejor los estándares además de estar disponible para casi cualquier plataforma. Además es software libre por lo cual garantizamos que nunca estaremos atados a una determinada marca o arquitectura.

En el caso de GNU/Linux™ sobre INTEL® 386 también es necesario usar un ensamblador. El ensamblador usado es GNU Assembler, que acompaña a GNU C Compiler.

El resto de herramientas usadas para desarrollar pDI-Tools como OpenJade, xsltproc, etc. no afectan a la construcción del software, sino tan sólo a la generación de la documentación. Para evitar hacer disponer y configurar estas herramientas se entrega la documentación ya generada.

# Capítulo 4. Esfuerzo y coste

En este capítulo se presenta como se ha estructurado el trabajo de creación de pDI-Tools y se estima el coste de éste si se hubiese realizado como un proyecto de “software” real.

El capítulo se divide en dos partes: planificación y coste.

En la primera parte se presenta la planificación de trabajo del proyecto en fases. La finalización de una fase representa una cantidad de trabajo sustancial y un hito en la historia del desarrollo de pDI-Tools. Además nos permite ver a grandes rasgos como se esperaba conducir, y como se condujo, el desarrollo de pDI-Tools: momentos más críticos, cuanto tiempo se dedicó a investigar, cuando pDI-Tools empezó a tener su aspecto definitivo, etc.

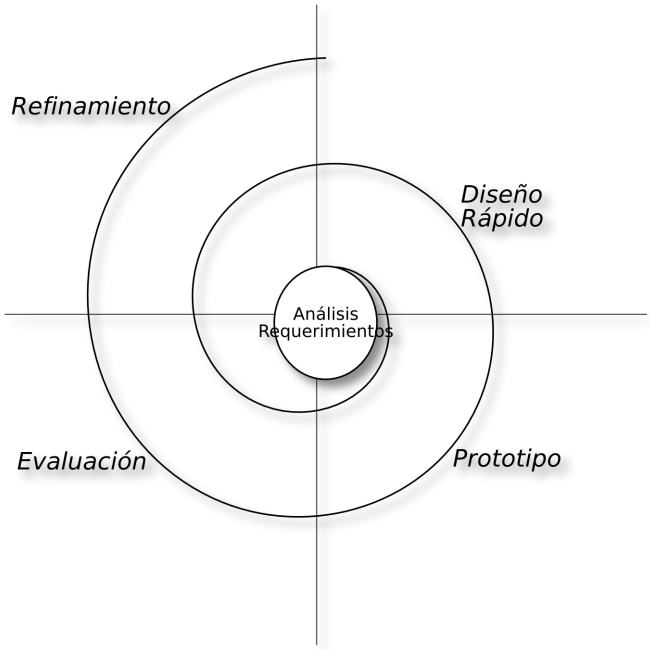
En la segunda parte se reune toda la información de distribución de tiempo de la primera parte y en base a ella se decide como se reparten costes: cuanta gente debería trabajar en el proyecto, con que remuneración, etc. Y al final del mismo presentamos el coste económico que hubiese tenido el proyecto.

## 4.1. Planificación

Inicialmente se quería portar DITools de Irix a GNU/Linux™. Este proyecto se empezó sin total seguridad de poder completarse o no. Por un lado no sabíamos si la implementación de ELF de GNU/Linux™ permitiría realizar las mismas operaciones que la implementación de Irix, y por otro lado no teniamos conocimiento sobre el grado de portabilidad de DITools.

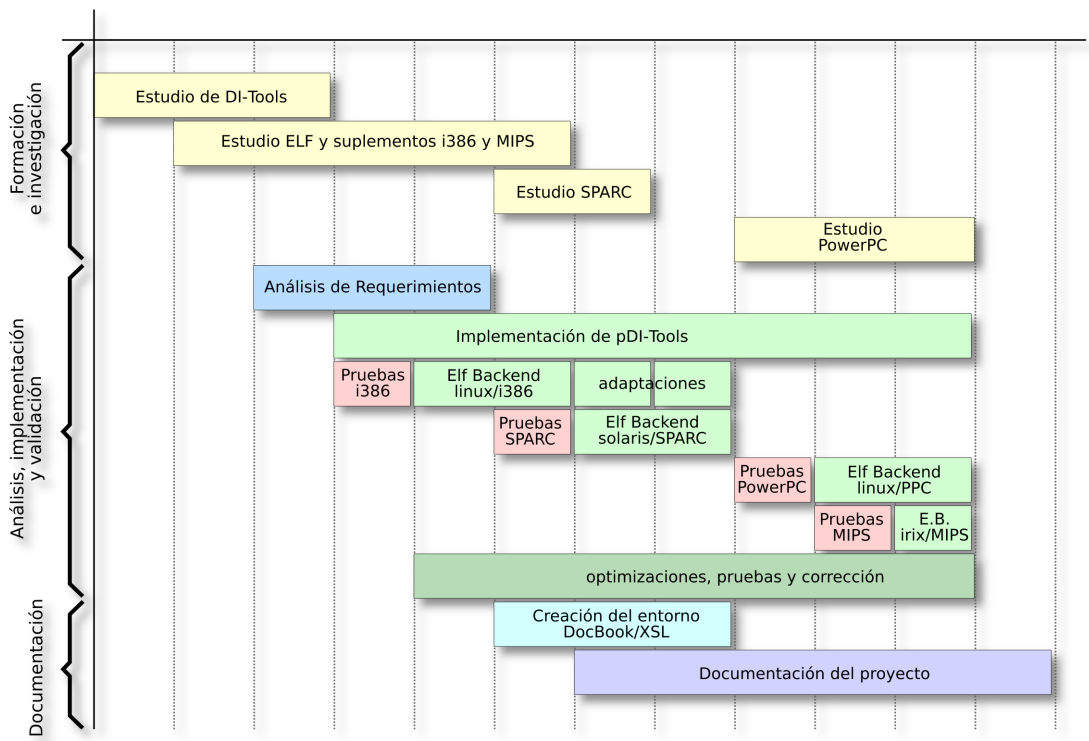
Terminado un proceso de investigación sobre ELF e ingeniería inversa sobre DITools se decidió reescribir la aplicación desde cero. Como se vió que sería un proyecto con un diseño básico que se iría perfeccionando y afinando con el tiempo se optó por un desarrollo basado en prototipos.

Esta metodología de diseño consiste en partir de un análisis de requerimientos y posteriormente se realiza un diseño rápido, una implementación del prototipo, su evaluación y finalmente su refinamiento. Si es necesario se itera sobre estos cuatro pasos hasta llegar al producto final. Sólo en casos extremos, en los cuales se ha partido de una mala base, se debe repetir el análisis de requerimientos.




En este diagrama se explica el modelo en diseño mediante prototipos. La espiral parte de un análisis de requerimientos y finalizará cuando se obtenga un prototipo que se pueda considerar un producto final.

No obstante este modelo en espiral de prototipos no es totalmente fiel a la planificación seguida para desarrollar este proyecto. A pesar de que hubo algunas iteraciones, hubo en realidad muchos procesos independientes en paralelo. En el siguiente gráfico se muestra a grandes rasgos como se planificó y ejecutó el desarrollo del proyecto a lo largo de doce meses:




En este gráfico se muestran las tareas realizadas. Cada división representa un mes de trabajo.

Entrando en detalle describimos las actividades de esta planificación:

A. **Estudio de DITools.** En esta fase se estudió DITools con el objetivo de decidir si era viable portarlo a GNU/Linux™. Al final resulto que el coste de portarlo era casi mayor que el coste de comenzarlo desde cero, así que se realizo un estudio para exprimir toda la experiencia que esta herramienta contiene. Además se depuró y se reprogramarón algunas partes para hacer el código más legible. El estudió tuvo como objetivo reunir experiencia para escribir una nueva aplicación.

B. **Estudio ELF y suplementos INTEL® 386 y MIPS®.** En esta fase se recogió información de cualquier tipo sobre ELF, arquitecturas, sistemas operativos, técnicas de reenlace, etc.

   Se hizo una comparativa, almenos superficial, entre formatos específicos de enlace dinámico para tener una visión global de como se implementan estos en diferentes plataformas.

   Se investigó también que técnicas de infección e interposición de código se han desarrollado o ensayado en ELF.

   También se profundizó en el conocimiento de las arquitecturas INTEL® 386 y MIPS®, tanto para poder entender DITools como para poder diseñar el nuevo pDI-Tools.

C. **Estudio SPARC.** En el quinto mes, ya con un conocimiento avanzado sobre ELF, se estudió una plataforma totalmente distinta tanto en arquitectura como en implementación. La plataforma escogida fue Solaris/SPARC debido a que parece de todas la más sencilla.

   Este estudio va tanto orientado a la implementación del “driver” para Solaris sobre SPARC como para poder extrapolar luego una interfaz portable para los “drivers”.

D. **Estudio PowerPC™.** Se investigó como funciona esta plataforma, la cual no está totalmente estandarizada a nivel ELF. Este estudio es el doble de costoso de lo habitual porque la especificación de 32 bits no se asemeja en nada a las recomendaciones para 64 bits.

E. **Análisis de requerimientos.** En el tercer mes, con la experiencia obtenida hasta ese momento, se empezó el análisis de requerimientos y a establecer el diseño inicial de pDI-Tools.

   Para poder empezar a programar lo antes posible se dividió pDI-Tools en grandes partes y se fijaron, sólo comenzar, a grandes rasgos las estructuras de datos principales. Estas se pudieron fijar rápidamente ya que ya se venían madurando durante el estudio de DITools.

   Este proceso duró tres meses aunque la mayor parte del trabajo está contenido principalmente en el primer mes.

F. **Implementación de pDI-Tools.** Esta fase contiene todo el desarrollo de la parte no dependiente de la arquitectura de pDI-Tools. El código desarrollado procura reciclar el máximo código posible sin mermar para nada la portabilidad.

   Se escribió todo el código encargado de configurar pDI-Tools, de procesar los ficheros de comandos, de realizar comprobaciones, gestionar “backends”, etc. Es decir, el código encargado de realizar la mayor parte de gestiones de pDI-Tools.

G. **“Elf Backend” Linux™/INTEL® 386.** En esta fase se implementó el código específico para realizar reenlaces, redefiniciones e interposiciones mediante el mecanismo de “callback” en GNU/Linux™ sobre INTEL® 386.

   Esta fase se dividió en pruebas de concepto, implementación y adaptaciones. Las pruebas de concepto sirvieron tanto para garantizar la viabilidad de las interposiciones como para ensayar su posterior implementación. Estas pruebas son pequeños programas con “hacks” que simulan la creación de las distintas interposiciones.

   Sigue la implementación del “driver”. A la vez se escribe, en la parte independiente el API, el protocolo a seguir para interaccionar con este “driver”.

Este código fué readaptado en dos ocasiones para poder dar soporte a la arquitectura Solaris/SPARC. Esto fue debido a nuestro desarrollo por prototipos: el primer prototipo era demasiado específico para INTEL® 386, así que se empezo un nuevo prototipo también apto para SPARC 32 bits. Luego se hizo otro prototipo (adaptación) para poder funcionar en entornos de 64 bits (SPARC 64).

H. **"Elf Backend" Solaris/SPARC.** Se crea un "driver" para un nuevo sistema operativo y una nueva arquitectura. El objetivo es desarrollar ambos "drivers" (GNU/Linux™ y Solaris) a la vez para poder definir la interfaz que compartiran para comunicarse con pDI-Tools.

   Esta fase tuvo repercusión sobre el resto del código ya que se portó la aplicación a un entorno de 64 bits. Esto sumado a que se cambió tanto de "hardware" como de sistema operativo expuso muchos problemas de portabilidad en el primer prototipo de pDI-Tools.

I. **"Elf Backend" Linux™/PowerPC™.** Consiste en ampliar el "driver" para GNU/Linux™ para que soporte las arquitecturas PowerPC™ de 32 y 64 bits.

   Como a estas alturas la interfaz entre la parte no dependiente y el "driver" estaba prácticamente congelada, el desarrollo de este "driver" apenas implicó cambios.

   No obstante sirvió para plantear como un "driver" puede soportar varias arquitecturas. Así se desarrolló una segunda capa de abstracción, totalmente interna al "driver" de GNU/Linux™, que permite reciclar mucho código relacionado con el sistema operativo y no con la CPU.

   Además se sabía de antemano que es una de las plataformas más duras de trabajar ya que PowerPC™ 64 bits es una plataforma todavía a medio desarrollar, desde el punto de vista de ELF.

J. **"Elf Backend" Irix/MIPS®.** El último "driver" que se escribió fue para Irix sobre máquinas MIPS®.

   Este "driver" se implemento en un tiempo record. Se escribió con objetivo de medir la facilidad de portar de golpe pDI-Tools a una nueva plataforma y arquitectura.

K. **Optimizaciones, pruebas y corrección.** La mayor parte de pruebas y optimizaciones se han realizado a lo largo del desarrollo, especialmente en el tema de la corrección del programa. No obstante una vez acabado el "software" se siguió sometiendo a varias revisiones y evaluaciones con objetivo de detectar donde se debía mejorar y, si era posible, se realizaban las mejoras.

L. **Creación del entorno DocBook/XSL.** Esta tarea consistió en construir un entorno de proceso de documentos XML en formato DocBook. Con este entorno de trabajo se simplificó la construcción y escritura de los documentos relacionados con esta aplicación.

   Esta fase incluye la creación de los ficheros `makefile` (responsables de construir la documentación), las hojas de transformación y estilo (que transforman un conjunto de ficheros XML en un fichero, y algunos "scripts" en Perl encargados de automatizar algunas tareas (por ejemplo la creación de las marcas de índice para poder indexar el texto DocBook/XSL), etc.

   Este desarrollo se dividió en dos partes: creación de lo más básico del entorno para poder escribir el documento "pDI-Tools: Descripción del proyecto" y después se añadió, según se iba necesitando, soporte a características más complejas como índices o imágenes.

M. **Documentación del proyecto.** Esta etapa se corresponde con la escritura del documento "pDI-Tools: Descripción del proyecto" y de la presente memoria.

   Para este proceso, gracias a la tecnología usada, sólo se necesitó un editor de textos (en concreto Vi IMproved) y editores de imágenes (Inkscape y GIMP principalmente, aunque no se estaba ligado a ninguno en especial).

Inicialmente se estimo que el proyecto tendría un coste total de 960 horas de trabajo. Esto correspondería a 12 meses de trabajo a 4 horas por día asumiendo que hay 20 días laborables cada mes.

Como suele suceder, se fue demasiado optimista y se subestimo la cantidad de trabajo, especialmente en cuanto a la documentación se refiere. En cambio casi se acerto el tiempo estimado para ingeniería inversa, el cual fue ligeramente superior debido al estado del código fuente DITools y la ausencia de documentación. Pero sin duda se acerto con el tiempo requerido para la programación de la parte no dependiente de pDI-Tools:

**Tabla 4-1. Comparación del tiempo estimado y empleado realmente**

| | Tiempo planificado | Tiempo real |
|---|---|---|
| Formación e investigación | 200 h. | 293 h. |
| Diseño | 60 h. | 78 h. |
| Ingeniería inversa | 100 h. | 125 h. |
| Programación | 400 h. | 399 h. |
| Documentación | 200 h. | 305 h. |
| total | 960 h. | 1.200 h. |

Para poder terminar el proyecto se tuvo que trabajar todos los días, excepto 5 de cada mes (asumimos un mes de 30 días). La cantidad de horas por día se mantuvo a 4 ya que más era imposible debido a que este proyecto se compagino con un trabajo de 5 horas diarias. Esto sumo un trabajo total de 1.200 horas de trabajo, superando en 240 horas las estimaciones iniciales.

Seguidamente, y con el fin de hacer una valoración del coste del proyecto, desglosamos todas estas horas de trabajo en tareas, y cada tarea la desglosamos en diferentes categorías. De esta forma podremos evaluar mejor la complejidad del proyecto y así calibrar el coste que hubiese tenido.

**Tabla 4-2. Tiempo empleado en cada tarea**

| etapa | formación e investigación | diseño | ing. inversa | programación | documentación | horas por etapa |
|---|---|---|---|---|---|---|
| A | 90 h. | - | 70 h. | 20 h. | - | 180 h. |
| B | 140 h. | - | - | - | - | 140 h. |
| C | 20 h. | - | - | - | - | 20 h. |
| D | 35 h. | - | - | - | - | 35 h. |
| E | - | 70 h. | - | - | - | 70 h. |
| F | - | - | - | 150 h. | - | 150 h. |
| G | - | - | 30 h. | 85 h. | - | 115 h. |
| H | - | - | 10 h. | 40 h. | - | 50 h. |
| I | - | - | 10 h. | 20 h. | - | 30 h. |
| J | - | - | 5 h. | 10 h. | - | 15 h. |
| K | - | - | - | 30 h. | - | 30 h. |
| L | 8 h. | 8 h. | - | 44 h. | - | 60 h. |
| M | - | - | - | - | 305 h. | 305 h. |
| total | 293 h. | 78 h. | 125 h. | 399 h. | 305 h. | 1.200 h. |

# 4.2. Valoración económica del esfuerzo

Para este proyecto es ideal que el diseño y el desarrollo lo ejecute una única persona. Esto se debe a que es necesario que tanto el diseñador como el programador tengan un profundo conocimiento de ELF, de los mecanismos de enlace dinámico de las plataformas objetivo y sobre sus arquitecturas.

Esto tiene como contrapartida que para ser justos se debe contratar al desarrollador como diseñador y programador a la vez.

Para hacer estimar el coste de esta aplicación asumiremos que un diseñador suele cobrar unos 20 € por hora de trabajo. Por otra parte también asumimos que alquilar una máquina del CEPBA tiene un coste de 1 €/hora.

**Tabla 4-3. Desglose del coste de pDI-Tools**

| concepto | unidades | precio unidad | total |
|---|---:|---:|---:|
| Horas de desarrollo | 1.200 h. | 20 €/hora | 24.000 € |
| Horas de máquina CEPBA | 160 h. | 1 €/hora | 160 € |
| Red Hat Linux™ 7.3[a] | 1 | 50 € | 50 € |
| Debian GNU/Linux™ | 1 | 0 € | 0 € |
| Entorno de desarrollo GNU | 1 | 0 € | 0 € |
| OpenOffice.org | 1 | 0 € | 0 € |
| Programa de ilustración Inkscape | 1 | 0 € | 0 € |
| Programa de ilustración GIMP | 1 | 0 € | 0 € |
| Herramientas DocBook/XSL | 1 | 0 € | 0 € |
| Impresiones y encuadernación | 1 | 150 € | 150 € |
| total | | | 24.360,00 € |

Notas de tabla:
a. Aunque puede ser gratuito para una empresa es aconsejable adquirir el producto para tener un mínimo de soporte

Podemos concluir que el proyecto tal como está costaría alrededor de 24.000 €. Un precio quizás algo elevado aunque en realidad ha representado mucho trabajo. Esto junto a que es un “software” con unos fines muy específicos, y por lo tanto con pocos clientes, debe ser el motivo por el cual no es una rama muy explotada por la empresa privada (de hecho la oferta casi se limita sólo a Dyninst y se trata de un producto creado por la Universidad de Maryland).

Según lo visto en mi experiencia en la empresa privada, más de seis años, rara vez se dedica tanto tiempo en pulir y perfeccionar tanto un producto. La empresa privada sólo quiere y necesita que el producto cumpla las funcionalidades básicas, y no se crea tanta documentación. Por ello el coste de desarrollar una aplicación similar a esta, en la empresa privada, sería bastante menor en tiempo y dinero. Las empresas tienen como objetivo maximizar beneficios, y por lo tanto evita malgastar recursos en pulir demasiado un proyecto: a corto plazo sería contraproducente (la empresa privada generalmente nunca mira a largo plazo).

# Capítulo 5. Conclusiones

Por último, en este apartado se hace una valoración personal del proyecto, tanto a nivel de conclusiones como conocimiento adquiridos con él. Finalmente se añade otra sección donde explico algunos objetivos futuros que podrían incrementar la potencia y funcionalidad de este “software”.

## 5.1. Conclusión y valoración

Tanto la instrumentación de código, como la manipulación de ejecutables y los ABI son asuntos muy poco tratados en general pero que, desde mi punto de vista, me han parecido muy interesantes. Este proyecto me ha permitido trabajar, investigar y aprender muchisimo sobre diferentes arquitecturas, sobre como funciona realmente el proceso de carga y el enlace en tiempo de ejecución y me ha introducido en el mundo de la instrumentación de código.

Ha sido todo un desafio, tanto de aprendizaje como de programación, en el que he podido aprender cosas que muy dificilmente hubiese podido aprender en la empresa privada o por cuenta propia.

Desde siempre he sentido un interés por el ensamblador, el “hardware” a bajo nivel y por saber como funciona realmente un sistema operativo. Cuando cogí este proyecto, al igual que mucha gente, tenía una visión muy simplificada y lejana de la realidad de los ABI. Generalmente se explican los formatos de enlace dinámico con cuatro diagramas y sin profundizar sobre como funciona en realidad. Cuando se mira detenidamente se descubre que lo que parece simple no es nada simple. El formato de los binarios es una de las bases de un sistema operativo, y decide en gran parte el rendimiento de un sistema[1]. El formato ELF es un formato de enlace donde esta cuidado hasta el último detalle, nunca falta o sobra información. Es lo suficiente compacto como para funcionar en maquinas empotradas y lo suficiente potente como para ser la base de grandes servidores.

Debido a que sólo se ha trabajado con estructuras ELF, sin usar información de depuración ni funcionalidades extraordinarias del sistema operativo, el resultado final no sólo es altamente portable y apenas tiene impacto en el rendimiento de la aplicación instrumentada, sino que además pDI-Tools funciona con cualquier tipo de ejecutable. Esto es muy bueno ya que abre un abanico de posibilidades amplísimo a pDI-Tools para que en el futuro sea usado no sólo en entornos con máquinas paralelas. Por ejemplo podría ser usado también para realizar ejecuciones conducidas y así testear “software”.

Otro gran punto enriquecedor de este proyecto ha sido la gran cantidad de arquitecturas y sistemas operativos sobre los que he trabajado: desde la plataforma más común INTEL® 386 hasta carísimos PowerPC™ 64 o MIPS®. En total se ha portado pDI-Tools sobre siete ABI’s y tres sistemas operativos distintos, y se trabajó durante un corto periodo de tiempo sobre Itanium®[2]. Esto me ha permitido tener una visión muy clara y a la vez cuidadosa sobre como se implementa ELF (y algunos aspectos de COFF) en los sistemas Unix™ actuales.

Además, haber portado pDI-Tools de nuevo a la plataforma original de DITools (Irix para MIPS®) permite que se porte OMPItrace, una aplicación de instrumentación desarrollada por el CEPBA, a pDI-Tools. De esta forma en el futuro se podrá portar casi automáticamente OMPItrace a otras arquitecturas.

Como hemos ya comentado, hay muy poca información sobre estos temas, y se ha tenido que realizar una gran labor de investigación. De hecho, a parte de las especificaciones, no se han encontrado apenas artículos o documentos “serios” sobre ELF y se ha recurrido en muchas ocasiones a “ezines” de “hackers”, listas de correos, notas en algunas páginas web, foros y sobretodo código fuente y en ocasiones a desensamblar partes del sistema operativo *a mano*. Esto sin duda ha sido una de las partes más duras, pero también más interesante, del proyecto. Me he visto obligado en muchas ocasiones a agudizar el ingenio y ha hecho que el resultado sea muchísimo más satisfactorio.

Además ha sido una aplicación difícil de escribir ya que en este tipo de software no se pueden tolerar errores. En un programa como este es crítico revisar su correcto funcionamiento, ya que un mínimo fallo puede significar mediciones incorrectas en la instrumentación y en el peor de los casos, si aborta, puede representar muchas horas de trabajo y recursos perdidos. Por ello ha sido un desarrollo duro ya que se ha repasado cuidadosamente cada función para garantizar un buen funcionamiento.

Por otro lado este proyecto me ha permitido refinar mis conocimientos en interfaces de usuario ofreciendo en pDI-Tools una facilidad de uso superior al antiguo DITools. Se ha simplificado mucho la detección de errores o interacciones entre interposiciones, además de que se ha ocultado al máximo al usuario las limitaciones y características del "Runtime Linker" ofreciendole una interfaz sencilla y clara con la cual poder crear herramientas de instrumentación altamente portables.

Otro objetivo que tenía con pDI-Tools era la creación de un proyecto de "software" libre de principio a fin, es decir, no sólo entregar el código bajo una licencia libre (LGPL), sino un proyecto con una documentación libre en un formato libre, su sistema de configuración e instalación portable, su propia página web del proyecto y que esté presente en un repositorio público de "software" libre (http://savannah.gnu.org/projects/pditools/). Este objetivo me ha obligado a aprender muchisimo no sólo sobre tecnología, sino sobre el movimiento GNU, su filosofía y sobre el "open source" en general.

Por último he aprovechado el hecho de escribir esta documentación para ponerme al día sobre XML, SGML y otras tecnologías del W3C®. Para escribir la documentación se ha usado DocBook/XSL, un lenguaje XML especial para escribir libros. Es usado por la editorial O'Reilly® para todas sus publicaciones y permite escribir documentos sin centrarse en su presentación, sino tan sólo en su contenido. Esto me ha obligado a aprender muchísimo y crear un entorno de trabajo basado en DocBook/XSL con sus "scripts" de renderización de documentos y proceso de documentos XML, hojas de transformación XSLT y hojas de estilo DSSSL.

En último lugar, pero no precisamente es lo menos importante, es que este proyecto se ha compaginado a diario con mi trabajo de 5 horas diarias en una empresa de programación de herramientas de tratamiento de lenguaje por computador. Esto ha provocado que sólo se le haya podido dedicar unas 4 o 5 horas diarias de trabajo al proyecto, que además no podían ser muy intensas debido a que ya venía de una jornada de trabajo en otro sitio. Aún así estoy orgulloso del resultado por la cantidad y calidad del trabajo realizado teniendo en cuenta el poco tiempo del que disponía.

## 5.2. Trabajo futuro

pDI-Tools esta actualmente en un punto donde ofrece mucha más funcionalidad de la esperada. Inicialmente sólo se pedía un programa equivalente a DITools, pero este hito se superó hace tiempo y en estos momentos es mucho más fácil de usar, más portable y más potente. No obstante hay algunos aspectos que se podrían mejorar para dar mejores funcionalidades.

La funcionalidad más interesante sería añadir nuevos tipos de interposiciones que permitiesen instrumentar las llamadas dentro de un mismo DSO. Es decir, interceptar llamadas a funciones que no usan los métodos de enlace dinámico. Esto es muy complejo, y aunque creo que sería posible implementarlo sin introducir cambios estructurales en pDI-Tools, necesitariamos usar información de depuración muy dependiente del compilador y del sistema operativo. Pero sin duda esta modificación pondría a pDI-Tools al nivel de Dyninst.

Otra interesante modificación sería alterar un poco la sintaxis de los comandos de reenlace para obtener un lenguaje más flexible y próximo a una verdadera "shell". Actualmente sólo se pueden aplicar las interposiciones de una en una o todas de golpe sobre un conjunto de elementos con un comodín (*). En el futuro sería interesante poder usar "wildcards", expresiones regulares o conjuntos de elementos. Además podría ser interesante potenciar el lenguaje con condicionales y/o variables para poder realizar "scripts" de interposición de código más complejos.

Por último, sería interesante terminar de desligar la parte no-dependiente de la arquitectura del formato ELF y portar pDI-Tools a otros entornos distintos como Microsoft® Windows o Mac OS™ X (este último, aunque es Unix™, no usa el formato de reenlace ELF). Sobretodo sería interesante soportar Microsoft® Windows ya que entonces pDI-Tools estaría disponible en un mercado muy masivo, y por lo tanto más al alcance de muchos usuarios.

# Notas

1. Como anécdota, el sistema operativo Mac OS™ X implementa un ABI inadecuado en su sistema operativo, provocando que las llamadas a funciones de otros DSO desde el programa principal sean más lentas que si hubiese seguido la especificación ELF para PowerPC™. Esto puede llegar a tener, teoricamente, una ejecución hasta un 10% más lenta (ver [Mach-O-ABI] (http://www.unsanity.org/archives/000044.php)).
2. Pero al terminar la cesión por Bull de la máquina Itanium® que tenía el CEPBA se tuvo que dejar de lado esta arquitectura.

# III. Manual de usuario

## Introducción

En este manual se describe como usar e implementar software de instrumentación sobre pDI-Tools.

El manual explica los tres puntos básicos del uso diario de pDI-Tools: como configurarlo, como establecer las interposiciones de código y como crear los “backends” (librerías con el código que interpondremos), que debe contener y realizar el `runtime` y el por último se explica el API de pDI-Tools.

# Capítulo 6. Configuración de pDI-Tools

En este capítulo describiremos los mecanismos que ofrece pDI-Tools para ser configurado. Se hará especial hincapié en describir la configuración mediante ficheros de texto, novedad de pDI-Tools, que pretende sustituir la clásica configuración de DITools mediante variables de entorno o en tiempo de compilación.

## 6.1. Variables de entorno

Originalmente DITools se configuraba mediante variables de entorno. pDI-Tools mantiene este método por compatibilidad, aunque ofrece mecanismos de configuración mucho más potentes y mejores. Seguidamente describimos las variables de entorno a las que responde pDI-Tools:

DI_CFG_FILE

Esta variable de entorno hace de puente entre el antiguo sistema de configuración y el actual basado en ficheros. Sirve para especificar cual es el fichero de configuración a procesar. En caso de no estar definida se asume que el fichero de configuración se llama `pdi.cfg` y se busca en los siguientes directorios, por este orden:

- directorio actual
- `~/etc`
- `~/etc/pdi`
- `~/etc/pDI`
- `~/etc/pdi-tools`
- `~/etc/pDI-Tools`
- En el directorio de configuración de la instalación de pDI-Tools (por defecto *`PREFIX`*`/etc`).
- `/etc`
- `/etc/pdi`
- `/etc/pDI`
- `/etc/pdi-tools`
- `/etc/pDI-Tools`

DI_CONFIG_FILE

Añade un fichero de comandos a la lista de ficheros de comandos. Se añadirá al principio de todo de la lista, ya que se procesan siempre primero las variables de entorno y luego los ficheros de configuración.

DI_RUNTIME_FILE

Especifica el fichero de comandos `runtime` que se usará en esta sesión. Si se vuelve a definir el parámetro *`runtime`* en un fichero de configuración se producirá un error.

DI_FEEDBACK

Con esta variable se activa el máximo nivel de verbosidad de pDI-Tools en los logs. Es muy útil cuando se están depurando ficheros de comandos o el mismo pDI-Tools. Es equivalente a asignar el máximo nivel de verbosidad al parámetro *`verbose`* en un fichero de configuración.

Por compatibilidad sólo se tiene en cuenta su presencia y no su contenido. DITools sólo tenía dos niveles de verbosidad: ninguno o máximo dependiendo de la presencia o ausencia de esta variable de entorno.

DI_DEBUG

Con esta variable se activa el modo de depuración de pDI-Tools. Equivale a activar el parámetro *`debug`* en un fichero de configuración. Básicamente imprime mucha información adicional y realiza más comprobaciones de lo normal.

Cuando se activa el modo de depuración automáticamente se sube la verbosidad de pDI-Tools a nivel 3, el máximo. Se hace esto ya que toda la información de depuración se escribe sólo en nivel 3, y no tiene sentido activar el modo de depuración sin imprimir ningún mensaje.

DI_LOG_FILE

Esta variable de entorno, si existe y tiene definido un contenido, indica la ruta al fichero donde se dejará la información de "log" generada por pDI-Tools y los "backends". Esto incluye mensajes de error, advertencia e información de depuración a todos los niveles.

Por defecto toda la salida del sistema de "log" se lanza por la salida de error estándar (`stderr`).

DI_FOR_CHAPMAN

Esta variable se usaba para activar un "hack" en DITools. Actualmente no tiene ninguna función y si está definida provoca una advertencia durante la ejecución de pDI-Tools.

## 6.2. Ficheros de configuración de pDI-Tools

pDI-Tools incorpora la posibilidad de ser configurado mediante ficheros de texto. Este método no sólo facilita la configuración, sino que es mucho más fácil de controlar y ofrece mucha más potencia que las variables de entorno.

### 6.2.1. Como se estructura un fichero de configuración

Un fichero de configuración se estructura en secciones. La secciones son bloques dentro de un fichero de configuración. Estos bloques son indivisibles, y por lo tanto todas las secciones se encuentran al mismo nivel. Estas secciones permiten dividir el fichero de configuración en porciones que podemos activar, desactivar y reutilizar fácilmente.

En un fichero de configuración, dentro de las secciones, podemos encontrar los siguientes elementos:

- **Comentarios.** Nos permiten documentar las decisiones de configuración o desactivar temporalmente determinadas opciones.
- **Asignaciones.** Están limitadas a una sola línea en la que a un determinado parámetro le asignamos un valor. Los parámetros están predefinidos y es un error referirse a un parámetro que no existen. Los valores válidos dependen del tipo de parámetro.

Por regla general, ha de tenerse en cuenta que si se realiza más de una vez la misma asignación, el valor asignado será el de la última asignación procesada. Existen excepciones (como *`config`*) que pueden definirse varias veces, creándose una lista de valores.

- **Acciones.** Cuando se encuentra uno de estos elementos se ejecuta la acción correspondiente. Las acciones nunca reciben parámetros. Por ejemplo *`reset_config`* destruye la lista construída hasta el momento con el parámetro *`config`*.
- **Comandos.** Son palabras reservadas del lenguaje de ficheros de configuración. Permiten incluir ficheros de configuración dentro de un fichero de configuración, abortar la lectura o emitir mensajes.

#### 6.2.1.1. Sintaxis

Un comentario se reconoce porque el primer carácter de la línea que no sea un blanco es una almohadilla #.

Las secciones, tal como hemos explicado, dividen un fichero de configuración en diferentes bloques. Una sección comienza con el identificador de sección, encerrado entre corchetes, y termina con el comienzo de una nueva sección (con otro identificador de sección), o con el propio fin del fichero de configuración.

En todo fichero de configuración siempre existe la sección implícita `global`. Si no se indica, al comenzar un fichero, ningún nombre de sección, se asume que todo pertenece a esta sección. También es posible introducir contenidos en ella explicitamente abriendo una sección con su nombre.

Seguidamente mostramos un fichero de configuración de ejemplo, comentado, dividido en tres secciones: `global` (de forma implícita), `sección_de_ejemplo` y `otra_sección`:

```
# .. asignaciones y comandos de la sección 'global' ..

[seccion_de_ejemplo]
  # .. asignaciones y comandos de la sección 'seccion_de_ejemplo' ..

[otra_sección]
  # .. asignaciones y comandos de la sección 'otra_sección' ..
```

Un último detalle es que las secciones pueden estar partidas por otras secciones dentro de un mismo fichero, pero cuando se cargue el fichero se procesarán como un todo todos los trozos de la sección sin verse afectada por estas divisiones. Por ejemplo, supongamos que queremos recoger la configuración contenida en la sección `AAA` del siguiente fichero de configuración:

```
[AAA]
  # .. asignaciones y comandos de la sección 'AAA' ..

[BBB]
  # .. asignaciones y comandos de la sección 'BBB' ..

[AAA]
  # .. MAS asignaciones y comandos de la sección 'AAA' ..
```

El resultado final es que se procesará la primera división con el nombre `AAA`, se ignorará el contenido de la sección `BBB`, y cuando se vuelva a encontrar la sección `AAA` se seguirá procesando el fichero hasta el final.

Las asignaciones se componen de un parámetro y un valor separados por un signo de igualdad (=). Una asignación debe ocurrir en su totalidad dentro de una misma línea.

Ambos lados de una asignación pueden estar entre comillas dobles que delimiten sus campos. Los espacios a los lados del signo de igualdad se ignorarán y no formarán parte ni del parámetro ni del valor. Si se delimita el parámetro o el valor mediante comillas es posible incluir comillas escapándolas con una contrabarra (\). Ejemplo de asignaciones:

```
[asignaciones]
  debug = 1
  be_path = "/home/gerardo/backends:/usr/local/pdi-tools/backends"
  "mi parametro" = "Toma como valor este \"string\" entre comillas"
```

Las siguientes herramientas de configuración de que disponemos son los comandos. Actualmente hay los siguientes:

**Include** {*fichero* [:*sección*] | :*sección*}

Se sustituye por el contenido de la sección `global` o *`sección`* del fichero *`fichero`*. Si no se especifica el fichero se busca la sección en el fichero actual. Debe vigilarse con esta opción porque actualmente no hay ningún control de recursiones y ¡es posible entrar en bucles infinitos de inclusiones!

Si en *`sección`* se especifica el valor %PLATFORM% se incluye el contenido de la sección que tenga el nombre de la plataforma en uso.

**Log** {*`mensaje`*}

Imprime el mensaje *`mensaje`* en el “log” de ejecución de pDI-Tools.

**Warning** {*`mensaje`*}

Emite el mensaje de advertencia *`mensaje`* en el “log” de ejecución de pDI-Tools.

**Error** {*`mensaje`*}

Imprime el mensaje de error *`mensaje`* en el “log” y termina la ejecución de pDI-Tools.

Si ejecutásemos el siguiente ejemplo en una máquina con GNU/Linux™:

```
[global]
Include :%PLATFORM%

[common]
verbose=3

[linux-gnu]
include :common
runtime=linux.cfg
```

```
log "Se cargo la configuración para Linux."
...

[irix]
runtime=irix.cfg
warning "La configuración para Irix es muy precaria!"
...
```

Se cargaría la configuración de la sección `global`. No obstante se interrumpiría el proceso de esta sección para seguir procesando la sección `linux-gnu`. Esta a su vez interrumpiría su proceso para incluir la sección `common`. Una vez procesada la sección `common` se volvería a procesar el resto de la sección `linux-gnu`. Es en este momento cuando se imprime el mensaje “Se cargo la configuración para Linux”. Terminada de procesar esta sección se termina de procesar la sección `global` justo donde se dejó al comenzar con el primer **Include**.

#### 6.2.1.2. La sección `global`

Esta sección es un poco especial. En principio, si no se indica el nombre de ninguna sección y empiezan a aparecer asignaciones, se asume que estas pertenecen a esta sección. Así podemos tener un fichero como:

```
verbose = 2

[global]
debug = on

[otra_seccion]
# ...
```

Se interpretará que *`verbose`* y *`debug`* están la sección global.

La sección `global` (y sólo ella) es la sección que se procesa por defecto si no se ha indicado el nombre de ninguna sección. Así si alguien ejecuta el comando `Include "linux.cfg"` se procesará la sección `global` del fichero `linux.cfg` ignorando el resto de secciones.

#### 6.2.1.3. La sección `defaults`

Esta sección existe por convención y no porque tenga alguna característica especial. Se aconseja crearla en el fichero de configuración global de pDI-Tools e introducir en ella los parámetros por defecto más usuales para el sistema.

Si se separan los parámetros por defecto de esta forma es posible incluirlos fácilmente desde un fichero de configuración de usuario sin tener que copiar todos los parámetros a mano a nuestro fichero de configuración.

#### 6.2.1.4. La sección `runtime`

Al igual que la sección `defaults`, esta también existe por convención. Sirve para tener separada la asignación al atributo *`runtime`*. De esta forma es fácil poder incluir el `runtime` por defecto sin tener que copiarlo del fichero de configuración por defecto.

## 6.2.2. Configuración basada en varios ficheros

Mediante el comando **Include** es posible distribuir la configuración en diferentes ficheros de configuración, o al menos, en diferentes secciones dentro de un fichero de configuración.

Esto permite a los usuarios crear pequeños ficheros de configuración que importan partes de otros más grandes y de esta forma reciclar configuraciones. Además reciclar nos permite trabajar de una forma más ordenada y no tan propensa a errores.

## 6.2.3. Parámetros de funcionamiento de pDI-Tools

Seguidamente describimos y clasificamos los parámetros de funcionamiento que se pueden fijar en los ficheros de configuración.

Siempre que no se indique lo contrario, si se redefine un parámetro, el valor vigente del parámetro será el asignado en la última aparición del mismo.

### 6.2.3.1. Ficheros de log y depuración

pDI-Tools incorpora su propio sistema de logs y depuración. Así evita usar librerías de terceros, polucionar lo menos posible el entorno de ejecución del programa que monitoriza e interaccionar accidentalmente con otras librerías. Seguidamente comentamos los parámetros que controlan la verbosidad y mecanismos de depuración de pDI-Tools.

*`logfile`* - Fichero de log de pDI-Tools

Si se indica un fichero, se imprimirán en él todos los mensajes producidos durante la ejecución de pDI-Tools sin invadir para nada las salidas estándar (`stdout` y `stderr`).

Si no se indica el parámetro o se deja vacío, los mensajes de pDI-Tools se lanzarán a la salida de error estándar (`stderr`).

*`verbose`* - Nivel de verbosidad

pDI-Tools se puede configurar para que emita los mensajes según la importancia de los mismos. Este parámetro toma como valor un entero que indica el nivel de verbosidad. Cuanto mayor el valor, más detalle:

- *0* - Silencio total (excepto errores).
- *1* - Sólo se muestran advertencias y errores.
- *2* - Imprime todos los mensajes excepto los de depuración.
- *3* - Imprime toda la información de depuración disponible (ver seguidamente el parámetro *`debug`*).

*`debug`* - Modo de depuración

Si este flag está activo se realiza un montón de trabajo extra para verificar que las cosas andan bien. No se aconseja usarlo en producción, ya que a parte de lento, imprime demasiada información en el log (implica poner el parámetro *`verbose`* a 3).

Por defecto está desactivado.

#### 6.2.3.2. Límites

Los siguientes parámetros permiten variar ciertos límites establecidos durante la inicialización de pDI-Tools. pDI-Tools hace uso en muchas ocasiones de estructuras estáticas (pools) para ganar velocidad no reservando dinámicamente memoria. Con estos parámetros podemos ajustar el consumo de memoria de pDI-Tools sin sacrificar velocidad. También es posible desactivar algunos límites y hacer que se ejecuten algoritmos que reservan memoria dinámicamente.

`max_objects` - Número máximo de objetos en memoria

Número máximo de objetos que puede haber en memoria (entre el binario, librerías y los backends). Es conveniente que sea un número algo alto, pero tampoco demasiado grande ya que si no consumimos mucha memoria.

Por defecto es 40.

`max_threads` - Número máximo de “threads”

El parámetro `max_threads` fija la máxima cantidad de “threads” que se pueden monitorizar. Es conveniente que el número no sea demasiado bajo, ya que si se superase el número de “threads” abortaría la ejecución del programa. Este parámetro reserva una tabla de `max_threads` punteros.

Por defecto su valor es 100.


Sólo tiene efecto si se realizan interposiciones de código mediante el mecanismo de “callback”.

`num_threads` - Número de “threads” que se deben pre-inicializar

No puede tener un valor mayor que `max_threads`. Indica el número de “threads” que se espera usar. De esta forma la inicialización y reserva de memoria se hace al inicio del programa y no durante la ejecución, evitando que la instrumentación tenga mucho impacto en la ejecución del programa.

Se pueden introducir los siguientes valores:

- *-1* - Inicializa todos los “threads” a la vez (en total `max_threads`) de golpe durante la inicialización. Esto implica el peor caso en consumo de memoria.
- *0* - No inicializamos ningún “thread”. Se irán inicializando según se necesiten.
- `+x` - Inicializa *x* “threads”. Si se supera el número el resto se van inicializando durante la ejecución del programa hasta un máximo de `max_threads`.

Por defecto vale 0.


Sólo tiene efecto si se realizan interposiciones de código mediante el mecanismo de “callback”.

`cb_max_stubs` - Cantidad máxima de “stubs” que pueden haber en memoria

Indica cuantos “stubs” como máximo habrá en memoria. Es importante considerar que si vamos a poner un “callback” sobre un objeto que llama a X funciones diferentes, es conveniente que este parámetro tome un valor mayor que X o si no será imposible instalar el “callback”.

El valor por defecto varía según la plataforma ya que se calcula con la función `_pdi_arch_defaultMaxStubs()`. Esta función devuelve, generalmente, el tamaño de página de la arquitectura dividida por el tamaño de un "stub". Por ejemplo, en INTEL® 386 esta función asigna 204 como valor por defecto a *`cb_max_stubs`* (4096 bytes por página y 20 bytes por "stub").


Sólo tiene efecto si se realizan interposiciones de código mediante el mecanismo de "callback".

*`cb_stack_size`* - Tamaño de la pila por "thread" para las interposiciones mediante "callback"

Al monitorizar los "threads" necesitamos una zona de memoria donde guardar información sobre que se está monitorizando en un determinado momento. Evidentemente, cada "thread" ha de tener su espacio separado. Esto se realiza mediante una pila cuyo tamaño viene definido por esta variable. Si durante una monitorización hay problemas ("Segmentation Fault"), pruebe a incrementar esta variable. Si con valores grandes sigue teniendo problemas quizás debería empezar a sospechar que hay una monitorización recursiva.

Por defecto vale 1024.


Sólo tiene efecto si se realizan interposiciones de código mediante el mecanismo de "callback".

### 6.2.3.3. Backends y runtime

Mediante los siguientes parámetros se construye la lista de ficheros de comandos que irán cargando los "backends" y realizando las interposiciones de código.

Estos parámetros no se comportan igual que el resto cuando se repiten. En el caso del parámetro *`runtime`*, se emite un error y se termina la ejecución si se define más de una vez. En el caso del parámetro *`config`*, cada vez que aparece se añade el valor que le acompaña al final de una lista.

Cuando se terminan de procesar todos los ficheros de configuración, se antepone el valor del parámetro *`runtime`* a la lista de ficheros de comandos formada por los parámetros *`config`*.

Existen también acciones (atributos sin valor) que pueden reiniciar los valores asignados a *`runtime`* y a *`config`*.

Seguidamente enumeramos y explicamos estos atributos.

*`runtime`* - Fichero de configuración de sistema

Es un fichero de comandos de reenlace pero a nivel de sistema. Es el primero en cargar de todos los ficheros de configuración, y además es único. Sólo puede haber un fichero de configuración del runtime. Generalmente sirve para cargar backends sobre funciones críticas del sistema como fork(2) o exec(2).

*`config`* - Fichero de comandos de reenlace

Indica un fichero de comandos de reenlace. Puede indicarse este parámetro tantas veces como ficheros de comandos de reenlace se tengan.

*reset_runtime* - Olvida el runtime configurado actualmente

A este parámetro no se le puede asignar ningún valor. Cuando se encuentra provoca que se olvide el valor asignado al parámetro *runtime*, pudiéndolo volver a definir si se desea.

*reset_config* - Olvida la lista de ficheros de comandos configurada actualmente

A este parámetro no se le puede asignar ningún valor. Cuando se encuentra provoca que se olvide la lista de ficheros de comandos configurada hasta el momento con el parámetro *config*.

### 6.2.3.4. Rutas de búsqueda

Estos parámetros establecen rutas de búsqueda. Sirven para poder especificar los backends y ficheros de comandos sólo mediante su nombre, pDI-Tools con ayuda de estos parámetros ya tratará de resolver la ubicación de los archivos.

*be_path* - Rutas de búsqueda de los backends

Aquí indicamos los lugares donde podemos encontrar los backends. La sintaxis es sencilla: directorios separados por dos puntos (:).

Si se especifica varias veces este parámetro, el resultado final es como haber concatenado uno tras otro (separados por dos puntos) los distintos valores asignados.

Por defecto *be_path* contiene el directorio al fichero de runtime y al directorio de los "backends", ambos configurados por Autoconf (generalmente estos directorios son *PREFIX*/backends y *PREFIX*/runtime)

*becfg_path* - Rutas de búsqueda de los ficheros de comandos

Aquí indicamos los lugares donde podemos encontrar los ficheros de comandos de reenlace. La sintaxis y uso es igual que en el parámetro *be_path* acabado de comentar.

Por defecto *becfg_path* contiene el directorio a los ficheros de comandos de backends y al directorio de configuración de pDI-Tools, ambos directorios configurados por Autoconf (generalmente son *PREFIX*/becfg y *PREFIX*/etc).

*lib_path* - Rutas de búsqueda de las librerías y DSO (objetos compartidos).

Este parámetro indica donde se deben buscar las librerías y DSO que se van a usar. Se inicializa con el contenido de la variable de entorno LD_LIBRARY_PATH y con los directorios /lib y /usr/lib.

Si se asigna el valor especial %LD_LIBRARY_PATH% se sustituirá por el valor de la variable de entorno LD_LIBRARY_PATH.

En el resto la sintaxis y uso es igual que en los anteriores parámetros.

*reset_be_path* - Olvida las rutas de búsqueda de los backends

A este parámetro no se le puede asignar ningún valor. Cuando se encuentra provoca que se olvide el valor asignado a los parámetros *be_path*.

*reset_becfg_path* - Olvida las rutas de búsqueda de los ficheros de comandos de reenlace

A este parámetro no se le puede asignar ningún valor. Cuando se encuentra provoca que se olvide el valor asignado a los parámetros *becfg_path*.

`reset_lib_path` - Olvida las rutas de búsqueda de las librerías y objetos compartidos

A este parámetro no se asigna ningún valor. Cuando se encuentra provoca que se olvide el valor asignado a los parámetros `lib_path`.

### 6.2.3.5. Otros parámetros

Por último explicamos algunos parámetros que no se clasifican en ninguna de las anteriores categorías.

`allow_lib_as_be` - Cualquier función de cualquier DSO puede tomar el control en una interposición de código

En principio está prohibido redigirir una llamada desde un DSO a otro DSO que no sea un "backend". No obstante es posible que en algún caso pueda ser interesante relajar esta restricción y esto es para lo que sirve este parámetro si se activa.

Por defecto esta opción está desactivada.

Aún así, si esta opción está activada, pDI-Tools seguirá emitiendo un mensaje de aviso cada vez que se cree un reenlace hacia una función que no pertenezca a un "backend".

`donttouch_backends` - No instrumentar los backends

Si está activada esta opción, no se podrá instrumentar sobre los "backends", o lo que es lo mismo, no se podrán hacer reenlaces sobre funciones de los "backends". Es aconsejable tener activada esta opción.

Por defecto esta opción está activada.

Si esta opción está activada los comodines no engloban a los backend, sino sólo a los objetos regulares (y a `libpdi.so` si la opción `donttouch_pdi` está desactivada).

`donttouch_pdi` - No instrumentar a pDI-Tools

Si esta opción está desactivada se podrá instrumentar el DSO `libpdi.so`. Evidentemente se desaconseja seriamente desactivar esta opción a no ser que se desee, por algún motivo, que pDI-Tools se autoinstrumente.

Por defecto esta opción está activada.

`cb_allow_handler` - Permitir usar un "callback handler" alternativo al definir un "callback"

Parámetro booleano. Si está activado se permite usar un "callback handler" propio. Si está desactivado, pDI-Tools obliga a tener vacio o poner a `NULL` el cuarto parámetro de las reglas "callback".

Por defecto esta opción está desactivada.

`no_check_on_config` - No comprueba los ficheros durante la creación de un "script" de reenlace.

Parámetro booleano. No se chequea si un objeto existe mientras se está creando una configuración de interposiciones ("script" de reenlace). Si este "flag" está activo se detectará si el objeto existe o no durante la instalación de las interposiciones.

Por defecto este “flag” está desactivado.

## 6.3. Ejemplo completo

Seguidamente mostramos dos ficheros de configuración, uno global para todo el sistema y uno de usuario y explicamos posteriormente como interaccionan entre ambos para dar la configuración final.

Ambos ficheros se llaman `pdi.cfg`, pero el de sistema (global) reside en el directorio `/usr/local/pditools/etc` (pDI-Tools se encuentra instalado en el directorio `/usr/local/pditools`) y el de usuario en el directorio actual.

El fichero de configuración global del sistema es el siguiente:

```
##############################################################################
# FICHERO DE CONFIGURACIÓN GLOBAL DEL SISTEMA
##############################################################################

##############################################################################
# SECCIÓN GLOBAL
#
#   Esta sección no debería de usarse directamente, así que emitimos un warning
#   avisando de ello y cargamos la configuración por defecto.
#

[global]
Warning Se está usando el fichero de configuración del sistema directamente.
Include ":defaults"
Include ":runtime"


##############################################################################
# SECCIÓN DEFAULTS
#
#   En esta sección se ponen los valores por defecto para todo el sistema.
#

[defaults]
# Desactivamos el modo de depuración
debug = off

# No dejamos instrumentar los backends y a pDI-Tools
donttouch_backends = on
donttouch_pdi = on

# Asumimos que una aplicación no tendrá nunca más de 50 objetos en memoria.
max_objects = 50

# Suponemos que no habrán nunca más de 32 threads simultaneos y se inicializan
# todos al iniciarse pDI-Tools.
max_threads = 32
num_threads = 32

# Por si acaso instrumentamos un DSO con muchas funciones creamos una pool
# grande de stubs para callback
```

```
cb_max_stubs = 1000

# No suelen haber instrumentaciones reentrantes, asi que tomamos una pila para
# los threads pequeña
cb_stack_size = 512

# Ponemos las rutas de búsqueda de los backends y sus ficheros de comandos
# NOTA: Se añaden a las rutas de búsqueda por defecto
be_path    = ~/lib/backends:~/backends:~/lib:./
becfg_path = ~/etc/:~/becfg:./


###############################################################################
# SECCIÓN RUNTIME
#
#   Aquí incluimos el backend de runtime que gestionará algunas funciones
#   críticas.
#

[runtime]
runtime = /usr/local/pditools/lib/runtime.so

# FIN DEL FICHERO DE CONFIGURACIÓN GLOBAL DEL SISTEMA
```

Podemos observar que el fichero de configuración global se divide en tres secciones: la sección `global`, la sección `defaults` y la sección `runtime`.

La sección `global` se usa explicitamente y sólo contiene tres comandos. Lo primero que hace es advertir de que se está usando la sección global de este fichero y luego incluye el contenido de las otras dos secciones.

Se da esta advertencia ya que no recomendamos usar esta configuración directamente, sino a traves de otros ficheros de configuración. Se supone que no cargarán directamente la sección global para evitar la aparición de este “warning”. Cuando pDI-Tools no encuentra ningún fichero de configuración usa el fichero de configuración del sistema y comienza a ejecutarlo, como cualquier otro fichero, por su sección global. Con esta advertencia avisa al usuario de que está usando la configuración por defecto ya que probablemente éste ha olvidado establecer su propia configuración.

Mientras que el fichero de configuración privado del usuario es el siguiente:

```
###############################################################################
# CONFIGURACIÓN PRIVADA DEL USUARIO
###############################################################################

# Configuración global de este fichero
[global]
verbose = 3
Log Vamos a procesar la configuración por defecto
Include "/usr/local/pditools/etc/pdi.cfg:defaults"
Include "/usr/local/pditools/etc/pdi.cfg:runtime"
Log Vamos a procesar la configuración de plataforma
Include ":%PLATFORM%"

# Configuración específica para plataformas linux
[linux-gnu]
Log Estamos en un sistema Linux
config = linux.cfg
```

```
debug = off

# Configuración específica para plataformas Solaris 8
[solaris2.8]
Log Estamos en un sistema Solaris
config = solaris.cfg
debug = on

# Configuración específica para Irix
[irix6.5]
Log Estamos en un sistema Irix
config = irix.cfg
debug = on

# FIN DEL FICHERO DE CONFIGURACIÓN PRIVADA DEL USUARIO
```

Este fichero también está dividido en secciones, pero de una forma bastante diferente. Comienza al igual que el anterior con la sección `global`. Esta sección, sólo comenzar sube la verbosidad al máximo. De esta forma podremos ver los mensajes de "log" que imprimirán los ficheros de configuración. Seguidamente se encarga de traer la configuración contenida en las secciones `defaults` y `runtime` de la configuración global del sistema. Una vez procesadas esas secciones del fichero global, se procesa una de las secciones específicas para la plataforma de este fichero. Cada una de estas secciones usa un fichero de comandos de reenlace distinto según la plataforma en la que estamos, y si es Irix o Solaris se activa el sistema de depuración de pDI-Tools.

Suponiendo que usamos estos ficheros de configuración en un sistema GNU/Linux™ con un kernel 2.6.8 y GNU C Library 2.3.2, tendríamos una salida como la siguiente (mostramos tan sólo los mensajes generados por el sistema de configuración y no la salida completa de una instrumentación):

```
ger@pandero:~/test$ LD_PRELOAD=/usr/local/pditools/libpdi.so ./mytest
pdiconfig.c:/home/ger/test/pdi.cfg:31:Vamos a procesar la configuración por def
ecto.
pdiconfig.c:/home/ger/test/pdi.cfg:33:Vamos a procesar la configuración de plat
aforma.
pdiconfig.c:/home/ger/test/pdi.cfg:40:Estamos en un sistema Linux.
------------------------------------------------------------------------
pDI-Tools version 0.1.0, Copyright (C) 2004, 2005 Gerardo García Peña
pDI-Tools comes with ABSOLUTELY NO WARRANTY; for details read the
'COPYING' file that comes with this library. This is free software,
and you are welcome to redistribute it under certain conditions;
read the 'COPYING' file for details.
------------------------------------------------------------------------
init.c:initLiblink:Se ha cargado la configuración de pDI-Tools.
ger@pandero:~/test$
```

La línea que ejecutamos contiene dos comandos en una: el primero establece la variable de entorno LD_PRELOAD que fuerza la carga de pDI-Tools y el segundo ejecuta el programa.

Obsérvese que los mensajes de los ficheros de configuración aparecen antes de la nota de copyright de pDI-Tools. Esto es debido a que esta nota de copyright se escribe tambien con el mecanismo de "logs" estándar, y este a su vez no termina de estar completamente inicializado hasta que se termina de procesar los ficheros de configuración.

A partir de la nota de copyright pDI-Tools ya ha procesado toda la configuración, y de hecho informa de ello una vez que la ha impreso mediante un mensaje de “log”. Llegados a este punto no se vuelven a consultar ni las variables de entorno ni los ficheros de configuración.

# Capítulo 7. Backends

La función final de pDI-Tools es interceptar llamadas a funciones entre objetos en memoria (por ejemplo el ejecutable y las librerías que usa) y redireccionarlas hacia el código de instrumentación. El código de instrumentación está siempre contenido en unos objetos llamados “backends”.

Un “backend” es un objeto compartido dinámicamente, o también llamado DSO (“Dynamic Shared Object”) o DLL (“Dynamic Link Library”). Estos objetos de por si no indican que código y como se ha de interceptar. Estos ficheros tan sólo contienen el código responsable de manejar las intercepciones y que será de hecho el código que se interpone.

Este capítulo mostramos la sintaxis y funcionamiento de los ficheros de comandos y como se resuelven las dependencias entre “backends” que aparecen en ellos. Seguidamente se describe como gestionar los distintos tipos de interposiciones y se finaliza explicando como se estructura un “backend” generalmente.

## 7.1. Ficheros de comandos

Los ficheros de comandos indican como deben realizarse las interposiciones de código y sobre que DSO deben realizarse. Son ficheros de texto que contienen básicamente reglas en las que explican los “backend” que intervienen y como intervienen sobre los DSO.

### 7.1.1. Sintaxis

Un fichero de comandos se divide en dos partes:

- **Lista de objetos y “backend”.** Al principio de un fichero de comandos se puede indicar todos los DSO (y “backend”) que intervendrán en los comandos de la siguiente sección.

  Declarándolos previamente se obtiene una serie de ventajas: de entrada, concentrando dicha lista al principio del fichero se puede saber a primera vista que DSO intervienen en un fichero de comandos.

  Otra gran ventaja es que se debe indicar si el fichero es un DSO normal o un “backend”. De esta forma pDI-Tools puede detectar reglas equivocadas debido a descuidos o distracciones.

  Por último, se permite asociar un alias a cada objeto (ya sea un DSO normal o un “backend”) y así hacer luego los comandos de interposición de código más legibles y más fáciles de modificar. Por ejemplo, en GNU/Linux™ con GNU C Library 2.5.x la librería matemática suele estar en `/lib/libm.so.6`, mientras que en las últimas versiones está en `/lib/tls/libm.so.6`. Trabajando con un alias modificar todas las reglas que afectan a esta librería es tan sencillo como alterar la línea donde se define.

  Hay tres alias predefinidos: `LIBC`, `MAIN` y `PDI`. El primero se asigna a la librería de C (por ejemplo a `/lib/libc.so` si fuese el caso), la siguiente se asigna al DSO del programa en ejecución y el último a la librería que implementa pDI-Tools. Estos alias se corresponden con las constantes `PDI_ALIAS_LIBC`, `PDI_ALIAS_MAIN` y `PDI_ALIAS_PDI` definidas en el fichero de cabecera `ebeif.h` de pDI-Tools.
- **Comandos.** En esta sección simplemente se encuentran, uno tras otro, los comandos de interposición de código. Se pueden usar en ellos los alias creados en la sección anterior.

La sintaxis de un fichero de comandos es muy sencilla. El elemento más simple de un fichero de comandos es el comentario: cualquier línea que comienza por un punto y coma (;) o blancos seguidos de un punto y coma (;) es ignorada hasta el próximo salto de línea.

Como ya hemos dicho, un fichero de comandos comienza por la lista de objetos y "backend". En esta sección, cada línea no vacía ni con un comentario, representa un DSO normal o un "backend". La sintaxis es la siguiente:

[#backend | #object | #define] {*path_dso*} [*alias*]

Obsérvese que la línea puede comenzar por `#backend`, `#define`, `#object` o directamente con el path a un DSO. El significado de cada opción es el siguiente:

- **`#backend`.** Se va a listar un "backend".
- **`#object`.** Indica que se va a listar, y seguramente asignarle un alias a, un DSO que no es un "backend" y que vamos a instrumentar. Se mantiene por compatibilidad con el antiguo DITools la directiva `#define`.
- Si una la línea comienza directamente por la ruta de acceso al DSO se asume que es un DSO normal que no es un "backend". Esta sintaxis se mantiene por compatibilidad con el antiguo DITools.

La ruta al DSO puede escribirse directamente o encerrarla entre dobles comillas si fuese el caso de que ésta contuviese espacios.

Después de la ruta al DSO podemos definir, opcionalmente, un alias al objeto o "backend". Este no puede contener espacios.

Esta sección se acaba, dando comienzo a la sección de comandos mediante las marcas `#commands` o `#relinks`.

En esta sección cada línea no vacía ni comentario representa un comando que explica como debe instalarse una interposición de código. La sintaxis de cada línea varía según el tipo de interposición. En las siguientes secciones comentamos la sintaxis de cada una de las interposiciones.

### 7.1.1.1. Sintaxis de los reenlaces

La sintaxis de un comando de reenlace es la siguiente:

{F | R} {*trg_obj*} [ {*trg_func*} {*backend*} {*be_func*} ]

El primer parámetro es una letra y puede ser `F` o `R`. No hay diferencia entre usar una u otra ya que ambas indican que este comando instalará un reenlace. Se mantienen dos opciones por compatibilidad con DITools.

El siguiente par de parámetros (*`trg_obj`* y *`trg_func`*) indican que llamadas a una función, desde un determinado objeto, se deben interceptar.

El último par de parámetros (*`backend`* y *`be_func`*) indican la función del "backend" que recibirá el control cuando se ejecute la llamada a la función el control.

Es posible asignarle un comodín al parámetro *`trg_obj`*. Asignándole a dicho parámetro un asterisco se instalará el reenlace sobre todos los DSO instrumentables (ver los parámetros de configuración de pDI-Tools: *`donttouch_backends`* y *`donttouch_pdi`*).

#### 7.1.1.2. Sintaxis de las redefiniciones

La sintaxis de un comando de redefinición es la siguiente:

```
D {trg_obj} {trg_func} {backend} {be_func}
```

La primera letra indica que se va a instalar una redefinición.

El siguiente par de parámetros (*trg_obj* y *trg_func*) indican la función que redefinimos en el objeto DSO. Evidentemente las redefiniciones no admiten ni pueden admitir comodines en ninguno de los dos parámetros.

El último par de parámetros (*backend* y *be_func*) indican la función del "backend" que recibirá el control cuando se ejecute la llamada a la función redefinida.

#### 7.1.1.3. Sintaxis de los "callback"

Las dos posibles sintaxis de un comando de "callback" son las siguientes:

```
{ [C] [F] [R] } {trg_obj} * {backend} [be_func]
```

En el caso de indicarse en el primer parámetro la letra `C` se deja claro que la intención es instalar un "callback". En cambio se ofrece, por compatibilidad con el antiguo DITools, la posibilidad de usar las letras `F` o `R`. pDI-Tools deducirá del asterisco en el tercer parámetro (función interceptada) que se desea instalar en realidad un "callback".

El siguiente parámetro *trg_obj* indica el objeto del cual interceptaremos todas las llamadas. Este parámetro va seguido siempre de un asterisco, que indica que se instalará el "callback" sobre todas las funciones del objeto[1].

El parámetro *backend* indica el "backend" que gestionará el "callback".

También está el parámetro *be_func*, que sólo se puede usar si la opción de configuración *cb_allow_handler* está activada. Si se indica este parámetro se usará un "callback handler" alternativo en lugar del que viene de serie con pDI-Tools. Generalmente se indica `NULL` o ningún valor.

### 7.1.2. Resolución de dependencias entre ficheros de comandos

Como se ha visto hasta ahora, con las opciones de configuración *runtime* y *config* es posible indicar más de dos ficheros de comandos. Estos ficheros de comandos establecen un orden de carga de los "backend" y de aplicación de los comandos que contienen. Como a veces un determinado "backend" requiere que se inicialize antes otro determinado "backend" esta lista de ficheros de configuración se ha de procesar de manera adecuada para respetar las ficheros.

El problema aparece cuando en varios ficheros de comandos se establecen distintos ordenes. pDI-Tools al recoger todos los ordenes de estos ficheros de comandos debe finalmente establecer un orden único de carga en el cual se respeten dichas dependencias. Así si por ejemplo, tenemos la lista de "backend" `A.so`, `B.so` y `C.so`, y el fichero de comandos `L1.CFG`:

```
; Fichero L1.CFG
; lista de backends y objetos
#backend A.so A
#backend B.so B

  ; .. otros backends y objetos ..

#commands
  ; .. comandos ..
```

y el fichero de comandos `L2.CFG`:

```
; Fichero L2.CFG
; lista de backends y objetos
#backend B.so B
#backend C.so C

  ; .. otros backends y objetos ..

#commands
  ; .. comandos ..
```

pDI-Tools deduce de aquí el siguiente grafo de dependencias:

- El “backend” `B.so` debe inicializarse después de `A.so`, por lo tanto B depende de A.
- Igualmente “backend” `C.so` debe inicializarse después de `B.so`, por lo tanto C depende de B.

Esto nos da a entender que los ficheros de comandos como mucho establecen grafos de dependencia lineales que posteriormente debemos combinar en un único grafo del que debemos deducir un orden de carga.

Siguiendo con el ejemplo, si combinamos ambos grafos y expresamos las dependencias como una flecha desde el objeto al objeto del que depende obtenemos el siguiente grafo:

$$A \longleftarrow B \longleftarrow C$$

De este grafo se puede deducir un único orden de carga de los “backend” en el cual se respetan las dependencias: si recorremos el grafo comenzando por un nodo que no depende de otro nodo obtenemos el orden: A, B, C. No obstante no podemos fiarnos de que siempre el orden sea único ya que no siempre se obtienen grafos lineales. Por ejemplo, supongamos dos ficheros de configuración que generan estos dos grafos:

- $A \longleftarrow B$
- $A \longleftarrow C$

al unirlos obtenemos el siguiente grafo:

$$B \longrightarrow A \longleftarrow C$$

del cual surgen dos posibles ordenes, ambos correctos:

- A, B, C

- A, C, B

Como se puede observar, si dos nodos están a la misma profundidad respeto a un nodo padre de ambos, estos se pueden cargar en distinto orden. Esto también afecta a nodos que no tienen padre. Por ejemplo la unión de:

- B ⟵ A
- C ⟵ A

genera este grafo de dependencias:

B ⟵ A ⟶ C

Los dos posibles ordenes que surgen, son:

- B, C, A
- C, B, A

La pregunta inmediata: ¿que pasa si surge un bucle? Por ejemplo:

- A ⟵ B ⟵ C
- B ⟵ C ⟵ A

Generaría un grafo con la siguiente forma:

C ⟵ A ⟵ B ⟵ C

Obsérvese que en este caso no hay ningún nodo que no dependa de otro: no sería posible elegir un primer fichero a cargar y la ejecución de pDI-Tools terminaría ipso-facto. Por regla generalmente es difícil encontrar dependencias circulares entre ellos, aunque no es imposible.

## 7.2. Tipos de interposiciones

pDI-Tools incorpora tres mecanismos distintos de interposición de código. Los dos principales son los más sencillos de usar y están disponibles en todas las plataformas: los reenlaces y las redefiniciones. El otro mecanismo son los “callback” que permiten instrumentar una función de una forma totalmente invisible (ver motivos en la sección *callback*. En contrapartida es el mecanismo menos portable de los tres y en ocasiones es bastante complejo de usar.

En las secciones siguientes se comenta y explica como se implementa y funciona cada uno de ellos, dando también ejemplos de su uso.

### 7.2.1. reenlaces

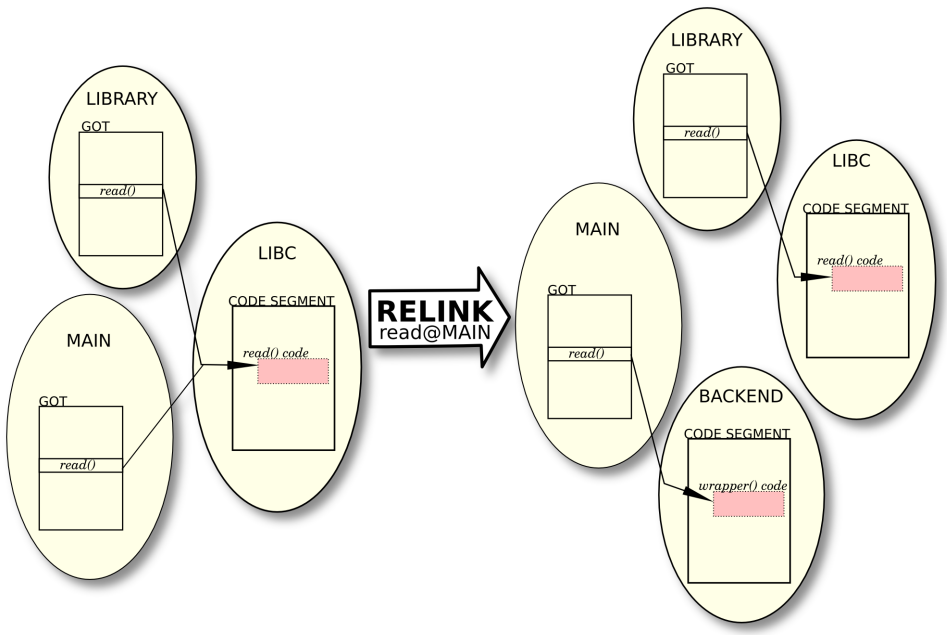
Un reenlace o “relink” es la interposición de código más simple que se puede realizar, y a la vez la más fácil de controlar y que menos problemas causa.

Para especificar un reenlace se necesitan los siguientes parámetros:

- *DSO objeto*. Es el objeto sobre el que se instala el “relink”. Generalmente se le llama “target object”.

- *Función objeto*. Se interceptarán las llamadas desde el objeto “target object” a esta función. Suele recibir el nombre de “target function”.
- El “backend” que gestionará el reenlace.
- La función de dicho “backend” que tomará el control.

Un reenlace consiste en sustituir la dirección la función “target function” en el GOT del objeto “target object”, por la dirección del “wrapper” del “backend”. De esta forma cada vez que el “target object” haga una llamada a la “target function”, se transferirá el control al “backend”. Evidentemente ningún otro DSO se verá afectado por esta operación ya que sólo se modifica la entrada del GOT que afecta al “target object”.




En este esquema vemos como se instala un reenlace en la aplicación, exactamente sobre el programa principal, sin afectar al resto de objetos en memoria.

## 7.2.2. redefiniciones

Las redefiniciones se usan y configuran de un modo muy similar a los reenlaces. Sin embargo su semántica y efectos son muy distintos.

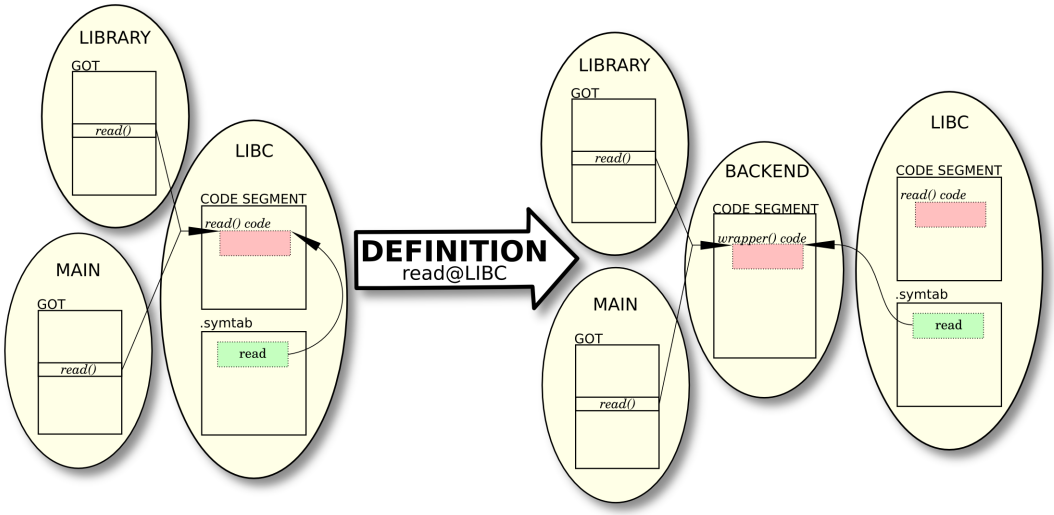
La idea de un reenlace es introducir una única interposición de código. En cambio una redefinición sustituye una función por otra.

Para especificar un reenlace se necesitan los siguientes parámetros:

- *DSO objeto*. El objeto que contiene la implementación de la función que queremos redefinir. Este objeto recibe el nombre de “target object”.
- *Función a redefinir*. Nombre de la función que se redefine. Recibe el nombre de “target function”.
- El “backend” que contiene la nueva implementación de la función a redefinir.
- El nombre de la función del “backend” que usaremos.

La redefinición consiste en modificar en la tabla de símbolos de “target object” la entrada de la función a redefinir. En dicha entrada indicamos la dirección de nuestra implementación de la función. A partir

de ahora, cada vez que un objeto use el símbolo redefinido en realidad se dará el control a la nueva definición.




En este esquema vemos como se sustituye la función read(2) de `libc.so` por nuestra versión de la función. Como se puede ver, esta sustitución afecta a todos los DSO de la aplicación.

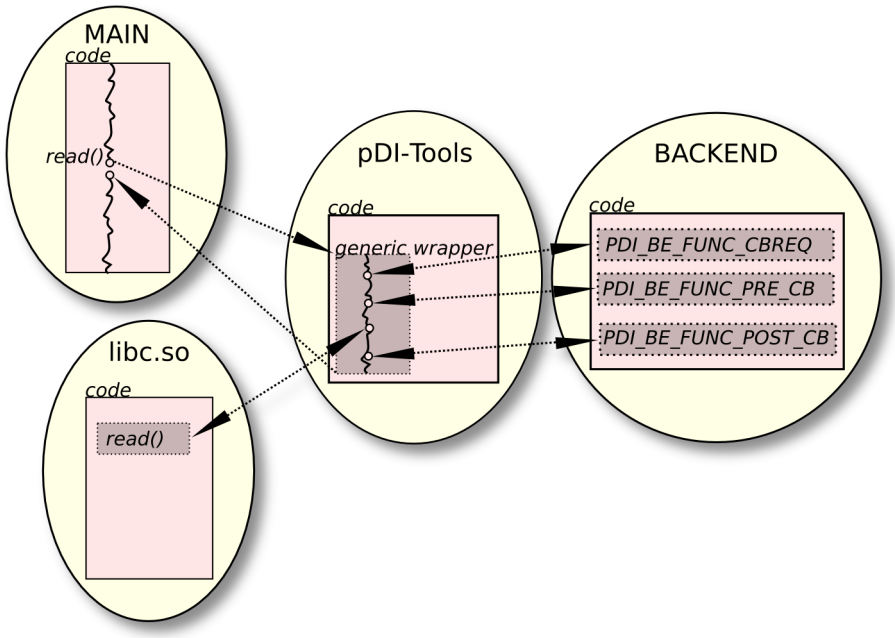
Si el parámetro de configuración *`donttouch_backends`* está activado (por defecto sí), la redefinición no afectará a los “backend”. En caso de activarse esta opción se debe trabajar con mucho cuidado ya que es fácil que se generen intercepciones recursivas. Por último, pDI-Tools sólo se verá afectado por las redefiniciones en caso de desactivarse el parámetro *`donttouch_pdi`*.

## 7.2.3. “callback”

El “callback”, al contrario que los reenlaces y las redefiniciones, desde el punto de vista de un DSO no actúa sobre una única función sino sobre todas. Un “callback” permite instrumentar todas las llamadas que realiza un DSO a otros DSO.

Con los reenlaces o las redefiniciones, durante la ejecución del programa instrumentado, sólo se ejecuta código de los “backend” ya que pDI-Tools se ha encargado de redirigir el control hacia el código de los “backend”. En el caso del “callback” interviene tanto código del “backend” como de pDI-Tools y ambos cooperan para realizar la interposición. En estas interposiciones quien recibe primero el control es la rutina “callback handler” (también llamada en ocasiones “generic wrapper”). Éste es un código que forma parte de pDI-Tools y se encarga de gestionar la intercepción. Este código exige al “backend” tener implementadas dos o tres funciones. La primera de todas, `PDI_BE_FUNC_CB_REQ`, debe decidir si la intercepción que ha realizado pDI-Tools interesa o no. Y las otras dos, `PDI_BE_FUNC_PRE_CB` y `PDI_BE_FUNC_POST_CB`, aportan el código a interponer antes y después de la llamada.

En el siguiente gráfico mostramos como cooperan entre si pDI-Tools y el “backend” para ejecutar la interposición de código.




En este diagrama se observa como al llamar a la función printf(2) desde `MAIN` en realidad se transfiere el control al "generic wrapper". Éste se encarga de transferir el control a las funciones de control de "callback" del "backend" y al verdadero código de printf(2).

El "callback handler" es una función, también llamada "generic wrapper", encargada de llamar a la función interceptada y a las funciones del "backend" encargadas de gestionar el "callback": `PDI_BE_FUNC_CB_REQ`, `PDI_BE_FUNC_PRE_CB` y `PDI_BE_FUNC_POST_CB`.

La función "callback handler" está escrita en ensamblador y es muy compleja y específica para la arquitectura. Es la encargada de realizar una interposición de tal forma que no haga falta copiar los parámetros de nuevo para poder ejecutar el código de la función interceptada, especialmente porque no se conocen. Esto nos obliga a salvar el estado de la pila y los registros. Salvando esta información es posible ejecutar el código pre-interpuesto, luego restaurar el estado inicial, y acto seguido saltar a la función interceptada. Al restaurar el estado exacto la función podrá recoger todos sus parámetros (sean por pila o registros) sin verse afectada por nuestras operaciones.

Al terminar la ejecución de la función el "callback handler" retomará el control, salvará el estado de los registros y la pila, ejecutará el código post-interpuesto, finalmente restaurará el estado final y retomará la ejecución donde tendría que haber vuelto la función.

Observemos que salvamos el estado de la aplicación antes de ejecutar el código interpuesto por las funciones `PDI_BE_FUNC_PRE_CB` y `PDI_BE_FUNC_PRE_CB`. Esto implica que estas funciones no pueden modificar los parámetros de entrada ni el valor retornado por la función.

Por último queda explicar la relación entre el mecanismo de "callback" y algunos parámetros de configuración que seguidamente enumeramos.

Antes de usar el mecanismo de "callback" debemos tener claro cuantos "threads" se usarán. Por cada "thread" el mecanismo de "callback" necesita una pila y estructuras de información sobre el "thread". Debido a ello debemos inicializar los parámetros *`max_threads`* y *`num_threads`*. El primero indica cuantos "threads" podemos instrumentar como mucho, mientras que el segundo indica cuantos "threads" esperamos usar.

La idea es que el parámetro *`max_threads`* se prepara para manejar un número máximo de "threads", mientras que el segundo, *`num_threads`* decide cuantos "threads" pre-inicializamos antes de comen-

zar la instrumentación. Si no se tiene claro cuantos “threads” se van a usar se aconseja un número alto en `max_threads` y 0 en `num_threads`.

Cada “thread” (una vez inicializado) tendrá asociado una pila donde guardará información de estado. El tamaño de esta pila viene determinado por el parámetro `cb_stack_size`. Por defecto se inicializa con un tamaño de 1024 bytes, que es más que suficiente para la mayoría de los casos. Sólo debería aumentarse si se realizan instrumentaciones que pueden desembocar en varios niveles de profundidad.

Un parámetro muy importante es `cb_max_stubs`. Este parámetro indica la cantidad de “stubs” para los que se reservará memoria. Por cada función de un DSO que se instrumenta se consume un “stub”. Así si instrumentamos dos librerías, una con 100 funciones y la otra con 200, este parámetro debe valer como mínimo 300.

Por último tenemos el parámetro `cb_allow_handler` que permite usar, si lo deseamos, nuestro propio “generic wrapper” en las interposiciones mediante “callback”. Evidentemente esta opción especial para usuarios muy avanzados.

Si se instala un “callback” sobre un DSO luego ya no se puede instalar ningún otro tipo de interposición sobre cualquier función del mismo DSO ya que ambas entrarían en conflicto.

## 7.3. Estructura de un backend

Como ya se ha dicho en la introducción, un “backend” es una librería compartida y como tal nunca será un programa completo y carecerá de una función `main()`. El código de un “backend” en su mayor parte está orientado a tomar el control y gestionar cada llamada que pDI-Tools intercepte.

Un “backend” se suele dividir en las siguientes partes:

- **Inclusión del fichero `backend.h`.** Todo “backend” debería incluir el fichero `backend.h` al principio. Este fichero define constantes, macros, símbolos y nombres de funciones especialmente útiles a la hora de escribir un “backend”.
- **Código de inicialización y finalización del “backend”.** Consiste en dos funciones, totalmente opcionales. La primera, la de inicialización, se ejecuta antes del código a instrumentar y antes de que se instalen sus propios reenlaces. La segunda se ejecuta una vez finalizado el programa a instrumentar, y una vez deshechos todos los reenlaces. El código de inicialización generalmente son operaciones de reserva de memoria, inicialización de estructuras, etc. Mientras que el de finalización realiza las operaciones de terminación del “backend”.
- **Funciones de interceptación o funciones “wrapper”.** Las interposiciones de código más sencillas, que son los reenlaces y redefiniciones, consisten en redirigir la ejecución hacia una función de un “backend” cuando interceptan una llamada a un determinado DSO. Estas funciones reciben el nombre de “wrapper”.
- **Gestión de los “callback”.** Es un tipo de interposición de código especial, no siempre disponible en todas las arquitecturas debido a su complejidad, que sirve para interponer el mismo sobre todas las funciones de un DSO.
- **Código de gestión interna del “backend”.** Un “backend”, a pesar de no ser un programa tradicional, puede llegar a ser bastante complejo. Exceptuando los “backend” de ejemplo, en la mayoría de ocasiones un “backend” llega a ser tan complejo como cualquier otra aplicación, necesitando

usar estructuras de datos, funciones, variables globales... y no sólo las funciones “wrapper” y/o funciones de gestión de los “callback”.

### 7.3.1. El fichero `backend.h`

La principal función de este fichero es definir los prototipos de funciones que usa pDI-Tools para interaccionar con el “backend”. De esta forma hacemos que el compilador compruebe por nosotros que la signatura de las funciones que implementamos sea correcta.

No obstante no se recomienda usar las funciones directamente, sino mediante un alias que les asignamos en el mismo fichero `backend.h`. Los alias son simples constantes del preprocesador con el prefijo `PDI_BE_FUNC_`. Cada una de ellas se sustituye por el verdadero nombre de la función en tiempo de compilación. Por ejemplo la constante `PDI_BE_FUNC_INIT` se traducirá en el nombre de función `di_init_backend()`.

Los motivos de recomendar usar estos alias en lugar del verdadero nombre de la función son varios.

El primer motivo viene de intentar mantener la nomenclatura usada en todo el programa intentando romper lo menos posible la compatibilidad hacia atrás con DITools. DITools espera encontrar funciones con nombres como `di_init_backend()`, `di_callback_required()` o `di_post_event_callback()`. Si se observan detenidamente se verá que no siguen las convenciones decididas para los nombres de funciones de pDI-Tools (ver las convenciones en Sección 2.2.2). Ocultando los nombres de estas funciones al programador mediante alias que siguen las convenciones usadas se simplifica la labor del programador sin sacrificar la compatibilidad directa con antiguos “backends” de DITools.

Otro motivo es que si se diese una situación donde es necesario usar otros nombres de función para estas funciones, cambiarlos sería tan fácil como editar el fichero `backend.h` y recompilar pDI-Tools.

Un último motivo es que con el tiempo se dejarán de usar estos nombres de función por los motivos antes expuestos. Si los “backend” usan los alias en lugar de los nombres de función la transición será transparente.

Por último el fichero `backend.h` añade otro juego de constantes con el prefijo `PDI_STR_BE_FUNC_`. Cada constante es un string con el nombre de la función contenido en la constante con prefijo `PDI_BE_FUNC_`. Estas constantes son usadas por pDI-Tools para realizar búsquedas e imprimir errores referentes a estas funciones. Se ponen también a disposición del usuario para que haga con ellas el uso que más le convenga.

Un inventario de las funciones y sus constantes se puede ver en la Tabla 7-1. La primera columna es el nombre de función por defecto, la segunda es la constante que contiene el verdadero nombre de la función, y la tercera es otra constante calculada a partir de la anterior con un “string” con el nombre de función.

**Tabla 7-1. Funciones y constantes en `backend.h`**

| función | alias | string |
|---|---|---|
| `di_init_backend()` | `PDI_BE_FUNC_INIT` | `PDI_STR_BE_FUNC_INIT` |
| `di_fini_backend()` | `PDI_BE_FUNC_FINI` | `PDI_STR_BE_FUNC_FINI` |
| `di_callback_required()` | `PDI_BE_FUNC_CB_REQ` | `PDI_STR_BE_FUNC_CB_REQ` |
| `di_pre_event_callback()` | `PDI_BE_FUNC_PRE_CB` | `PDI_STR_BE_FUNC_PRE_CB` |
| `di_post_event_callback()` | `PDI_BE_FUNC_POST_CB` | `PDI_STR_BE_FUNC_POST_CB` |

Como hemos dicho sólo empezar se incluyen sus prototipos de función. Así el compilador puede

comprobar que la implementación del “backend” coincide con los prototipos de funciones que pDI-Tools espera encontrar. Estos prototipos de funciones se pueden ver en la Tabla 7-2.

**Tabla 7-2. Prototipos de funciones en `backend.h`**

| prototipo de función |
|---|
| `int PDI_BE_FUNC_INIT(void);` |
| `int PDI_BE_FUNC_FINI(void);` |
| `int PDI_BE_FUNC_CB_REQ(char *func_name);` |
| `void PDI_BE_FUNC_PRE_CB(int virtual_processor, int event_id, ...);` |
| `void PDI_BE_FUNC_POST_CB(int virtual_processor, int event_id, int retval);` |

Por último notar que al incluir el fichero `backend.h` no se incluye automáticamente ningún otro fichero de cabecera que importe otros símbolos de pDI-Tools. Si se desea usar alguna función de la API que ofrece pDI-Tools deberá incluirse explícitamente el fichero de cabecera adecuado.

## 7.3.2. Inicialización y finalización

El código de inicialización y finalización de un “backend” es algo totalmente opcional. Además la existencia del código de inicialización no fuerza a que exista un código de finalización o viceversa.

El código de inicialización se debe encapsular dentro de una función exportada públicamente (es decir, visible desde el exterior del módulo) con un determinado nombre. Por defecto recibe el nombre `di_init_backend()`, aunque se debería referir siempre a ella a través de la constante `PDI_BE_FUNC_INIT` (ver los motivos más arriba).

La inicialización del “backend” sucede después de la carga de los ficheros de configuración, pero antes de aplicar los ficheros de comandos. Evidentemente esto sucede antes de comenzar la ejecución del programa a instrumentar.

Los “backend” se inicializan según el orden de aparición en los ficheros de comandos, y en caso de haber varios ficheros de comandos, según el orden de aparición y dependencias establecidas entre ellos. Para saber más sobre el orden de carga e inicialización vea la Sección 7.1.2.

El código de finalización, al igual que el de inicialización, debe estar en una función de un determinado nombre: `di_fini_backend()`, aunque por los mismos motivos el programador debe referirse a ella mediante la constante `PDI_BE_FUNC_FINI`.

El código de finalización de los “backend” sólo se ejecuta una vez el programa ha finalizado y después de que pDI-Tools deshaga todas las interposiciones.

Por último comentar que es posible aprovechar el segmento de código de inicialización y finalización del estándar ELF (`.init` y `.fini`) para ejecutar un código de inicialización justo cuando pDI-Tools ha cargado el “backend” en memoria, o el de finalización cuando este “backend” sea descargado. No obstante esto nunca se ha probado y el resultado no es fácilmente predecible. Aunque en principio debería funcionar, se desaconseja seriamente.

### 7.3.3. Wrappers

Conceptualmente, el modo más sencillo de interceptar una llamada a una función sería poniendo en el lugar de la función original la función interceptadora. Esta estrategia es la que siguen aproximadamente los reenlaces y las redefiniciones: llamar a una función del “backend”, llamada “wrapper”, en lugar de la original. Cuando termina la ejecución de dicha función se devuelve el control al programa instrumentado.

Debido a los motivos comentados en el párrafo anterior, los “wrapper” deben tener el mismo aspecto (signatura) que la función a la que interceptan.

Los “wrapper” permiten, de una forma sencilla, alterar o registrar el resultado de una función de una librería. Generalmente suelen ser funciones que, a parte de sus tareas específicas, durante su ejecución suelen transferir el control a la función que están interceptando.

Por último remarcar que los “wrapper” deben ser visibles desde fuera del “backend”, para poder ser referenciados por pDI-Tools.

Un ejemplo de código de un “backend” con funciones “wrapper”:

```
/* backend de ejemplo: 'myfirstbe.so' */

#include<stdio.h>
#include<unistd.h>
#include<backend.h>

#define TRUE (1==1)

static int read_no;
static int write_no;

ssize_t read_wrapper(int fd, void *buf, size_t nbyte)
{
  ++read_no;
  fprintf(stderr, "read()\n");
  return read(fd, buf, nbyte);
}

ssize_t write_wrapper(int fd, const void *buf, size_t nbyte)
{
  ++write_no;
  fprintf(stderr, "write()\n");
  return write(fd, buf, nbyte);
}

int di_init_backend(void)
{
  printf("Inicializando backend.\n");
  read_no = write_no = 0;
  return TRUE;
}

void di_fini_backend(void)
{
  printf("Cerrando el backend.\n"
         "  number of reads  = %d\n"
         "  number of writes = %d\n", read_no, write_no);
}
```

Y este “backend” se usaría ejecutando los siguientes comandos, por ejemplo:

```
; Fichero: 'commands.cfg'
; Definimos los alias a las librerías y los backends que usamos
#backend BACKEND        ./myfirstbe.so

; Comandos de reenlace
# commands
R MAIN read BACKEND read_wrapper
R MAIN write BACKEND write_wrapper
```

Por último se puede usar este fichero de comandos tan sólo asignando el nombre del fichero de comandos (`commands.cfg`) a la variable de entorno DI_CONFIG_FILE o mediante un fichero de comandos como el siguiente:

```
# Fichero de configuración 'pdi.cfg'
config = commands.cfg
```

El resultado de la ejecución sería un mensaje por cada read(2) y write(2) del programa principal y, al finalizar la aplicación, un resumen con el total de las veces que han sido ejecutadas cada una de ellas.

### 7.3.4. Callbacks

Si un “backend” desea interceptar llamadas mediante el mecanismo de “callback” debe implementar las siguientes funciones (públicamente):

- Una función con un nombre determinado (definido por la constante `PDI_BE_FUNC_CB_REQ`) que al ejecutarse decida si la llamada interceptada desea procesarse o no.
- Dos funciones encargadas (ver constantes `PDI_BE_FUNC_PRE_CB` y `PDI_BE_FUNC_POST_CB`) que se ejecutan sólo cuando se intercepta una llamada y la anterior indica que debe procesarse. Una que se ejecuta antes que se ejecute el código interceptado, y otra que se ejecuta después. Se puede implementar una de las dos o las dos a la vez: son independientes y opcionales.

Debe tenerse en cuenta que estas funciones han de ser visibles desde fuera del “backend”, para poder ser referenciadas por pDI-Tools.

Terminamos esta descripción de los “callback” con un ejemplo de “backend” que los usa:

```
/* backend de ejemplo: 'callbacks.so' */

#include<stdio.h>
#include<unistd.h>
#include<backend.h>

#define TRUE (1==1)

/* Lista de funciones que procesará el callback */
char *funcs[] = { "fputc", "printf", "fprintf", "puts", "tell", NULL };
```

```
int PDI_BE_FUNC_INIT(void)
{
  printf("Inicializando el backend.\n");
  return TRUE;
}

void PDI_BE_FUNC_FINI(void)
{
  printf("Cerrando el backend.\n");
}

int PDI_BE_FUNC_CB_REQ(char *funcname)
{
  int i;

  printf("Identificado '%s'\n", funcname);
  for(i = 0; funcs[i]; i++)
    if(!strcmp(funcname, funcs[i]))
      return i+1;

  return 0;
}

void PDI_BE_FUNC_PRE_CB(int virtual_processor, int event_id, ...)
{
  printf("Se va a ejecutar una llamada a '%s'\n", funcs[event_id-1]);
}

void PDI_BE_FUNC_POST_CB(int virtual_processor, int event_id, int retval)
{
  printf("La llamada a '%s' a devuelto %d.\n", funcs[event_id-1], retval);
}
```

Y este "backend" se usaría ejecutando los siguientes comandos:

```
; Definimos los alias a las librerías y los backends que usamos
#backend BACKEND        ./callbacks.so

; Comandos de reenlace
# commands
C MAIN * BACKEND
```

De nuevo usaríamos este fichero de comandos con la variable de entorno DI_CONFIG_FILE o mediante un fichero de configuración.

Si el programa principal consiste en:

```
/* Programa de ejemplo */
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
```

```
  fputc('+', stdout);
  if(fputc('*', stdout) != (int) '*')
    printf("\nLa función 'fputc()' funcional mal!\n");
  else
    printf("\nLa función 'fputc()' parece funcionar bien\n");
}
```

El resultado de la instrumentación sería:

```
gerardo@kobetai:~/pdi/src/link$ ./runtest
Inicializando el backend.
Identificado '__libc_start_main'                          (1)
Identificado 'fputc'                                      (2)
Se va a ejecutar una llamada a 'fputc'
+La llamada a 'fputc' a devuelto 43.
Identificado 'fputc'                                      (3)
Se va a ejecutar una llamada a 'fputc'
*La llamada a 'fputc' a devuelto 42.
Identificado 'puts'                                       (4)
Se va a ejecutar una llamada a 'puts'

La función 'fputc()' parece funcionar bien
La llamada a 'puts' a devuelto 44.
Identificado 'exit'                                       (5)
Cerrando el backend.
gerardo@kobetai:~/pdi/src/link$
```

**(1)** La primera sorpresa que nos encontramos es una llamada a una función interna de C. Esta función es la encargada de inicializar las librerías de C `libc.so` y preparar el entorno de ejecución del programa y llamarlo. Esta función nunca vuelve, es decir, que si la instrumentamos con un "callback" sólo se ejecuta el pre-tratamiento y nunca el post-tratamiento. No vuelve debido a que al acabar ejecuta la llamada al sistema _exit(2).

**(2)** En este punto se ha ejecutado la función `fputc('+');`

**(3)** En este punto se ha ejecutado la función `fputc('*');`

**(4)** Otra sorpresa que nos llevamos es que en este punto no se ejecuta la función printf(2), sino un puts(3). El motivo de esto es que el optimizador del compilador ha decidido que para este caso es más eficiente llamar a esta función. Este tema es muy delicado ya que no depende ya ni de pDI-Tools ni del sistema operativo, sino totalmente de las decisiones del compilador.

**(5)** Si `main()` no llama explícitamente a exit(3), el código de finalización de la aplicación se encarga de hacerlo por él implícitamente. Este es el motivo de que se haya detectado esta llamada.

## 7.3.5. Código de gestión interna del "backend"

Se recomienda que un "backend" sólo exporte los símbolos necesarios y así evitar posibles colisiones con símbolos públicos de la aplicación instrumentada. Siempre es una buena política declarar como static cualquier símbolo global que no sea necesario ver desde fuera del "backend". Esta palabra reservada de C, usada en el ámbito global, impide que un símbolo pueda ser visto desde fuera del fichero objeto al que pertenece.

# Notas

1. Se fuerza a esta sintaxis para permitir que en el futuro se pueda implementar la posibilidad de instalar un “callback” sobre un conjunto de funciones y no necesariamente sobre todas.

# Capítulo 8. El `runtime`

El `runtime` es un "backend" que implementa unas cuantas interposiciones especiales que cooperarán con pDI-Tools para llevar a cabo con éxito la instrumentación del "software". Estas interposiciones instrumentan determinadas llamadas a GNU C Library que pueden alterar el estado de las estructuras del "Runtime Linker" y ELF, como por ejemplo la carga de un objeto compartido, o del flujo natural de la ejecución del programa, como una llamada a exit(3).

El `runtime` también es responsable de incorporar una rutina que asigne un identificador numérico a cada "thread", en caso de que la plataforma a la que está orientado soporte el mecanismo de interposición mediante "callback".

Seguidamente enumeramos unos cuantos casos de funciones que un `runtime` debería cubrir mediante definiciones o reenlaces y posteriormente explicamos como el `runtime` resuelve el problema antes comentado.

## 8.1. Casos que debería cubrir

pDI-Tools es una herramienta que debido a su naturaleza puede verse afectada de forma indeseada por algunos componentes del sistema y/o la aplicación. Se intenta controlar los fenómenos que pueden dejar las estructuras de pDI-Tools desincronizadas respecto a la información del "Runtime Linker" aunque hay cosas que caen fuera del control de pDI-Tools como llamadas al sistema como fork(2), exit(3), exec(2), abort(3), etc. pDI-Tools no dispone de ningún mecanismo para instrumentar directamente llamadas al sistema y como mucho puede interceptar las llamadas a las librerías de C que llamarán a estas llamadas al sistema.

Por ello de la gestión de estas llamadas se encargará un módulo que llamamos `runtime`. Este módulo es un "backend" normal a excepción de que es el primero que se instala, que interpone el código necesario para gestionar las llamadas antes descritas. No obstante este módulo actualmente no forma parte de pDI-Tools y es obligación del programador implementarlo para sus herramientas de instrumentación.

Las funciones que debe instrumentar el `runtime` varían en cada sistema operativo. Aquí sólo describimos algunas de las más importantes, pero que permiten ver la naturaleza de las funciones que se deben interceptar.

- **fork(2) y __clone(3).** Estas llamadas interesa capturarlas ya que su ejecución puede implicar cambios que alteren el funcionamiento de pDI-Tools o de los "backends". Además puede interesar partir los "log" en otros ficheros o mantener estructuras de memoria propias.

  Existen algunas llamadas de algunas librerías como MPI que pueden implicar acciones como la creación o destrucción de otro proceso. Se debería cubrir también estos casos.

- **dlopen(3) y dlclose(3).** La ejecución de estas llamadas altera la lista de objetos cargados en memoria. El `runtime` debería hacer una llamada a `_pdi_arch_refreshObjectList()` para poner al día la réplica de pDI-Tools.

- **exec(2) y execve(2).** Estas llamadas en el fondo son como llamar a exit(3): va implicar el cese de la aplicación y pDI-Tools y en algunos casos puede hacer falta tenerla instrumentada para forzar la finalización de pDI-Tools o simplemente para realizar algunas operaciones antes de transferir el control a otro programa.

- **Llamadas especiales al "Runtime Linker".** El "Runtime Linker" en algunas plataformas exporta un API especial que permite modificar sus estructuras. Es deber del programador detectarlas e

implementar los reenlaces adecuados para mantener pDI-Tools actualizado respecto a ellas.

## 8.2. Gestión de “threads” en los “callback”

Como se ha explicado en el anterior capítulo, el mecanismo de “callback” requiere que se implemente una función que llamamos “thread id resolver”. Esta función asigna un identificador numérico a cada “thread” que está en ejecución.

Esta función ha de reutilizar identificadores comenzando desde 0, es decir, que si un “thread” muere, el próximo que se creé debe tener el identificador del antiguo “thread”. Así si en un determinado momento hay 8 threads en ejecución todos sus identificadores estarán entre 0 y 7.

Respetar esta restricción es importante, ya que se asociará una pila de ejecución a cada “thread”, y estas pilas estarán almacenadas en una tabla de como mucho *`max_threads`* (ver parámetro de configuración) entradas. Si asignamos un identificador distinto a cada “thread” que se cree podemos agotar pronto las entradas de dicha tabla y/o consumir más memoria de la necesaria.

La función que identifica un “thread” no recibe ningún parámetro y sólo devuelve un entero positivo a partir de 0.

El “thread id resolver” se configura, habitualmente, desde el `runtime` con las funciones `_pdi_ebe_getThreadIdResolver()` y `_pdi_ebe_setThreadIdResolver()`. La primera obtiene un puntero al “thread id resolver” actual, mientras que la segunda permite asignar uno nuevo.

Mostramos un ejemplo de `runtime` que instala un “thread id resolver” para la librería pthread:

```
/* Runtime de ejemplo que instala un "thread id resolver" para pthreads */
#include<config.h>
#include<backend.h>
#include<log.h>
#include<pdiconfig.h>
#include<threadid.h>
#include<pthread.h>

#define FALSE 0
#define TRUE (1==1)

/* Esta es la tabla que contendrá la relación de threads en ejecución */
pthread_t *thconv = NULL;

/* Usaremos un mutex para hacer los accesos a la tabla de forma atómica */
pthread_mutex_t mutex_thconv = PTHREAD_MUTEX_INITIALIZER;

/* Esta función será la encarga de darle un identificador al thread */
static int idres(void)
{
  int i;
  int fz;
  pthread_t pt;

  if(thconv)
  {
    /* Buscamos el thread */
    pthread_mutex_lock(&mutex_thconv);
    pt = pthread_self();
```

```
    fz = PDICFG.max_threads + 1;
    for(i = 0; i < PDICFG.max_threads; i++)
    {
      /* si lo encontramos lo devolvemos */
      if(thconv[i] == pt)
      {
        pthread_mutex_unlock(&mutex_thconv);
        return i;
      }
      /* nos apuntamos esta posición, si está libre */
      if(fz > PDICFG.max_threads && !thconv[i])
        fz = i;
    }

    /* no hemos encontrado el thread, lo damos de alta */
    thconv[fz] = pt;
    pthread_mutex_unlock(&mutex_thconv);

    return fz;
  } else
    return 0;
}

/* este wrapper se encargará de interceptar la muerte de un */
/* thread y así darlo de baja en la tabla 'thconv'          */
int pthread_exit_wrapper(void *ret)
{
  int id;

  if(thconv)
  {
    id = idres();
    _pdi_log(THIS, "mato a %d\n", id);
    thconv[id] = 0;
  }

  pthread_exit(ret);
}

/* funciones de inicialización y finalización */
int PDI_BE_FUNC_INIT(void)
{
  _pdi_log(THIS, "Inicializando backend '" __FILE__ "'.\n");

  if(PDICFG.max_threads > 0)
  {
    if((thconv = malloc(sizeof(int) * PDICFG.max_threads)) == NULL)
    {
      _pdi_error(THIS, "No hubo memoria para 'thconv'.");
      return FALSE;
    }

    memset(thconv, 0, sizeof(int) * PDICFG.max_threads);

    _pdi_ebe_setThreadIdResolver(idres);
  } else {
    _pdi_log(THIS, "En este programa no se usan threads.");
```

```
  }

  return TRUE;
}

void PDI_BE_FUNC_FINI(void)
{
  _pdi_log(THIS, "Cerrando mi backend.\n");

  if(thconv)
  {
    free(thconv);
    thconv = NULL;
    _pdi_ebe_setThreadIdResolver(NULL);
  }
}
```

Como podemos ver este `runtime`, sólo inicializarse instala, si hay instrumentación con “threads”, un “thread id resolver” e inicializa una tabla de conversión. Esta tabla de conversión será ampliada cada vez que se ejecute `idres`() sobre un nuevo “thread”.

La tabla de conversión mantiene una biyección entre un identificador pthread y el identificador numérico que le asignamos. Se dan de alta los nuevos “threads” de tal modo que siempre se les asignará el identificador más bajo posible.

Para mantener sólo los “threads” vivos en la tabla se instala una redefinición sobre la función `pthread_exit`() que va dando de baja los “threads” que terminan. El fichero de comandos del `runtime` sería el siguiente:

```
; Fichero de comandos runtime
#backend RUNTIME runtime.so
#object  PTHREADS /lib/libpthread.so

#commands
D PTHREADS pthread_exit RUNTIME pthread_exit_wrapper
```

Es posible que no se hayan contemplado casos de la librería de pthread en este ejemplo, pero tampoco pretende ser un `runtime` exhaustivo y real. Simplemente se ilustra como se implementaría un pequeño sistema de obtención de identificadores de “thread”.

# Capítulo 9. API de pDI-Tools

Finalmente se describe el API de pDI-Tools. Esta API permitirá al programador de “backends” realizar desde su código muchas operaciones de pDI-Tools como cargar “backends”, instalar interposiciones, gestionar ficheros de comandos, etc.

En este capítulo se subdivide el API en diferentes partes según el fichero de cabecera donde está definida cada función.

## 9.1. `beconfig.h` - Gestión de ficheros de comandos

Este módulo ofrece un conjunto de herramientas para leer y manipular ficheros de configuración para los “backends”, y también para aplicar dicha configuración en el proceso actual. En dichos ficheros se especifican los objetos y funciones sobre los que realizaremos interposiciones de código, y evidentemente, los “backends” y sus funciones de interposición.

Esta API se ofrece para que los “backend”, si lo desean, puedan cargar y procesar ficheros de comandos y así instalar configuraciones complejas de interposiciones de un modo casi automático.

```
BECFG_OBJ_INFO *_pdi_becfg_addCommand(BECFG_CONFIG *cfg, int type,
BECFG_OBJ_INFO *trgObj, char *trgFunc, BECFG_OBJ_INFO *backend, char
*wrapper);
```

**Descripción.** Añade un nuevo comando de interposición a la configuración *`cfg`*.

**Parámetros.**

- ***`cfg`***. Configuración que manipulamos.
- ***`type`***. Tipo de comando. Puede tomar los valores `BECFG_IT_RELINK`, `BECFG_IT_REDEFINITION` o `BECFG_IT_CALLBACK`.
- ***`trgObj`***. Objeto que afectara esta definición (`NULL` si se usa un comodín).
- ***`trgFunc`***. Función a la que afectará (o `NULL` si comodín).
- ***`backend`***. El “backend” encargado de gestionar esta interposición.
- ***`wrapper`***. El nombre de la función “wrapper” (o nuevo “generic wrapper” en caso de ser un “callback”) que gestionará la interposición. `NULL` si no se indica ninguna.

**Valor devuelto.** 0 en caso de éxito, -1 en caso de error.

```
BECFG_OBJ_INFO *_pdi_becfg_addObject(BECFG_CONFIG *cfg, char *o_path, char
*o_alias, int is_backend);
```

**Descripción.** Añade un nuevo objeto a la lista de objetos en la configuración.

**Parámetros.**

- ***`cfg`***. Configuración que manipulamos.

- ***o_path*.** Ruta al binario.
- ***o_alias*.** Alias que le asignamos.
- ***is_backend*.** Distinto de cero si declaramos un “backend”.

**Valor devuelto.** Un puntero a la información del nuevo objeto en caso de éxito, `NULL` en caso de error.

```
int _pdi_becfg_applyBackendConfig(BECFG_OBJ_INFO *cfg);
```

**Descripción.** Ejecuta la lista de comandos de interposición contenida en la configuración *`cfg`*.

**Parámetros.**

- ***cfg*.** Configuración a aplicar.

**Valor devuelto.** 0 en caso de éxito, o -1 en caso de error.

```
void _pdi_becfg_destroyConfig(BECFG_CONFIG *cfg);
```

**Descripción.** La función `_pdi_becfg_destroyConfig()` destruye la estructura BECFG_CONFIG devuelta por `_pdi_becfg_newConfig()` y libera memoria.

**Parámetros.**

- ***cfg*.** Puntero a la configuración a destruir.

**Valor devuelto.** La función no devuelve nada.

```
void _pdi_becfg_finalizeRead(BECFG_CONFIG *cfg);
```

**Descripción.** `_pdi_becfg_finalizeRead()` termina la lectura del fichero de comandos activa en la estructura *`cfg`*.

**Parámetros.**

- ***cfg*.** Puntero a la configuración sobre la que trabajaremos.

**Valor devuelto.** La función `_pdi_becfg_finalizeRead()` no devuelve nada.

```
int _pdi_becfg_initializeRead(BECFG_CONFIG *cfg, FILE *f);
```

**Descripción.** Prepara la estructura de configuración *`cfg`* para empezar a leer un fichero de comandos a través del descriptor fichero *`f`*.

**Parámetros.**

- ***cfg*.** Puntero a la configuración sobre la que trabajaremos.
- ***f*.** Puntero a un descriptor de fichero desde donde leeremos el fichero de comandos.

**Valor devuelto.** La función `_pdi_becfg_initializeRead()` devuelve 0 en caso de éxito, o -1 en caso de error.

```
BECFG_OBJ_INFO *_pdi_becfg_getObjectByPath(BECFG_CONFIG *cfg, char *path);
BECFG_OBJ_INFO *_pdi_becfg_getObjectByAlias(BECFG_CONFIG *cfg, char
*alias);
BECFG_OBJ_INFO *_pdi_becfg_getObject(BECFG_CONFIG *cfg, char *x);
```

**Descripción.** La primera, `_pdi_becfg_getObjectByPath()`, busca la información sobre un objeto en una configuración por la ruta a su binario. `_pdi_becfg_getObjectByAlias()` busca el objeto por su alias y la última, `_pdi_becfg_getObject()` busca primero por el alias y en caso de no encontrarse vuelve a buscar por la ruta al binario.

**Parámetros.**

- ***cfg*.** Puntero a la configuración donde haremos la búsqueda.
- ***alias*.** Cadena con el alias a buscar.
- ***path*.** Cadena con el binario a buscar.
- ***x*.** Cadena con la ruta al binario o el alias a buscar.

**Valor devuelto.** Estas funciones devuelven `NULL` si no se encontró el objeto, en caso de éxito se devuelve un puntero a una estructura BECFG_OBJ_INFO.

```
char *_pdi_becfg_getObjectName(BECFG_OBJ_INFO *object);
```

**Descripción.** Esta función devuelve el nombre del objeto (la ruta al binario, o el alias en su defecto).

**Parámetros.**

- ***object*.** Puntero con información sobre el objeto.

**Valor devuelto.** Una cadena con el nombre del objeto, o `NULL` en caso de error.

```
BECFG_OBJ_INFO *_pdi_becfg_loadBackendConfigFromFile(FILE *f, char
*wrapper);
BECFG_OBJ_INFO *_pdi_becfg_loadBackendConfig(char *fname);
```

**Descripción.** Estas funciones permiten cargar desde fichero(s) la configuración de los “backend” en memoria. No comprueban que los símbolos, objetos, “backends”, etc. existan, tan solo los

declara.

**Parámetros.**

- ***fname*.** Nombre del fichero a abrir y a procesar.
- ***f*.** Fichero desde el cual se lee el fichero de comandos.

**Valor devuelto.** `NULL` en caso de error, un puntero a una configuración en caso de éxito.

```
int _pdi_becfg_mergeBackendConfigFiles(BECFG_OBJ_INFO *ctrg,
BECFG_OBJ_INFO *csrc);
```

**Descripción.** Combina las configuraciones *`csrc`* y *`ctrg`* y deja el resultado en la configuración *`ctrg`*.

**Parámetros.**

- ***csrc*.** Configuración origen.
- ***ctrg*.** Configuración destino.

**Valor devuelto.** 0 en caso de éxito, o -1 en caso de error.

Si la ejecución falla (devuelve un valor distinto de cero), se desaconseja usar la configuración *`ctrg`* ya que esta puede quedar en un estado corrupto.

```
BECFG_CONFIG *_pdi_becfg_newConfig(char *name);
```

**Descripción.** La primera función, `_pdi_becfg_newConfig()`, reserva memoria para una nueva lista de comandos de interposición.

**Parámetros.**

- ***name*.** Nombre del fichero de comandos (generalmente suele ponerse el nombre del fichero desde donde la hemos leído).

**Valor devuelto.** Devuelve un puntero a una configuración o `NULL` en caso de error.

```
int _pdi_becfg_setObjectAlias(BECFG_OBJ_INFO *object, char *new_alias);
```

**Descripción.** Esta función devuelve el nombre del objeto (la ruta al binario, o el alias en su defecto).

**Parámetros.**

- ***object*.** Puntero con información sobre el objeto.

- ***new_alias*.** Nuevo alias a asignar. `NULL` si se desea quitarle el alias actual.

**Valor devuelto.** 0 en caso de éxito, -1 en caso de error

```
int _pdi_becfg_showObjects(int nivel, BECFG_OBJ_INFO *cfg);
```

**Descripción.** Imprime por el canal de "log" la lista de objetos que contiene la configuración *`cfg`*. El parámetro *`nivel`* puede tomar uno de los valores `LOG_LEVEL_*` definidos en `log.h` e indica a partir de que nivel de verbosidad debe mostrarse la lista.

**Parámetros.**

- ***nivel*.** Nivel de verbosidad con el que mostramos la lista.
- ***ctrg*.** Configuración que queremos examinar.

**Valor devuelto.** Nada.

## 9.2. `ebeif.h` - Interfaz del "Elf Backend"

Esta interfaz exporta las funciones necesarias para poder manipular los objetos en memoria, cargar y descargar "backends" e instalar y desinstalar interposiciones. Todas estas funciones son independientes de la arquitectura, y por lo tanto están disponibles sobre todas las plataformas.

```
void *_pdi_ebe_getBackendSymbol(PDI_ELFOBJ *backend, char *symbol);
```

**Descripción.** Obtiene un puntero a un símbolo *`symbol`* del backend *`backend`*.

**Parámetros.**

- ***backend*.** Puntero a la información de objeto del "backend" en el que buscaremos el símbolo.

**Valor devuelto.** Un puntero al punto de entrada del símbolo, o `NULL` en caso de que este no exista.

```
char *_pdi_ebe_getObjectName(PDI_ELFOBJ *obj);
```

**Descripción.** Obtiene la ruta o el alias en su defecto del objeto *`obj`*.

**Parámetros.**

- ***obj*.** Puntero a la información de objeto.

**Valor devuelto.** Un puntero a la cadena que identifica el objeto.

```
int _pdi_ebe_installInterposition(int type, PDI_ELFOBJ *trgObj, char
*trgFunc, PDI_ELFOBJ *backend, char *wrapper);
```

**Descripción.** Instala una interposición de tipo *type*, sobre el objeto *object* y la función *func*. Los tipos soportados de interposición son: `PDI_IT_RELINK`, `PDI_IT_REDEFINITION` y `PDI_IT_CALLBACK`.

**Parámetros.**

- ***type*.** Tipo de interposición.
- ***trgObj*.** Objeto sobre el que instalamos la interposición.
- ***trgFunc*.** Función sobre la que instalamos la interposición. Debe valer `NULL` en caso de instalarse un "callback".
- ***backend*.** Objeto "backend" que gestionará la interposición.
- ***wrapper*.** Función "wrapper" que recibirá la interposición. Normalmente, en el caso del "callback", este parámetro es `NULL`.

**Valor devuelto.** 0 en caso de éxito, o -1 en caso de error.

```
PDI_ELFOBJ *_pdi_ebe_loadBackend(char *path);
```

**Descripción.** Permite cargar e inicializar un "backend" a partir de la ruta a su binario.

**Parámetros.**

- ***path*.** Ruta del fichero "backend".

**Valor devuelto.** Un puntero al PDI_ELFOBJ que representa a este backend si ha habido éxito en la carga. En caso contrario la función devuelve `NULL`.

```
char *_pdi_ebe_mainFilename(void);
```

**Descripción.** Obtiene el path o el alias del ejecutable instrumentado.

**Parámetros.** Esta rutina no recibe parámetros.

**Valor devuelto.** Un puntero a la cadena que identifica al ejecutable instrumentado.

```
PDI_INTERCEPT *_pdi_ebe_searchInterposition(PDI_ELFOBJ *object, char
*path);
```

**Descripción.** Busca una interposición que afecte a la función *func* del objeto *object*.

**Parámetros.**

- ***object*.** Objeto en el que buscamos.

- ***func*.** Función a la que debe afectar la interposición.

**Valor devuelto.** Devuelve un puntero a la información de interposición si hay éxito, o `NULL` en caso de no existir.

```
PDI_ELFOBJ *_pdi_ebe_searchObjectByPath(char *path);
PDI_ELFOBJ *_pdi_ebe_searchObjectByAlias(char *path);
PDI_ELFOBJ *_pdi_ebe_searchObject(char *x);
```

**Descripción.** La primera, `_pdi_ebe_searchObjectByPath()`, busca la información sobre un objeto en una configuración por la ruta a su binario. `_pdi_ebe_searchObjectByAlias()` busca el objeto por su alias y la última, `_pdi_ebe_searchObject()` busca primero por el alias y en caso de no encontrarse nada vuelve a buscar por la ruta al binario.

**Parámetros.**

- ***alias*.** Cadena con el alias a buscar.
- ***path*.** Cadena con el binario a buscar.
- ***x*.** Cadena con la ruta al binario o el alias a buscar.

**Valor devuelto.** Estas funciones devuelven `NULL` si no se encontró el objeto, en caso de éxito se devuelve un puntero a una estructura PDI_ELFOBJ.

```
int _pdi_ebe_setObjectAlias(PDI_ELFOBJ *obj, char *alias);
```

**Descripción.** Le asigna al objeto *`object`* el alias *`alias`*. Si *`alias`* es `NULL` se le quita al objeto su alias actual (si lo tiene).

**Parámetros.**

- ***object*.** Objeto al que queremos quitarle o ponerle un alias. Los alias `PDI_ALIAS_LIBC`, `PDI_ALIAS_MAIN` y `PDI_ALIAS_PDI` están reservados. El primero es un alias que identifica al DSO de pDI-Tools (`libpdi.so`), y el segundo es un alias que identifica al programa principal.
- ***object*.** Nuevo alias. Si es `NULL` se le quita el alias actual al objeto.

**Valor devuelto.** 0 si todo fue bien, -1 en caso de error.

```
int _pdi_ebe_unloadAllBackends(void);
```

**Descripción.** Descarga de memoria todos los “backends”. Pero ojo, no ejecuta el código de finalización de los mismos.

**Parámetros.** Esta función no recibe parámetros.

**Valor devuelto.** 0 si todo fue bien, otro valor en caso de error.

```
int _pdi_ebe_unloadBackend(PDI_ELFOBJ *be);
```

**Descripción.** Desinstala las interposiciones, finaliza y descarga el “backend” *be*.

**Parámetros.**

- ***be*.** Puntero a la información de objeto del “backend” que descargaremos.

**Valor devuelto.** 0 si todo fue bien, otro valor en caso de error.

```
int _pdi_ebe_uninstallAllInterpositions(void);
int _pdi_ebe_uninstallBackendInterpositions(PDI_ELFOBJ *backend);
int _pdi_ebe_uninstallInterposition(PDI_ELFOBJ *object, PDI_INTERCEPT *i);
int _pdi_ebe_uninstallInterpositions(PDI_ELFOBJ *object);
```

**Descripción.** La primera función desinstala todas las interposiciones instaladas actualmente. La segunda, `_pdi_ebe_uninstallBackendInterpositions()`, desinstala todas las interposiciones que gestiona un determinado “backend”. La función `_pdi_ebe_uninstallInterposition()` desinstala una única interposición y la última desinstala todas las interposiciones sobre un determinado objeto.

**Parámetros.**

- ***backend*.** Objeto “backend” que queremos que no gestione más interposiciones.
- ***i*.** Interposición que deseamos desinstalar.
- ***object*.** Objeto del que desinstalamos una o más interposiciones.

**Valor devuelto.** 0 si todo fue bien, -1 en caso de error.

## 9.3. `log.h` - Sistema de “log”

Este módulo es un conjunto de funciones encargadas de emitir los distintos mensajes de error, advertencias, sucesos (“log”) o mensajes de depuración de un modo sencillo y uniforme.

```
#define LOG_LEVEL_ERROR 0
#define LOG_LEVEL_WARNING 1
#define LOG_LEVEL_LOG 2
#define LOG_LEVEL_DEBUG 3
```

**Descripción.** Son constantes que identifican a cada nivel de “log” posible. Cuanto más alto es el número que identifican, menos alta es su prioridad.

```
#define THIS (char *) __FILE__, (char *) __FUNCTION__
```

**Descripción.** Esta constante es muy útil ya que si se puede usar en lugar de los parámetros *`fichero`* y *`funcion`*, y de esta forma ahorrarse muchos errores debidos al "copy'n'paste" y facilitando la escritura de los mensajes de "log".

Seguidamente se muestra un ejemplo de como usarlo:

```
#include"backend.h"
#include"log.h"

int fputc_wrapper(int c)
{
  _pdi_log("backend.so", "putchar_wrapper", "putchar escribe '%c'.", c);
  return c;
}

int fputs_wrapper(char *s, FILE *f)
{
  _pdi_log(THIS, "fputs escribe '%s'.", s);
  return 0;
}
```

Las dos indican el fichero al que pertenecen y la función desde donde se imprime el mensaje, sólo que la segunda es más resistente a cambios y más corta.

```
void _pdi_debug(char *fichero, char *funcion, char *formato, ...);
void _pdi_log(char *fichero, char *funcion, char *formato, ...);
void _pdi_warning(char *fichero, char *funcion, char *formato, ...);
void _pdi_error(char *fichero, char *funcion, char *formato, ...);
```

**Descripción.** En este grupo de funciones imprimen todas un mensaje de "log". Sólo les diferencia el nivel de "log" en que cada una imprime el mensaje. En orden respectivo: `LOG_LEVEL_DEBUG`, `LOG_LEVEL_LOG`, `LOG_LEVEL_WARNING` y `LOG_LEVEL_ERROR`.

**Parámetros.**

- ***`fichero`*.** Nombre del fichero con que relacionamos el mensaje de "log", aunque puede ser cualquier otra cosa o `NULL` en caso de no querer mostrar este campo.
- ***`funcion`*.** Nombre de la función con que relacionamos el mensaje de "log", aunque puede ser cualquier otra cosa o `NULL` en caso de no querer mostrar este campo.
- ***`formato`*.** Cadena de formato en la que intervendrán los parámetros siguientes.
- ***`...`*** Lista de parámetros variable. Depende del parámetro *`formato`*.

**Valor devuelto.** Estas funciones no devuelven nada.

```
void _pdi_log_level(int nivel, char *fichero, char *funcion, char
*formato, ...);
```

**Descripción.** Esta función imprime un mensaje de "log" en el nivel que se le indique. Es útil cuando la importancia de un mensaje de "log" puede variar en tiempo de ejecución.

**Parámetros.**

- ***nivel*.** Categoría del mensaje de error. Puede valer `LOG_LEVEL_DEBUG`, `LOG_LEVEL_LOG`, `LOG_LEVEL_WARNING` o `LOG_LEVEL_ERROR`.
- ***fichero*.** Nombre del fichero con que relacionamos el mensaje de "log", aunque puede ser cualquier otra cosa o `NULL` en caso de no querer mostrar este campo.
- ***funcion*.** Nombre de la función con que relacionamos el mensaje de "log", aunque puede ser cualquier otra cosa o `NULL` en caso de no querer mostrar este campo.
- ***formato*.** Cadena de formato en la que intervendrán los parámetros siguientes.
- `...` Lista de parámetros variable. Depende del parámetro *`formato`*.

**Valor devuelto.** Esta función no devuelve nada.

## 9.4. `pdiconfig.h` - Gestión de la configuración

Este fichero de cabecera exporta una estructura de tipo PDI_CONFIG con todos los parámetros de configuración de pDI-Tools. Esta estructura recibe el nombre `PDICFG` y seguidamente describimos cada uno de sus campos:

```
int allow_lib_as_be;
```

En principio está prohibido redirigir una llamada desde un DSO a otro DSO que no sea un "backend". No obstante es posible que en algún caso pueda ser interesante relajar esta restricción y esto es para lo que sirve este parámetro si se activa. Por defecto esta opción suele estar desactivado.

```
char **be_path;
int n_be_path;
char **becfg_path;
int n_becfg_path;
```

Estos campos indican las rutas de búsqueda a los "backend", en el caso de *`be_path`*, y a los ficheros de comandos en el caso de *`becfg_path`*. Estos campos son dos tablas de punteros, donde cada entrada es un puntero a una cadena con la ruta de búsqueda. Cada una de estas tablas tiene *`n_be_path`* y *`n_becfg_path`* elementos, respectivamente.

```
char **beconfig_files;
int n_beconfig_files;
```

Es la lista de ficheros de comandos que se han configurado para usar en pDI-Tools. El primero suele ser el `runtime`, y el resto ficheros de comandos estándar. Se organiza como una tabla de punteros a cadenas, donde cada entrada es la ruta a un fichero de configuración. Esta tabla contiene *`n_beconfig_files`* entradas.

```
int cb_allow_handler;
```

Parámetro booleano. Si está activado se permite usar un "callback handler" propio. Si está desactivado, pDI-Tools obliga a tener vacío o poner a `NULL` el cuarto parámetro de las reglas "callback".

Por defecto esta opción está desactivada.

`int cb_max_stubs;`

Indica cuantos “stubs” como máximo habrá en memoria. Es importante considerar que si vamos a poner un “callback” sobre un objeto que llama a X funciones diferentes, es conveniente que este parámetro tome un valor mayor que X o si no será imposible instalar el “callback”.

El valor por defecto varía según la plataforma ya que se calcula con la función `_pdi_arch_defaultMaxStubs()`. Esta función devuelve, generalmente, el tamaño de página de la arquitectura dividida por el tamaño de un “stub”. Por ejemplo, en INTEL® 386 esta función asigna 204 como valor por defecto a *`cb_max_stubs`* (4096 bytes por página y 20 bytes por “stub”).

`int cb_stack_size;`

Al monitorizar los “threads” necesitamos una zona de memoria donde guardar información sobre que se está monitorizando en un determinado momento. Evidentemente, cada “thread” ha de tener su espacio separado. Esto se realiza mediante una pila cuyo tamaño viene definido por esta variable. Si durante una monitorización hay problemas (“Segmentation Fault”), pruebe a incrementar esta variable. Si con valores grandes sigue teniendo problemas quizás debería empezar a sospechar que hay una monitorización recursiva.

Por defecto vale 1024.

`int debug;`

Si está activo se realiza un montón de trabajo extra para verificar que las estructuras de datos, tanto de ELF como de pDI-Tools, son coherentes.

`int donttouch_backends;`

Si esta opción está activada, no se podrán realizar interposiciones sobre un “backend”. Es aconsejable, por seguridad, tener activada esta opción.

Cuando está activada esta opción, los comodines no engloban a los “backend”, sino sólo a los objetos regulares (y a `libpdi.so` si la opción *`donttouch_pdi`* está desactivada).

`int donttouch_pdi;`

Si esta opción está desactivada se podrá trabajar sobre `libpdi.so`. Evidentemente autoinstrumentar pDI-Tools es algo muy arriesgado y complejo, por lo tanto se desaconseja desactivar esta opción.

```
char **lib_path;
int n_lib_path;
```

Este campo indica las rutas de búsqueda de las librerías. Este campo, por defecto, es una copia de la variable de entorno LD_LIBRARY_PATH. Al igual que antes, *`lib_path`* es una tabla de punteros donde cada entrada es un puntero a una cadena con la ruta de búsqueda. Esta tabla contendrá *`n_lib_path`* entradas.

```
char *log_filename;
FILE *logfile;
```

Ruta al fichero y descriptor del fichero donde se imprimirán los mensajes de “log”.

`int max_objects;`

Número máximo de objetos que puede haber en memoria (entre el binario, librerías y los "backends"). Es conveniente que sea un número algo alto, pero tampoco demasiado grande ya que si no consumimos mucha memoria.

Por defecto es 40.

`int max_threads;`

Este parámetro fija la máxima cantidad de "threads" que se pueden monitorizar. Es conveniente que el número no sea demasiado bajo, ya que si se superase el número de "threads" abortaría la ejecución del programa.

`int num_threads;`

No puede tener un valor mayor que *`max_threads`*. Indica el número de "threads" que se espera usar. De esta forma la inicialización y reserva de memoria se hace al inicio del programa y no durante la ejecución, evitando que la instrumentación tenga mucho impacto en la ejecución del programa.

Se pueden introducir los siguientes valores:

- *-1* - Inicializa todos los "threads" a la vez (en total *`max_threads`*) de golpe durante la inicialización. Esto implica el peor caso en consumo de memoria.
- *0* - No inicializamos ningún "thread". Se irán inicializando según se necesiten.
- *+x* - Inicializa *x* "threads". Si se supera el número el resto se van inicializando durante la ejecución del programa hasta un máximo de *`max_threads`*.

Por defecto vale 0.

`int verbose;`

Establece el nivel de verbosidad de los mensajes de "log": 0, silencio total (excepto errores); 1, solo advertencias y errores (valor por defecto); 2, imprime también mensajes normales y 3, que imprime también los mensajes de depuración.

## 9.5. `threadid.h` - Gestión del "thread id resolver"

Con este subconjunto del API podemos configurar la función "thread id resolver". Esta función es usada por el mecanismo de interposición por "callback" para identificar a cada "thread" de la aplicación.

```
int (*_pdi_ebe_getThreadIdResolver(void))(void);
```

**Descripción.** Obtiene un puntero al punto de entrada del "thread id resolver" actual.

**Parámetros.** Esta función no recibe parámetros.

**Valor devuelto.** `_pdi_ebe_getThreadIdResolver()` devuelve un puntero a la función actualmente configurada, o `NULL` si no hay ninguna.

```
void _pdi_ebe_setThreadIdResolver(int (*thresolver)(void));
```

**Descripción.** Establece la función apuntada por ***`thresolver`*** como el nuevo “thread id resolver” actual.

**Parámetros.**

- ***`thresolver`*****.** Puntero al nuevo “thread id resolver”.

**Valor devuelto.** Esta función no devuelve nada.

# IV. Manual de instalación

# Capítulo 10. Manual de instalación

En este capítulo se describe el método de instalación basado en Autoconf y Automake de pDI-Tools.

Si se está ansioso por comenzar la instalación en la siguiente sección se describe lo básico para realizar una instalación.

El resto del apéndice es una descripción bastante genérica sobre como configurar, compilar e instalar la aplicación.

## 10.1. Instalación básica

La aplicación se presenta como un árbol de directorios con los "scripts" y el código fuente para construir la aplicación. Se entrega todo empaquetado dentro de un fichero `.tar` (tal cual o, generalmente, comprimido). No se incluyen binarios del programa en la distribución.

El "shell script" `configure` intenta averiguar los valores correctos para las variables dependientes de la arquitectura usadas durante la compilación. Usa estos valores para crear un `Makefile` en cada directorio del paquete pDI-Tools. También crea algunos ficheros `.h` que contienen definiciones dependientes del sistema. Finalmente crea el "shell script" `config.status` que usted puede ejecutar en el futuro para recrear la configuración actual, y un fichero `config.log` que contiene la salida del compilador (fichero especialmente útil para depurar a `configure`).

También se puede usar un fichero opcional (generalmente llamado `config.cache` y que se activa con la opción *`--cache-file=config.cache`* o simplemente *`-C`*) que salva los resultados de sus tests para acelerar la reconfiguración. (Esta funcionalidad está desactivada por defecto para evitar posibles problemas con valores anticuados).

Si necesita hacer a mano algo para compilar el paquete ya que no lo hace `configure`, por favor intente averiguar como `configure` podría hacerlas y envíenos los "diffs" o las instrucciones a la dirección dada en el fichero `README` para que puedan ser consideradas/incluidas en la próxima versión. Si está usando la cache y en algún momento `config.cache` contiene resultados que no desea guardar, puede borrarlo o editarlo.

El fichero `configure.in` es usado para crear a `configure` por un programa llamado Autoconf. Usted sólo necesitará el fichero `configure.in` si usted desea cambiar o regenerar a `configure` usando una nueva versión de Autoconf.

El método más sencillo para compilar este programa es:

1. Cambie al directorio que contiene la distribución de pDI-Tools y escriba

   `./configure`

   para configurar pDI-Tools para su sistema. Si esta usando el "shell" **csh** o una versión antigua de "System V", usted puede necesitar escribir

   `sh ./configure`

   para evitar que **csh** intente ejecutar `configure` el mismo.

   Ejecutar `configure` toma un tiempo. Mientras se ejecuta imprime mensajes indicando que pruebas está realizando.

2. Ejecute **make** para compilar el paquete.
3. Opcionalmente, ejecute **`make check`** para ejecutar los tests que vienen con pDI-Tools.
4. Ejecute **`make install`** para instalar los programas, los ficheros de datos y la documentación.

5. Puede borrar los binarios y los ficheros objetos del código fuente ejecutando **`make clean`**. Para borrar también los ficheros creados por `configure` (de tal forma que pueda compilar el paquete para otro tipo de ordenador), escriba **`make distclean`**. También puede usar el comando **`make maintainer-clean`**, pero está pensando principalmente para ser usado por el desarrollador del paquete. Si usted lo usa, puede resultar que necesite instalar todo tipo de programas para poder regenerar los ficheros que venían con su distribución.

# 10.2. Compiladores y opciones

Algunos sistemas requieren opciones inusuales para compilar o enlazar de las cuales `configure` no tiene conocimiento. Ejecute **`./configure --help`** para obtener detalles sobre las variables de entorno pertinentes.

Puede darle a `configure` valores iniciales para sus parámetros de configuración inicializando variables en la línea de comandos o mediante variables de entorno. Por ejemplo:

```
./configure CC=c89 CFLAGS=-O2 LIBS=-lposix
```

# 10.3. Directorios y nombres de archivos al instalar

Por defecto, **`make install`** instalará los ficheros del paquete en el directorio `/usr/local/bin`, `/usr/local/man`, etc. Puede especificar otro prefijo en lugar de `/usr/local` llamando a `configure` con la opción *`--prefix=PATH`*.

Puede especificar prefijos de instalación separados para los ficheros específicos de la arquitectura y los ficheros independientes de la arquitectura. Si se da a `configure` la opción *`--exec-prefix=PATH`*, el paquete usará *`PATH`* como el prefijo para instalar los programas y librerías. La documentación y los otros ficheros de datos seguirán usando el prefijo normal.

Adicionalmente, si desea una distribución de directorios especial, se pueden usar opciones como *`--bindir=PATH`* para especificar diferentes valores para diferentes tipos de fichero. Ejecute **`configure --help`** para obtener una lista de los directorios que puede configurar y que tipos de ficheros van en ellos.

Puede hacer que los programas se instalen con un prefijo o sufijo extra en sus nombres dándole a `configure` las opciones *`--program-prefix=PREFIX`* o *`--program-suffix=SUFFIX`*.

# 10.4. Características opcionales

El “script” de pDI-Tools reconoce algunas opciones de la forma *`--enable-FEATURE`*, donde *`FEATURE`* indica una parte opcional del paquete. El fichero `README` menciona las opciones *`--enable-`* y *`--with-`* que pDI-Tools reconoce.

Aunque pDI-Tools de momento no usa el sistema de ventanas X (X Window), `configure` puede encontrar automáticamente sus ficheros de cabecera y librerías. En caso de no poder, se puede configurar manualmente con las opciones *`--x-includes=DIR`* y *`--x-libraries=DIR`*.

## 10.5. Especificando el tipo de sistema

Puede haber algunas características que `configure` no sea capaz de reconocer automáticamente, pero que puede determinar sabiendo el tipo de máquina sobre el que funcionará el programa. Generalmente, asumiendo que se construye el paquete sobre la misma arquitectura en la que se ejecutará, `configure` deduce por regla general el tipo de máquina. No obstante a veces puede imprimir un mensaje anunciando que es incapaz de averiguar sobre que tipo de máquina se está ejecutando. Cuando eso suceda use la opción *`--build=TYPE`*. *`TYPE`* puede ser tanto un nombre corto para el sistema, como por ejemplo `sun4`, o un nombre canónico de la forma:

```
CPU-COMPANY-SYSTEM
```

Donde `SYSTEM` puede tomar una de estas dos formas:

- `OS`
- `KERNEL-OS`

Vea el fichero `config.sub` para saber los posibles valores de cada campo.

Si desea usar un compilador cruzado, que genere código para una plataforma diferente de la plataforma de construcción, debería considerar especificar la plataforma “host” (la plataforma en la que los programas generados funcionarán) con la opción *`--host=TYPE`*.

## 10.6. Compartiendo los valores por defecto

Si desea compartir un conjunto de valores por defecto para todos los “scripts” `configure`, puede emplazarlos en un “shell script” llamado `config.site` que de los valores por defecto a las variables de entorno como CC, cache_file, prefix, etc. `configure` comprueba si existe, o bien el fichero *`PREFIX`*`/share/config.site` existe o el fichero *`PREFIX`*`/etc/config.site`. O también puede establecer la variable de entorno CONFIG_SITE con la ruta al fichero.


No todos los “scripts” `configure` comprueban si existe el “script” `config.site`.

## 10.7. Definiendo variables

Las variables no definidas en el fichero `config.site` pueden ser definidas en el entorno pasado a `configure`. Sin embargo, algunos paquetes pueden incluir en su distribución pDI-Tools y volver a ejecutar `configure` de nuevo durante la ejecución del `configure` principal y los valores especiales para estas variables puede perderse. Para evitar este problema, se debería inicializarlas en la llamada a `configure` usando la sintaxis ***`VAR=value`***. Por ejemplo:

```
./configure CC=/usr/local2/bin/gcc
```

hará que el **gcc** especificado se use como compilador de C (a no ser que tome el valor del fichero `config.site`).

# 10.8. Invocación de `configure`

Se puede controlar como opera `configure` con las siguientes opciones:

*`--help`*
*`-h`*

Imprime un resumen de las opciones que acepta `configure` y termina.

*`--version`*
*`-V`*

Imprime la versión de Autoconf que se uso para generar el “script” `configure` y termina.

*`--cache-file=FILE`*

Activa la cache: usa y guarda los resultados de los tests en *`FILE`*, tradicionalmente `config.cache`. *`FILE`* por defecto es `/dev/null` para desactivar la cache.

*`--config-cache`*
*`-C`*

Alias para *`--cache-file=config.cache`*.

*`--quiet`*
*`--silent`*
*`-q`*

No imprime mensajes explicando que pruebas se están realizando. Para suprimir toda la salida normal, redirijala a `/dev/null` (cualquier mensaje de error se seguirá mostrando).

*`--srcdir=DIR`*
*`-V`*

Busca el código fuente del paquete en el directorio *`DIR`*. Generalmente `configure` puede determinar este directorio automáticamente.

`configure` también aceptar algunos otros, no tan comúnmente usadas, opciones. Ejecute **`configure --help`** para más detalles.

# V. Apéndices

# Apéndice A. GNU Free Documentation License

Version 1.2, November 2002



## 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as

Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

# 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

# 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

# 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

# 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

# 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

# 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

# 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

# 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

# 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

# ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.2
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

# Apéndice B. GNU LESSER GENERAL PUBLIC LICENSE

Version 2.1, February 1999



[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

## Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author’s reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and

is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

# GNU LESSER GENERAL PUBLIC LICENSE

*TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION*

1. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

   A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

   The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

   "Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any

associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

2. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. The modified work must itself be a software library.

   b. You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

   c. You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

   d. If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

      (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

   In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to

this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

5. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

   If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

6. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

   When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

   If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

   Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

7. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

   You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

   a. Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under

Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c. Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d. If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e. Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

8. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

   a. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

   b. Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

9. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to

do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

11. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

12. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

    If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

    It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

    This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

13. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

14. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

15. 14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

*NO WARRANTY*

16. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
17. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

*END OF TERMS AND CONDITIONS*

# How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the library's name and a brief idea of what it does.>
> Copyright (C) <year> <name of author>
>
> This library is free software; you can redistribute it and/or
> modify it under the terms of the GNU Lesser General Public
> License as published by the Free Software Foundation; either
> version 2.1 of the License, or (at your option) any later version.
>
> This library is distributed in the hope that it will be useful,
> but WITHOUT ANY WARRANTY; without even the implied warranty of
> MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
> Lesser General Public License for more details.
>
> You should have received a copy of the GNU Lesser General Public
> License along with this library; if not, write to the Free Software
> Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the library
> ‘Frob’ (a library for tweaking knobs) written by James Random Hacker.
>
> <signature of Ty Coon>, 1 April 1990
> Ty Coon, President of Vice

That’s all there is to it!

# Bibliografía


[PPC64ELF] Ian Lance Taylor, *64-bit PowerPC™ ELF Application Binary Interface Supplement*, Edition 1.7.1, 1999-2004, 2003,2004, July 21, 2004.

[Teensy] Brian Raiter, *A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux: (or, "Size Is Everything").*

[SiulHacky] SiuL y Hacky, *About Introducing Your Own Code*, Fravia's page of reverse engineering.

[Hongjiu] Hongjiu Lu, *ELF: From The Programmer's Perspective*, May 17, 1995.

[Dev970FX] *Developing Embedded Software For The IBM® PowerPC™ 970FX Processor*, 2004, July 9, 2004.

[DITools2] Serra Albert, Navarro Nacho, y Cortes Toni, *DITools: Application-level Support for Dynamic Extension and Flexible Composition*, June 2000.

[DITools] Serra Albert y Navarro Nacho, *Extending the Execution Environment with DITools*, 1999.

[IA32arch1] *IA-32 INTEL® Architecture Software Developer's Manual: Volume 1: Basic Architecture*, 1997-2004, 2004.

[IA32arch2a] *IA-32 INTEL® Architecture Software Developer's Manual: Volume 2A: Instruction Set Reference, A-M*, 1997-2004, 2004.

[IA32arch2b] *IA-32 INTEL® Architecture Software Developer's Manual: Volume 2B: Instruction Set Reference, N-Z*, 1997-2004, 2004.

[IA32arch3] *IA-32 INTEL® Architecture Software Developer's Manual: Volume 3: System Programming Guide*, 1997-2004, 2004.

[GkiRobGNU] Eleftherios Gkioulekas y Marcelo Roberto Jimenez, *Learning the GNU development tools*, Edition 0.11.4, April 3, 2003.

[Stillldh] Michael Still, *Linux ld hacking howto*, 2002, 2002.

[MachOABI] slava, *Mach-O ABI*, 2002, 2002, October 21.

[POPEN32] *PowerOpen Application Binary Interface Big-Endian 32-Bit Hardware Implementation*, Edition 1.7.1, PowerOpen Association, Inc, June 30, 1994.

[PPC64Kernel] David Engebretsen, Mika Corrigan, y Peter Bergner, *PowerPC™ 64-bit Kernel Internals*.

[PPCSpec1] Joe Wetzel, Ed Silha, Cathy May, y Brad Frey, *PowerPC™ User Instruction Set Architecture: Book I*, Version 2.01, September 2003, IBM.

[PPCSpec2] Joe Wetzel, Ed Silha, Cathy May, y Brad Frey, *PowerPC™ Virtual Environment Architecture: Book II*, Version 2.01, December 2003, IBM.

[PPCSpec3] Joe Wetzel, Ed Silha, Cathy May, y Brad Frey, *PowerPC™ Operating Environment Architecture: Book III*, Version 2.01, December 2003, IBM.

[AtmelSparc7] *SPARC 7 Instruction Set*, 2002, Atmel Corporation.

[ClarkeSPARC9] Bill Clarke, *SPARC V9 Instruction-Set Syntax Specification*, The Australian National University, November 22, 2000.

[Sys5ABI] *System V Application Binary Interface*, Edition 4.1, 1990-1996, 1990-1992, The Santa Cruz Operation, Inc, March 18, 1997.

[S5ABIDRAFT] *System V Application Binary Interface - DRAFT - 24 April 2001*, 1997-2001, The Santa Cruz Operation, Inc, 24 April 2001.

[Sys5Sparc64] *System V Application Binary Interface: Generic 64-Bit Extensions*, Delta Document 1.4, SPARC International Confidential, October 9, 1996.

[Sys5i386] *System V Application Binary Interface: Intel386 Architecture Processor Supplement*, Fourth Edition, 1990-1996, 1990, The Santa Cruz Operation, Inc, March, 1997.

[Sys5mips] *System V Application Binary Interface: MIPS® RISC Processor Supplement*, 3rd Edition, 1990-1996, The Santa Cruz Operation, Inc, February, 1996.

[Sys5Sparc] *System V Application Binary Interface: SPARC Processor Supplement*, Third Edition, 1990-1996, 1990, The Santa Cruz Operation, Inc.

[Sys5SparcV9] *System V Application Binary Interface: SPARC Version 9 Processor Supplement*, Delta Document 1.35x, SPARC International Confidential, January 2, 1997.

[PPC32ELF] Steve Zucker y Kari Karhi, *System V Application Binary Interface: PowerPC™ Processor Supplement*, SunSoft, A Sun Microsystems, Inc. Business, September, 1995.

[SparcV8] SPARC International, Inc., *The SPARC Architecture Manual: Version 8*, SPARC International, Inc., 1992, 0-13-825001-4.

[SparcV9] Editado por David L. Weaver Editado por y Tom Germond, SPARC International, Inc., *The SPARC Architecture Manual: Version 9*, PTR Prentice Hall, 2000, 0-13-825001-4.

[PPCComp] Steve Hoxey, Faraydon Karim, Bill Hay, y Hank Warren, *The PowerPC™ Compiler Writer's Guide*, Warthman Associates, January, 1996, 0-9649654-0-2.

[ELF95] *Tool Interface Standard (TIS) Executable and Linking Format Specification*, Version 1.2, TIS Committee, May, 1995.

[Cederqvist] Per Cederqvist et al, *Version Management with CVS*, For CVS 1.11.1p1, Signum Support AB.

# Índice