Sistemas informáticos
Curso 2005-2006

Emulador de llamadas al sistema de Linux bajo arquitectura PowerPC

Ricardo Jesús García Gómez
Juan José Sánchez Esteban
José María Vivas Rebuelta

Dirigido por:
Luis Piñuel Moreno

Facultad de Informática
Universidad Complutense de Madrid

Índice

Pags.

Resumen

El proyecto consiste en darle a un simulador PowerPC, que ya existe, la capacidad de ejecutar programas reales Linux. Dicho simulador pertenece a la conocida herramienta de modelado de arquitecturas SimpleScalar.

Para esto, hemos modificado, en el simulador, el modulo que se ocupa de la carga del programa en memoria para su ejecución y el modulo que implementa el interfaz de llamadas al sistema.

La modificación realizada en el simulador, ha sido probada con los test de carga propios de SimpleScalar y con el paquete de benchmarks SPEC 2000, para comprobar el correcto funcionamiento del repertorio de llamadas al sistema en programas con carga real de trabajo.

La importancia de esta modificación radica en la gran difusión del simulador estático. Usando los módulos modificados del proyecto se puede comprobar el funcionamiento de cambios en otros módulos, como predictores de salto, en el conjunto Linux-PPC.

The aim of this proyect is to make an existent PowerPC simulator able to run Linux real programs. That simulator owns to the wide known tool SimpleScalar, a processors architecture simulation tool.

We have modified, in the simulator, the loader module, which loads the program into memory before running it, and the syscalls interface module.

This simulator update has been tested with SimpleScalar tests and with the benchmarks SPEC 2000, that test the correct syscall operation with real load programs.

The importance of this update lies in the great spreading of the static simulator. Using the updated modules of our proyect, the operation of changes made in oher modules, like branch predictors, can be tested on a Linux-PPC system.

Palabras clave

- ***AIX*** (Advanced Interactive eXecutive): sistema operativo desarrollado por la compañía IBM basado en Unix. Se trata de un sistema operativo diseñado principalmente para estaciones de trabajo y servidores corporativos.
- ***ELF*** (Executable and Linkable Format): es el formato estándar de archivo binario para Linux.
- ***Enlazado estático***: crea un programa autónomo sin necesidad de acceder dinámicamente a librerías durante su ejecución, pero al precio de agrandar el tamaño del ejecutable binario.
- ***Linux***: versión bajo la licencia GPL/GNU (que permite la copia y distribución junto al código fuente y sólo se paga el "medio físico") del conocido systema operativo UNIX. Es un sistema multitarea multiusuario para PC´s. Linux es una implementación del sistema operativo UNIX (uno más de entre los numerosos clónicos del histórico Unix), pero con la originalidad de ser gratuito y a la vez muy potente, que sale muy bien parado (no pocas veces victorioso) al compararlo con las versiones comerciales para sistemas de mayor envergadura y por tanto teóricamente superiores.
- ***Llamada al sistema***: función que ofrece el sistema operativo a los programas, que junto con otras forman la API, y que sirven de interfaz para la comunicación entre los programas y el sistema operativo.
- ***PowerPC***: familia de microprocesadores fruto del acuerdo de fabricación, implementación y comercialización establecido entre IBM, Apple y Motorola. Los tres integrantes del consorcio pretenden estandarizar los ordenadores de todo tipo y tamaño -desde portátiles a mainframes- en base a este procesador.
- ***SimpleScalar***: simulador desarrollado conjuntamente por las Universidades de Massachusetts y Texas, que permite emular la ejecución de programas reales para las arquitecturas de computadores actuales sin necesidad de disponer de las mismas.
- ***SPEC*** (Standard Performance Evaluation Corporation): es un consorcio sin fines de lucro que incluye a vendedores de computadoras, integradores de sistemas, universidades, grupos de investigación, publicadores y consultores de todo el mundo. Tiene dos objetivos: crear un benchmark estándar para medir la performance de computadoras y controlar y publicar los resultados de estos tests.
- ***Unix***: es una familia de sistemas operativos tanto para ordenadores personales como para mainframes. Soporta gran número de usuarios y posibilita la ejecución de distintas tareas de forma simultánea (multiusuario y multitarea). Su facilidad de adaptación a distintas plataformas y la portabilidad de las aplicaciones (está escrito en lenguaje C) que ofrece hacen que se extienda rápidamente.

1.- Introducción

El proyecto consiste en dotar a un simulador PowerPC, que ya existe, la capacidad de ejecutar programas reales Linux. Estos programas deberán ser enlazados estáticamente en una máquina PowerPC que cuente con dicho sistema. Para darle dicha capacidad, es necesario llevar a cabo una emulación del interfaz de llamadas al sistema sobre otro tipo de plataforma distinta a la simulada. El simulador debe llegar a tener la capacidad de ejecutar programas reales con una carga de trabajo representativa.

¿Por qué PowerPC?

Es una arquitectura de gran relevancia que, en la actualidad, podemos encontrar en numerosos ámbitos, desde consumibles informáticos hasta supercomputadores, en sistemas especializados o de alto rendimiento como consolas, servidores o sistemas empotrados. Los procesadores PowerPC representan la cima del funcionamiento en la categoría de un solo núcleo, además de no cesar su incremento de cuota de mercado.

¿Por qué Linux?

Por su uso en muchos de los ámbitos antes descritos. La extensión del sistema Linux en estos ámbitos se debe a que es fácil de actualizar y automatizar, poniendo gran énfasis en el trabajo en red, la estabilidad y la gran variedad de usos que tiene, además de ser el sistema que, como servidor, está experimentando un crecimiento más rápido.

El simulador

La herramienta elegida sobre la que hemos trabajado y hemos modificado ha sido SimpleScalar, la cual nos permite simular el comportamiento de programas reales ejecutados en una gama de procesadores y de sistemas modernos. SimpleScalar es capaz de emular diferentes arquitecturas de procesadores así como sus respectivos repertorios de instrucciones, para servir de esta forma de banco de pruebas y de rendimiento de programas diseñados para dichas plataformas sin necesidad de trabajar directamente sobre ellas.

La parte del simulador que se tenía que modificar era la correspondiente a la especialización de la herramienta, es decir, su cargador de archivos y las llamadas al sistema (Figura 1.1) las cuales debíamos emular. Así la modificación se centra sobre el interfaz entre la máquina cuyo comportamiento deseamos simular y la aplicación, que se ejecuta en un host de arquitectura diferente a la simulada.

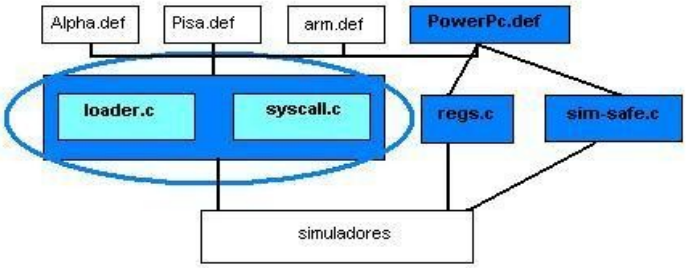


Figura 1.1: Esquemático del proyecto (Fuente: [1])

La emulación del interfaz de llamadas al sistema a la que se hacía mención en el comienzo de esta introducción se realiza de dos formas distintas según el tipo de llamada de la que se trate:

- Si la llamada no realiza cambios sobre el procesador se envía directamente al sistema operativo.
- En el caso de que si realice cambios en el procesador, debemos encargarnos de realizar esos cambios en nuestra arquitectura, es decir, actualización de memoria, registros, sistema de ficheros, señales, etc.

En este proyecto nos hemos familiarizado con el simulador, con las técnicas que se utilizan para introducir en él nuevos elementos y por supuesto también con la salida generada por la simulación concreta que utilizábamos, y de esta forma, poder realizar los cambios necesarios para la correcta ejecución de los programas Linux-PPC.

Para realizar nuestro trabajo disponíamos de un servidor Linux-Intel, en el que se ejecutaba el simulador, y de un servidor Linux-PowerPC (un MiniMac: Figura 1.2) en el que realizábamos el enlace estático de los programas que posteriormente portaríamos y ejecutaríamos en nuestro simulador para comprobar el correcto funcionamiento del programa en una plataforma distinta de la que se había usado para realizar el enlace como se requería.

Figura 1.2: Apple Mini Mac

Para la prueba de ejecución de carga real se han usado unos paquetes de prueba estandarizados y de gran difusión. Los benchmarks SPEC son paquetes de tests de funcionamiento estandarizado destinados a la métrica. Estos son programas reales a ejecutar, lo cual nos es necesario para probar que el simulador, ya dotado de la capacidad de ejecutar archivos Linux-PPC, puede ejecutar una carga real de trabajo. Como cualquier otro test deben ser enlazados estáticamente en la máquina Linux-PPC y llevados a ejecutar en la máquina que cuenta con una plataforma distinta y en la que tenemos el simulador.

Con todo esto hemos dado al simulador la capacidad de evaluar el comportamiento del conjunto procesador-sistema, en nuestro caso Linux-PowerPC, pudiendo así modelar y probar aplicaciones reales, para sistemas concretos, como los antes mencionados, que aprovechen al máximo las posibilidades del conjunto incluso antes de disponer del mismo.

2.- SimpleScalar (PowerPC/AIX)

Los procesadores modernos son maravillas de la ingeniería increíblemente complejas que están siendo más y más difíciles de evaluar cada día. El paquete de herramientas SimpleScalar implementa una simulación rápida, flexible y precisa de los procesadores modernos. La herramienta toma binarios compilados para la arquitectura de SimpleScalar y simula su ejecución en varios simuladores dados. Usamos paquetes de binarios precompilados más una versión modificada de GNU GCC que permite compilar nuestros propios test de prueba en código C.

Las ventajas de la herramientas SimpleScalar [1] son la alta flexibilidad, portabilidad y que es fácilmente ampliable. La herramienta es totalmente portable, haciendo uso únicamente de herramientas GNU. Este paquete de herramientas ha sido probado ampliamente en numerosas plataformas. El paquete de herramientas es fácilmente ampliable. Esta creado para la fácil escritura de instrucciones sin tener que volver a fijar el objetivo del compilado para cambios posteriores. La definición de instrucciones hace fácil la escritura de nuevas simulaciones y hace que, las ya escritas, sean más fáciles de ampliar. Finalmente los simuladores han sido mejorados agresivamente y pueden ejecutar códigos que se aproximen a tamaños reales en cantidades tratables de tiempo. En un Pentium Pro 200Mhz el simulador diseñado que ofrece menos detalles de la ejecución, el más rápido, simula aproximadamente cuatro millones de ciclos máquina por segundo cuando el más detallado simula 150.000 por segundo.

La versión actual de la herramienta ha sufrido una significativa mejora con respecto a las anteriores. Incluye mejor documentación, compatibilidad con más plataformas, interfaces más claros, paquetes de manejo estadístico, un depurador y una herramienta para trazar la ejecución en el simulador fuera de orden.

La herramienta SimpleScalar (versión 3.0) puede simular las arquitecturas Alpha y PISA. Para nuestro proyecto hemos utilizado una ampliación de la herramienta la cual soporta la arquitectura PowerPC en modo de trabajo big-endian [2] y [3]. Actualmente, solo se contempla la implementación de 32 bits. Futuras versiones tal vez incluyan la arquitectura de 64 bits.

2.1.- Descripción de la arquitectura PowerPC

La arquitectura PowerPC tiene características que la hacen diferente de las arquitecturas Alpha y PISA. Por ejemplo, la arquitectura Alpha tiene 215 instrucciones con 4 formatos de instrucción y la arquitectura PISA tiene 135 instrucciones con 4 formatos de instrucción. Por otra parte, la arquitectura PowerPC tiene 224 instrucciones con 15 formatos de instrucción. No todas estas instrucciones están implementadas en el simulador. A continuación describiremos las características fundamentales de la arquitectura PowerPC que se encuentran implementadas en nuestro simulador.

- **Los registros**

La arquitectura PowerPC tiene definidos 32 registros de propósito general (GPR) y 32 registros de punto flotante (FPR). Los GPRs son de 32 bits mientas que los FPRs son de 64 bits de longitud. Un registro de condición (CR) esta lógicamente dividido en 8 subcampos, de 4 bits de longitud, desde CR0 a CR7. Este registro contiene los códigos de condición.

El registro de unión (LR) se usa para transferir el flujo de programa y el registro contador (CTR) se usa para los bucles. Algunas instrucciones comparan el CTR con cero para detectar la condición de terminación de un bucle. El CTR puede también ser usado para transferir el flujo de programa.

El estado de la unidad de punto flotante se salva en un registro de 32 bits llamado registro de control de estado de punto flotante (FPSRC). Para las instrucciones en punto fijo existe un registro de 32 bits que contiene el estado de las excepciones generadas en su ejecución, llamado de excepción en punto fijo (XER).

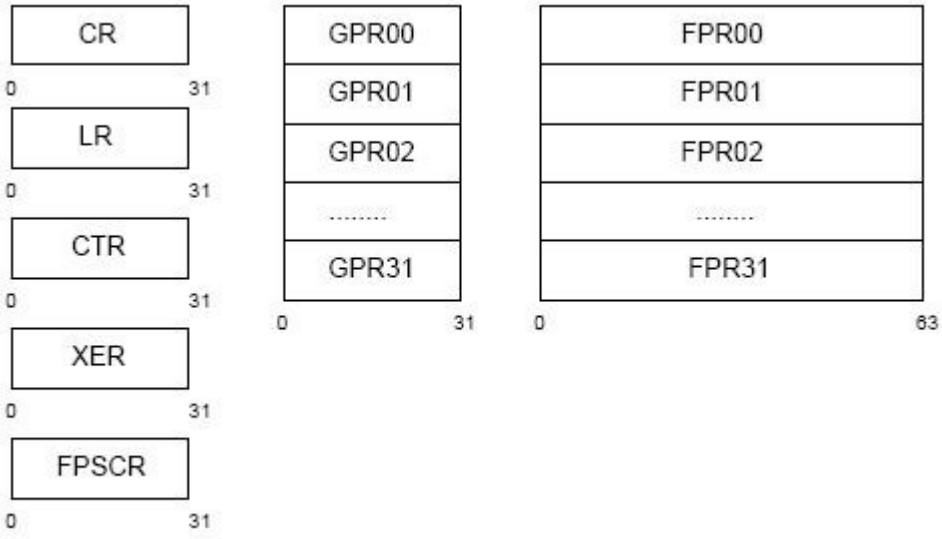


Figura 2.1: Registros de la arquitectura PowerPC (Fuente: [2])

- **Instrucciones**

PowerPC cuenta con instrucciones de 4 bits con alineamiento de palabra. Los bits 0-5 siempre dan el código de operación. Muchas de las instrucciones tienen un código de operación extendido. Algunas instrucciones tienen campos reservados que deben ser puestos a cero.

Las instrucciones ilegales no definidas invocan al sistema de manejo de instrucciones ilegales. En el simulador, durante la decodificación de instrucción, se invoca una llamada a *panic* en caso de ejecutarse una instrucción no contemplada en el repertorio de instrucciones simulado. Esta llamada detiene la ejecución del programa y muestra un mensaje de error. No todas las instrucciones definidas en la arquitectura están implementadas en el simulador. Solo están definidas las instrucciones a nivel de usuario de 32 bits.

- **Modelo de almacenamiento**

Los tipos de datos primitivos que define PowerPC son byte, media palabra y palabra. Los bytes se numeran en memoria de forma consecutiva empezando por el 0. Cada número es la dirección del byte correspondiente. Los operandos pueden ser de tamaño byte, media palabra, palabra o doble palabra, y en el caso de las instrucciones Load/Store múltiples y Move, una secuencia de bytes o palabras. La dirección del operando es la dirección de su primer byte. Las direcciones sin alineamiento están permitidas para accesos a datos.

La arquitectura PowerPC soporta tanto big-endian como little-endian en el ordenamiento de byte, aunque en el simulador solo esta contemplado el formato big-endian (el byte de menor peso en la posición más alta de memoria). Mencionar que existe una versión en formato little-endian disponible en la página del SimpleScalar.

2.2.- Descripción del simulador SimpleScalar

- **El sistema operativo**

Los dos principales problemas para portar SimpleScalar a una nueva arquitectura son:

1. El cargador de archivos
2. Las llamadas al sistema

1. El cargador de archivos

Desde que el simulador toma el archivo binario de entrada, necesitamos saber el formato de dicho archivo binario así como las tareas que realiza el cargador antes de empezar a ejecutar el programa. El cargador introduce el programa en memoria y resuelve las referencias relocables a direcciones de memoria. Las llamadas al sistema las cuales están en el archivo binario como referencias relocables son resueltas por el cargador.

También existen otros problemas menores como el paso de variables de entorno y argumentos de programa los cuales deben ser manejados por el cargador.

2. Llamadas al sistema

Como implementamos solo instrucciones a nivel de usuario, las llamadas al sistema deben ser implementadas usando la máquina en la que tenemos el simulador como un proxy para ejecutarlas. Cuando el programa simulado hace una llamada al sistema, el simulador obtiene los argumentos que se pasan a la llamada y la ejecuta llamando a la correspondiente función a nivel usuario. En ocasiones es posible que esta metodología no nos resuelva el problema como ya veremos más adelante, ya que la utilización de la herramienta SimpleScalar implica utilizar sus propios módulos simulados (memoria, registros, sistema de ficheros, sistema de señales, etc) que son totalmente independientes de los del sistema operativo. Este hecho, implica además que que no nos podamos servir totalmente de la implementación realizada por el sistema operativo de las llamadas al sistema en estos casos.

- **Implementación**

La herramienta SimpleScalar es modular y puede ser modificada para dar soporte a nuevas características de arquitecturas y micro-arquitecturas. Las diferentes estructuras simuladas como la cache, la memoria, los registros, la emulación de instrucciones y la micro arquitectura están en diferentes archivos. Todos los simuladores contenidos en la herramienta, *sim-fast*, *sim-safe*, *sim-cache*, *sim-profile*, *sim-bpred* y *sim-outorder* comparten estos archivos comunes.

- **Emulación de instrucciones**

Los cinco simuladores comparten las definiciones de instrucciones contenidas en el archivo llamado *machine.def*. Este archivo contiene el código para la emulación de las instrucciones (en C) así como el registro y dependencias funcionales de la instrucción. La corrección de las dependencias en una instrucción no afecta a su definición. Incluso si algunas dependencias estuvieran mal, el simulador funcional, el simulador de cache y la predicción de saltos funcionarían. Sin embargo, para el correcto funcionamiento del simulador de sincronización, estas dependencias deben estar bien definidas. Algunas de las instrucciones están definidas solo en el modo de 64 bits y el simulador se detiene con un error de instrucción ilegal cuando encuentra alguna de estas instrucciones.

La arquitectura PowerPC implementa la especificación aritmética de punto flotante según el estándar IEEE 751-1985. El procesador de punto flotante eleva el número de excepciones y soporta cuatro modos de redondeo. Para simular el procesador de punto flotante se adopta una aproximación de dos puntos. Dependiendo del host las instrucciones de punto flotante se ejecutan de forma nativa o son emuladas si el host es no-nativo.

Implementación nativa de punto flotante

La mayoría de las instrucciones de cómputo en punto flotante modifican un gran número de flags/campos en el registro FPSCR. Las instrucciones de cómputo son aquellas que realizan suma, resta, multiplicación, división, raíz cuadrada, redondeo, conversión, comparación, y combinaciones de estas operaciones.

En un host nativo, una emulación real del procesador en punto flotante se alcanza ejecutando la instrucción de forma nativa. Por emulación real entendemos que es la que realiza un cambio en la máquina simulada después de la ejecución igual que lo haría (en registros y memoria) en la máquina real.

Las variables de estado que se ven afectadas por una instrucción de cómputo en punto flotante son:

- Uno de los FRPs.
- El registro de estado de punto flotante.
- El registro de condición.

El archivo de registro de la máquina simulada se salva como una variable en el simulador. El FPSCR y el CR son campos en este archivo de registro. Para ejecutar una instrucción de cómputo en punto flotante se dan los siguientes pasos:

- Copiar el FPSCR de la máquina simulada en el FPSCR del host.
- Ejecutar la instrucción en punto flotante en el host en el que el simulador se está ejecutando. Esto afectara al estado del FPSCR en la máquina real. La salida generada por la ejecución se copia al archivo de registro del simulador.
- Copiar el valor del FPSCR desde el host en la estructura de datos del archivo de registro.

Implementación no nativa de punto flotante

En este caso las modificaciones sobre el FPSCR se ignoran. En un host no nativo el contenido del FPSCR se ignora el modo de redondeo del compilador que se usa para compilar el simulador permanece siempre activo. En la simulación de los SPEC se aprecia que ignoran los cambios en el FPSCR no afecta a la ejecución.

Algunas de las instrucciones de cómputo en punto flotante modifican el registro de condición (CR). De acuerdo con el resultado de la instrucción, CR1 (los segundos cuatro bits de CR) se ponen a 0, 1, 2 o 3. En el simulador se realiza comparando el resultado generado después de la ejecución. Este paso no varia entre los sistemas nativos y no nativos.

Accesos no-alineados

La arquitectura PowerPC permite las direcciones no alineadas para el acceso a datos. Para dar cabida a esto en el simulador, el alineamiento de cada lectura y escritura en memoria es comprobado y para cada no-alineamiento (lectura/escritura), las dos palabras consecutivas son leídas y los bytes correctos se juntan y se devuelven.

Toda lectura de palabra no alineada de memoria tiene como resultado dos lecturas simuladas y como consecuencia fallos de página y errores de cache simulados. Cada escritura de palabra no alineada en memoria tiene como resultado dos lecturas de memoria para leer las dos palabras alineadas en los límites afectados por la lectura, dos escrituras en memoria para escribir ambas palabras modificadas y consecuentemente los fallos de cache y fallos de página simulados que producen estos cuatro accesos.

- **Hosts little-endian**

El soporte para hosts little-endian está basado en las macros de memoria de Simplescalar 3.0. En los hosts little-endian se trabaja reordenando los bytes antes de ser escritos o leídos de la memoria simulada. Durante la ejecución del programa a la memoria se accede de cuatro maneras:

1. Cargando el programa: El cargador del sistema copia el segmento de código a memoria cuando se ha cargado el programa. En el simulador el código se lee del archivo binario y se escribe en la memoria simulada.
2. Segmentos de datos, argumentos de programa y variables de entorno: Estos valores son escritos en memoria por el cargador.
3. Llamadas al sistema: Algunas llamadas al sistema, leen o escriben datos en bufferes. Por ejemplo, la llamada al sistema *fread* lee un bloque desde un archivo y lo escribe en un buffer en memoria.
4. Instrucciones Load/Store: Instrucciones que leen o escriben valores de los registros en memoria.

Estos cuatro tipos de acceso a memoria pasan a través de la misma macro de acceso a memoria en el simulador. Para dotar de soporte para el paso entre los formatos big y little-endian, en host little-endian, los bytes escritos en memoria son reordenados antes de escribirse y después de leerse de la memoria simulada. Reordenando los bytes de esta manera se garantiza que el contenido de la memoria sea big-endian independientemente de si el host lo es o no. Reordenando los contenidos usando las macros se obtienen los valores correctos en un host little-endian cuando estos son usados en el cómputo de instrucciones de la sección de emulación del simulador. Las macros de acceso a memoria están definidas en *memory.h*.

- **Cargador**

  Como antes se ha mencionado, hay dos funciones principales del cargador:

  1. Inicializar la pila de programa (Figura 2.2): variables de entorno y argumentos de entrada.

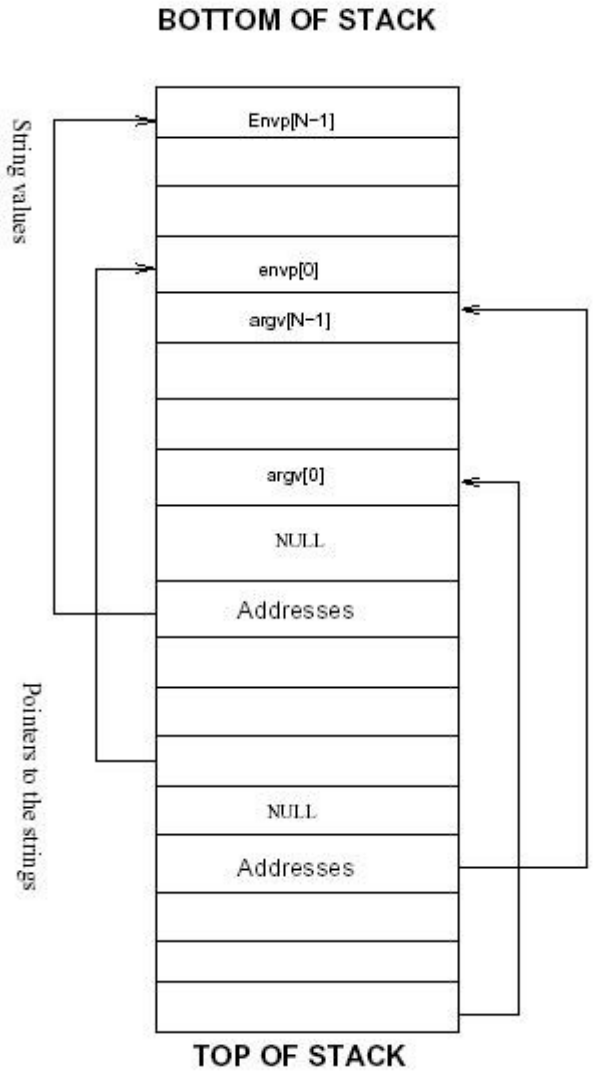


Figura 2.2: Pila de programa (Fuente: [3])

  2. Generar en memoria la imagen del programa. Asignar referencias en el segmento de programa a posiciones de memoria. Dichas referencias son objetos cuya dirección de memoria se determina en tiempo de ejecución por el cargador.

Las llamadas al sistema están presentes como referencias en el segmento del cargador. Este, determina las direcciones de estas llamadas y escribe sus valores en memoria cuando el programa se carga. Del cargador que hemos implementado para nuestro proyecto hablaremos más detalladamente en otro apartado de la memoria ya que no funciona de este modo.

- **Ejecutando las llamadas al sistema**

Una llamada al sistema es exactamente igual a una llamada a una función, excepto que, es código del sistema operativo y no es visible al usuario. En el simulador, una llamada al sistema es el resultado de una instrucción SC emulada. Primero examinamos R0 para ver de que llamada se trata. Las llamadas pasan sus argumento como otra función de usuario cualquiera, en los registros R3-R31. Los argumentos son ubicados en variables del simulador y la función de nivel usuario correspondiente a la llamada al sistema es ejecutada con los argumentos.

- **Diferentes simuladores contenidos en SimpleScalar**

El desarrollo de este proyecto se ha basado en SimpleScalar 3.0. Simplescalar esta compuesto por un conjunto de herramientas y de varios simuladores base. El código fuente está disponible de forma abierta, por lo que resulta fácilmente modificable para que los investigadores puedan adaptarlo a sus necesidades. Simplescalar permite la simulación de múltiples arquitecturas, entre ellas PowerPC.

Dentro de SimpleScalar podemos encontrar los siguientes simuladores implementados (Figura 2.3):

Sim-Safe: simulador funcional que interpreta y ejecuta las instrucciones de los programas de prueba. Muy simple, de ejecución relativamente rápida dada su simplicidad. Comprueba el alineamiento de memoria y las llamadas al sistema. Es el que hemos modificado por ser el más estructurado y por las facilidades que da para seguir las trazas de programa.

Sim-Outorder: simulador temporal complejo de un procesador superescalar fuera de orden. De simulación más lenta que el anterior, sim-outorder modela la temporización interna de un procesador ciclo a ciclo. Es el más complejo de todos y cuya ejecución es la más costosa en tiempo.

Sim-fast: Es el simulador más rápido, no modela temporización ni genera ninguna salida especial, además no realiza las comprobaciones de seguridad que si hace *sim-safe*.

Sim-profile: Proporciona gran cantidad de información estadística. Muy adecuado para el modelado de aplicaciones sin necesidad de disponer del sistema.

Sim-cache sim-cheetah: estos simuladores son muy adecuados si el efecto de la cache en tiempo de ejecución no se necesita. Realizan una simulación funcional aportando información sobre las estadísticas de la cache.

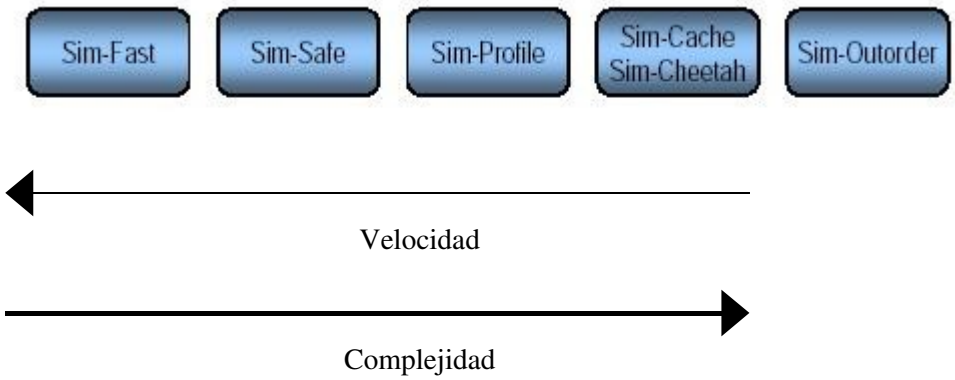


Figura 2.3: Características de los diferentes simuladores contenidos en SimpleScalar (Fuente: [3])

Breve descripción de los archivos de código del simulador:

- bitmap.h: contiene macros para la manipulación de mapas de bits.
- bpred.[c,h]: lleva la creación, funcionalidad y actualización de predictores de salto.
- cache.[c,h]: contiene funciones generales para soportar múltiples tipos de cache. Usa liked-lists para la comparación en cache de baja asociatividad y tablas hash para caches con alta asociatividad.
- dlite.[c,h]: contiene el código del depurador Dlite.
- endian.[c,h]: define un grupo de funciones simples que determinan el orden de byte y de palabra en las plataformas host y destino.
- eval.[c,h]: Contiene código para evaluar las expresiones usadas en Dlite.
- eventq.[c,h]: define macros y funciones para mantener ordenadas las colas de eventos (usada para el ordenamiento del write back).
- loader.[c,h]: carga el programa objeto en la memoria. Define el tamaño de segmento y las direcciones, define la primera pila de llamadas y obtiene el punto de comienzo del programa a ejecutar.
- main.c: realiza toda la inicialización y lanza la función principal del simulador.
- memory.[c,h]: contiene las funciones para leer desde, escribir en, inicializar y borrar el contenido de la memoria principal. La memoria se implementa como un espacio dividido, en el que cada porción es adjudicada por petición.
- misc.[c,h]: contiene numerosas funciones útiles como *panic*( ), *fatal*( ), *debug*( ), etc.
- options.[c,h]: contiene las opciones del paquete de código de SimpleScalar, se usa para procesar los argumentos de la línea de comandos y las opciones especificadas en los archivos de configuración. Las opciones se registran en una base de datos.

- ptrace.[c,h]: contiene código para producir y tomar las trazas del pipeline desde simoutorder.
- range.[c,h]: tiene el código que interpreta los comandos de rango de programa usados en Dlite.
- regs.[c,h]: contiene funciones para inicializar los archivos de registro y vaciar su contenido.
- resource.[c,h]: contiene código para manejar las unidades funcionales, divididas en clases. Las funciones definidas en el árbol crean tablas de ocupación y los grupos de recursos tomando un recurso de su grupo especifico si esta disponible.
- sim.[c,h]: contiene unas pocas declaraciones de variables externas y prototipos de funciones.
- stats.[c,h]: contiene rutinas para manejar estadísticas midiendo asi el comportamiento del programa ejecutado. Como con las opciones del paquete, los contadores son registrados por tipo con una base de datos interna. El paquete de estadísticas permite tambien medir distribuciones.
- symbol.[c,h]: maneja los símbolos de programa y las líneas de información (se usan en Dlite).
- syscall.[c,h]: contiene el código que funciona como interfaz entre las llamadas al sistema de SimpleScalar y las llamadas al sistema del host en el que se ejecute la herramienta.
- sysprobe.c: comprueba el orden de byte y palabra de la plataforma del host y genera los flags adecuados.
- version.h: define el numero de versión y la fecha de revisión de la distribución.

3.- Interfaz de llamadas de Linux

3.1.- Introducción al interfaz de llamadas de Linux

Cuando encendemos el ordenador, el primer programa que se ejecuta es el Sistema Operativo. Este programa se encarga de controlar toda la actividad que se produzca en el ordenador. Esto incluye quién se conecta, la gestión de la memoria y los discos, el uso de la CPU y la comunicación con otras máquinas.

Los programas se comunican con el Sistema Operativo por medio de llamadas al sistema. Una llamada al sistema es normalmente una demanda al sistema operativo (núcleo) para que haga una operación de hardware/sistema específica o privilegiada. Una llamada al sistema es similar a las llamadas a procedimientos. Los parámetros pasados a *syscall*() son el número de la llamada al sistema seguida por el argumento necesario. Los números de llamadas al sistema se pueden encontrar en <*linux/unistd.h*>.

En la arquitectura i386, las llamadas al sistema están limitadas a 5 argumentos además del número de llamada al sistema, debido al número de registros del procesador. Si se usa Linux en otra arquitectura se puede comprobar el contenido de <*asm/unistd.h*> para ver cuántos argumentos admite el hardware.

La llamada al sistema la invoca un proceso de usuario (o mejor dicho un proceso en modo usuario) y es servida por el núcleo (el mismo proceso en modo núcleo). Una llamada al sistema implica pasar o *saltar* del código del usuario al código del núcleo. Este *salto* conlleva un cambio en el modo del funcionamiento del procesador. El procesador debe pasar de modo usuario (acceso restringido a los recursos) a modo supervisor o privilegiado.

Las llamadas al sistema implican un cierto coste. Mientras que una llamada a una función o procedimiento puede ser llevada a cabo con unas pocas instrucciones máquina, una llamada al sistema requiere salvar el estado completo de la máquina, permitir al Sistema Operativo tomar el control de la CPU para que ejecute las funciones que tenga que realizar mediante el lanzamiento de una instrucción de interrupción, volver a recuperar el estado y finalmente devolver el control al usuario.

3.2.- Interfaz con el Sistema Operativo

Podemos distinguir fácilmente dos puntos de vista en la interfaz ofrecida por las llamadas al sistema. Por una parte tenemos la interfaz ofrecida por el núcleo del Sistema Operativo.

- Es una interfaz definida a nivel de lenguaje ensamblador.
- Depende directamente del hardware sobre el que se está ejecutando el S.O. (Registros del procesador, cómo se cambia de modo y se salta del código de usuario al código del núcleo etc.)
  Por otro lado nos encontramos con la interfaz ofrecida al programador o usuario (API)

- Todas las implementaciones de UNIX disponen de unas bibliotecas de usuario que esconden la implementación concreta de las llamadas al sistema.
- Se ofrece al programador una interfaz desde un lenguaje de alto nivel como es C.
- Presenta la ventaja añadida de la portabilidad entre distintas versiones de UNIX y entre diferentes arquitecturas.
- Todas las llamadas al sistema se encuentran documentadas en la sección 2 del manual de UNIX.

3.3.- Biblioteca de llamadas al sistema

La biblioteca que contiene todas las llamadas al sistema es la "libc". Esta biblioteca se encarga de ocultar los detalles de la interfaz de llamadas al sistema del núcleo en forma de funciones de C (pasa de nivel ensamblador a un lenguaje de alto nivel). Dichas funciones se encargan de trasladar los parámetros que reciben a los registros apropiados del preprocesador, seguidamente invocan al S.O., y finalmente recogen el código de retorno asignándolo normalmente a la variable errno.

En la arquitectura i386, Linux devuelve el código de retorno de la llamada al sistema en el registro %eax. Cuando la llamada no ha tenido éxito, el valor devuelto es negativo. Si es negativo, la biblioteca copia dicho valor sobre una variable global llamada errno y devuelve -1 como valor de retorno de la función Aun así, algunas llamadas realizadas con éxito pueden devolver un valor negativo (p.ej. lseek). La biblioteca debe ser capaz de determinar cuándo el valor devuelto es un error y tratarlo de forma adecuada. . La rutina syscall_error es la encargada de hacerlo.

La variable errno está declarada en la propia biblioteca y contiene el código de error de la última llamada que falló. Una llamada que se realice con éxito no modifica errno.

Existen casos en los que algunas supuestas llamadas al sistema ofrecidas por la biblioteca son implementadas completamente por ella misma, con lo que el núcleo del S.O. no llega a invocarse. El código de la biblioteca no pertenece al núcleo del S.O, sino que está en el espacio de direcciones del usuario. Por tanto, el código de las bibliotecas se ejecuta todavía en modo usuario.

Inicialmente se usaba la biblioteca libc pero actualmente se utiliza la biblioteca de GNU (glibc). Existen dos versiones de esta biblioteca con idéntica funcionalidad:

- libca: Se enlaza de forma estática y está incluida en el propio programa ejecutable del usuario.
- libc.so: De enlace dinámico. Se incorpora al proceso de usuario solo cuando es necesario, y su código es compartido por todos los procesos que la utilizan.

La biblioteca de GNU es realmente compleja pues los mismos fuentes se pueden compilar sobre multitud de sistemas UNIX y sobre diferentes arquitecturas.

El código de la mayoría de las llamadas al sistema se genera en tiempo de compilación, dependiendo su valor del S.O. y la arquitectura para la cual se estén compilando las funciones. O sea, el código en ensamblador que realiza la llamada al sistema no existe en un fichero sino que se crea y compila a partir de unos ficheros de especificaciones del S.O. y la arquitectura de destino.

El resultado de compilar los fuentes de la libc son los ficheros libc.a y libc.so, versión estática y dinámica de la biblioteca respectivamente. Dentro de los ficheros de biblioteca están empaquetados, como si de un fichero *arj* o *zip* se tratara, los bloques de código máquina de todas las funciones.

3.4.- Llamadas al sistema a nivel de usuario

Para ver cómo se realiza una llamada al sistema desde el modo usuario, vamos a tomar un programa ejemplo sencillo, cuyo código se muestra a continuación:

$ cat syscall.c

```
main() {
printf("Hola\n");
exit(0);
}
```

Si compilamos el programa, y vemos el código ensamblador generado, obtenemos:

```
$ gcc s syscall.c
$ cat syscall.s
```

```
    .file "syscall.c"

    .version "01.01"
    gcc2_compiled.:
      .section .rodata
    .LC0:
      .string "hola\n"
      .text
      .align 4
      .globl main
      .type main,@function
    main:
      pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      subl $12, %esp
      pushl $.LC0
      call printf
      addl $16, %esp
      leave
      ret
    .Lfe1:
      .size main,.Lfe1-main
      .ident "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.1
      2.96-81)"
```

Podemos observar cómo el programa invoca a la función "**printf**" de la biblioteca. Para ver el código generado por la biblioteca, vamos a compilar el código de nuestro programa con la biblioteca de forma estática (por defecto se utiliza la biblioteca de forma dinámica).

```
$ gcc -o syscall syscall.c -static
```

Una vez hecho esto, para poder visualizar el código utilizamos el programa *objdump* que nos permite desensamblar el código:

```
$ objdump -D syscall

080481e0 <main>:
80481e0: 55                push %ebp
80481e1: 89 e5                   mov %esp,%ebp
80481e3: 83 ec 08          sub $0x8,%esp
80481e6: 83 ec 0c          sub $0xc,%esp
80481e9: 68 a8 e2 08 08    push $0x808e2a8
80481ee: e8 85 04 00 00    call 8048678 <_IO_printf>
80481f3: 83 c4 10          add $0x10,%esp
80481f6: 83 ec 0c          sub $0xc,%esp
80481f9: 6a 00                   push $0x0
80481fb: e8 a8 02 00 00    call 80484a8 <exit>
...
;IO_printf invoca a libc_write a través de varias funciones internas de la biblioteca
...
08060e20 <__libc_write>:
8060e20: 53                push %ebx
8060e21: 8b 54 24 10             mov 0x10(%esp,1),%edx
8060e25: 8b 4c 24 0c             mov 0xc(%esp,1),%ecx
8060e29: 8b 5c 24 08             mov 0x8(%esp,1),%ebx
8060e2d: b8 04 00 00 00    mov $0x4,%eax
8060e32: cd 80                   int $0x80
8060e34: 5b                pop %ebx
8060e35: 3d 01 f0 ff ff    cmp $0xfffff001,%eax
8060e3a: 0f 83 e0 33 ff ff       jae 8054220 <__syscall_error>
8060e40: c3                ret
8060e41: eb 0d                   jmp 8060e50 <__libc_lseek>
```

3.5.- Llamadas al sistema a nivel de núcleo

En el ejemplo del apartado anterior podemos observar cómo el programa de usuario genera la interrupción 0x80 que produce un salto a la zona de código del Sistema Operativo. Concretamente se salta a la función system_call. En el proceso de salto:

- El procesador pasa de "modo usuario" a "modo supervisor".
- Se cambia el puntero de pila para que apunte a la pila del núcleo del proceso y se guardan en dicha pila algunos registros (Figura 3.1).

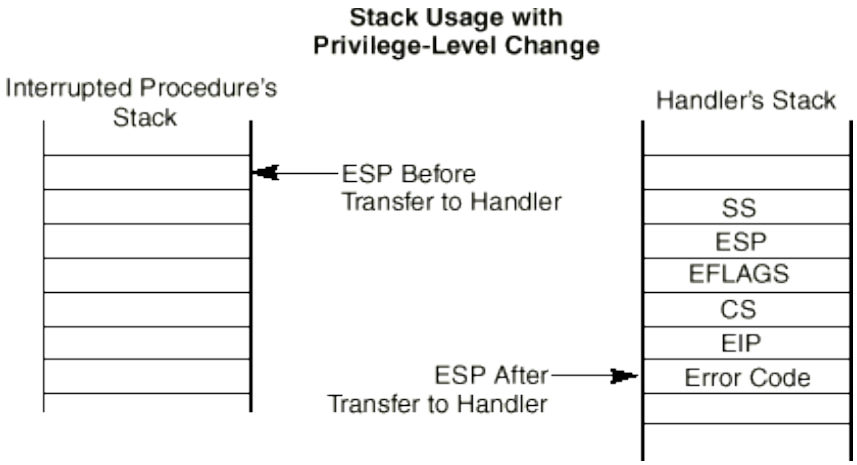


Figura 3.1: Pila de proceso (Fuente: [6])

Veamos qué ocurre en en el kernel del sistema. Para ello, examinaremos el código de la versión 2.4.20 del kernel de Linux.

En el archivo ***linux/include/asm/hw_irq.h*** podemos encontrar el vector que se utiliza para realizar las llamadas al sistema:

```
/*
 * linux/include/asm/hw_irq.h

 *
 * (C) 1992, 1993 Linus Torvalds, (C) 1997 Ingo Molnar
 *
 * moved some of the old arch/i386/kernel/irq.h to here. VY
 *
 * IRQ/IPI changes taken from work by Thomas Radke
 * <tomsoft@informatik.tu-chemnitz.de>
 */

#include <linux/config.h>
#include <asm/atomic.h>
#include <asm/irq.h>

/*
 * IDT vectors usable for external interrupt sources start
 * at 0x20:
 */
#define FIRST_EXTERNAL_VECTOR 0x20

#define SYSCALL_VECTOR 0x80

```

En el archivo ***linux/arch/i386/traps.c***, vemos como al vector 0x80 se le asigna el manejador *system_call(),* que es el que trata las llamadas al sistema.

```
*
2 * linux/arch/i386/traps.c
3 *
4 * Copyright (C) 1991, 1992 Linus Torvalds
5 *
6 * Pentium III FXSR, SSE support
7 * Gareth Hughes <gareth@valinux.com>, May 2000
8 */
9
10 /*
11 * 'Traps.c' handles hardware traps and faults after we have saved
some
12 * state in 'asm.s'.
13 */
14 #include <linux/config.h>
...
53 asmlinkage int system_call(void);
...
set_trap_gate(0,&divide_error);
967 set_trap_gate(1,&debug);
968 set_intr_gate(2,&nmi);
969 set_system_gate(3,&int3); /* int3-5 can be called from all */
970 set_system_gate(4,&overflow);
971 set_system_gate(5,&bounds);
972 set_trap_gate(6,&invalid_op);
973 set_trap_gate(7,&device_not_available);
974 set_trap_gate(8,&double_fault);
975 set_trap_gate(9,&coprocessor_segment_overrun);
976 set_trap_gate(10,&invalid_TSS);
977 set_trap_gate(11,&segment_not_present);
978 set_trap_gate(12,&stack_segment);
979 set_trap_gate(13,&general_protection);
980 set_intr_gate(14,&page_fault);
981 set_trap_gate(15,&spurious_interrupt_bug);
982 set_trap_gate(16,&coprocessor_error);
983 set_trap_gate(17,&alignment_check);
984 set_trap_gate(18,&machine_check);
985 set_trap_gate(19,&simd_coprocessor_error);
986
987 set_system_gate(SYSCALL_VECTOR,&system_call);
988
989 /*
990 * default LDT is a single-entry callgate to lcall7 for iBCS
991 * and a callgate to lcall27 for Solaris/x86 binaries
992 */
993 set_call_gate(&default_ldt[0],lcall7);
994 set_call_gate(&default_ldt[4],lcall27);
995
```

Podemos ver cómo la función set_system_gate() ajusta la entrada de la IDT (Interrupt Descriptor Table) y la copia en su lugar (0x80). En el archivo ***include/asm-x86_64/desc.h***, encontramos:

```
#define copy_16byte(dst,src) memcpy((dst), (src), 16)
83
84 static inline void _set_gate(void *adr, unsigned type, unsigned
long func, unsigned dpl, unsigned ist)
85 {
86 struct gate_struct s;
87 s.offset_low = PTR_LOW(func);
88 s.segment = __KERNEL_CS;
89 s.ist = ist;
90 s.p = 1;
91 s.dpl = dpl;
92 s.zero0 = 0;
93 s.zero1 = 0;
94 s.type = type;
95 s.offset_middle = PTR_MIDDLE(func);
96 s.offset_high = PTR_HIGH(func);
97 copy_16byte(adr, &s);
98 }
99
. . .
110 static inline void set_system_gate(int nr, void *func)
111 {
112 _set_gate(&idt_table[nr], GATE_INTERRUPT, (unsigned long)
func, 3, 0);
113 }
```

3.6.- Manejador de la llamada al sistema

El manejador de llamadas al sistema lo encontramos en el archivo ***linux/arch/i386/entry.s***, que es el punto de acceso al kenel para cualquier evento, no solo las llamadas al sistema. El fragmento que nos interesa es el correspondiente a la función system_call() y ret_from_sys_call() que implementan las llamadas al sistema y el retorno de una llamada al sistema, respectivamente:

```
/*
2 * linux/arch/i386/entry.S
3 *
4 * Copyright (C) 1991, 1992 Linus Torvalds
5 */
6
7 /*
8 * entry.S contains the system-call and fault low-level handling
routines.
9 * This also contains the timer-interrupt handler, as well as all
interrupts
10 * and faults that can result in a task-switch.
11 *
. . .
#define SAVE_ALL \
88 cld; \
89 pushl %es; \
90 pushl %ds; \
91 pushl %eax; \
92 pushl %ebp; \
93 pushl %edi; \
94 pushl %esi; \
95 pushl %edx; \
96 pushl %ecx; \
97 pushl %ebx; \
98 movl $(__KERNEL_DS),%edx; \
99 movl %edx,%ds; \
100 movl %edx,%es;
101
102 #define RESTORE_ALL \
103 popl %ebx; \
104 popl %ecx; \
105 popl %edx; \
106 popl %esi; \
107 popl %edi; \
108 popl %ebp; \
109 popl %eax; \
110 1: popl %ds; \
111 2: popl %es; \
112 addl $4,%esp; \
113 3: iret; \
. . .

202 ENTRY(system_call)
203 pushl %eax # save orig_eax
204 SAVE_ALL
205 GET_CURRENT(%ebx)
206 testb $0x02,tsk_ptrace(%ebx) # PT_TRACESYS
207 jne tracesys
208 cmpl $(NR_syscalls),%eax
209 jae badsys
210 call *SYMBOL_NAME(sys_call_table)(,%eax,4)
211 movl %eax,EAX(%esp) # save the return value
212 ENTRY(ret_from_sys_call)
```

La función system_call() comienza salvando en la pila el número de la llamada al sistema (línea 203) y todos los registros de la CPU que pueden ser utilizados por el manejador de la excepción (excepto eflags, cs, eip, ss, y esp, que fueron salvados automáticamente por la Unidad de Control). Esto lo realiza la macro **SAVE_ALL** (líneas 88 a 100). El orden en el que se insertan en la pila es importante (Figura 3.2). Los últimos en insertarse en la pila son los primeros parámetros en la declaración de la función en C.

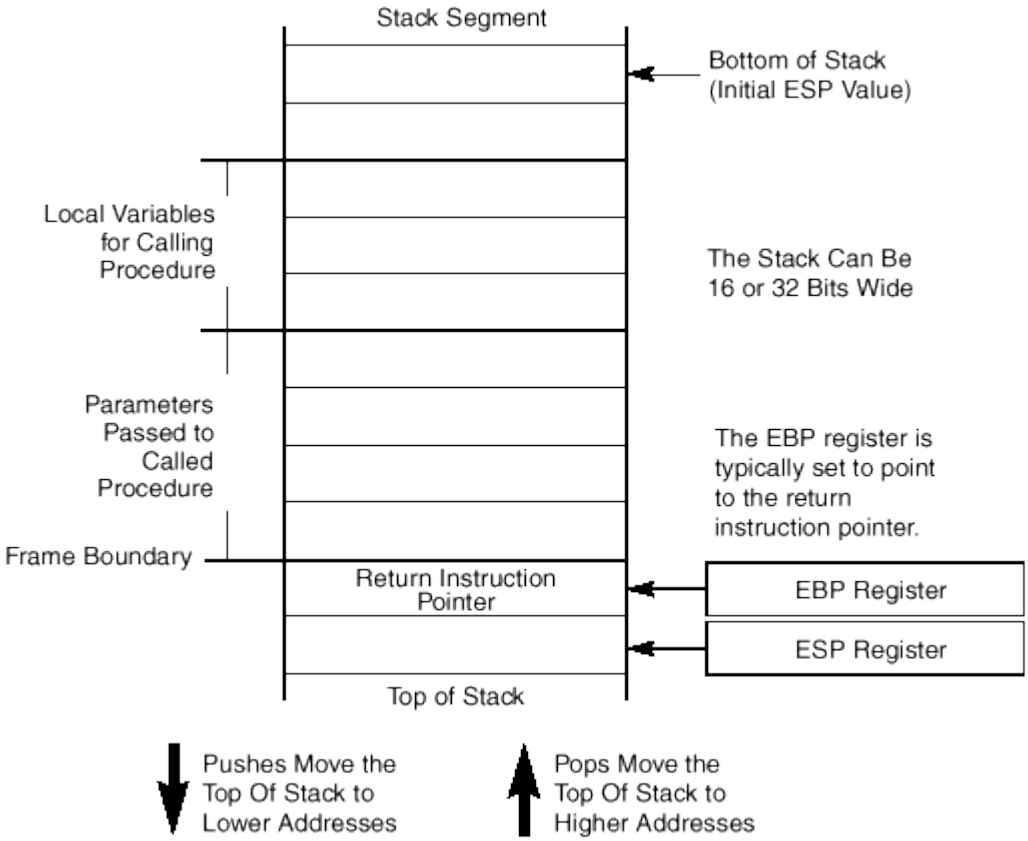


Figura 3.2: Operaciones sobre la pila de programa (Fuente: [6])

Después se realiza una comprobación para asegurarnos de que el número de llamada que pasamos desde el modo usuario es válido (línea 208). Si el número que pasamos es mayor o igual que el máximo de llamadas al sistema (NR_syscalls), entonces terminamos (línea 209). La función badsys no se recoge en el fragmento de código anterior.

Para finalizar, invocamos en la línea 210 a la rutina específica que tratará la llamada al sistema realizada por el proceso. Está rutina se encargará de ejecutar la función C cuya dirección se encuentra en la tabla sys_call_table, que es una matríz de punteros a las funciones específicas de cada llamada al sistema. La declaración call *sys_call_table(, %eax, 4) accede a la función apuntada por la entrada indicada por eax. El 4 se debe a que cada entrada de la tabla tiene un tamaño de 4 bytes. El kernel calcula la entrada multiplicando por 4 el número de la llamada al sistema. Cuando la llamada al sistema finaliza, system_call() obtiene el código de retorno de eax y lo almacena en la posición de la pila donde se salva el valor del registro eax en modo usuario.

Al finalizar la ejecución de la llamada al sistema se ejecuta el código que se encuentra en ret_from_sys_call (línea 212). Este código lleva a cabo algunas comprobaciones antes de volver a modo usuario.

- Se deshabilitan las interrupciones.
- Durante la ejecución de la llamada al sistema, el proceso ha podido cambiar de estado, y por lo tanto, haber surgido la necesidad de planificar de nuevo. Esto se comprueba y se soluciona aquí.
- Se comprueba si el proceso activo tiene señales pendientes, en cuyo caso se le envían en el código de signal_return.
- Finalmente se restauran todos los registros con RESTORE_ALL (línea 219), incluido el %eax que contiene el código de retorno y se retorna de la interrupción, reponiendo el modo del procesador y la pila a la que tiene acceso el proceso en modo usuario.

Resumiendo (Figura 3.3):

1. La biblioteca mete en los registros del procesador los parámetros de la llamada.
2. Se produce la interrupción software (trap) 0x80. El descriptor de interrupción. 0x80 apunta a la rutina system_call.
3. Se guarda en el registro %eax. El resto de los registros con SAVE_ALL.
4. Se verifica que el número de llamada al sistema corresponde con una llamada válida.
5. Se salta a la función que implementa el servicio pedido: **call *SYMBOL_NAME(sys_call_table)(,%eax,4**)
6. Si mientras se atendía la llamada al sistema se ha producido algún evento que requiera llamar al planificador, se llama en este punto.
7. Si el proceso al que se va a retornar tiene señales pendientes, se le envían ahora.
8. Se restauran los valores de los registros con RESTORE_ALL y se vuelve de la interrupción software.
9. La biblioteca comprueba si la llamada ha producido algún error, y en este caso guarda en la variable errno el valor de retorno.

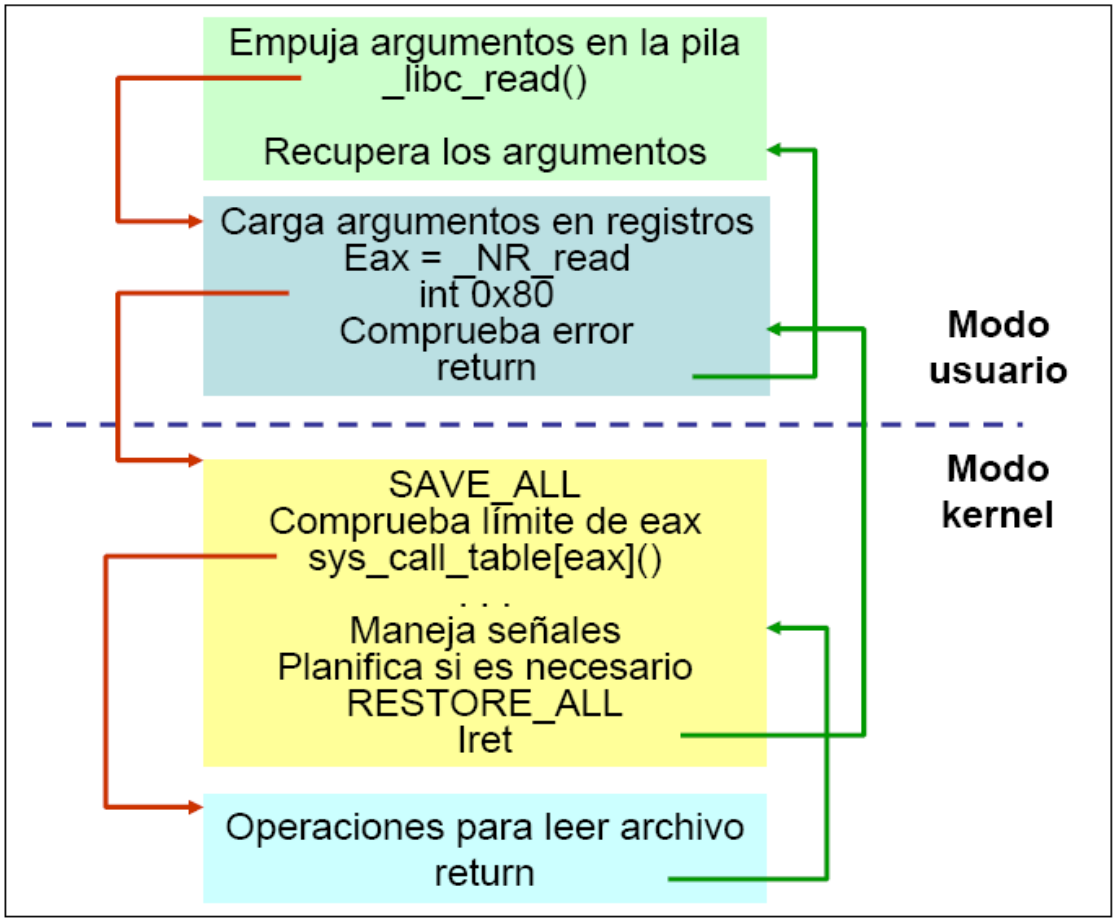


Figura 3.3: Resumen del proceso de llamadas al sistema de Linux (Fuente: [5])

4.- Desarrollo del proyecto

4.1.- Carga del ejecutable (ELF)

El primer problema con el que nos encontramos durante el desarrollo del proyecto fue la diferencia de formatos de los ficheros binarios de la arquitectura PowerPC con respecto a los ficheros ejecutables con los que trabajan los sistemas Unix. Mientras que la versión PowerPC de nuestra herramienta SimpleScalar estaba preparada para ejecutar binarios con formato XCOFF, las plataformas basadas en Unix ejecutan ficheros binarios con formato ELF. Recordemos que los programas que deseamos ejecutar son compilados en una máquina con arquitectura PowerPC/AIX (*urbion*) cuyo sistema operativo es una plataforma Unix, más concretamente Linux Debian Sarge.

XCOFF es una versión mejorada y ampliada del formato del fichero objeto de COFF definido por IBM y usado en las versiones de PowerPC/AIX, mientras que el formato **ELF** (*Executable and Linkable Format*) es un formato de archivo para ejecutables, código objeto, librerías compartidas y volcados de memoria. Fue desarrollado por el UNIX System Laboratory (*USL*) como parte de la ABI de su sistema operativo. En principio fue desarrollado para plataformas de 32 bits, a pesar de que hoy en día se usa en gran variedad de sistemas. ELF es el formato ejecutable usado mayoritariamente en los sistemas tipo UNIX como Linux, BSD, Solaris, Irix.

Evidentemente ambos formatos contienen todos los datos necesarios para cargar debidamente programas en memoria y poder ejecutarlos correctamente. Aunque el contenido (instrucciones, datos, punteros, etc) referente a los programas que ambos formatos almacenan sea prácticamente el mismo, la forma de almacenarlos y de acceder a ellos es significativamente diferente. Este hecho obligó a implementar desde cero el cargador de archivos de nuestra herramienta SimpleScalar (loader.c) para la versión PowerPC/AIX.

Para la implementación de nuestro cargador de binarios ELF dispusimos no obstante de ayuda, la herramienta SimpleScalar contiene entre los procesadores que es capaz de emular una versión para la arquitectura **ARM** para ejecutar binarios ELF. Además como el problema del cargador de archivos no estaba planteado al princinpio del desarrollo del proyecto, se nos proporcionó una versión del **SimpleScalar dinámico** [4] con capacidad para llevar acabo la emulación de ficheros binarios ELF para la arquitectura PowerPC/AIX. Dicha versión disponía de un cargador universal, con la capacidad de cargar en memoria ficheros binarios de diferentes formatos, entre ellos ELF.

Nuestro problema se redujó entonces a llevar acabo una adaptación de la versión dinámica del cargador de ficheros a nuestra herramienta. Para poder llevar acabo dicha conversión era necesario previamente documentarnos debidamente sobre la estructura y característica del formato de ficheros binarios ELF [8].

4.1.1.- Estructura del formato ELF

Los ficheros binarios ELF participan tanto en la construcción del programa (linking view) como en la ejecución del mismo (execution view), generando una imagen en memoria de nuestro ejecutable. Por conveniencia y eficiencia, el formato de los ficheros binarios ELF proporciona una visión paralela del contenido del fichero, reflejando las diferentes necesidades de aquellas actividades que son propias de los ficheros objeto de cualquier plataforma.

Para poder acceder adecuadamente a las diferentes estructuras que componen un fichero ELF fue necesario incluir en nuestra herramienta SimpleScalar un fichero de cabecera (***target-ppc/elf.h***) que contuviese todas las estructuras, tipos y macros necesarias para poder llevar acabo dicha tarea. Este fichero fue tomado directamente de la versión dinámica del SimpleScalar, aparte de que forma parte de la librería GNU de C.

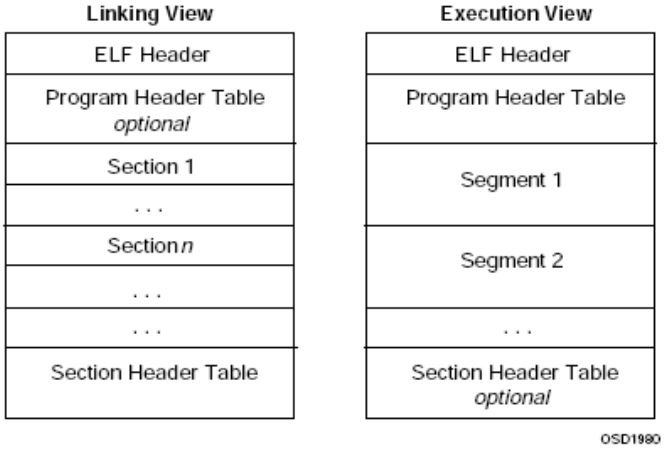


Figura 4.1: Estructura del formato ELF (Fuente: [8])

Dicho fichero de cabecera dispone de soporte para multiples plataformas, ya que no solo los sistemas Unix utilizan el formato binario ELF para sus ejecutables (p. ej. Solaris). Además está preparado para trabajar con diferentes precisiones de datos (de 8 bits a 64 bits) dependiendo de la arquitectura bajo la que nos encontremos. En nuestro caso, el emulador PowerPC/AIX de nuestra herramienta SimpleScalar trabaja con 32 bit.

En la cabecera del formato ELF (***ELF Header***) residen los campos necesarios para llevar acabo la identificación del ejecutable como un fichero binario ELF, así como un mapa de encaminamiento a las diferentes estructuras del fichero y su contenido. Hace uso de la estructura ***Elf32_Ehdr*** que contiene la siguiente información:

- *e_ident*: información de identificación ELF.
- *e_type*: tipo de fichero objeto (fichero ejecutable, librería de enlace dinámico, fichero de relocalización, fichero core, etc).
- *e_machine*: especifica la arquitectura sobre la que el fichero fue compilado (PowerPC).
- *e_version*: versión ELF utilizada.
- *e_entry*: dirección virtual de entrada del programa (dirección con la que será inicializado el registro contador de programa).
- *e_phoff*: desplazamiento en bytes para acceder a Program Header Table.
- *e_shoff*: desplazamiento en bytes para acceder a Section Header Table.
- *e_flags*: flags específicas del procesador.
- *e_ehsize*: tamaño en bytes de ELF Header.
- *e_phentsize*: tamaño en bytes de cada una de las entradas de Program Header Table.
- *e_phnum*: número de entradas de Program Header Table.
- *e_shentsize*: tamaño en bytes de cada una de las entradas de Section Header Table.
- *e_shnum*: número de entradas de Section Header Table.
- *e_shstrndx*: índice de entra de Section Header Table asociado con la Section Name String Table.

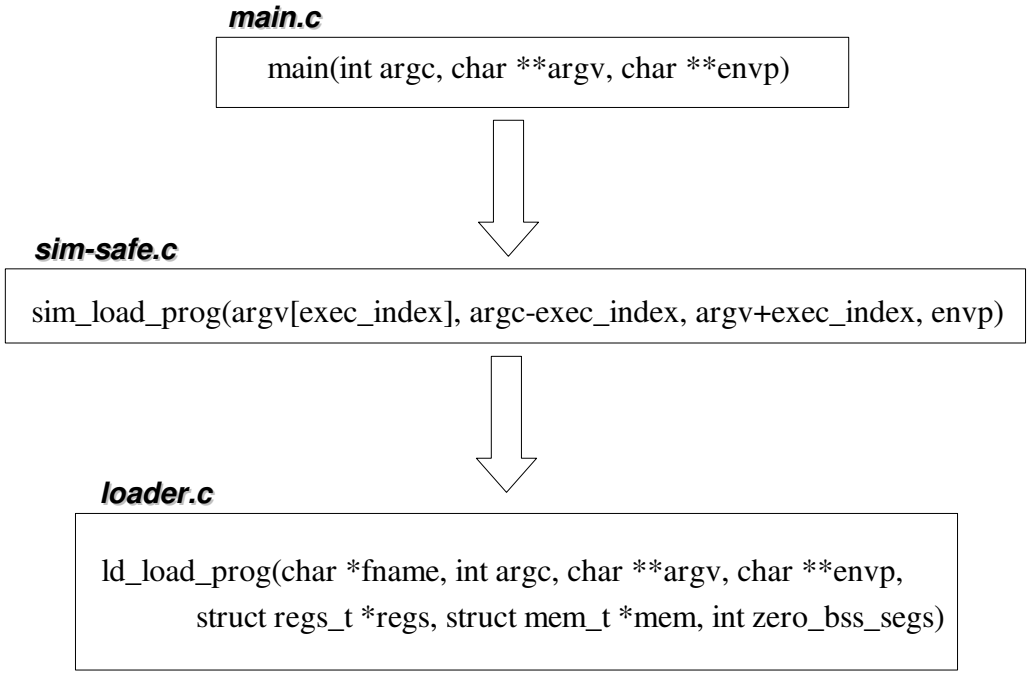
Una vez que hemos realizado la identificación de nuestro fichero binario ELF y obtenido la información de encaminamiento necesaria para acceder a las diferentes estructuras podemos desplazarnos hasta la tabla de cabecera de programa (***Program Header Tabla***). Cada una de las entradas de dicha tabla es una sección y para leerlas hacemos uso de la estructura ***Elf32_Shdr*** que contiene la siguiente información:

- *sh_name*: especifica el nombre de sección.
- *sh_type*: tipo de sección (intrucciones, datos, datos sin inicializar, segmento de carga del programa, información de enlace dinámico, tabla de símbolos, etc).
- *sh_flags*: atributos de los datos contenidos en la sección.
- *sh_addr*: dirección virtual de memoria que deben de ocupar los datos contenidos en la sección.

- *sh_offset*: desplazamiento dentro de la sección para acceder a la región de datos.
- *sh_size*: tamaño en bytes de la región de datos de la sección.
- *sh_link*: índice de enlace a una entrada de Section Header Table. Dependiendo del tipo de sección este campo será utilizado o ignorado.
- *sh_info*: información extra. Dependiendo del tipo de sección este campo será utilizado o ignorado.
- *sh_addralign*: algunas secciones tienen atributos de alineamiento de dirección. Por ejemplo: intrucciones de tamaño doble palabra.
- *sh_entsize*: algunas secciones contienen una tabla de símbolos con entradas de tamaño fijo.

4.1.2.- Implementación del cargador de ficheros

Cuando ejecutamos nuestro simulador sim-safe lo primero que se invoca es la función *main()* (main.c) que se encarga de llevar acabo operaciones de inicialización y configuración de nuestro simulador, para posteriormente invocar a *sim_main()* (sim-safe.c) que es nuestro emulador PowerPC/AIX propiamente dicho. Entre las operaciones llevadas acabo por la función *main()* se encuentra la de carga de nuestro fichero binario ELF. La función encargada de llevar acabo dicha operación es *sim_load_prog()* que se encuentra ubicada en el fichero sim-safe.c. Esta función llama a su vez a la función *ld_load_prog()* ubicada en loader.c y que se sirve de una serie de funciones y estructuras para generar la imagen de nuestro programa en memoria.



Para la lectura de nuestro fichero binario el cargador se sirve de dos estructuas, la primera de ellas es *elf_t* que se encuentra definida en elf.h, mientras que la segunda es *executable* que se encuentra definida en loader.h.

- Estructura ***elf_t***:

```
typedef struct
{
    int fd;          /* filedescriptor of elf file */
    Elf32_Ehdr head; /* elf header */
    int pt_len;      /* sections */
    Elf32_Phdr *pt;  /* section table */
} elf_t;
```

La estructura *elf_t* es capaz de almacenar toda la información relevante para nuestro simulador contenida en el fichero binario.

- *fd*: descriptor de fichero Unix de nuestro binario.
- *head*: cabecera de nuestro fichero ELF.
- *pt_len*: número de secciones.
- *pt*: array de secciones de nuestro fichero ELF.

- Estructura ***executable***:

```
typedef struct executable
{
  char * data;
  md_addr_t data_vaddress;
  int data_size;
  char * text;
  md_addr_t text_vaddress;
  int text_size;
  char * bss;
  md_addr_t bss_vaddress;
  int bss_size;
  char * program;
  md_addr_t program_vaddress;
  int program_size;
  md_addr_t text_base;
  md_addr_t data_base;
  md_addr_t data_break;
  md_addr_t prog_entry;
  md_addr_t break_point;
  /* Used to store any variables other than those above. */
  unsigned int umisc1;
  unsigned int umisc2;
  signed int smisc1;
  signed int smisc2;
} Executable;
```

El objetivo de la estructura *executable* es desglosar la información contenida en nuestro fichero binario a partir de la estructura *elf_t* para facilitarnos en la medida de lo posible nuestro trabajo. Nos permite llevar acabo la identificación de aquellas secciones cuyo contenido son comunes (instrucciones, datos, datos sin inicializar, etc) para agrupar sus contenidos en un único vector de datos, así como sus direcciones virtuales de comienzo y el tamaño completo de las mismas. A continuación vamos a mencionar aquellas campos de la estructura que utilizamos en la implementación de nuestro cargador y su función:

- *data*: vector de datos.
- *data_vaddress*: dirección virtual de comienzo de los datos de nuestro programa.
- *data_size*: tamaño en bytes de la región de datos.
- *text*: vector de instrucciones.
- *text_vaddress*: dirección virtual de comienzo de las instrucciones de nuestro programa.
- *text_size*: tamaño en bytes de la región de instrucciones.
- *bss*: vector de datos sin inicializar (generalmente son ceros).
- *bss_vaddress*: dirección virtual de comienzo de los datos sin inicializar de nuestro programa.
- *bss_size*: tamaño en bytes de la región de datos sin inicializar.
- *program*: vector del segmento de carga del programa.
- *program_vaddress*: dirección virtual de comienzo del segmento de carga del programa.
- *program_size*: tamaño en bytes de la región del segmento de carga del programa.
- *prog_entry*: dirección virtual de comienzo de la ejecución de nuestro programa.

Mencionar que la dirección virtual de comienzo de nuestro programa, la cual es cargada como dirección inicial en el registro contador de programa, no tiene porque coincidir con la dirección virtual de comienzo de la región de instrucciones. Generalmente los programas están diseñados modularmente, sirviéndose de funciones y procedimientos para su funcionamiento. El código máquina de estass funciones se encuentra ubicado antes que las instrucciones asociadas a la función principal *main ()* del programa.

Otro problema que surgió durante el desarrollo es la diferencia existente entre la arquitectura PowerPC/AIX y la arquitectura Intel (plataformas Unix sobre las que trabajamos) a la hora de almacenar información en memoria. Mientras PowerPC utiliza un formato ***big-endian***, Intel utiliza ***little-endian***. La diferencia radica en que con *big-endian* el byte de mayor peso se almacena en la dirección de memoria más baja. Sin embargo, con *little-endian* es justamente al revés, el byte de menor peso se almacena en la dirección de memoria más baja.

Así, un Long Int de 4 bytes (Byte3 Byte2 Byte1 Byte0) se almacenará con *little-endian* en memoria de la siguiente manera:

```
Dirección_Base + 0 ===> Byte0
Dirección_Base + 1 ===> Byte1
Dirección_Base + 2 ===> Byte2
Dirección_Base + 3 ===> Byte3
```

Sin embargo, con un formato *big-endian* el resultado sería:

```
Dirección_Base + 0 ===> Byte3
Dirección_Base + 1 ===> Byte2
Dirección_Base + 2 ===> Byte1
Dirección_Base + 3 ===> Byte0
```

La compilación de cualquier programa en *urbion* generará ficheros binarios ELF con datos en el formato *big-endian*. Es en el proceso de lectura del fichero ELF cuando llevamos acabo una operación de ***SWAP*** (intercambio) para solucionar el problema. Las macros encargadas de realizar el cambio de formato se encuentran definidas en *target-ppc/elf.h*.

La función que se encarga finalmente de la carga del programa propiamente dicho es la función ***ld_load_prog( )***. Veamos cual es su estructura:

- Declaración del puntero *Executable *exec*.
- Llamada a la función *ld_load_binary(char * filename, Executable * e, struct mem_t *mem)*:
    - Declaración de la variable *elf_t elf*.
    - Llamada a la función *elf_open(const char *filename, elf_t * e)*:
        - Indentificación del fichero binario ELF (operaciones swap).
        - Lectura del fichero binario ELF: cabezera y secciones (operaciones swap).
        - Devuelve *elf_t e* actualizado (Fichero binario ELF leido).
    - Operación de desglosamiento de *elf_t elf*: *Executable *exec* atualizado.
    - Actualización de *ld_prog_entry*.
    - Actualización de *ld_text_base* y *ld_text_size*.
    - Actualización de *ld_data_base* y *ld_data_size*.

- Operaciones a partir de *Executable *exec*:
    - Escritura en memoria de la región de instrucciones.
    - Escritura en memoria de la región de datos.
    - Escritura en memoria del segmento de carga del programa.
    - Escritura en memoria de la región de datos sin inicializar.
- Llamada a la función *ld_load_stack(char * fname, int argc, char ** argv, char ** envp, struct regs_t * regs, struct mem_t * mem)*:
    - Escritura en memoria de la pila de programa.
    - Actualización de *ld_stack_base* y *ld_stack_size*.
    - Actualización de *ld_environ_base*.
    - Llamada a la función *writemillicodepredecode(struct mem_t * mem)*: escritura en memoria de los datos de preprocesamiento (tomados directamente de la versión original).
- Inicialización del contador de programa: *regs_PC = ld_prog_entry*.

4.2.- Implementación de las llamadas al sistema

4.2.1.- Mecanismo de implementación de llamadas al sistema

El objetivo fundamenteal del proyecto es emular las llamadas al sistema de la arquitectura PowerPC/AIX haciendo uso del repertorio de llamadas al sistema que nos proporciona Linux. Una llamada al sistema en cualquier arquitectura es un salto a una determinada subrutina que se lleva acabo mediante una operación de interrupción. Evidentemente, antes de ejecutar dicho salto será necesario que se ejecuten una serie de operaciones previas que denominamos operaciones de preparación. En nuestro caso, estas operaciones no nos importan demasiado, lo que si que nos importa es ser capaces de recoger toda la información necesaria (argumentos de entrada de la llamada al sistema) para poder realizar correctamente la emulación de la llamada.

La invocación a una llamada al sistema se realizar mediante una instrucción específica del repertorio de instrucciones. Veamos como está definida dicha intrucción en el repertorio de intrucciones de nuestro simulador SimpleScalar (*machine.def*):

```
#define SC_IMPL
{
      dosyscall(&regs, mem, mem_access);
}

DEFINST(SC, 0x11, "sc", "",
      FUClass_NA,  F_TRAP,
       DNA, DNA, DNA, DNA, DNA,
      DNA, DNA,DNA, DNA, DNA)
```

La instrucción asociada a las llamadas al sistema se denota por SC (*syscall*) y su código de opercación asociado es 0x11 (notación hexadecimal). La emulación de dicha instrucción se realiza llamando a la función ***dosyscall( )*** (*syscall.c*). Esta función recibe como parámetros tanto los registros como la memoria de nuestro simulador. Entre las operaciones previas a la instrucción SC se encuentra la carga en el registo regs_R[0] del identificador de la llamada al sistema que se desea ejecutar.

Existen llamadas al sistema que se pueden resolver directamente haciendo uso del repertorio de llamadas al sistema de Linux, mientras que existen otras que involucran cambios en los registros o memoria de nuestro emulador, y debemos de tener muy en cuenta que no es lo mismo los registros o la memoria de nuestro sistema opetativo Linux que la de nuestra herramienta simuladora SimpleScalar. Este hecho, conlleva que en muchas ocasiones deberemos de hacer uso de las macros y funciones definidas en SimpleScalar para modificar los registros o memoria, o bien implementar nuestras propias funciones.

Una vez se llama a la función *dosyscall( )*, esta examina el contenido del registro regs_R[0] y en función del valor contenido en el mismo (identificador de llamada al sistema) se llama a la función encargada de simularla o bien de imprimir un mensaje por pantalla de que dicha llamada al sistema no se encuentra implementada por el simulador. Todas estas funciones se encuentran implementadas en el fichero *syscall.c* y todas ellas tienen el siguiente formato:

```
void syscall_nombre(struct regs_t *regs, struct mem_t *mem)
{
      ....
}
```

Mencionar que la emulación de las llamadas al sistema ha requerido llevar acabo la implementación de un ***sistema de ficheros*** y de **señales** simulados, los cuales ya explicaremos más adelante.

La implementación propiamente dicha de una llamada al sistema se puede dividir en tres fases diferentes:

1. Captura del estado arquitectónico: en el momento en que se produce la llamada al sistema mediante la instrucción SC, todos los argumentos e información necesarias para implementar dicha llamada se encuentra en nuestro sistema de registros enteros (*regs_R[]*).
2. Modificaciones en los sistemas de emulación: muchas llamadas requieren llevar acabo modificaciones en los sistemas de simulación de nuestra herramienta SimpleScalar (registros, memoria, ficheros, señales, etc).
3. Llamada al sistema Linux: emulación de la llamada al sistema sirviéndonos del repertorio de llamadas de Linux. Estas llamadas pueden consultadas en el *manual 2 de Linux* para conocer su funcionamiento (parámetros de entrada, valor devuelto, etc).

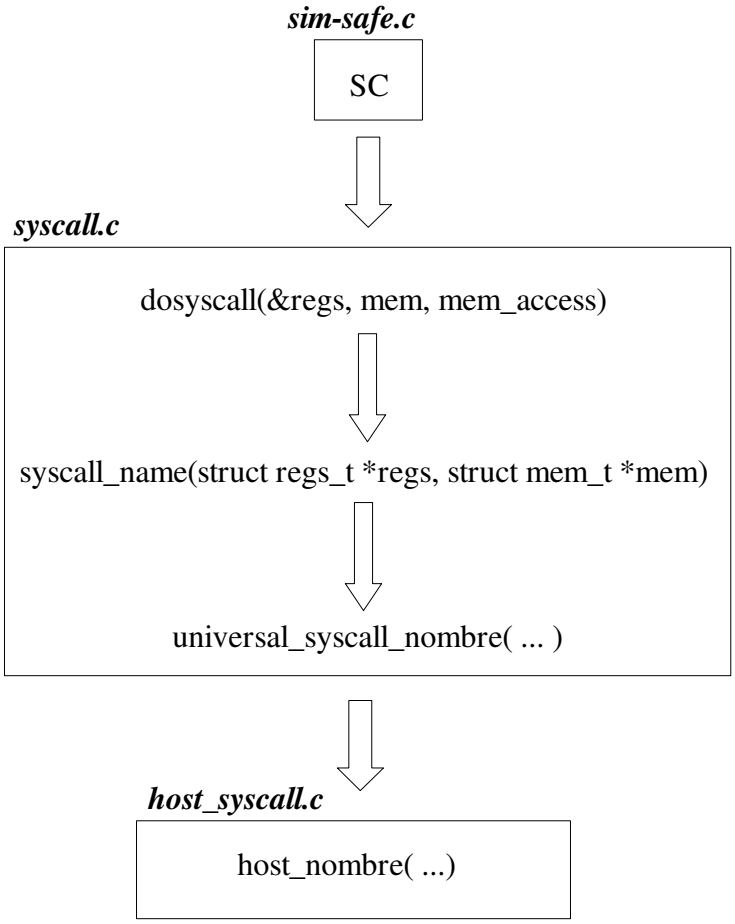
También es importante mencionar que las tres fases anteriormente mencionadas no son rígidas. Existen llamadas que es posible que se salten alguna de las fases porque no la necesiten.

La metodología utilizada para la implemetación de las llamadas al sistema es la misma que usa la versión dinámica del SimpleScalar. Esta versión implementa un sistema ***universal*** de llamadas al sistema independiente de la arquitectura emulada. Para ello implementa una serie de tipos, estructuras y macros que son comunmente utilizadas por todas las arquitecturas. Su gran virtud es la de definir funciones universales que permiten modificar los diferentes sistemas simulados de nuestro emulador. Estas funciones han sido fácilmente utilizables en nuestra versión sin más que modificarlas ligeramente o bien incluyéndolas directamente en nuestra implementación. Todos estos tipos, estructuras y macros han sido incluidos directamente por comodidad en *syscall.h*. En el caso de las funciones asociadas a operaciones han sido incluidas en *syscall.c* y *memory.c* (las asociadas con operaciones en memoria).

Así mismo cada función *syscall_nombre( )* tiene asociada una función *universal _syscall_name( )* que es la encargada de hacer uso de las funciones universales anteriormente mencionadas. Posteriormente, aquellas llamadas al sistema que requieren hacer uso de las llamadas al sistema proporcionadas por Linux realizan la llamada *host_nombre( )* desde su función universal. Estas funciones están implementadas en el fichero *host_syscall.c*.

Resumiendo, a partir del momento en que se llama a la función *syscall_nombre( )* el procedimiento para implementar las llamadas al sistema es el siguiente:

- *syscall_nombre( )*: captura del estado arquitectónico. Llamada a la función *universal_ syscall_ name( )* mediante el uso de la macro *DOSYSCALL* (*syscall.h*) que se encarga de registrar posibles errores e informar de ellos al usuario.
- *universal_syscall( )*: realiza las modificaciones necesarias en los diferentes sistemas de nuestro simular. Llamada a *host_syscall( )*. Llamada a *sys_return( )* (*syscall.c*): se encarga de modificar el registro regs_R[3] para informar al flujo del programa si la llamada ha tenido éxito, en caso contrario devuelve -1 en dicho registro.
- *host_syscall( )*: llamada a la función correspondiente del repertorio de Linux.



Exiten llamadas al sistema que obtienen información poco relevante e incluso que pasa desapercibida durante la ejecución de un programa, mientras que por contra la gran mayoría de ellas son esenciales para el correcto funcionamiento del programa. Para saber las llamadas al sistema que ejecuta cada programa se definio la constante *-DEBUG_PRINT_SYSCALLS* dentro del fichero de compilación *Makefile*. Si dicha variable se encuentra activa, cada llamada al sistema que ejecute nuestro programa mostrará por pantalla información acerca de la misma (información del sistema, señales enviadas, tratamiento de ficheros, valores devueltos, etc).

4.2.2.- Descripción de las llamadas al sistema implementadas

En total se han implementado 48 llamadas al sistema. A continuación vamos a explicar brevemente cada una de ellas:

- ***Syscall uname*** (Código de identificación 122): obtiene el nombre e información acerca del actual kernel del sistema. Hace uso de la estructura universal *UniversalUnam*. Llama a *universal_syscall_uname( )* que a su vez llama *a host_uname( )* la cual hace uso de la llamada al sistema de Linux *uname*. La información devuelta por dicha función se copia en la estructura *UniversalUnam* y se copia en la memoria simulada en la dirección de memoria especificada en regs_R[3] (este funcionamiento es común a muchas de las llamadas implementadas) la información más relevante (sistema operativo, nombre del nodo, version del sistema operativo, fecha de la última actualización, tipo de arquitectura de la máquina, etc).

- ***Syscall getpid*** (20): obtiene el identificador de proceso. Llama a *universal_syscall_getpid*( ) la cual llama a *host_getpid*( ) que hace uso de la llamada al sistema de Linux *getpid*. Su única función es la de mostrar por pantalla dicho identificador a modo informativo.

- ***Syscall getgidx*** (16): obtiene el identificador de grupo del proceso. Llama a *universal _syscall_getgidx*( ) la cual llama a *host_getgidx*( ) que hace uso de la llamada al sistema de Linux *getgid*. Su única función es la de mostrar por pantalla dicho identificador a modo informativo.

- ***Syscall getuidx*** (17): obtiene el identificador real de usuario del proceso. Llama a *universal _syscall_getuidx*( ) la cual llama a *host_getuid*( ) que hace uso de la llamada al sistema de Linux *getuid*. Su única función es la de mostrar por pantalla dicho identificador a modo informativo.

- ***Syscall geteuid*** (49): obtiene el identificador efectivo de usuario del proceso. Llama a *universal_syscall_geteuid*( ) la cual llama a *host_geteuid*( ) que hace uso de la llamada al sistema de Linux *geteuid*. Su única función es la de mostrar por pantalla dicho identificador a modo informativo.

- ***Syscall getuid*** (24): obtiene el identificador real de usuario del proceso. Llama a *universal _syscall_getuid*( ) la cual llama a *host_getuid*( ) que hace uso de la llamada al sistema de Linux *getuid*. Su única función es la de mostrar por pantalla dicho identificador a modo informativo.

- ***Syscall getegid*** (50): obtiene el identificador de grupo del proceso. Llama a *universal _syscall_getegid*( ) la cual llama a *host_getegid*( ) que hace uso de la llamada al sistema de Linux *getegid*. Su única función es la de mostrar por pantalla dicho identificador a modo informativo.

- ***Syscall getgid*** (47): obtiene el identificadorde grupo del proceso. Llama a *universal _syscall_getgid*( ) la cual llama a *host_getgid*( ) que hace uso de la llamada al sistema de Linux *getgid*. Su única función es la de mostrar por pantalla dicho identificador a modo informativo.

- ***Syscall sbrk*** (66): cambia el tamaño del segmento de datos. Llama a *universal _syscall_sbrk*( ) la cual llama a la función *do_brk*( ) (memory.c). Esta última función recibe como parámetro de entrada el contenido del registro regs_R[3] que indica el número de bytes en que debe de ser incrementado el tamaño del segmento de datos y realiza las oportunas modificaciones sobre la memoria simulada.

- ***Syscall brk*** (45): cambia el tamaño del segmento de datos. Llama a *universal _syscall_brk*( ) Esta última función recibe como parámetro de entrada el contenido del registro regs_R[3] que indica la dirección de memoria final del segmento de datos. Si la dirección de memoria es 0, entonces el tamaño del segmento de datos no se ve modificado, mientras que si la dirección supera la cota fijada por *ld_break_point* (final del segmento de datos) se llama a la función *do_brk*( ) pasándole como paramétro el número de bytes en que es necesario aumentar el tamaño del segmento de datos para que lleve acabo los cambios necesarios sobre la memoria simulada.

- ***Syscall open*** (5): abre y posibilita la creación de un fichero o dispositivo.
    - Regs_R[3]: dirección virtual de memoria en donde se encuentra el nombre del fichero.
    - Regs_R[4]: flags.
    - Regs_R[5]: modo de apertura.

  Llama a la función *universal_syscall_open*( ) que se sirve de las funciones *ofd_map_name*( ) y *ofd_open*( ) de nuestro sistema de ficheros simulado. La función *ofd_map_name*( ) asigna un prefijo válido al fichero en nuestra base de datos. Se llama a *host_open*( ) que hace uso de la llamada al sistema de Linux *open*. Finalmente se llama a *ofd_open*( ) para dar de alta al fichero en la base de datos de nuestro sistema de ficheros simulado.

- ***Syscall close*** (6): cierra un descriptor de fichero.
    - Regs_R[3]: descriptor de fichero.

  Llama a la función *universal_syscall_close*( ) que se sirve de las funciones *ofd_close*( ) y *ofd_get*( ) para eliminar el fichero de la base de datos del sistema de ficheros simulado y llamar a la función *host_close*( ). Esta función hace uso de la llamada al sistema de Linux *close*.

- ***Syscall creat*** (8): crea un fichero o dispositivo.
    - Regs_R[3]: dirección virtual de memoria en donde se encuentra el nombre del fichero.
    - Regs_R[4]: modo de creación.

  Llama a la función *universal_syscall_creat*( ) que se sirve de las funciones *ofd_map_name*( ) y *ofd_open*( ) de nuestro sistema de ficheros simulado. La función *ofd_map_name*( ) asigna un prefijo válido al fichero en nuestra base de datos. Se llama a *host_creat*( ) que hace uso de la llamada al sistema de Linux *creat*. Finalmente se llama a *ofd_open*( ) para dar de alta al fichero en la base de datos de nuestro sistema de ficheros simulado.

- ***Syscall readlink*** (85): leer el valor de un enlace simbólico.
    - Regs_R[3]: dirección virtual de memoria en el que se encuentra el nombre del fichero que se va a enlazar.
    - Regs_R[4]: dirección virtual de memoria del fichero enlace.
    - Regs_R[5]: tamaño del buffer del enlace.

  Llama a la función *host_readlink*( ) que copia el contenido del fichero enlazado en un buffer. Finalmente se copia el buffer en memoria para crear la imagen del enlace.

- ***Syscall unlink*** (10): elimina un enlace simbólico.
    - Regs_R[3]: dirección virtual de memoria donde se encuentra el nombre del enlace simbólico.

  Llama a la función *universal_syscall_unlink*( ) que se sirve de la función *ofd_map_name*( ) para obtener el prefijo asignado al fichero y llamar a *host_unlik*( ). Esta función hace uso de la llamada al sistema *unlik* para eliminar el enlace. En este caso hay que tener en cuenta que los enlaces simbólicos no se almacenan en la base de datos de nuestro sistema de fichero simulado.

- ***Syscall access*** (33): llamada al sistema ignorada.

- ***Syscall times*** (43): obtiene los tiempos de proceso (tiempo de usuario, tiempo de sistema y tiempo de usuario y de sistema de muerte de los procesos hijos). Llama a la función *universal_syscall_times*( ) la cual hace uso de la estructura *UniversalTms* para llamar a *host_times*( ). Esta función hace uso de la llamada al sistema de Linux *times*. Finalmente se copia la información obtenida en la dirección de memoria indicada por el registro regs_R[3].

- ***Syscall rename*** (38): cambia el nombre de un fichero.
    - Regs_R[3]: dirección virtual de memoria donde se encuentra el nombre del fichero.
    - Regs_R[4]: dirección de memoria virtual donde se encuentra el nuevo nombre del fichero.

  Llama a la función *universal_syscall_rename*( ) la cual realiza dos llamadas a la función *ofd_map_name*( ) para obtener los prefijos de sendos nombres de fichero. Finalmente llama a la función *host_rename*( ). Esta función hace uso de la llamada al sistema de Linux *rename*. Debemos de recordar que el cambio de nombre no influye en el descriptor de fichero almacenado en la base de datos del sistema de ficheros simulado.

- ***Syscall gettimerofday*** (78): obtiene el tiempo actual y para ello se sirve de dos estructuras definidas en <time.h> y <sys/time.h> respectivamente:

```
struct timeval {
   time_t  tv_sec;         /* seconds */
   suseconds_t  tv_usec;   /* microseconds */
};

struct timezone {
   int  tz_minuteswest;  /* minutes W of Greenwich */
   int  tz_dsttime;      /* type of dst correction */
};
```

  Llama a la función *universal_gettimerofday*( ) y haciendo uso de las estructuras *UniversalTimeval* y *UniversalTimezone* llama a la función *host_gettimerofday*( ). Esta función hace uso de la llamada al sistema *gettimerofday*( ). Finalmente copia en memoria las estructuras *timeval* y *timezone* en las direcciones de memoria especificadas en los registros regs_R[3] y regs_R[4] respectivamente.

- ***Syscall gettimerid*** (27): llamada al sistema ignorada.

- ***Syscall incinterval*** (25): permite la medición de un intervalo de tiempo. Para ello fue necesario modificar *sim-safe.c* para proporcionarle un sistema de captura de tiempos mediante la utilización de la estructura *itimerstruc* (syscall.h).

```
struct sc_timeval {
       sword_t  tv_sec;  /* 32bit time_t value */
       sword_t  tv_nsec;
};

struct itimerstruc_t {
      struct  sc_timeval  it_interval;  /* timer interval */
      struct  sc_timeval  it_value;     /* current value */
};

struct itimerstruc_t timerinfo;
```

En el momento de producirse la llamada al sistema *incinterval* en el registro regs_R[MD_REG_A1] se encuentran todos los datos que necesitamos.

```
it_value.tv_sec = regs_R[MD_REG_A1]
it_value.tv_nsec = regs_R[MD_REG_A1] + 4
it_interval.tv_sec = regs_R[MD_REG_A1] + 8
it_interval.tv_nsec = regs_R[MD_REG_A1] + 12
```

La función *syscall_incinterval*( ) llama a la función *universal_syscall_incinterval*( ) cuyo función es imprimir por pantalla dichos valores a modo de información en el caso de que la constante *DEBUG_PRINT_SYSCALLS* este definida.

- ***Syscall setitimer*** (104): permite la medición de un intervalo de tiempo. Su funcionamiento es prácticamente idéntico al de *incinterval*. La diferencia radica en que en este caso se utiliza la estructura *universal_itimerval*. Los datos que antes obteniamos del registro regs_R[MD_REG_A1] ahora los obtenemos accediendo a memoria con dicha estructura a la dirección especificada por el registro regs_R[3]. Finalmente se llama a universal _syscall_incinterval( ).

- ***Syscall sigcleanup*** (13): llamada al sistema ignorada.

- ***Syscall sigaction*** (67): llamada al sistema para cambiar la acción llevada acabo por un proceso al recibir una señal específica.
    - Regs_R[3]: identificador de señal de la nueva señal.
    - Regs_R[4]: dirección virtual de memoria en donde se encuentra ubicada la nueva señal.
    - Regs_R[5]: dirección virtual de memoria en donde se encuentra ubicada la antigua señal.

  Se hace uso de la estructura *UniversalSigaction* para la lectura de memoria de la nueva y antigua señal. Llamada a *universal_syscall_sigaction*( ) para la sustitución en el sistema de señales simulado de la nueva señal por la antigua.

- ***Syscall rt_sigaction*** (173): su comportamiento es el mismo que el que proporciona la llamada al sistema *sigaction*( ). La diferencia radica en que en este caso es necesario que el registro regs_R[6] contenga el tamaño por defecto de una señal en el sistema Linux (LINUX_SIGSET_SIZE, definida en *signal.h*), en caso contrario se producirá un error.
- ***Syscall sigprocmask*** (126): llamada utilizada para cambiar la lista de señales bloqueadas actualmente mediante la modificación de la máscara del sistema de señales simulado.
    - Regs_R[3]: flags de acción (bloquear, desbloquear o cambio de máscara).
    - Regs_R[4]: contiene la nueva máscara.
    - Regs_R[5]: dirección virtual de memoria en donde se encuentra la máscara actual del sistema de señales.

  Haciendo uso de la estructura *UniversalSigset* se lee de memoria la máscara actual del sistema de señales simulado y en función de la nueva máscara y de los flags de acción se realizan los cambios pertinentes en la máscara del simulador.

- ***Syscall rt_sigprocmask*** (174): su comportamiento es el mismo que el que proporciona la llamada al sistema *sigprocmask*( ). La diferencia radica en que en este caso es necesario que el registro regs_R[6] contenga el tamaño por defecto de una señal en el sistema Linux (LINUX_SIGSET_SIZE, definida en *signal.h*), en caso contrario se producirá un error.

- ***Syscall sigaltstack*** (185): llamada al sistema utilizada para modificar el estado de la pila de señales en espera.
    - Regs_R[3]: dirección virtual de memoria del nuevo estado de la pila de señales.
    - Regs_R[4]: dirección virtual de memoria del estado actual de la pila de señales.

  Lectura de memoria del estado actual de la pila de señales en espera mediante la estructura *universal_sigaltstack*. Llamada a *universal_syscall_sigaltstack*( ) para modificar el tamaño de la pila de señales en espera de nuestro sistema de señales simulado. Comentar que la implementación realizada no soporta los flags de control de la llamada.

- ***Syscall lseek*** (19): modificación de la posición del puntero de lectura/escritura de un fichero.
    - Regs_R[3]: descriptor de fichero.
    - Regs_R[4]: número de bytes del desplazamiento.
    - Regs_R[5]: punto de partida del desplazamiento (*SEEK_SET, SEEK_CUR, SEEK_END*). Para más información mirar en la página 2 del manual (*man 2 lseek*).

  Llamada a *universal_syscall_lseek*( ) la cual llama a *ofd_get(*) y *host_lseek*( ). Esta función hace uso de la llamada al sistema de Linux *lseek*.

- ***Syscall klseek*** (140): su funcionamiento es el mismo que el de la llamada al sistema *lseek*. Se llama a la función *universal_syscall_klseek*( ) que a su vez llama a *ofd_get*( ) y *host_lseek*( ). La única diferencia radica en que en este caso el desplazamiento puede tomar valores mayores, haciendo uso para ello de los registro regs_R[4] y regs_R[5]. En este caso el punto de partida del desplazamiento de toma del registro regs_R[7].

- ***Syscall setrlimit*** (75): modifica los límites del recurso. Llamada al sistema parcialmente implementada. Toma del registro regs_R[3] el recurso y llama a *universal_syscall _setrlimit*( ) para mostrar por pantalla el recurso.

- ***Syscall getrlimit*** (76): obtiene los límites del recurso.
    - Regs_R[3]: identificador de recurso.
    - Regs_R[4]: dirección virtual de memoria en donde se almacenará el límite del recurso.

  Llamada a *universal_syscall_getrlimit*( ) que hace uso de la estructura *UniversalRlimit* para llamar a *host_getrlimit*( ). Una vez obtenido el límite creamos la imagen en la memoria.

- ***Syscall kread*** (3): lectura desde un descriptor de fichero.
    - Regs_R[3]: descriptor de fichero.
    - Regs_R[4]: dirección virtual de memoria en la que vamos a crear la imagen de lo leido.
    - Regs_R[5]: tamaño en bytes de lo que vamos a leer.

  Llamada a *universal_syscall_kread*( ) la cual llama a *ofd_get(*) y *host_read*( ) que copia en un buffer la información leida del fichero haciendo uso de la llamada al sistema de Linux *read.* Finalmente escribimos en memoria lo leido en la dirección de memoria especificada.

- ***Syscall kwrite*** (4): escribe en un desciptor de fichero.
    - Regs_R[3]: descriptor del fichero en que vamos a escribir.
    - Regs_R[4]: contiene una dirección virtual de memoria y que es la ubicación en memoria de lo que deseamos escribir.
    - Regs_R[5]: tamaño en bytes de lo que vamos a escribir.

  Llama a *universal_syscall_kwrite*( ) que se encarga de leer de memoria a un buffer el contenido de lo que deseamos escribir. Finalmente se llama a *host_write*( ) la cual hace uso de la llamada al sistema de Linux *write* para escribir el buffer en el descriptor de fichero especificado.

- ***Syscall mmap*** (90): mapea archivos o dispositivos en memoria.
    - Regs_R[3]: dirección virtual de memoria en la que va a ser mapeado el fichero.
    - Regs_R[4]: tamaño en bytes de la información a copiar.
    - Regs_R[5]: máscara de protección (la memoria simulada de SimpleScalar no dispone de protección de páginas de memoria, con lo que será ignorada).
    - Regs_R[6]: flags (*PROT_EXEC*, *PROT_READ*, *PROT_WRITE* o *PROT_NONE*).
    - Regs_R[7]: descriptor de fichero.
    - Regs_R[8]: desplazamiento dentro del fichero.

  Llamada a *universal_syscall_mmap*( ). Esta función se sirve de la funcióm *mem_mmap*( ) (*memory.c*) que se encarga de reservar e inicializar las páginas de memoria necesarias. Posteriormente se llama a *host_lseek*( ), para posicionar debidamente el puntero de lectura en nuestro fichero, y a *host_read*( ) para leer del fichero el fragmento deseado a un buffer. Finalmente se escribe en memoria el buffer creandose la imagen deseada.

- ***Syscall munmap*** (91): elimina la imagen en memoria de un archivo o dispositivo.
    - Regs_R[3]: dirección virtual de la imagen en memoria.
    - Regs_R[4]: tamaño en bytes de la imagen.

  Llamada a *universal_syscall_munmap*( ) la cual llama a *mem_munmap*( ) (*memory.c*) que se encarga de eliminar la imagen de memoria y de liberar correctamente las páginas de memoria.

- ***Syscall writev*** (146): escribe datos de múltiples bufferes. Se define la estructura *iovec* (*syscall.h*) que es la encargada de definir la posición de memoria donde comienzan los datos que queremos escribir así como su tamaño en bytes.

```
struct iovec {
     char  *iov_base;  /* Base address. */
     size_t  iov_len;  /* Length. */
};
```

    - Regs_R[3]: descriptor de fichero en que se va a escribir.
    - Regs_R[4]: dirección virtual de memoria en la que se encuentra iovec.
    - Regs_R[5]: tamaño en bytes de lo que queremos escribir.

  Lectura de memoria de la estructura *iovec*. Si el tamaño de lo que deseamos escribir supera el tamaño máximo fijado para el buffer de lectura, entonces es necesario leer de memoria varios bufferes y llamar a *host_write*( ) para cada uno de ellos.

- ***Syscall kill*** (37): envia una señal a un proceso.
    - Regs_R[3]: identificador de proceso.
    - Regs_R[4]: identificador de la señal que se desea enviar.

  Llama a *universal_syscall_kill*( ) la cual hace uso de la estructura *UniversalSiginfo* para generar la señal que será enviada a nuestro sistema de señales simulado. Finalmente se comprueba que el proceso actual coincide con el identificador de proceso (para ello se hace uso de *host_getpid*( ) y se envia la señal haciendo uso de la función *send_siginfo*( ) (signal.c).

- ***Syscall kioctl*** (54): control de dispositivo.
    - Regs_R[3]: descriptor de fichero del dispositivo.
    - Regs_R[4]: petición o solicitud al dispositivo.
    - Regs_R[5]: argumentos de control de la petición.

  Llamada a *universal_syscall_kioctl*( ) la cual llama a *ofd_get*( ) y *host_ioctl*( ). Esta función hace uso de la llamada al sistema de Linux *ioctl*.

- ***Syscall kfcntl*** (55): manipula un descriptor de ficheros.
    - Regs_R[3]: descriptor de fichero.
    - Regs_R[4]: comando o acción que se va a llevar acabo sobre el descriptor.
    - Regs_R[5]: argumentos para el manipulador.

  Llamada a *universal_syscall_kfcntl*( ) la cual llama a *ofd_get(*) y *host_fcntl*( ). Esta función hace uso de la llamada al sistema de Linux *fcntl*.

- ***Syscall stat*** (106): obtiene el status de un fichero (datos y características del fichero: device, permisos, propietario, fecha de creación, última modificación, tamaño en bytes, número de bloques, etc).
    - Regs_R[3]: dirección virtual de memoria del nombre del fichero.
    - Regs_R[4]: dirección virtual de memoria en la que se copiarán los resultados del análisis.

  Llama a la función *universal_syscall_stat*( ) que se sirve de la estructura *UniversalStat* (*syscall.h*) para llamar a *ofd_map_name*( ) y *host_stat*( ). Esta función hace uso de la llama al sistema de Linux *stat*. Finalmente se copian los resultados en memoria.

- ***Syscall stat64*** (195): obtiene el status de un fichero con formato de 64 bits.
    - Regs_R[3]: dirección virtual de memoria del nombre del fichero.
    - Regs_R[4]: dirección virtual de memoria en la que se copiarán los resultados del análisis.

  Llama a la función *universal_syscall_stat64*( ) que se sirve de la estructura *UniversalStat64* (*syscall.h*) para llamar a *ofd_map_name*( ) y *host_stat64*( ). Esta función hace uso de la llama al sistema de Linux *stat64*. Finalmente se copian los resultados en memoria.

- ***Syscall fstat*** (108): obtiene el status de un fichero. La diferencia con la llamada *stat* es que en este caso disponemos del descriptor de fichero en vez de su nombre.
    - Regs_R[3]: descriptor de fichero.
    - Regs_R[4]: dirección virtual de memoria en la que se copiarán los resultados del análisis.

  Llama a la función *universal_syscall_fstat*( ) que se sirve de la estructura *UniversalStat* (*syscall.h*) para llamar a *ofd_get*( ) y *host_fstat*( ). Esta función hace uso de la llama al sistema de Linux *fstat*. Finalmente se copian los resultados en memoria.

- ***Syscall fstat64*** (197): obtiene el status de un fichero con formato de 64 bits. La diferencia con la llamada *stat64* es que en este caso disponemos del descriptor de fichero en vez de su nombre.
    - Regs_R[3]: descriptor de fichero.
    - Regs_R[4]: dirección virtual de memoria en la que se copiarán los resultados.

  Llama a la función *universal_syscall_fstat64*( ) que se sirve de la estructura *UniversalStat64* (*syscall.h*) para llamar a *ofd_get*( ) y *host_fstat64*( ). Esta función hace uso de la llama al sistema de Linux *fstat64*. Finalmente se copian los resultados en memoria.

- ***Syscall lstat*** (107): su comportamiento es el mismo que el de la llamada *stat* excepto en el caso en que el fichero tenga un enlace simbólico. En este caso, dicho enlace no aparece como información obtenida acerca del fichero.
    - Regs_R[3]: dirección virtual de memoria del nombre del fichero.
    - Regs_R[4]: dirección virtual de memoria en la que se copiarán los resultados.

  Llama a la función *universal_syscall_lstat*( ) que se sirve de la estructura *UniversalStat* (*syscall.h*) para llamar a *ofd_map_name*( ) y *host_lstat*( ). Esta función hace uso de la llama al sistema de Linux *lstat*. Finalmente se copian los resultados en memoria.

- ***Syscall getcwd*** (182): obtiene el directorio de trabajo actual.
    - Regs_R[3]: dirección virtual de memoria en donde se copia el directorio de trabajo actual.
    - Regs_R[4]: tamaño del buffer de escritura.

  Llama a *universal_syscall_getcwd*( ) la cual llama a *host_getcwd*( ). Esta función hace uso de la llamada al sistema de Linux *getcwd* que nos devuelve el directorio de trabajo actual, el cual se copiará en un buffer. Finalmente este buffer es escrito en memoria.

- ***Syscall getrusage*** (77): obtiene el uso de un recurso.
    - Regs_R[3]: recurso (puede ser del proceso actual o de sus hijos).
    - Regs_R[4]: dirección virtual de memoria donde se escribirán los resultados obtenidos (tiempo de usuario, tiempo de sistema, instancias de página, fallos de página, señales recibidas, mensajes recibidos, ect). Para más información mirar la estructura *rusage* en *man 2 getrusage*.

  Llamada a *universal_syscall_getrusage*( ) la cual hace uso de la estructura *UniversalRusage* y llama a *host_getrusage*( ). Esta función hace uso de la llamada al sistema de Linux *getrusage*. Finalmente se copian los resultados en memoria.

- ***Syscall exit*** (1): finalización de la ejecución del actual programa. Llama a la función *exit_now*( ) definida en *main.c*, la cual recibe como parámetro de entrada el contenido del registro regs_R[3]. Éste parámetro determina diferentes formas de finalizar la ejecución de un programa (internamente utiliza la función *exit* de Linux).

4.2.3.- Sistema de ficheros simulado

Entre la necesidades de nuestro simulador para poder emular llamadas al sistema, se encontraba la necesidad de dotar al SimpleScalar de un sistema de ficheros simulado (*file.c*). El objetivo de este apartado es llevar acabo una descripción de la implementación realizada del mismo.

El sistema de ficheros simulado se fundamenta en una base de datos que contiene toda la información que podamos necesitar sobre todos los ficheros que se encuentran abiertos en cada momento por nuestro simulador. Para ello disponemos de las siguientes estructuras (*file.h*):

```
#define MAX_OPEN_FILES 128

struct openfile_record {
      int active;         /* bit de activo */
      char filename[256]; /* Nombre del fichero */
      int true_handle;    /* Descriptor de fichero asociado */
      int flags;          /* flags del fichero */
      mode_t mode;        /* Modo en el que se ha abierto el fichero */
      double checksum;    /* Información de corrección checksum */
      off_t offset;       /* Posición del puntero del fichero */
};

struct openfile_database {
      struct openfile_record dat[MAX_OPEN_FILES];
      int ind_list[MAX_OPEN_FILES];
};
```

La estructura *openfile_record* contiene toda la información que podamos necesitar sobre nuestros ficheros, mientras que la estructura *openfile_database* contiene un array de estructuras *openfile_record* que hará las funciones de base de datos.

Para crear nuestro sistema de ficheros es necesario inicializarlo llamando a *ofd_init_first*( ). Esta función es llamada desde la función *sim_init*( ) que se encarga de inicializar los diferentes sistemas de nuestro simulador (memoria, registros, control de tiempo, señales, etc).

A su vez, *ofd_init_first*( ) invoca a la función *ofd_reset_stdio*( ) que se encarga de reservar las tres primeras entradas de nuestra base de datos para *sim_progin*, *sim_progout* y *sim_progerr*, que son la entrada, salida y salida con error de nuestro simulador por defecto. Estas variables se encuentran declaradas en *sim.h* como strings y son utilizados por las funciones definidas en *misc.h* para leer de la entrada del simulador, mostrar información o resultados por el terminal o bien notificar de errores durante la ejecución de los programas.

Además de estas funciones hay que destacar las siguientes:

- *ofd_open*( ): añade un nuevo fichero a la base da datos.
- *ofd_map_name*( ): dado el nombre de un fichero, busca dicho fichero en la base de datos de nuestro sistema de ficheros simulado y nos devuelve el nombre con el que ha sido almacenado en la base de datos.
- *ofd_get*( ): dado el descriptor de fichero, busca dicho fichero en la base de datos para proporcionar el descriptor de fichero con el que ha sido almacenado en la base de datos.
- *ofd_close*( ): elimina un fichero de la base de datos.

Debemos de tener en cuenta que el nombre o el descriptor de fichero asociado a un fichero bajo el sistema Linux no tiene porque ser el mismo que el asignado en nuestro sistema de ficheros simulado. De ahí, que necesitemos funciones que nos permitan realizar búsquedas en la base de datos y proporcionarnos los nombres y descriptores asociados en nuestro sistema de ficheros SimpleScalar.

4.2.4.- Sistema de señales simulado

El sistema de señales simulado (*signal.c*) era otra de las necesidades que requeria ser implementada e incorporada a nuestra herramienta SimpleScalar para la correcta implementación de todas aquellas llamadas al sistema que tenían relación con el sistema de señales.

Cuando atendemos a una señal es necesario llevar acabo un salvado del marco o estado de la arquitectura (puntero de pila, contenido de los registros, contador de programa, etc). Se produce un efecto parecido al que se genera cuando saltamos a una subrutina. En este caso la implementación de las señales es simulada, y el salvado del estado de la máquina es necesario ya que en ocasiones el atender a una señal determinada requiere de la utilización de los registros, lo que implica que se puedan perder datos almacenados en los mismos antes de la ejecución de la señal.

La información necesaria para poder almacenar llamadas al sistema se encuentra en la estructura *UniversalSiginfo* (*syscall.h*). En dicha estructura se encuentra almacenada la información más inportante y relevante (número de señal, código de identificación, proceso que la ha enviado, etc).

Para poder almacenar las llamadas que van siendo enviadas durante la ejecución de un programa necesitamos de una cola de señales, que establezca un orden sobre las mismas. En este caso, se trata de una cola FIFO (primero en entrar, primero en salir). Dicha cola es proporcionada por la estructura *sig_queue*.

```
struct sig_queue {
      struct sig_queue  *next;
      UniversalSiginfo  *signal;
};
```

Para el correcto mantenimiento del estado de la arquitectura utilizamos otra cola. Cada vez que una señal es invocada se almacena el estado de la máquina (registros) en memoria y cuando termina se restaura. En este caso, se trata de una cola con estructura LIFO (último en entrar, primero en salir). Dicha cola es proporcionada por la estructura *regs_queue*.

```
struct regs_queue {
      UniversalSiginfo  *signal;
      md_addr_t  saved_pointer;     /* SP of previous frame */
      md_addr_t  saved_frame_top;   /* SP of this frame */
      struct regs_queue  *old_regs; /* old registers' queue */
      UniversalSigset  oldsigmask;
};
```

El sistema de señales dispone de las siguientes funciones:

- *signal_init*( ): inicializa el sistema de señales simulado. Dicha función es invocada desde *sim_init* (sim-safe.c).
- *send_siginfo*( ): envia una señal desde un proceso. Esta función llama a su vez a la función *add_signal*( ), que añade la señal a la cola de espera.
- *do_signal*( ): función que se encarga de atender a la siguiente señal en turno, una vez atendida la elimina de la cola de espera mediante la llamada a la función *dequeue_signal*( ). Además hace uso de la función *signal_setup_frame*( ) antes de ejecutar la señal para salvar el estado de la arquitectura.
- *signal_setup_frame*( ): salva el estado de la arquitectura haciendo uso de la memoria y de diferentes estructuras universales (*syscall.h*).
- *restore_frame*( ): restaura el estado de la arquitectura.

Además fue necesario llevar acabo modificaciones en el simulador (*sim-safe.c*). Tras la ejecución de cada nueva instrucción el sistema de señales es llamado para que atienda a la siguiente señal en cola de espera. Para ello basta con invocar a la función *do_signal*( ) en cada ciclo de ejecución.

4.3.- Compilación, pruebas y depuración

4.3.1.- Compilación

Para la compilación de nuestro proyecto basado en la herramienta SimpleScalar hay que seguir los siguientes pasos:

- Descomprimir el fichero *ss3-ppc.tgz* en nuestro directorio *home*:

```
$ cd $HOME
$ tar -zxvf ss3-ppc.tgz
```

- Acceder al directorio del proyecto y compilar el simulador *sim-safe* para la arquitectura PowerPC:

```
$ make clean config-ppc sim-safe
```

La opción *clean* para la compilación solamente es necesaria cuando queramos borrar cualquier rastro de compilaciones anteriores durante la sesión. El fichero de compilación *Makefile* se encuentra en el directorio raiz del proyecto (*$HOME/ss3-ppc*). Hay que destacar dos aspectos importantes de dicho fichero de compilación:

- En Makefile se define la variable de entorno CC utilizada por el compilador GNU GCC del siguiente modo:

```
CC =  gcc-3.4 -DPPC_SWAP -DDEBUG_PRINT_SYSCALLS
```

Es necesario llevar acabo la compilación con la versión 3.4 de GCC, para versiones superiores o inferiores el comportamiento del simulador puede no ser el adecuado. Esto se debe a que el simulador está especificamente diseñado para trabajar y ser compilado bajo esta versión.

La opción *-DPPC_SWAP* es necesario definirla en caso de que los ficheros binarios que vayamos a ejecutar sean para una arquitectura con un formato *endian* diferente al de la máquina sobre la que vamos a ejecutar nuestro simulador.

En este caso en concreto, nuestros ficheros binarios son compilados en un host (*urbion*) con arquitectura PowerPC que trabaja con formato *big-endian*, mientras que el host en el que ejecutamos nuestro simulador trabaja bajo arquitectura Intel que utiliza un formato *little-endian*. Por lo tanto, se hace necesario definir la operación *SWAP* como operación necesaria para la correcta carga y ejecución de nuestros binarios.

La opción *-DDEBUG_PRINT_SYSCALLS* únicamente debe ser definida en el que caso en el que queramos obtener información sobre las llamadas al sistema que son invocadas durante la ejecución de nuestros programas. La definición de esta opción mostrará por pantalla una pequeña información relativa a cada llamada al sistema y el resultado que produce.

Si por el contrario no precisamos de dicha información o sencillamente no deseamos que se muestre por pantalla para que la salida por pantalla de nuestros programas sean más legibles, únicamente deberemos eliminar dicha opción y recompilar.

4.3.2.- Pruebas y depuración

Las pruebas realizadas a nuestro proyecto se han basado en dos tipos de tests diferentes:

- Tests de la ampliación de la herramienta SimpleScalar a la arquitectura PowerPC, y sobre la que hemos modificado e implementado.
- Benchmarks SPEC 2000: programas enteros y en punto flotante.

Todas los programas que han sido utilizados para depurar y probar nuestro simulador se proporcionan ya compilados y preparados para que sean directamente probados. Aún así, explicaremos a continuación como es necesario compilar los tests y como se instalan y se compilan las benchmarks SPEC 2000.

4.3.2.1.- Tests SimpleScalar PowerPC

Los tests de SimpleScalar PowerPC vienen junto con el paquete original y se pueden encontrar en nuestro proyecto en el directorio *ss3-ppc/tests-ppc/bin*. En este directorio se encuentran los ficheros de código fuente (**.c*) de todos los tests. Los ficheros ya compilados se encuentran en el directorio raiz del proyecto con el nombre *test-nombredeltest*.

Para compilar los tests es necesario seguir los siguientes pasos:

- Llevar el fichero *.c* correspondiente a *urbion* (arquitectura PowerPC).
- Compilarlo estáticamente del siguiente modo:

```
$ gcc-3.4 -static -O2 nombredeltest.c -o test-nombredeltest
```

Hay que tener en cuenta que los programas que ejecuta nuestro simulador deben de estar necesariamente compilados de forma estática, ya que nuestra versión de SimpleScalar no admite enlazado dinámico de librerias.

La opción *-O2* se refiere al nivel de optimización de la compilación, en todas las compilaciones de programas para nuestro simulador hemos fijado como convenio este valor. Valores mayores de optimización pueden aumentar significativamente el número de instrucciones por programa, la mayoría de las cuales no eran relevantes y lo único que hacían era dificultar posibles depuraciones.

- Comprobamos que el test compilado funciona correctamente en *urbion*:

  ```
  $ ./test-nombredeltest
  ```

- Llevamos el fichero compilado a nuestro host (lo ubicamos en el directorio raiz del proyecto), generalmente hemos utilizado el servidor *aneto* (arquitectura Intel).

- Ejecutamos el test (debemos estar en el directorio raiz del proyecto):

  ```
  $ ./sim-safe test-nombredeltest
  ```

El primero de los tests con el que trabajamos, y que fijamos como primer objetivo para que funcionará correctamente el proyecto fue *test-lswlr*. La ejecución de este test es muy básica, lo único que hace es mostrar por pantalla el mensaje *Hello world*. Basándonos en este test comezamos el desarrollo de nuestro cargador de ficheros binarios ELF y la implementación de las primeras llamadas al sistema.

La depuración de un programa de este tipo puede llegar a ser algo complicada, sobre teniendo en cuenta que la versión de SimpleScalar PowerPC de la que hemos partido puede contener errores, ya que se trata de un software que está en continua actualización y mejora.

Para la depuración de muchos de los programas nos hemos basado en la ejecución del programa en la versión dinámica del SimpleScalar. Si la ejecución del programa en la versión dinámica era correcta ya disponiamos de un punto de partida que nos permitía conocer el flujo de instrucciones correcto del programa.

La depuración se podía realizar de varias formas:

- Utilizando algun depurador: en nuestro caso hemos solido usar *ddd*.

- Obteniendo la versión ensamblador del programa mediante *objdump* como opción de compilación para *gcc*.

La mayoría de las veces nos hemos decantado casi siempre por el depurador *ddd*, que nos proporciona un entorno gráfico más manejable y ameno de usar. La forma de depurar se ha basado en observar los cambios realizados por las instrucciones en cada ciclo de ejecución sobre los registros y la memoria por parte de cada una de las dos versiones. Este sistema nos permitía ver con detalle cual era el flujo del programa y el estado de la arquitectura en cada momento, hasta llegar al punto en el que dichos programas no coincidieran en su comportamiento.

Evidentemente este sistema de trabajo era bastante aceptable cuando el programa tenía un tamaño razonable, entorno a los 10 millones de instrucciones. Para programas mayores se hacía algo más tedioso, pero nada que no pudiese ser tratado con algo de paciencia. El problema aparecía cuando el programa tenía un tamaño bastante considerable, entorno a los billones de instrucciones. Estos programas pueden necesitar incluso de horas para ejecutarse y resultar una carga considerable incluso para los propios programas depuradores.

Curiosamente muchos de los errores que hemos encontrado a lo largo del proceso de depuración eran instrucciones incorrectamente definidas en *machine.def*. A continuación vamos a hacer una descripción de los diferentes test ejecutados:

- *test-fmath*: programa de matemáticas muy sencillo que realiza operaciones muy básicas (suma, resta, multiplicación, división y exponenciación) y muestra los resultados en diferentes precisiones (32 y 64 bits).
    - Ejecuta las llamadas *uname*, *brk*, *fstat64*, *mmap*, *kwrite*, *munmap* y *exit*.
    - Número de instrucciones: 38585.

- *test-llong*: programa que realiza operaciones muy básicas (suma, resta, multiplicación y división) con operandos en doble precisión y muestra los resultados en notación hexadecimal.
    - Ejecuta las llamadas: *uname*, *brk*,*fstat64*, *mmap*, *kwrite*, *munmap* y *exit*.
    - Número de instrucciones: 1548.

- *test-lswlr*: muestra por pantalla *str=Hello world...*
    - Ejecuta las llamadas: *uname*, *brk*, *fstat64*, *mmap*, *kwrite*, *munmap* y *exit*.
    - Número de instrucciones: 9005.
    - Depuración: fue el primer test en probarse y nos permitio localizar una instrucción incorrectamente definida en *machine.def*: *MULHWU* (multiplicación doble palabra sin signo) realizaba la multiplicación con signo.

- *test-math*: programa que realiza operaciones matemáticas complejas (exponenciación, logaritmos, funciones trigonométricas, raices, etc.) y muestra los resultados en doble precisión.
    - Ejecuta las llamadas: *uname*, *brk*, *fstat64*, *mmap*, *kwrite*, *munmap* y *exit*.
    - Número de instrucciones: 111929.

- *test-printf*: test de formato de salida en un gran número de notaciones.
    - Ejecuta las llamadas: *uname*, *brk*, *fstat64*, *mmap*, *kwrite*, *munmap* y *exit*.
    - Número de instrucciones: 1108958.

- *test-stream_d*: mide la razón de transferencia de memoria en MB/s para una simple cooperación computacional entre núcleos en C.
    - Compilación (*urbion*):

      ```
      $ gcc-3.4 -static -O2 stream_d.c second_cpu.c -o test-stream_d -lm
      ```

      La opción *-lm* realiza un enlace estático a la librería de matemáticas de C.
    - Ejecuta las llamadas: *uname*, *brk*, *fstat64*, *mmap*, *kwrite*, *getrusage* (mide la razón de transferencia de memoria), *munmap* y *exit*.
    - Número de instrucciones: 100068578.

  Mecionar que ninguno de los *test* requiere de ficheros de entrada.

4.3.2.2.- Benchmarks SPEC 2000

Las benchmarks *SPEC 2000* [9] son proporcionadas en el paquete *SPEC2000.tgz*. El primer paso es instalarlas para la arquitectura específica del host en el que nos encontremos. Evidentemente en nuestro caso y como siempre hemos hecho, es necesario instalarlas y compilar los programas en *urbion* para posteriormente llevarnos los programas ya compilados a *aneto*. Para llevar acabo la instalación en *urbion* es necesario seguir los siguientes pasos:

```
$ cd $HOME
$ tar -zxvf SPEC2000.tgz
$ cd SPEC2000
$ ./install
```

El gestor de instalación nos pregunta entonces para que arquitectura deseamos instalar las benchmarks, en nuestro caso debemos de elegir *linux-ppc*, que además es la única opción a elegir que nos proporciona.

Los *SPEC 2000* proporcionan un conjunto de programas lo suficientemente amplio y completo como para medir el rendimiento de los procesadores. En nuestro caso no estamos tan interesados en el rendimiento, sino más bien en el correcto funcionamiento de los programas que lo componen para de este modo asegurarnos del funcionamiento de nuestro proyecto.

Una vez llevada acabo la instalación es necesario compilar los programas que deseemos probar. Los programas se encuentran ubicados en el directorio *SPEC2000/benchspec*. Dicho directorio contiene a su vez dos directorios *CINT2000* y *CFP2000*, que contiene los programas para enteros y los programa para punto flotante respectivamente.

Si nos introducimos por ejemplo en el directorio *CINT2000* y hacemos un listado (comando *ls*), veremos los siguientes directorios:

```
164.gzip
175.vpr
176.gcc
181.mcf
186.crafty
197.parser
252.eon
253.perlbmk
254.gap
255.vortex
256.bzip2
300.twolf
```

Cada uno de estos directorios contiene el código fuente, los ficheros de entrada de prueba, ficheros de configuración y documentación de los diferentes programas que componen las *SPEC 2000* para enteros.

Antes de compilar ningún programa es necesario modificar el fichero de configuración de compilación de nuestra arquitectura para adaptarlo a las necesidades de nuestro simulador. Para ello accedemos al directorio *SPEC2000/config* y editamos el fichero *ppc32_linux.cfg*. Definimos las variables de entorno *CC* (compilador de C), *CXX* (compilador de C++) y *FC* (compilador de Fortran) del siguiente modo:

```
CC = gcc-3.4 -static
CXX = g++ -static
FC = g77 -static
```

Veamos ahora cuales son los pasos necesarios para realizar la compilación de un programa en concreto (*164.gzip*):

```
$ cd $HOME/SPEC2000
$ sh
$ . ./shrc
$ runspec --config=ppc32_linux --action=build gzip
$ runspec --config=ppc32_linux --action=validate --input=test gzip
```

El comando *sh* nos permite entrar en el shell de Linux y ejecutar *shrc* que se encarga de definir todas las variables de entorno y comandos necesarios para compilar los diferentes programas.

La primera llamada a *runspec* realiza la compilación del programa, generando todos los ficheros objeto necesarios así como el fichero binario ELF final del propio programa. Con la segunda llamada a *runspec* validamos la compilación anterior y nos proporciona todos los ficheros de prueba e información necesaria para poder ejecutar correctamente los programas. Todos los ficheros anteriormente citados son ubicados en el directorio *164.gzip/run/00000002*, los cuales tienen que ser llevados al servidor *aneto* para su ejecución con nuestro simulador.

Entre los fichero que se encuentran en el directorio *run/00000002* hay que destacar dos:

- *gzip_base.ppc32_linux*: fichero binario ejecutable ELF del programa.
- *speccmds.cmd*: ejecutando **`more speccmds.cmd`** se nos proporcionar la información necesaria sobre como hay que ejecutar el programa (ficheros de entrada, opciones del programa, salida, etc).

Recordar que todos los ficheros de entrada necesarios para ejecutar el programa se encuentran en el directorio run/00000002. Además es necesario comprobar que el programa compilado funciona correctamente en *urbion* antes de llevarlo a *aneto*. Una vez compilados y probados en *urbion,* los programas que componen las *SPEC* se encuentran ubicados en el directorio *ss3-ppc/qstat* de nuestro proyecto. Dentro de este directorio hay un directorio por cada programa cuyo nombre es el nombre del programa, así en *ss3-ppc/qstat/gzip* se encuentra la *SPEC* gzip. El procedimiento de compilación y prueba que acabamos de explicar es análogo para cualquier programa que compone las *SPEC 2000*.

Los programas que componen las *SPEC 2000* son programas complejos cuya ejecución puede durar un tiempo considerable, alrededor de horas o incluso días. No se trata por lo tanto de programas que podamos ejecutar y depurar en el momento, sino que necesitamos de una cola de trabajos que nos permita ejecutar el programa y cuando este termine comprobar los resultados.

Para tal fin *aneto* proporciona acceso a un *cluster* de computadores de 8 nodos. Desde *aneto* podemos mandar trabajos, eliminarlos o simplemente ver el estado de la cola de espera de trabajos. Podemos además especificar que la salida estándar del programa así como la de errores sean volcadas en un fichero de texto para ver cual ha sido el comportamiento del programa y si este funciona correctamente.

La única condición es que para mandar trabajos debemos mandarlos como *scripts*, para ello lo único que hay que hacer es crear un fichero de texto cuyo único contenido debe de ser la línea con la que ejecutamos nuestro programa en un terminal de Linux, teniendo siempre cuidado de pasar la ruta completa de todos los fichero tomando como punto de partida nuestro *$HOME*.

Todos los *scripts* de los programas *SPEC 2000* se encuentran ubicados por comodidad en el directorio raiz del proyecto con el nombre *nombredelprograma.sh*. Mientras que la salida producida por la ejecución de los diferentes programas se encuentra en sus respectivos directorios en *ss3-ppc/qstat* con nombres*:*

- *nombredelprograma-ss.ox*: salida estándar del programa.
- *nombredelprograma-ss.ex*: salida de error por defecto del programa.

La *x* indica el número de trabajo en la cola de ejecución y puede ser ignorada. Para más información sobre el funcionamiento de la cola de trabajos consultar [10].

A continuación vamos a hacer una descripción de los programas *SPEC* que funcionan perfectamente para nuestro proyecto. Comenzamos por los programas enteros:

- *164.gzip* (Lenguaje de programación C): programa de compresión bajo formato gzip.
  - Ficheros de entrada: *input.combined*.
  - Número de instrucciones ejecutadas: 17910672105.

- *181.mcf* (C): programa de optimización combinatoria.
  - Ficheros de entrada: *inp.in*.
  - Número de instrucciones ejecutadas: 135810179.

- *255.vortex* (C): programa basado en una base de datos orientada a objetos.
  - Ficheros de entrada: *bendian.raw*.
  - Número de instrucciones ejecutadas: 272594.

- *256.bzip2*: (C) programa de compresión bajo formato *bzip2*.
    - Ficheros de entrada: input.random.
    - Número de instrucciones ejecutadas: 9026782779.

- *300.twolf* (C): simulador de rutado y emplazamientos.
    - Ficheros de entrada: es necesario especificarle el directorio donde se encuentran todos los ficheros de entrada. En nuestro caso *ss3-ppc/qstat/twolf*.
    - Número de instrucciones ejecutadas: 232546433.

Programas en punto flotante:

- *171.swim* (Fortran 77): programa de modelado de superficies de agua.
    - Ficheros de entrada: *swim.in*.
    - Número de intrucciones ejecutadas: 485721375.

- *172.mgrid* (Fortran 77): solucionador de redes multietapa para campos con potencial 3D.
    - Ficheros de entrada: *mgrid.in*.
    - Número de instrucciones ejecutadas: 21909921019.

- *173.applu* (Fortran 77): solucionador de ecuaciones diferenciales parciales: parábolas / elipses.
    - Ficheros de entrada: applu.in.
    - Número de instrucciones ejecutadas: 308227164.

- *177.mesa* (C): librería gráfica 3D.
    - Ficheros de entrada: *mesa.in*, *mesa.ppm*.
    - Número de instrucciones ejecutadas: 1822945988.

- *179.art* (C): reconocimiento de imagen / redes neuronales.
    - Ficheros de entrada: *c756hel.in*, *a10.img*.
    - Número de instrucciones ejecutadas: 1618393583.

- *183.equake* (C): simulador de propagación de ondas sísmicas.
    - Ficheros de entrada: *inp.in*.
    - Número de instrucciones ejecutadas: 972815228.

- *188.ammp* (C): computación química.
    - Ficheros de entrada: *ammp.in*.
    - Número de instrucciones ejecutadas: 6395596139.

El resto de programas que componen las *SPEC* y que no hemos mencionado no han podido ser probados por razones de compilación. Determinados programas necesitan ser modificados o tienen problemas que es complicado solucionar y que se sale del objetivo de nuestro proyecto. Aún así, lo más importante es que los programas probados y que hemos mencionado anteriormente conforman una parte lo suficientemente representativa.

Gracias a la ejecución de los programas de prueba *SPEC* volvimos a detectar instrucciones que se encontraban mal definidas dentro del repertorio del simulador SimpleScalar. Así la instrucción *DCBZ* tenía mal definido su código de operación, la instrucción *MULHWUD* realizaba la correspondiente multiplicación con signo cuando debería de ser sin signo.

Además nos permitió corregir pequeños errores tanto en el cargador de programa así como en la implementación de algunas llamadas al sistema. Con más tiempo, estamos convencidos de que podríamos haber hecho funcionar todos los programas que componen las *SPEC*.

Tabla resumen benchmarks SPEC 2000:

	Programa	Resultados
CINT2000	164.gzip	Funciona
	175.vpr	No funciona (SS/DSS)
	176.gcc	Problemas de compilación
	181.mcf	Funciona
	186.crafty	No funciona (SS/DSS)
	197.parser	No funciona (SS/DSS)
	252.eon	Problemas de compilación
	253.perlbmk	Problemas de compilación
	254.gap	No funciona (SS/DSS)
	255.vortex	Funciona
	256.bzip2	Funciona
	300.twolf	Funciona
CFP2000	186.wupwise	No funciona (SS/DSS)
	171.swim	Funciona
	172.mgrid	Funciona
	173.applu	Funciona
	177.mesa	Funciona
	178.galgel	Problemas de compilación
	179.art	Funciona
	183.equake	Funciona
	187.facerec	Problemas de compilación
	188.ammp	Funciona
	189.lucas	Problemas de compilación
	191.fma3d	Problemas de compilación
	200.sixtrack	No funciona (SS/DSS)
	301.apsi	No funciona (SS/DSS)

Conclusión

El haber llevado a cabo la modificación de la versión SimpleScalar PowerPC estática, y adaptarla para disponer de la capacidad de ejecutar programas reales y con carga representativa bajo Linux, tiene como objetivo disponer de un emulador capaz de ejecutar programas compilados en máquinas con arquitectura PowerPC sin disponer necesariamente de una máquina con dicha arquitectura. Las únicas condiciones que se requerirán a partir de ahora serán las de disponer de una máquina con sistema operativo Linux y nuestro simulador, con esto, e independientemente de las características arquitectónicas de la máquina, seremos capaces de ejecutar programas PowerPC-Linux.

Este tipo de simuladores no solamente es capaz de verificar el correcto funcionamiento de los programas para un amplio número de arquitecturas, sino que además es capaz de medir el rendimiento de los mismos sobre dichas arquitecturas, proporcionándonos una idea muy cercana sobre el comportamiento de los mismo en situaciones reales y trabajando sobre las arquitecturas para las que fueron diseñados e implementados.

SimpleScalar es el simulador arquitectónico más ampliamente extendido, tanto en investigación como en docencia. Su portabilidad y potencia le sirven como punto de partida para el análisis te nuevas arquitecturas, así como para modificaciones de las ya existentes.

Bibliografía

[1] Doug Burger, Todd M. Austin (University of Wisconsin-Madison): "The SimpleScalar Tool Set, Version 2.0". IEEE Computer Vol. 2, 2002. (Página web: http://www.simplescalar.com).

[2] Karthikeyan Sankaralingam, Ramadass Nagarajan, Stephen W. Keckler, Doug Burger (University of Texas). "SimpleScalar Simulation of the PowerPC Instruction Set Architecture, 2004".

[3] Todd M.Austin, Dan Emst, Eric Larson, Chris Weaver. "SimpleScalar PowerPC Version 4.0 Test Releases Tutorial". (Página web: htt://simplescalar.com/v4test.html).

[4] Dynamic SimpleScalar (University of Massachusetts Amherst and the University of Texas at Austin). (Página web: http://ali-www.cs.umass.edu/DSS/index.htm).

[5] PowerPC Microprocessor Family Programming Environments Manual for 64 and 32-Bit Microprocessors Version 2.0. IBM Microelectronics Division.

[6] José Antonio Gómez Hernández. "Llamadas al sistema en Linux, 2005". (http://lsi.ugr.es/~jagomez/sisopi_archivos/SystemCall.pdf).

[7] Juan Carlos Pérez y Sergio Sáez (Universidad Politécnica de Valencia). "Tema 5: Llamadas al sistema. Estudio y diseño de sistemas operativos". (http://futura.disca.upv.es/~eso/es/t5-llamadas-al-sistema/gen-t5-llamadas-al-sistema.html).

[8] TIS Committee. "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. Mayo 1995. (Página web: http://www.pdos.csail.mit.edu).

[9] Standard Performance Evaluation Corporation Version 2000 (Benchmarks SPEC 2000). (Página web: http://www.spec.org).

[10] Manual para la utilización de Aneto Cluster (http://mulhacen.dacya.ucm.es).