Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Persistencia en SqueakNOS

Alumno: Guido Martín Chari
L.U. 696/03
charig@gmail.com

Alumno: Javier Esteban Pimás
L.U. 482/03
jpimas@dc.uba.ar

Director: Lic. Hernán Wilkinson
hernan.wilkinson@gmail.com
Departamento de Computación
Universidad de Buenos Aires

Co-Director: Gerardo Richarte
gera@corest.com

Junio de 2011

Tesis para optar al grado de
Licenciado en Ciencias de la Computación

Índice general

Resumen

SqueakNOS es una reificación de los conceptos de *Computadora* y de gran cantidad de conceptos de los *Sistema Operativo* dentro del dialecto Squeak del lenguaje Smalltalk. La filosofía de SqueakNOS establece que el desarrollo del mismo debe hacerse completamente en Smalltalk, logrando minimizar el código de bajo nivel a los casos en que esto sea imposible o el deterioro de rendimiento extremadamente ostensible. El proyecto es un trabajo en preexistente y aún en desarrollo, y como tal, varias funcionalidades comunes a los sistemas operativos no han sido implementadas aún. En este trabajo se analizan varios interrogantes relacionados con la persistencia de los objetos, que se presentan al trabajar utilizando modelos de alto nivel en este entorno. Se desarrolló un controlador de discos ATA y un modelo de filesystem FAT32, lo que brinda compatibilidad con otros sistemas operativos y con el entorno Squeak genérico. Así, se logra acceder al código fuente de los métodos y se avanza hacia el grabado de la imagen, característica que aún no estaba disponible en el proyecto. Luego, se desarrolló una técnica de persistencia de la memoria de los objetos cuyo objetivo principal es la simplicidad y su principal desventaja el requerir una utilización importante e ineficaz de memoria. A pesar de sus desventajas, fue el primer paso para lograr simular la *atomicidad* necesaria para grabar los objetos mientras estos están siendo modificados. Finalmente, se implementó un esquema de manejo de memoria basado en paginación, modificando el mecanismo de manejo de interrupciones original de SqueakNOS para que pudiera funcionar en forma sincrónica, requisito indispensable para resolver los *fallos de página*. Este esquema desarrollado permite resolverlos completamente desde Smalltalk, lo cual da lugar a la experimentación y al desarrollo de formas novedosas de utilización del mismo. Gracias a esto, resulta posible implementar una técnica de persistencia de la imagen alternativa, que utiliza mucha menos memoria que la original, gracias a la asistencia del mecanismo de paginación, utilizando la técnica de *copy on write*. Por último, se analizan los aspectos relacionados con la manera de trabajar en este tipo de entornos y plataformas, sus dificultades y complicaciones.

Capítulo 1

Introducción

Este trabajo de tesis de licenciatura es transversal a dos importantes tópicos de la computación. Estos son, los *sistemas operativos* y el desarrollo de modelos bajo las premisas, y con las herramientas, del paradigma de *programación orientada a objetos*. El trabajo se relaciona con estas dos áreas debido a que se trabaja sobre y extendiendo *SqueakNOS* [Squb], el cual cumple la función de un sistema operativo y además esta desarrollado mayoritariamente en *Smalltalk*. También, porque para lograr los objetivos hubo que investigar y/o modelar (en Smalltalk) conceptos como *archivos, controladores de dispositivos, interrupciones de hardware y manejadores de memoria*, todos estos enmarcados dentro del área de sistemas operativos.

Estos dos conceptos podrían estar intensamente relacionados si se analiza la definición de cada uno independientemente del otro. Por un lado, los sistemas operativos son componentes de *software*, en general de gran complejidad, que a grandes rasgos interactúan muy fuertemente con el *hardware* y son los encargados de su administración [SGG08, Sta97]. Una definición más conceptual los exhibe como una máquina virtual que se le presenta al software abstrayendo la complejidad del hardware [Tan07]. Por otro lado, el paradigma de programación orientada a objetos es un concepto teórico importante bajo el cual se suele desarrollar la mayoría del software en la actualidad [Tio]. Entonces, por lógica, dadas estas dos descripciones, se tendería a pensar que muchos de los sistemas operativos se desarrollan con herramientas de programación orientada a objetos y se modelan según sus premisas. Pero luego de explorar el dominio se puede afirmar que son demasiado pocos los proyectos que logran relacionar estos dos conceptos. Siguiendo esa línea, hay importantes autores que dudan de que pueda extrapolarse la evolución del software hacia los sistemas operativos [Sta97, Han01]. Dado que ambos autores del trabajo están interesados en el área, y que además desarrollan en, y son parte de la comunidad de Smalltalk, esta poca interrelación y estas afirmaciones les generaron importantes interrogantes. Una de las primeras preguntas que se plantearon fue el por qué de esta poca relación. ¿Es simplemente un problema de factibilidad?, ¿es un problema de rendimiento?, ¿o se reduce a que no hay un extenso abordaje científico y pragmático al respecto?.

SqueakNOS es una primer aproximación hacia la relación de ambos temas dentro del ambiente Smalltalk y fue la plataforma bajo la cual se decidió intentar contestar estas y otras interesantes preguntas. El proyecto está en una de sus primeras etapas y su objetivo está basado en la siguiente

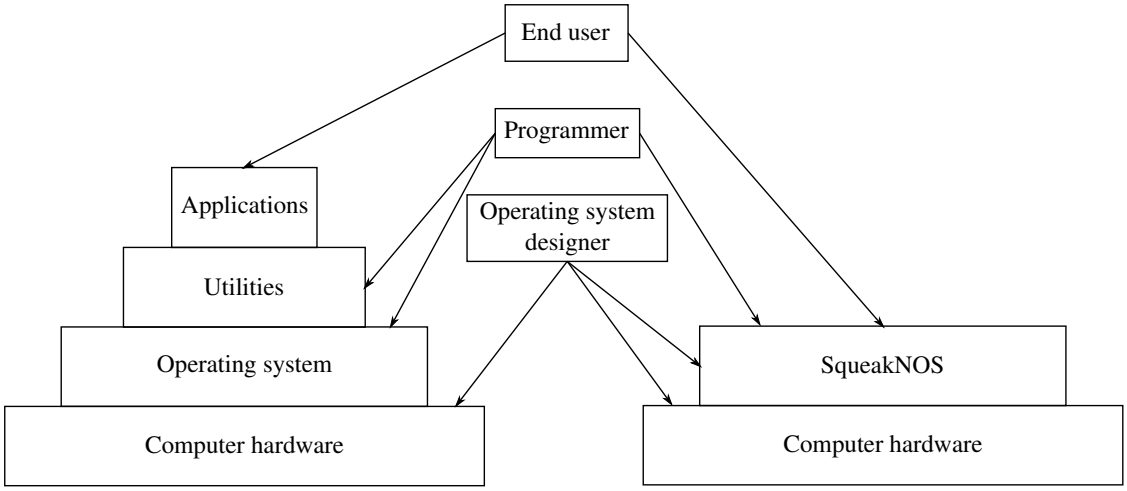


Figura 1.1: Comparación de arquitecturas entre un SO clásico y SqueakNOS

afirmación de Dan Ingalls: "Los sistemas operativos son una colección de cosas que no caben dentro de un lenguaje, no deberían existir" [Ing81]. La idea principal del proyecto es poder ejecutar el ambiente *Squeak*, sin necesitar el soporte de un sistema operativo clásico. Una síntesis muy general de su arquitectura y su diferencia con los sistemas operativos clásicos es presentada en la figura 1.1.

Los interrogantes que se plantearon son demasiado abarcativos para un trabajo de tesis y, como se describe más adelante en la sección 3, han sido parcialmente explorados, por lo cual se decidió restringir el trabajo al área de la persistencia para SqueakNOS y, con los resultados obtenidos dentro de esta área, intentar disipar las dudas ya existentes. Además, se sumaron nuevos interrogantes interesantes y algunos desafíos técnicos de implementación que son la esencia del trabajo. Lo primero que cabe destacar es que se decidió la exploración de esta área de trabajo porque SqueakNOS no ofrecía ningún soporte para persistencia al momento de comenzado el trabajo y eso resultaba uno de los obstáculos mas importantes del proyecto y uno de los más desafiantes para trabajar.

Con respecto a la persistencia, Squeak [Squa] (base de SqueakNOS) hace un fuerte uso del concepto de *archivos* con lo cual para lograr la persistencia en la plataforma hubo que desarrollar modelos que soporten este concepto. Para esto se implementó un controlador de disco ATA [ATA], un *sistema de archivos* FAT32 [FAT] y se adaptó y enriqueció el modelo de *Streams* con los cuales se interactúa con los archivos. Cabe destacar que otras soluciones menos prolijas y escalables hubieran sido posibles de implementar sin tener que implementar la capa del *sistema de archivos*. Una posibilidad era crear particiones particulares para cada uno de los archivos con los cuales fuera necesario interactuar y acceder a sus datos mediante el acceso a las posiciones de estas particiones en el dispositivo de almacenamiento. Esta técnica tiene importantes inconvenientes como no contar con metainformación de los archivos, tal como nombre, fechas de creación, jerarquías, etc. Además resulta imposible redimensionar un archivo sin reubicar todas las particiones subsiguientes, y se cuenta con un número de particiones limitado. Sin embargo, fue implementada inicialmente por conveniencia como paso previo al desarrollo del filesystem FAT32 completo.

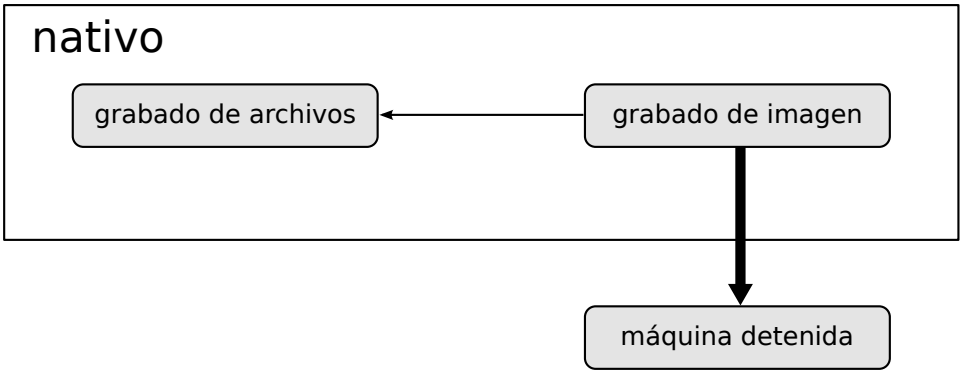
Es importante aclarar que desde el punto de vista teórico, parece un error persistir objetos en archivos secuenciales, originalmente pensados para persistir datos. Por lo tanto si se piensa a SqueakNOS como una plataforma no sólo de desarrollo, sino también ofreciendo los servicios de un sistema operativo (donde todos los elementos son objetos), no debería requerir de ninguno de los mecanismos relacionados a los archivos para su correcto funcionamiento. Pero queda fuera del alcance de este trabajo deshacerse de ellos. Además, SqueakNOS usa el *motor de objetos* [Row01] de Squeak y la idea es modificarlo o extenderlo lo menos posible, con lo cual el uso de archivos es un requerimiento que se hereda de Squeak, el cual los usa para persistir diversos elementos. En el plano pragmático, al entender a SqueakNOS como una plataforma integral, resulta necesario que posea los mecanismos para poder comunicarse con otros sistemas y los archivos actualmente son un medio esencial de organización de la información.

Los conceptos implementados que fueron descriptos hasta el momento parecieran ser suficientes para lograr persistencia en el ambiente. Sin embargo, primero es importante describir a que se le llamará persistencia en SqueakNOS en el marco de este trabajo y por qué esto es interesante para investigar y contestar las preguntas sustanciales de esta tesis. La plataforma *Squeak* está dividida en dos grandes elementos: el motor de objetos (también conocido como la *virtual machine*) y la memoria de los objetos (más conocida como *imagen*) [AG]. Hay tres elementos principales a persistir y todos están incluidos dentro del concepto de imagen. Estos son: el código fuente de los métodos, los cambios sobre ellos y finalmente la imagen misma respectivamente. El código fuente puede ser persistido y leído del medio donde se persiste a partir de poder interactuar con archivos y con el medio mismo, para lo cual alcanza con el driver ATA y el filesystem FAT32, más las adaptaciones comentadas, si el medio cumple con estos estándares. Sin embargo, persistir la imagen es un proceso complejo y no alcanza simplemente con estas herramientas.

Para persistir la imagen, el motor de objetos debe primero dejarla lista para este fin. Esto implica, entre otras cosas, limpiar los objetos que ya no tienen referencias en un proceso que se conoce como recolección de basura, el cual debe ser lo menos notorio posible para el usuario y que en Squeak está implementado con el algoritmo llamado *generation scavenge* [Ung]. Luego, él mismo es el responsable de persistirla en un archivo sin que esta pueda ser modificada en el interín. A este requerimiento (no ser modificada) es a lo que en este trabajo se denotará como el problema de la *atomicidad*. Se lo llama así ya que desde el punto de vista de la imagen ésta no se modifica en el transcurso de la escritura. De lo contrario, los objetos estarían cambiando mientras son persistidos y esto podría devenir en una imagen inconsistente, la cual al ser inicializada no posea las propiedades deseadas o directamente que no pueda ser cargada por el motor de objetos. Las propiedades deseadas son esencialmente que se mantenga el estado de toda la memoria previo a comenzar la persistencia.

Sin embargo, dado que no existe un sistema operativo soportándolo, SqueakNOS no permite usar persistencia desde el motor de objetos sino que todo el comportamiento para interactuar con dispositivos y archivos esta definido en la imagen. Debido a esto, no alcanza con tener implementado FAT32 y los controladores de disco, sino que es necesario lograr la atomicidad o por lo menos encontrar alguna manera de simularla, es decir, lograr el mismo resultado que si la operación fuera

Figura 1.2: Grabado de la imagen en un sistema operativo estándar



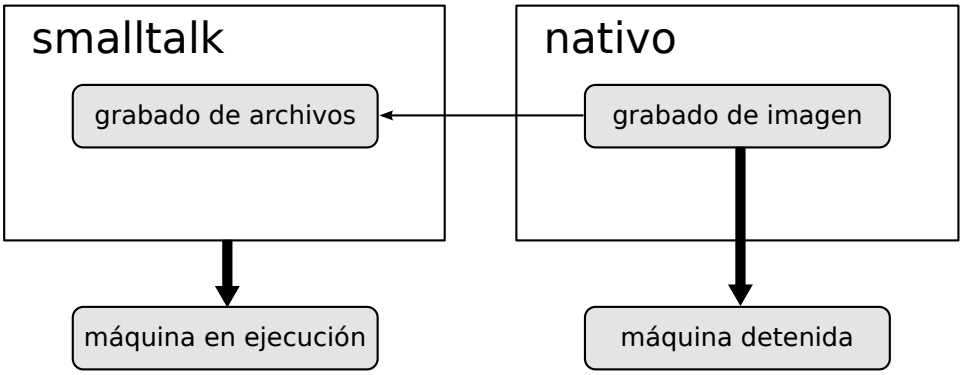


Figura 1.3: Grabado de la imagen en un SqueakNOS

atómica. Por lo tanto esta es una de las preguntas más importantes que surgen en este trabajo y es complementaria a las preguntas enunciadas con anterioridad: ¿Cómo se logra simular la atomicidad en SqueakNOS para poder persistir la imagen?.

En este trabajo se proponen dos técnicas que logran simular la atomicidad requerida. Ambas logran el mismo objetivo, pero recorren diversos caminos y utilizan distintas herramientas y técnicas. Una implica mayor trabajo en el motor de objetos y en algún punto va en contra de la filosofía de SqueakNOS por motivos que ya se explicarán en la sección 2.4. La otra utiliza intensamente el mecanismo de *paginación* [Sta97] provisto por la mayoría de las arquitecturas de procesadores de uso general en la actualidad para una eficiente administración de la memoria. Para esto último, hubo que modelar conceptos de administración de memoria y se abrieron interesantes incógnitas en cuanto a la factibilidad de poder resolver *fallos de página* [Sta97] completamente a alto nivel.

Para completar el capítulo introductorio resulta significativo enfatizar sobre el carácter integral que tuvo el proceso de investigación y desarrollo por el cual se llegó a los resultados aquí presentados. Los sistemas operativos son, en general, elementos de software de gran complejidad, que abarcan gran cantidad de temas y cuyos límites son bastante difusos. Se repasó bibliografía al respecto y surgieron a nivel pragmático y teórico muchos de estos temas en el transcurso del trabajo. Asimismo,

las máquinas virtuales [Wik], que poseen propiedades similares a los sistemas operativos en cuanto a complejidad, tamaño y muchas otras cuestiones fueron un tema central en el marco de este trabajo. Hubo que entender grandes porciones del código del motor de objetos de *Squeak* y se usaron máquinas virtuales de sistema [SN05] para poder probar SqueakNOS sin la necesidad de reiniciar el sistema ni crear particiones especiales para alojarlo. La necesidad de reiniciarlo fue muy frecuente en el comienzo, debido a que todavía no se contaba con soporte de persistencia y a que usualmente se estaba interactuando con modelos de muy bajo nivel que suelen generar errores críticos para el sistema.

También relacionado con el carácter integral de esta tesis, los lenguajes de programación tuvieron su papel. SqueakNOS está ideado para ser un paso adelante en el área de los lenguajes dinámicos y especialmente de Smalltalk. Sin embargo, el motor de objetos es traducido a lenguaje *C* antes de ser compilado en un programa objeto y fue necesario trabajar a ese nivel a la hora de investigar o modificar algunos de sus componentes. Como se trabajó a muy bajo nivel en ciertas cuestiones (manejo de memoria, controladores de dispositivos, interrupciones de hardware), también hubo que agregar algo de código *Assembler*. Aunque la idea fue siempre trabajar la mayor cantidad de tiempo posible dentro del lenguaje Smalltalk, en muchos momentos hubo que entender importantes porciones de comportamiento escritas en los lenguajes de bajo nivel mencionados y también en momentos particulares fue primordial cambiar o agregar algo de comportamiento a esos modelos. Además, fue importante entender los distintos requerimientos para lograr la interacción necesaria entre los distintos niveles en ciertas funciones que así lo exigían.

Finalmente, hay tres cuestiones centrales en cualquier proceso de desarrollo de *software* que no han sido pasadas por alto y que tienen sus particularidades dentro de SqueakNOS. Ellas son la *depuración*, el *testeo* y el previo análisis de riesgos. Es extremadamente complejo depurar código de bajo nivel y más aún cuando no hay un sistema operativo ofreciendo funcionalidades como la impresión de texto en una consola. Además, acostumbrados al proceso de desarrollo en Smalltalk, donde el dinamismo lleva a veces hasta a programar en el *depurador*, trabajar sobre un ambiente básico donde ni siquiera se pueden usar las herramientas de *depuración* de los lenguajes de bajo nivel, resulta extremadamente complejo. Y es realmente difícil desarrollar una aplicación sin lograr tener información durante su ejecución. Para atenuar estas dificultades, se decidió implementar una *consola de depuración* para lograr tener a mano información importante durante la ejecución del motor de objetos. Ya que ante caídas del sistema se perdía la información, se decidió implementar un pequeño *workaround* mediante el cual se recibe desde otra máquina, por puerto serie, la información de la consola de SqueakNOS, y de esta manera se logró mejorar considerablemente el proceso de desarrollo. Llegando al final del trabajo, y ante situaciones que requerían una profundidad mayor de información de la ejecución para ser superadas, se logró depurar al mismo nivel que en un entorno estándar gracias a las herramientas de depuración provistas por la máquina virtual de sistema usada (VMWare). Probablemente, la conjunción de esta con las herramientas implementadas lograron proveer más información todavía que la obtenida en los entornos estándar.

Con respecto al *testeo*, es complejo probar componentes de bajo nivel o testear el comportamiento de SqueakNOS sobre un ambiente de desarrollo. Hubiera resultado interesante una política más

agresiva de testing, pero igualmente se presentan bastantes *tests de unidad*. Se implementaron objetos especiales, que emulan partes de otros como por ejemplo emuladores de controladores de hardware. De esta manera es posible testear ciertos modelos, como por ejemplo el del sistema de archivos, sin depender de la plataforma de ejecución.

El *análisis de riesgos* previo a comenzar el trabajo dictaminó que los posibles problemas estaban casi exclusivamente centrados en la factibilidad de los objetivos propuestos. Además, hay un límite difuso con respecto a cuanto es necesario desarrollar en la imagen por sobre el motor de objetos para no caer en la clasificación de lo que sería un sistema operativo clásico. La idea es forzar ese límite lo más posible hacia el lado de la imagen. El problema de la factibilidad está relacionado simplemente con no cruzar ese umbral, o en su defecto con no situarlo en un lugar no deseado ni por los idearios del proyecto ni por los autores de este trabajo. El otro importante riesgo hallado estaba fuertemente relacionado con la casi nula existencia de proyectos similares o el fracaso de los pocos conocidos. La fuerte convicción a partir de los conocimientos adquiridos, sumado a la existencia de un proyecto de base y el convencimiento de sus creadores ayudaron enormemente a sentir que esos riesgos estaban lo suficientemente mitigados.

El resto de este trabajo se encuentra organizado de la siguiente manera: en el capítulo 2 se presentan diversas secciones explicando los diferentes conceptos importantes necesarios para comprender cabalmente este trabajo; en el capítulo 3 se presenta el estado del arte referido a los tópicos de esta tesis; en el capítulo 4 se enumeran y formalizan los problemas centrales que se intentan resolver; en el capítulo 5 se detalla la solución presentada, el núcleo del trabajo mismo, para luego, en el capítulo 6, presentar los problemas más importantes que fueron surgiendo en el transcurso de su implementación y exponer sus soluciones. En el capítulo 7 se presentan las conclusiones a las cuales se llegó para finalizar con el capítulo 8 en el cual se presentan las posibilidades de nuevos trabajos que surgen a partir del trabajo realizado.

Capítulo 2

Una breve reseña de Squeak y SqueakNOS

Para una mejor comprensión de los aspectos abarcados por esta tesis, es necesario comprender profundamente los proyectos Squeak y SqueakNOS, así como también las tecnologías subyacentes en ambos.

2.1. Sistemas operativos

Uno de los elementos más relevantes y complejos de *software* encontrados en casi todos (o todos) los sistemas digitales es, sin dudas, el sistema operativo. Entre la vasta mayoría de los mismos, existe alguna de las tantas variantes de este tipo de programas. Los sistemas operativos pueden ser desarrollados con muy distintos requisitos, objetivos y contextos, cubriendo el espectro que va desde los sistemas de propósito general (ostensibles en nuestras computadoras hogareñas) hasta los sistemas de tiempo real (estos últimos encontrados en dispositivos con requerimientos críticos como pueden ser los elementos de sensado en los aviones para dar un ejemplo concreto). Dado los diversos contextos de uso, la gran cantidad de funcionalidades que proveen, su no menos notoria complejidad, los distintos enfoques para desarrollar sistemas en la actualidad y su versatilidad, es una ardua tarea encontrar una definición concisa de estos avalada por la comunidad científica.

Algunos autores los definen como una interfaz entre el *hardware* y el usuario, siendo el software responsable de gestionar los recursos de una computadora [SGG08, Sta97, NL06]. En otras definiciones los sistemas operativos pueden ser considerados como una capa de software que reifica una máquina, tomando el rol de una máquina virtual [Tan07]. Esto significa que el sistema introduce numerosas abstracciones relacionadas con los componentes físicos que posee y ofrece interfaces para interactuar con ellos a alto nivel, logrando abstraer la complejidad del hardware. Comprender a los sistemas operativos de este modo, como abstracciones de alto nivel en contraste con simplemente denotarlos como administradores de recursos, parece encajar mejor con la evolución de las entidades de software. Además, se aproxima a las ideas presentadas por el paradigma de programación orientada a objetos, pilar de este trabajo.

Los primeros sistemas operativos fueron programados en *Assembler*. Hubo una importante revolución cuando el lenguaje *C* lo remplazó [SGG08]. Como *C* es un lenguaje de más alto nivel, se pudieron modelar mejores abstracciones y la tarea del desarrollador de sistemas operativos se volvió mas práctica [Han01]. Sin embargo, así como las computadores evolucionaron y hoy son considerablemente mas rápidas, los lenguajes de programación en particular y el desarrollo de software en general también evolucionaron y *C* hoy es más conocido como un lenguaje de bajo nivel. No obstante, los sistemas operativos siguen estando generalmente escritos en *C* y sus derivados, y la opinión general establece que es la única manera de hacerlo[Sta97]. Una de las conclusiones más fuertes que propone SqueakNOS, y que este trabajo intenta profundizar, es la incorrectitud de esa hipótesis en la actualidad.

2.2. Máquinas virtuales

Una forma de disminuir la complejidad de los sistemas es mediante la separación en niveles de abstracción delimitados por interfaces bien definidas. En los sistemas digitales, los niveles de abstracción están separados en jerarquías, siendo los niveles más bajos de estos implementados en *hardware* y los más altos en *software*. A pesar de las grandes ventajas de estos métodos, todavía existen problemas, como por ejemplo cuando programas compilados para correr con una interfaz (la ABI que implementa un sistema operativo) y se intentan ejecutar bajo otra. La *virtualización* provee una forma de relajar este tipo de problemas y aumentar la flexibilidad de los sistemas o programas. Formalmente, la virtualización implica un isomorfismo que mapea una máquina virtual con una máquina real [SN05].

Las distintas interfaces que provee una máquina virtual dependerán de la función que se espera de esta. En general implementan o una *arquitectura de instrucciones* (más conocida como ISA, que marca la división entre *hardware* y *software*) o una interfaz de aplicación binaria (ABI, que provee a los programas acceso a los recursos mediante *llamadas al sistema*) o una interfaz de programación (API, que especifica una abstracción de los servicios que involucran recursos privilegiados). Una máquina virtual es implementada agregando una capa de *software* a una máquina real (o virtual) para soportar la arquitectura deseada. Dependiendo del tipo de interfaz que implemente, tendrá funcionalidades totalmente distintas.

Así como los procesos y los sistemas operativos tienen una perspectiva totalmente distinta de la máquina subyacente a ellos, existen máquinas virtuales de proceso y de sistema que a grandes rasgos conllevan la misma diferencia. En las primeras, el software de virtualización emula las instrucciones de usuario y las llamadas al sistema, implementando la interface conocida como ABI (Aplication Binary Interface). Por su parte, las de sistema proveen un entorno completo implementando la interface a nivel ISA (Instruction Set Architecture), aunque no siempre corren sobre un ISA diferente al implementado. El software de virtualización en este caso es llamado *Virtual Machine Monitor* (VMM).

En el transcurso de este trabajo fueron utilizadas diversas máquinas virtuales. El ambiente Squeak corre una máquina virtual de proceso, el motor de objetos, que interpreta *bytecodes*. Además,

para correr SqueakNOS y poder hacer pruebas sin tener que reiniciar la máquina ni configurar particiones especiales o el boot loader, se usa *VMWare* [Vmw] o *VirtualBox* [Vbo] que son máquinas virtuales de sistema. También lo es *Bochs* [Boc], que tiene una herramienta de *depuración* que resulta muy útil para descubrir ciertos errores críticos de muy bajo nivel.

2.3. Squeak

Squeak es un dialecto de Smalltalk. Es una implementación *open-source* y sus dos componentes esenciales son los mismos que los de cualquier ambiente Smalltalk: el motor de objetos y la imagen. El primero tiene como sus dos principales componentes a un intérprete de *bytecodes* y un *recolector de basura*. La imagen por su parte, contiene los objetos, los cuales se envían mensajes entre sí, siendo el intérprete el responsable de ejecutar las instrucciones correspondientes a los *bytecodes*. Esta es una breve descripción del funcionamiento del ambiente de Smalltalk Squeak en particular, pero también de los ambientes de Smalltalk en general. Se profundizará un poco más en la descripción de los principales componentes de Squeak.

2.3.1. Motor de objetos

El lenguaje Smalltalk propone una única metáfora para desarrollar *software*, la de *objetos* enviándose *mensajes*. Esto conlleva una simplicidad conceptual, al entender los modelos computacionales como un concepto uniforme, proveyendo una herramienta de gran valor para el desarrollo bajo el paradigma de programación orientada a objetos [AG]. El encargado de ejecutar los cómputos relacionados con estas comunicaciones es lo que comúnmente se conoce como máquina virtual. Durante este trabajo se sigue la nomenclatura de Tim Rowledge [Row01] y se lo denota motor de objetos, ya que el término máquina virtual es demasiado general. Además, denominándolo motor de objetos, se enfatiza la inclusión de ciertos conceptos como el *recolector de basura* o de clases como Class o MethodContext que son de muy bajo nivel y están en gran medida circunscriptas al ámbito del motor de objetos. El motor de objetos es un programa objeto y es una máquina virtual en el sentido que le presenta una abstracción de una máquina a sus usuarios, que en este caso son los objetos que viven en la imagen. Cuando estos se envían mensajes, es este motor el encargado de buscar la implementación del mensaje solicitado, crear los objetos que representan los contextos de ejecución, actualizar el estado de la máquina, hacer los cálculos correspondientes, ejecutar primitivas en caso que corresponda y en el transcurso de todo este proceso manejar los posibles errores y la memoria.

Está mayormente implementado en *Slang*, un subconjunto del lenguaje Smalltalk. Su código fuente esta escrito dentro de una imagen de Smalltalk, y luego es traducido a *C* usando una herramienta llamada *VMMaker*. Además, puede compilarse con *plugins*, que son extensiones que debieran ser usadas solamente para mejorar el rendimiento, o cuando ciertos recursos necesitan una comunicación especial con la imagen (por ejemplo cuando usos especiales de memoria son requeridos). Los *Plugins* también son escritos en *Slang*.

El código traducido de *VMMaker* es finalmente añadido a código nativo específico para la plataforma donde se ejecutará el programa objeto y finalmente es compilado y enlazado para obtener el binario correspondiente al motor de objetos. Todo este proceso es relativamente complejo ya que el intérprete involucra mecanismos bastante complejos de por si y el *Slang* es bastante cercano al código de bajo nivel.

2.3.2. Imagen

Aunque es uno de sus componentes esenciales, no hay mucho para remarcar sobre la imagen. Básicamente, es una porción de la memoria donde se almacenan todos los objetos que son administrados por el motor de objetos. Squeak define su propio formato de objetos y una disposición especial de los mismos dentro de la porción de memoria correspondiente [Row01]. Por ejemplo, los objetos de más reciente creación se alojan en la parte alta de memoria y son recolectados por el *recolector de basura* con mayor frecuencia que los objetos más antiguos, implementando un *recolector de basura* de tipo *generation scavenge*. Por más que los cómputos los realice el motor de objetos, el desarrollador trabaja completamente en la imagen cuando está modelando ya que trabaja con los objetos. En este trabajo la persistencia de la imagen tiene una gran trascendencia y esto implica ni más ni menos que la persistencia de la porción de memoria que está siendo descripta en esta sección.

2.4. SqueakNOS

SqueakNOS es un proyecto cuyo principal objetivo es lograr deshacerse del sistema operativo bajo el cual corre el motor de objetos de Squeak. La filosofía que sigue el proyecto es la de desarrollar lo máximo posible dentro de la imagen (en Smalltalk) dejando lo mínimo indispensable para extender o adaptar el motor de objetos (*C y Assembler*). Fue iniciado en el año 2000 por Gerardo Richarte y Luciano Notarfrancesco y tuvo un segundo envión en 2006 después de estar inactivo durante algunos años.

SqueakNOS implementa conceptos comunes a los sistemas operativos, pero bajo la teoría de programación orientada a objetos. Interactúa con el hardware directamente desde Smalltalk, brindando un protocolo dentro del mismo ambiente a los demás objetos que necesiten utilizar los recursos. A diferencia de los sistemas operativos de uso general más utilizados, esta interacción es completamente dinámica y sus propiedades no son diferentes a las de cualquier otra comunicación entre objetos dentro del ambiente Smalltalk. Aprovechando el dinamismo inherente de Smalltalk, se pueden cambiar mensajes en tiempo de ejecución del sistema sin necesidad de un reinicio. Esto permite, por ejemplo, cambiar controladores de dispositivos mientras se están usando, un tipo de dinamismo impensado para el área de sistemas operativos.

En lugar de reemplazar la API del sistema operativo con su propia implementación en código C o Assembler, SqueakNOS implementa directamente el comportamiento para interactuar con los recursos del sistema en la imagen. Así logra cumplir con el objetivo de mantener la cantidad de

código de bajo nivel en los mínimos indispensables. Además, la idea es adaptar los objetos que implementan la interacción con el hardware a los objetos existentes en la imagen para tratar con recursos. Ejemplos de esto serían la jerarquía de clases de *FileStreams* para archivos o los eventos *Morphic* para el mouse y el teclado. De este modo se mantiene sin cambios (lo máximo posible) el núcleo de *Squeak*. Dado que el ambiente está en constante cambio, mantener atenuadas las modificaciones necesarias de la imagen, conlleva el beneficio de poder adaptar fácilmente nuevas versiones de Squeak a SqueakNOS.

SqueakNOS implementa la mayor parte de las extensiones necesarias al motor de objetos en un *plugin* escrito en *Slang* llamado *SqueakNOSPlugin*. Esto permite la separación prolija del código especifico de SqueakNOS con el código del motor de objetos. Algunos comportamientos específicos no son posibles sin antes hacer algunos cambios al motor de objetos, y otros como el manejo de interrupciones son específicos del hardware donde corre, por lo tanto, estas pequeñas porciones de comportamiento están escitas en *C* y *Assembler* para cada plataforma.

2.4.1. Evolución de la implementación de SqueakNOS

Al momento de iniciada la tesis, SqueakNOS contaba ya con soporte para teclados *PS/2*, *mouse*, *puerto serie*, *VESA*, *BIOS*, *PICs*, *PCI* y *placas de red estándar*. Se resumirá aquí como este soporte fue evolucionando, ya que al hacerlo se ofrecerá una visión incremental de los componentes del proyecto.

En la etapa de creación de SqueakNOS, el primer objetivo fue que la imagen y el motor de objetos modificados sean iniciados por algún gestor de arranque y esto se logró mediante uno ampliamente conocido y *open-source*. En primer lugar fue *Lilo* [Lil], para luego cambiar a *GRUB* [Gru]. Se debieron aplicar modificaciones a ambos para lograr cargar SqueakNOS. Al primero debido a que no permitía cargar imágenes del tamaño necesario. Al segundo, hubo que hacerle modificaciones para que permita cambiar el modo de video al inicializar. Grub es el arrancador por defecto de la mayoría de los sistemas *Linux* modernos y posee importantes ventajas como por ejemplo el manejo de inicialización de video (se le hicieron modificaciones ya que las primeras versiones no tenían todo el soporte necesario en SqueakNOS) y soporte para carga de módulos. Además, respeta el estándar *Multiboot* [Mul], lo cual permitiría cargar el kernel de SqueakNOS sin grandes modificaciones de existir otro gestor de arranque que implemente el mismo estándar.

Una vez logrado el primer paso, se implementó el soporte para interrupciones. Éstas, son atendidas por una rutina en código de bajo nivel genérica, que se encarga de notificar la emisión de las mismas a la imagen y de retornar de la interrupción. Dentro de la imagen, cada rutina de atención de interrupciones se encuentra en un proceso de alta prioridad distinto, bloqueado por un semáforo a la espera de que la interrupción ocurra. Cuando esta sucede, la rutina genérica libera el semáforo correspondiente y vuelve de la interrupción (avisándole al dispositivo que ya fue atendido). Es decir, si observáramos el pseudocódigo de la rutina de atención de interrupciones nativa veríamos los siguientes pasos:

1. Apilar todos los registros del procesador

```
installOn: aComputer
    self registerSemaphore.
    process := [
        [
            semaphore wait.
            self handleOn: aComputer.
            aComputer interruptController signalEndOfInterrupt: interruptNumber
        ] repeat.
    ] forkAt: Processor highIOPriority.
    process name: 'Iterrupt Dispatcher for IRQ#:', interruptNumber printString.
    aComputer interruptController enableIRQ: interruptNumber
```

Figura 2.1: Médodo para la instalación de las rutinas de atención de interrupciones

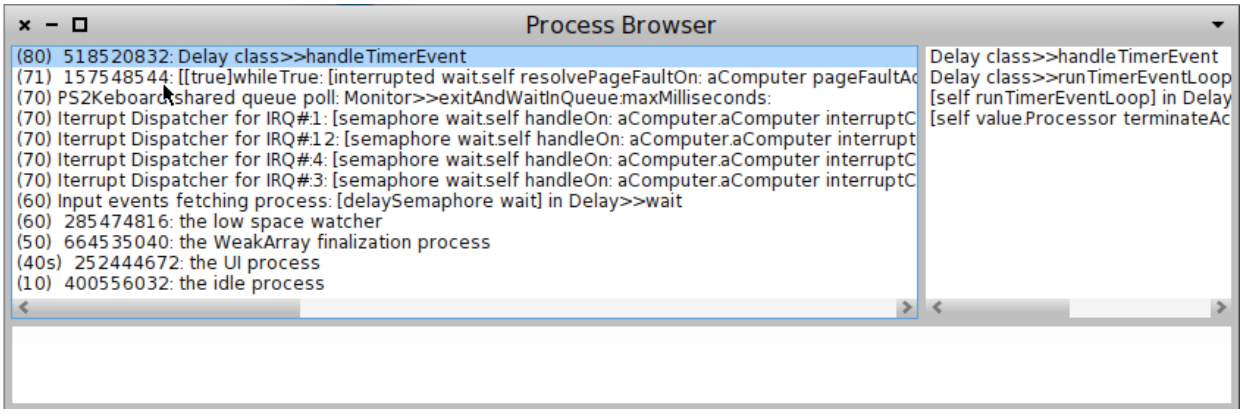


Figura 2.2: Procesos de las rutinas de atención de interrupciones suspendidas en espera

2. Desbloquear semáforo correspondiente a la interrupción

3. Restaurar todos los registros del procesador

4. Volver de la interrupción mediante *iret*

Como puede notarse, el código no atiende la interrupción directamente sino que habilita al proceso de Smalltalk que se encuentra esperando a que continúe ejecutando. Al retomar el control, el motor de ejecución detecta que un proceso de alta prioridad está listo para ser ejecutado y de esa manera finalmente se ejecuta, desde código Smalltalk, la atención de la interrupción. Finalmente, se implementaron algunos controladores de dispositivos básicos y se solucionaron los pequeños inconvenientes encontrados y el proyecto logró cargar la imagen sin un sistema operativo de uso general por detrás. En la figura 2.1 puede verse el código Smalltalk utilizado para la instalación de un proceso que quedará a la espera de la activación del semáforo cuando una interrupción ocurra. Además, en la figura 2.2 se muestra un explorador de procesos, en el cual puede observarse como los distintos controladores han instalado procesos que se encargarán de atender las interrupciones.

Como se explicó al comienzo de esta sección, SqueakNOS tiene modelados y soporta todos los componentes esenciales para ser usado, sin embargo, varias características importantes siguen faltando, como por ejemplo la persistencia. Ésta es relevante, fundamentalmente, para poder mejorar y testear el ambiente desde el mismo ambiente.

2.5. Controlador ATA

El estándar ATA es una interfaz utilizada para conectar dispositivos de almacenamiento como los discos rígidos y los discos de estado sólido. La especificación cubre a grandes rasgos dos áreas principales: la descripción física de los componentes por un lado, y el modelo de programación de los dispositivos por otro. De ellas, la que interactúa con este trabajo es solamente la última, el modelo de programación. Esta interfaz es el resultado de un desarrollo histórico incremental, realizado desde los comienzos de la *PC*. La misma ha ido evolucionando y sigue siendo utilizada en la actualidad. Cada versión actualizada incorpora nuevas funcionalidades, pero en su gran mayoría mantienen la compatibilidad hacia atrás, siendo la última versión la número 7. Una de sus más grandes ventajas es que debido a su compatibilidad, es posible acceder a la funcionalidad básica de los dispositivos más nuevos utilizando controladores diseñados para el estándar original, que es mucho más simple y está más documentado.

Según la especificación, un medio de almacenamiento es un gran arreglo de sectores conjuntos de bytes de tamaño fijo y único para el dispositivo. En la versión original cada sector era accedido mediante un índice que indicaba el cilindro, sector y cabezal donde se encontraba almacenado, pero esta forma de acceso fue reemplazada por otra más cómoda, en la que el índice es simplemente un identificador lógico del número de sector. No es posible acceder a los bytes por separado directamente, sino que para ello es necesario primero mover el contenido del sector a la memoria principal. ATA permite un acceso a los sectores de tipo aleatorio, y para un mayor desempeño permite también accesos en ráfaga a sectores contiguos. Estos accesos pueden realizarse mediante dos técnicas comunes: tanto por *polling*, un mecanismo de sencilla implementación pero bajo desempeño, como por interrupciones.

La configuración de un dispositivo, tanto para lectura como para escritura de datos, se realiza mediante la lectura y escritura de los registros del mismo. Cada operación se realiza indicando el número de sector inicial que se desea acceder, y la cantidad de sectores totales. Una vez que el dispositivo fue configurado, para obtener finalmente el contenido de los sectores se realiza la lectura de un registro de datos. Es necesario tanto antes de comenzar la configuración, como después de realizarla y antes de comenzar a leer, esperar a que el dispositivo se encuentre listo. Esto se realiza mediante la lectura de un registro de estado. En caso de acceder a más de un sector a la vez, es necesario esperar también a que el dispositivo esté listo después de cada lectura individual.

2.6. FAT32

FAT32 es una versión más (la más reciente), de la familia *FAT* (File Allocation Table) de *sistemas de archivos* diseñados por Microsoft [FAT]. La principal diferencia entre las versiones está en el tamaño de los archivos que soportan, siendo las versiones modernas las que soportan archivos de mayor tamaño. Además, cuanto más reciente la versión, pueden ocupar un mayor tamaño de disco. Esto es fundamentalmente debido a que las nuevas versiones utilizan más bits para describir los tamaños y espacios que usan. *FAT32* consta principalmente de cuatro elementos:

el *sector de booteo*, la *FAT*, los *clusters* y los datos. El disco no suele verse como una secuencia de sectores, sino como una de *clusters*, que son agrupaciones lógicas de sectores consecutivos. Los clusters tienen un tamaño fijo y único para todo el disco, y en general, equivalen a 1, 2, 4 u 8 sectores.

El sector de booteo, es el primer sector de la secuencia de estos asignados al *sistema de archivos*. Contiene la información de control necesaria para poder acceder a los datos tal como la cantidad de bytes por sector, la cantidad de sectores de cada cluster y la cantidad de sectores totales.

Luego, en la *FAT* se definen los metadatos y en los clusters los datos mismos. La primera es una estructura que esta compuesta por un arreglo de clusters. Dentro de este arreglo se definen listas simplemente encadenadas que representan todos los clusters de un archivo o directorio. La única manera de recomponer los datos de un archivo dispersos físicamente en el disco es mediante esta lista. Explorando el cluster correspondiente y luego siguiendo la cadena de clusters, se pueden encontrar los datos de cualquier archivo o directorio. Debido a su central importancia, la FAT está duplicada para lograr redundancia en caso de fallos.

La especificación define un directorio raíz en una posición fija, que es el primer cluster de la sección de datos. En la misma sección se encuentra el contenido de cada archivo, que se obtiene juntando los datos de los clusters correspondientes encadenados en la FAT. También se encuentra en esta sección el contenido de cada directorio. Para ellos, los clusters están divididos en porciones de 32 bytes que describen los archivos y subdirectorios existentes en él.

Los sistemas operativos modernos proveen por defecto otras opciones de sistemas de archivos, por supuesto que mucho más modernos también, y que ofrecen varias ventajas en cuanto a muy diversos parámetros. Sin embargo, *FAT*, sigue siendo frecuentemente usado por su simplicidad y su eficiencia en dispositivos de poca capacidad y donde la estructura de archivos no es tan compleja, como en los *Pendrives*. En la sección 5.1.5, se compartirán las razones por las cuales este sistema de archivos fue elegido.

2.7. Administración de memoria

La administración de memoria es una de las tareas mas complejas que realiza un sistema operativo. Si bien los costos de este recurso han disminuido considerablemente en la actualidad, de todas maneras nunca hay suficiente para todos los programas del sistema, sus estructuras de datos, sus datos, etc. Para mantener los procesadores activos, el sistema debe intercambiar lo más inteligentemente posible los datos entre la memoria y el disco y mantener la mayor cantidad posible de procesos en memoria. Para todo esto, existe el administrador de memoria, el cual debe satisfacer la mayor cantidad posible de los siguientes requerimientos: realocación, protección, compartición, organización lógica y organización física [Sta97].

Una de las técnicas mas comúnmente usadas para administrar la memoria es la de particionado. Los programas son particionados en bloques mas chicos y la memoria también es particionada en bloques. Las particiones pueden ser para separar el programa lógicamente como en la *segmentación*. En el caso de la *paginación*, las particiones son de tamaño fijo y esto permite reducir mucho la

posibilidad de fragmentación de la memoria. El programador debe estar al tanto de las particiones en el caso de la *segmentación*, sin embargo, el *paginado* es totalmente transparente para el mismo.

Ambos esquemas poseen todas las características para ser usados como *memoria virtual*. Las direcciones de los programas son lógicas y deben ser traducidas en tiempo de ejecución a sus correspondientes direcciones físicas. Además, sus particiones no deben estar en memoria física todas al mismo tiempo, sino sólo las que se necesitan ejecutar o referenciar por las instrucciones a ejecutar. Estas formas de administrar la memoria suelen funcionar bastante bien debido al principio de *localidad de las referencias*. En caso de que el sistema operativo pase demasiado tiempo trabajando en administrar la memoria y poco en ejecutar las instrucciones de los usuarios, se genera una disminución importante del tiempo de respuesta para el mismo y se conoce a esta situación como *trashing*.

Smalltalk maneja su propia porción de memoria de manera autónoma. Esto significa que le pide al sistema operativo una cantidad de espacio y luego el motor de objetos es el encargado de manejar la alocación, limpieza, y compactación de los objetos. Sin embargo, todos los dialectos de Smalltalk dependen del manejo del sistema operativo sobre la porción de memoria que ellos administran. Esto significa que el espacio donde residen los objetos puede no ser continuo o que parte de él puede estar en disco. SqueakNOS, en cambio, es el encargado de administrar el espacio de memoria donde viven sus objetos completamente, sin injerencia de otro ente. La versión con la que el trabajo fue iniciado, administra la memoria como un bloque contiguo que nunca decrece donde no existe ningún tipo de protección, ni organización lógica, ni posibilidad de tener más objetos que los que entren en la cantidad de memoria disponible.

2.7.1. Paginación

La paginación es el esquema de manejo de memoria virtual más usado en la actualidad. Consiste en particionar los programas en bloques al igual que la memoria. Los bloques de los programas son copiados a memoria cuando son referenciados o deben ser ejecutados. El sistema operativo debe proveer al procesador una *tabla de páginas* para el proceso que se está ejecutando. Esta tabla indica esencialmente para cada dirección lógica, en que bloque de memoria principal está, en caso de estar copiado en ella o, en caso de haber sido descargada a disco, simplemente su ausencia y el lugar del disco en donde se encuentra. El proceso de traducir las direcciones lógicas (página, offset) en direcciones físicas es responsabilidad del *hardware*.

La realocación en este esquema de manejo de memoria está garantizada ya que las direcciones son lógicas y son traducidas a direcciones físicas en tiempo de ejecución por el *hardware*. En la *tabla de páginas* se puede especificar si la página correspondiente es escribible o es de sólo lectura con lo cual el esquema provee algún tipo de protección. Además, el proceso puede referenciar solo las páginas que están en su *tabla de páginas* lo cual protege aún más al sistema. La compartición se logra referenciando una misma página desde la *tabla de páginas* de dos procesos diferentes.

El otro sistema de manejo de memoria más conocido es la *segmentación*. Bajo este esquema, el espacio de direcciones es dividido en unidades lógicas de tamaño variable llamadas *segmentos*. Los

segmentos tienen una dirección base indicando su comienzo en la memoria física. Cada programa tiene múltiples segmentos descriptos en una *tabla de segmentos*. La protección y compartición de memoria son más simples en los esquemas segmentados ya que el programa se divide en particiones lógicas (datos, código, etc) donde la separación la debe especificar el programador [NL06].

Sin embargo, es muy raro encontrar un sistema operativo importante que use como esquema de manejo de memoria la *segmentación*. En todo caso, se encuentran sistemas híbridos, donde la mayor parte del trabajo es realizado en la administración de la *paginación*. La mayoría de los procesadores modernos incluyen una *unidad de gestión de memoria (MMU)* con soporte para la *paginación* [Sta97].

Fallo de página

Cada vez que se ejecuta una instrucción que hace referencia a una posición de memoria, la MMU accede a la entrada correspondiente en la *tabla de páginas*, y verifica que esta pueda ser accedida, es decir, que esté marcada como presente en memoria, y con permiso de escritura, si es que la posición se accediera para escribirse. Si esta verificación diera que la página no está presente en memoria o que no se cuenta con permiso de escritura, se genera un *fallo de página*. En este caso el procesador, pone en un registro la dirección virtual que no pudo resolver y lanza una interrupción la cual debe ser atendida por el sistema operativo. El *administrador de memoria* es el encargado entonces de discernir a que se debió el fallo. Si fue porque la página no estaba presente, deberá traerla de disco, buscarle un lugar en la memoria y actualizar la *tabla de páginas* para que quede consistente con la nueva distribución de la memoria. Una vez terminado este proceso avisará al procesador que la interrupción ya fue atendida y este podrá continuar ejecutando ya que la dirección referenciada ahora está alocada en memoria [Sta97]. A este proceso de interactuar con la memoria secundaria para alojar las páginas de memoria mientras estas no están siendo usadas y luego volver a alojarlas en memoria se lo conoce como *swapping*. En el caso de que la interrupción fuera por un error de protección, esto podría sugerir que se está intentando acceder a una zona de memoria no debida, y se deberá actuar según corresponda.

En el caso del swapping, es importante la selección de las páginas que hay que remover de memoria para hacer lugar a nuevas en caso de que no haya lugar disponible y hay mucha teoría al respecto ya que diferentes algoritmos de remoción producen resultados asombrosamente distintos en un área como esta donde es tan crítica la celeridad. Sin embargo, en este trabajo no se ahondará en este problema sino en el de poder resolver los fallos de página desde un lenguaje de alto nivel.

Los fallos de página por errores de protección pueden utilizarse para tareas variadas, entre ellas se encuentra la implementación eficiente del mecanismo *copy-on-write* (COW) de memoria, el cual será sumamente importante en el proceso de persistir la imagen.

2.8. Resumen

Han sido descriptos los componentes más relevantes que subyacen a la tecnología y herramientas que están involucradas en este trabajo así como también los elementos centrales del trabajo mismo. Los conceptos expuestos en este capítulo son necesarios para comprender cabalmente el resto del trabajo. En el siguiente capítulo se detalla el análisis previo efectuado sobre los diversos trabajos que se relacionan con el problema resuelto.

Capítulo 3

Estado del arte

Por diversos motivos, el diseño de sistemas operativos con lenguajes de alto nivel no es un área muy prolífica. Entre ellos podemos destacar la complejidad inherente de este tipo de software, el cual abarca demasiadas áreas lo que desalienta a muchos a embarcarse en estos proyectos. Además, la imposibilidad de éxito, debido principalmente al mayor rendimiento de los programas escritos en lenguajes de bajo nivel por sobre los de alto nivel, es una creencia profundamente arraigada y la cual se intenta refutar en parte en esta tesis. Otro de las importantes motivos es la falta de recursos humanos con el conocimiento adecuado para este tipo de proyectos. Esto se debe esencialmente a que la gente tiende a especializarse en alto o bajo nivel, pero son pocos los que logran un conocimiento adecuado de ambos niveles. Sin esta capacidad, es muy complejo embarcarse en este tipo de proyectos y poder analizar sus ventajas, desventajas y posibles dificultades. Sin embargo, se analizan a continuación algunos pocos proyectos que intentan una refutación similar y están relacionados con SqueakNOS. Cabe aclarar que en general sus fines son diferentes a los planteados por SqueakNOS así como también sus metodologías, herramientas e implementaciones. Se presenta un esbozo de las principales características de cada uno de ellos. Finalmente, se describe un recolector de basura que utiliza técnicas similares a las de esta tesis.

3.1. JNode (Java OS)

JNode [JNo] (Java New Operating System Design Effort) es un proyecto de software libre planteado para crear un sistema operativo implementado en la plataforma *Java* [Jav]. Este tomó la dirección de desarrollar todo el código en Java con la excepción de un *microkernel* escrito con algo de código *assembler* para el booteo y el cargado de bibliotecas del sistema. El compilador de la máquina virtual de Java (que normalmente usa *just-in-time compilation*) es usado para generar binarios nativos fuera del código del núcleo Java. De esta manera, casi todo el sistema está escrito en si mismo.

Sin lugar a dudas este proyecto es el más similar al proyecto SqueakNOS. El enfoque general es el mismo, siendo en este caso implementar la mayor cantidad del código posible en Java en lugar de Smalltalk. Según algunas métricas no demasiado formales, publicadas en la página web del

proyecto, logran un velocidad de ejecución para código de alto nivel similar a la de correr Java bajo otras plataformas. Sin embargo, no enuncian nada respecto a la velocidad de los manejadores de dispositivos donde el rendimiento debería ser menor. El proyecto no planifica modificar el manejo de memoria, la cual actualmente es administrada como una porción plana de la memoria física, sin direcciones virtuales. Cabe destacar que el proyecto tiene un desarrollo de manejadores de dispositivos similar al de SqueakNOS.

Sin embargo, entre varias diferencias con SqueakNOS, hay dos que son sustanciales. Una de ellas es el fin de cada proyecto. El de SqueakNOS ya fue explicado, el de *Jnode* al parecer tiene que ver más con la seguridad, estabilidad y robustez que le brindaría a los programas Java, el ejecutarse sobre un ambiente *Java* nativo[lsa]. La otra gran diferencia entre los proyectos *Jnode* y SqueakNOS es simplemente la herramienta, es decir, el medio. De un lado esta el ambiente de *Java* y del otro Smalltalk. Pero esta diferencia no es menor, y los problemas técnicos, de factibilidad y de rendimiento pueden variar significativamente. Una diferencia importantísima que aparece al compararlos es que en *Java* no existe el concepto de imagen. Por lo tanto, no existe el problema de persistencia que se desarrolla en esta tesis. Pero además, al no tener este componente, el entorno de desarrollo y ejecución de *Java* es mucho menos dinámico que el de Smalltalk. Los autores creen fuertemente que el dinamismo del ambiente es una propiedad de una importancia superlativa para el desarrollo de modelos complejos como son los que se encuentran en los sistemas operativos. En la figura 3.1 se puede ver la arquitectura del proyecto, donde cabe destacar el predominio del color gris claro, que implica que el componente está implementado en *Java*, por sobre el gris oscuro, visualizado en el *microkernel*, que determina que el componente fue desarrollado con lenguajes de bajo nivel.

3.2. JX

JX es un sistema operativo desarrollado en Java que se enfoca en presentar una arquitectura robusta y flexible [JX]. Esta consiste de un conjunto de componentes desarrollados en Java que se ejecutan en un núcleo de JX el cual es el responsable de la inicialización del sistema, el cambio de contexto de la *CPU* y el manejo de dominios de bajo nivel. Estos componentes son cargados sobre dominios, luego verificados y finalmente traducidos a código nativo. Al igual que *Jnode*, los benchmarks que presentan les dan un rendimiento similar a los de los sistema operativo mainstream y uno de sus principales argumentos a favor es la seguridad que le provee correr sólo *bytecodes* permitidos por la máquina virtual Java. También es open-source.

Los distintos objetos corren en diferentes *dominios*, los cuales son unidades de protección que aíslan los objetos. De esta manera ofrecen seguridad entre el código del sistema y el código del usuario y también entre los diferentes códigos de usuarios. Cada dominio tiene su propio *heap* y su propio recolector de basura. El dominio cero contiene el código de bajo nivel que representa el *microkernel*. Los benchmarks presentados muestran un rendimiento de entre un 40 y un 100 por ciento de mejoras en los tiempos respecto a la ejecución en un *Linux* de diversos manejadores de dispositivos [MG]. El proyecto es muy similar a *Jnode*, sólo que parece un poco más prolífico en

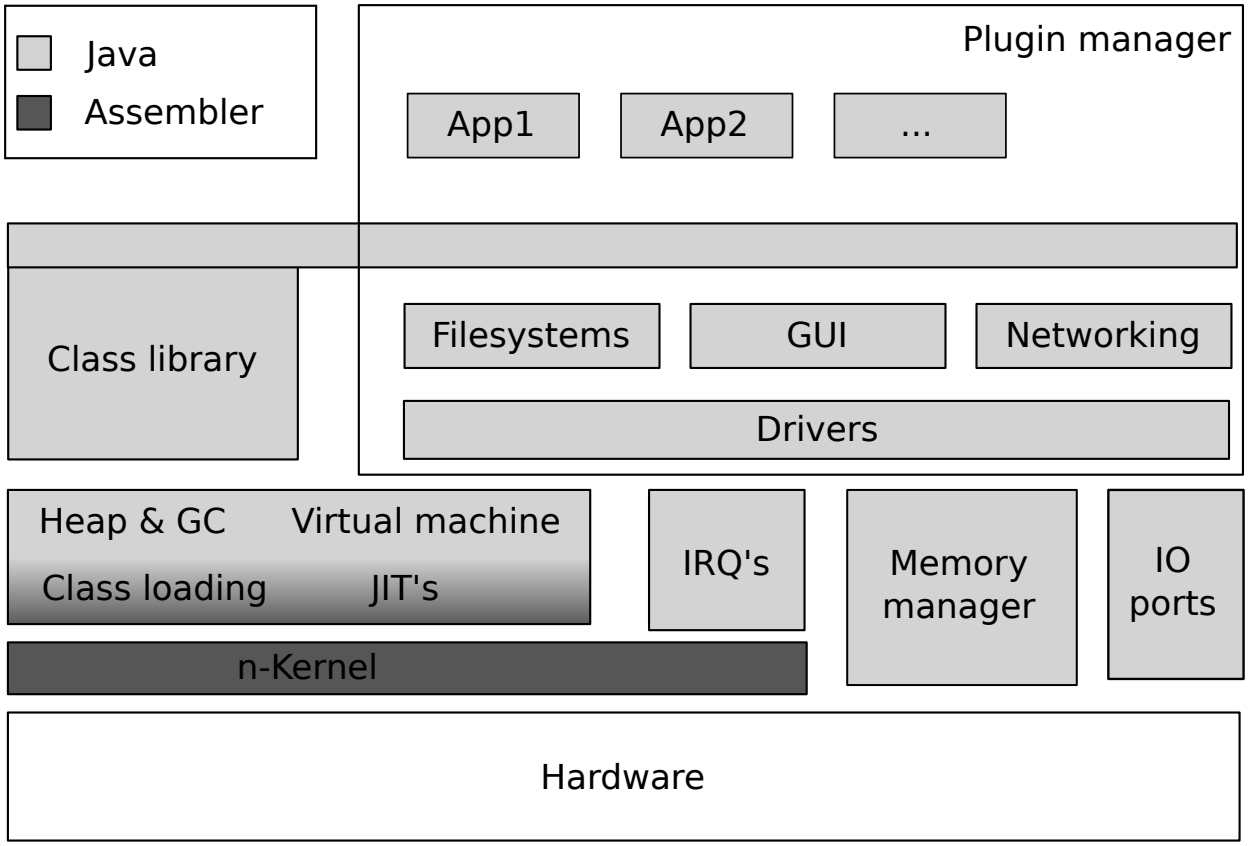


Figura 3.1: JNode arquitecture

cuanto a su documentación, lo que lo hace más valorable para ser citado. Sin embargo, comparte con *Jnode* las mismas similitudes y diferencias con respecto a SqueakNOS.

3.3. House

House [Hou] es un pequeño sistema operativo implementado casi completamente en el lenguaje funcional *Haskell*. Está implementado sobre la base de un proyecto previo conocido como *hOp*. El sistema incluye manejadores de dispositivos, un simple sistema de ventanas, un protocolo de pila para redes y un shell de comandos en el cual archivos binarios pueden ser cargados mediante *TFTP* y ejecutados en modo usuario. Es el primer sistema operativo escrito en lenguaje funcional que soporta ejecución de binarios arbitrarios (no sólo programas escritos en el lenguaje base).

El objetivo de este proyecto es lograr una interfaz monádica para interactuar con los dispositivos de bajo nivel [TH]. A partir de esto, logran enfocarse en la seguridad que provee este tipo de sistemas, al poseer todos los atributos de seguridad del lenguaje funcional, sumado a una separación de dominios. Logran demostrar interesantes propiedades basados en el fuerte tipado del lenguaje y la simplicidad de su sintaxis y semántica. El proyecto se centra fundamentalmente en este tipo de validaciones y no parece haber datos sobre el rendimiento del sistema.

Sin dudas, House es un proyecto muy interesante. Sin embargo, hay una diferencia importante

con SqueakNOS que tiene que ver con que el paradigma de desarrollo es absolutamente diferente, con lo cual las comparaciones pasan a ser probablemente inocuas. Además, al igual que los proyectos Jx y Jnode, no manejan el concepto de imagen, con lo cual la persistencia que utilizan es simplemente a nivel archivos y no documentan ningún problema similar al presentado en este trabajo.

3.4. Recolector de basura

Además de los proyectos relacionados con los sistemas operativos, existe un problema relacionado con la *atomicidad*, que fue resuelto usando técnicas de *memoria virtual*. El proyecto en cuestión exhibe un *recolector de basura* concurrente y de tiempo real, de tipo *parada y copia*, que hace uso de los mecanismos de paginación para implementar la sincronización necesaria entre el recolector y los diferentes mutadores (*threads* que modifican los objetos) [AWA]. Este algoritmo simula la *atomicidad* cuando esta recolectando la basura mientras que los mutadores están al mismo tiempo modificando los objetos de manera similar a la que en este trabajo se simula la atomicidad para persistir la imagen. SqueakNOS sería un ambiente ideal para implementar este tipo de algoritmos de recolección.

Capítulo 4

Problema

El trabajo cubre varias áreas y tiene un carácter integral en cuanto a la cantidad y la relevancia de los diversos tópicos con los que se relaciona. Además, es una extensión a un proyecto base (SqueakNOS), el cual todavía está en etapas de desarrollo y donde hay mucho por investigar y aportar. Esto generó que durante el transcurso del trabajo se fueran sumando interrogantes interesantes donde lo experimentado e investigado podía ser útil. Se puede añadir que los sistemas operativos son un software extremadamente complejo debido a la ingente cantidad de abstracciones con las cuales debe interactuar. Quizás debido a esto, varias preguntas relacionadas con la evolución de los diseños y herramientas para producir *software* todavía no han sido contestadas (o en su defecto, no lo suficientemente demostradas) al circunscribirse a ellos.

En esta sección, se presentan las preguntas que surgieron durante el trabajo, sumadas a los interrogantes iniciales. Además, dados los proyectos relacionados que fueron analizados, se demuestra que estos no cubren completamente esos interrogantes y que algunos de los mismos no han sido ni siquiera explorados. Finalmente, se justifica la relevancia que tiene contestar estas preguntas.

4.1. Sistemas Operativos y lenguajes de alto nivel

El interrogante inicial de este trabajo y de SqueakNOS en general surge de preguntarse,

> ¿Son posibles los sistemas operativos escritos en lenguajes de alto nivel? y más particularmente, ¿son posibles los sistemas operativos escritos en lenguajes orientados a objetos? ¿y qué pasa al circunscribirse al lenguaje Smalltalk?.

El software desarrollado a nivel mundial a cambiado considerablemente en su manera de ser concebido en las últimas décadas. Primero hubo un sustancial salto de abstracción cuando se paso de programar en lenguaje *ensamblador* a programar en lenguaje *C*. Un nuevo salto de abstracción apareció con los lenguajes de programación orientada a objetos, dejando a *C* en el conjunto de los lenguajes de bajo nivel. Sin embargo, este proceso no ha impactado de igual manera en el área de los sistemas operativos. Aunque hubieron algunos avances en esa dirección, hay una importantísima diferencia entre el impacto en el desarrollo de aplicaciones de usuario con respecto al impacto en

los sistemas operativos. Entre los principales argumentos está la pérdida de rendimiento y de la independencia con respecto a los lenguajes de programación usados en la capa de usuario. También podría verse como una desventaja el aumento de tamaño y complejidad dados por el uso de un motor de objetos, así como también el alejarse de la capa más cercana al hardware, aunque otros (los participantes de este trabajo entre ellos) crean que es más una ventaja que una desventaja.

Estudios muestran que los mayores generadores de errores dentro de los sistemas operativos son los manejadores de dispositivos [ACE]. Además, que estos errores tienden a estar presentes en el núcleo durante largo tiempo. Esto puede ser debido, entre otras cosas, a que las metodologías de desarrollo son antiguas y que las herramientas (lenguajes de bajo nivel) no son las adecuadas para lograr varias de las propiedades deseadas en un modelo computable (simplicidad, comprensibilidad, extensibilidad). Por ejemplo, el tener que recompilar todo un kernel, para hacer una prueba que implica modificar una constante de un manejador de dispositivo, parece una tarea extremadamente tediosa y no acorde a las herramientas contemporáneas. Por otro lado, las interfaces que presentan los dispositivos de hardware son cada día más complejas, pero se continúan utilizando desde lenguajes de bajo nivel, lo que aumenta los problemas. Las ventajas que pueden ofrecer los sistemas operativos escritos en entornos dinámicos y de alto nivel son muchísimas, y entre ellas resultan relevantes la influencia educativa que pueden aportar, la influencia en el campo de la *ingeniería reversa* de dispositivos, la alta interactividad que presentan y la influencia en una disminución considerable de los errores en los mismos (o por lo menos una baja en su tiempo de vida).

Sin embargo, luego de explorar el dominio, se desprende que los proyectos como *JNODE* y *JX* (secciones 3.1 y 3.2 respectivamente) parecen ser los únicos en responder los dos primeros interrogantes, mientras que el proyecto HOUSE (sección 3.3) corresponde solo al primero. Sin embargo, esta respuesta puede ser parcial y depender de qué se interprete o busque como sistema operativo. Por ejemplo los proyectos de Java no presentan un manejador de memoria con la posibilidad de manejar memoria virtual. Estas herramientas son tan populares en los sistemas operativos de uso general que la gran mayoría de los procesadores incluyen mecanismos de *paginación*. Entonces surge la duda de si es por un impedimento técnico o si se reduce a una decisión de diseño, lo cual no queda claro en los documentos que explican sus implementaciones. Además, son proyectos que no parecen tener una alta aceptación en la comunidad científica (hay muy poco escrito y referenciado respecto a ellos). Por otro lado, *HOUSE* parece responder afirmativamente el primer interrogante, pero el paradigma funcional es tan poco usado en el desarrollo de *software* en general que no sería suficiente por sí mismo como demostración.

Considerando lo poco explorado que está el interrogante en su nivel más general, es lógico pensar que no existe aún una respuesta a la pregunta restringida al mundo de Smalltalk en particular. En este mundo, la respuesta a esta pregunta es una incógnita aún mayor, ya que Smalltalk tiene características como lenguaje que Java no posee, y esto para el desarrollo de sistemas operativos puede plantear dificultades muy diferentes. Cualidades como la *reflexividad* (en Smalltalk las clases son objetos), la *meta-circularidad* o el dinamismo del lenguaje (en Smalltalk no hace falta compilar los programas y se puede modificar el código hasta desde el depurador) hacen que no sea equivalente para un desarrollador usar una herramienta o la otra.

Se desprende entonces que responder la pregunta sobre la factibilidad de modelar los sistemas operativos en Smalltalk no es algo sencillo ni trivial de hacer, y que lo más similar desarrollado al respecto tiene que ver con proyectos hechos en lenguaje Java. No se encontraron proyectos que hayan explorado ese camino en Smalltalk. Esta incógnita intenta ser contestada por SqueakNOS directamente y es uno de los objetivos mismos del proyecto. Aunque el proyecto se encuentra actualmente en desarrollo, el funcionamiento actual del mismo es claramente un ejemplo que responde afirmativamente a esa pregunta.

4.2. Limitaciones de los Sistemas Operativos en Smalltalk

Al restringirse a Smalltalk, y en particular al manejo de los distintos componentes de la jerarquía de memoria, el espectro de preguntas se amplía y pueden encontrarse otros interrogantes aún más interesantes.

> ¿Es posible manejar las distintas partes de la jerarquía de memoria del sistema directamente desde Smalltalk? ¿puede tenerse un controlador de disco y un sistema de archivos escritos completamente en Smalltalk y cuyo desempeño sea aceptable? ¿y qué tal un administrador de páginas, donde las restricciones son aún mayores?

Es interesante poder refutar algunas creencias generales que aún existen respecto a la conveniencia de implementar sistemas operativos utilizando lenguajes de bajo y alto nivel. En particular, la teoría de que la diferencia de desempeño es tan grande que hace imposible modelar en alto nivel el hardware e interactuar con él de forma eficiente. En el capítulo 3 se describieron proyectos que muestran resultados que refutan parcialmente esa hipótesis. Pero extender SqueakNOS con nuevas funcionalidades no cubiertas por los proyectos anteriores, revalida la idea de que la hipótesis es falsa.

Además, de ser afirmativas las respuestas a las preguntas de la sección anterior,

> ¿sería posible utilizar los modelos creados para manejar el hardware de forma novedosa y resolver así nuevos problemas?

Por los mismos motivos expresados en el párrafo anterior, más allá de lo presentado por el mismo SqueakNOS, no se logró detectar en los proyectos analizados conclusiones al respecto. Como este trabajo extiende SqueakNOS y se expone durante el mismo un claro ejemplo de respuesta afirmativa, resulta interesante remarcar y revalidar el concepto.

4.3. Problemas de implementación relevantes

Al comenzar este trabajo, una funcionalidad esencial aún faltaba en SqueakNOS: el poder persistir los objetos, o sea, la imagen misma. Desde un primer momento apareció el interrogante,

> ¿De qué manera puede implementarse en SqueakNOS el grabado de la imagen? ¿Es posible lograr la persistencia de la imagen en SqueakNOS sin contradecir su filosofía?

La persistencia de los objetos es imprescindible para poder mejorar el sistema desde el sistema mismo así como también para poder probarlo, entendiéndose por probarlo a usarlo y modificarlo mientras se lo está ejecutando. En lugar de eso, la forma de trabajar era lo que denominamos *offline*. Esto es, modificar los objetos en una imagen de Squeak corriendo bajo un sistema operativo, persistirla y luego hacer que esa imagen sea la que corra SqueakNOS. Esta manera de trabajar no es la deseada bajo la filosofía Smalltalk y tiene muchos inconvenientes, siendo los más importantes la pérdida de dinamismo -uno de los factores más fuertes de Smalltalk-, el mayor tiempo de desarrollo, errores por tener que hacer el trabajo bajo otras plataformas y la pérdida de cambios hechos si hay algún error crítico.

Por otro lado, además de no poder persistir los objetos, tampoco era posible persistir los cambios o leer los fuentes de los métodos. El ambiente *Squeak* guarda la imagen en un archivo *serializado*, pero utiliza también dos archivos importantes. En el primero guarda todos los fuentes de los métodos y en el segundo guarda los fuentes de los cambios realizados a esos métodos y los métodos o clases añadidas. Al SqueakNOS no soportar interactuar con el disco y no soportar la interacción con archivos, esta tarea era imposible. Por lo tanto, sumado al problema de la persistencia de los objetos, existía el obstáculo de que mientras se exploraba código, sólo se tenía acceso al mismo decompilado, el cual no muestra los nombres de las variables temporales, los comentarios, ni el formato del código. Tampoco existía la posibilidad de acceder a los cambios que se iban generando en el ambiente para poder recuperarlos en caso de fallas. Esto, sumado a la restricción de no poder persistir los objetos generaba que la mayor parte del tiempo el trabajo sea *offline* ya que era muy complejo hacerlo de otra manera con estas limitaciones.

Luego, para poder lograr que el trabajo sobre SqueakNOS pueda ser realizado *online*, resulta necesario permitir la persistencia no sólo de los objetos sino también de los fuentes en los respectivos archivos. El mayor desafío en este caso es poder lograr un desempeño razonable en el manejo básico de archivos que hace *Squeak* y lograr cambiar lo mínimo posible los objetos de alto nivel que Squeak ya provee para manejar los archivos.

Pero para comprender por qué la persistencia de los objetos no estaba ya implementada dos aspectos de SqueakNOS deben ser analizados en detalle: su filosofía y el mecanismo de *snapshotting*.

Con respecto a SqueakNOS mismo, como se explicó en la sección 2.4, este implementa conceptos de los sistemas operativos directamente en Smalltalk, en un modelo puro de objetos. Interactúa con el hardware directamente, brindando interfaces a los objetos de cualquier aplicación para interactuar con el mismo. A diferencia de los sistemas operativos clásicos, estas interfaces están implementadas en lenguajes dinámicos por lo que ellas mismas presentan un dinamismo imposible de proveer en otras implementaciones. Resumiendo, la filosofía de SqueakNOS es apoyarse en modelos de alto nivel en lugar de implementar sus propias interfaces de bajo nivel. Esto se traduce a que en SqueakNOS se modele la mayor parte del dominio en la imagen misma. De aquí se deduce entonces que la mejor solución posible al problema de snapshotting es desarrollar un sistema de archivos y un controlador de disco totalmente en Smalltalk. Como veremos más adelante, esta dependencia en la ejecución de código Smalltalk para grabar archivos afectará al mecanismo de snapshotting, ya que este último requiere el detenimiento de la unidad de ejecución de bytecodes.

Con respecto a la persistencia de objetos, el código para salvar la imagen está en el método primitiveSnapshot, el cual como su nombre indica es una primitiva. En *Squeak*, las primitivas son traducidas a lenguaje *C* y luego compiladas dentro del motor de objetos. Durante su ejecución, se podría afirmar que la imagen está parada. Más precisamente, el motor de objetos no está ejecutando *bytecodes*. La primitiva puede separarse claramente en dos partes. La primera deja el *object heap* consistente para ser serializado (recolecta la basura, compacta, finaliza objetos externos, etc.). La segunda corresponde a grabar este bloque de memoria en un archivo, añadiéndole algunos datos de formato. Esta implementación del guardado de la imagen aprovecha muy fuertemente que durante la ejecución de la primitiva, cuando llama a grabar en disco el bloque de memoria serializado, el motor de ejecución no está ejecutando bytecodes sino que está pausado. Si esto no fuera cierto, la primitiva podría grabar bloques de memoria que estén siendo modificados al mismo tiempo que son grabados en disco y esto podría devenir en una imagen inconsistente.

En Squeak, se puede hacer esta asunción debido a que la escritura de archivos es realizada mediante una llamada al sistema operativo, con lo cual no hay objetos de Smalltalk interactuando con este proceso. Pero en SqueakNOS, escribir o leer datos de un archivo deja de ser una primitiva. En lugar de eso hay objetos enviándose mensajes para buscar los archivos en el sistema de archivos y los clusters que éstos ocupan, para encontrar en que bloques de disco esos archivos están y para modificar los datos de los bloques correspondiente. Todo esto rompe el requisito de que el motor de ejecución esté pausado en el proceso de escribir, por lo tanto otro camino es necesario para resolver el problema. Lograr que el grabado de la imagen se haga con código Smalltalk solamente fue considerado, pero dejado de lado al instante mismo ya que su complejidad sería muy grande y no hay nada experimentado en ese camino. Esto hubiera requerido que la primitiva de grabado tuviera noción de cuales objetos son los que cambian durante la escritura, de manera que pueda lograr que ellos sean grabados antes de ser modificados. Una solución más simple sería engañar al proceso de salvado de alguna forma para que el resultado sea similar al de hacerlo con el motor de ejecución pausado.

Esta descripción del problema de grabado de la imagen (*snapshotting*) podría entenderse como un problema de *atomicidad*. Los problemas de atomicidad son muy comunes en el área de las bases de datos o en el área de sistemas (por ejemplo en la sincronización de procesos mediante semáforos o estructuras similares). En los casos de las bases de datos, es importante que una transacción se ejecute completamente o que no se ejecute en absoluto, para no dejar la base de un modo inconsistente. En cuanto a la sincronización de procesos, es sustancial que cuando se hacen algunas operaciones sobre estos (como leer o escribir sus posiciones de memoria), no se ejecute nada en el medio que pueda hacer que dos procesos entren juntos a la sección crítica. Al problema general que enmarca este tipo de requisitos se lo conoce como "thread safety". Respecto a este trabajo, sería necesario que para persistir la imagen, el proceso sea atómico desde el punto de vista de la imagen, o sea que no se ejecuten bytecodes en el transcurso del mismo. Las soluciones que logran esto rompen con las ideas centrales del proyecto sobre desarrollar lo máximo posible en Smalltalk ya que por definición del problema no deberían usar los modelos de Smalltalk para grabar los archivos. Por lo tanto, el objetivo fue lograr simular la atomicidad, o sea, aunque no sea atómica la operación, esto

Figura 4.1: Estado del *object heap* en el momento que debe ser escrito atómicamente a disco.

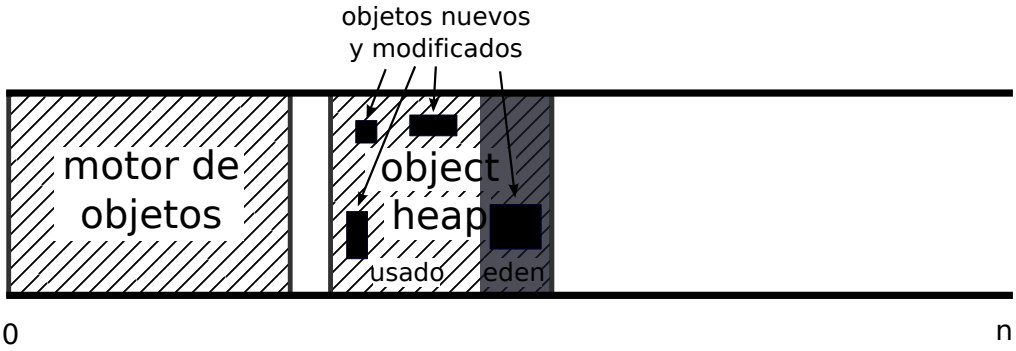


Figura 4.2: *Object heap* modificado por la ejecución del interprete.

sería suficiente para dejar en el archivo una imagen consistente.

Por lo tanto, lograr la persistencia en SqueakNOS es importante y alguna porción de la misma puede ser implementada fácilmente y otra conlleva un problema interesante para resolver donde no hay estudios al respecto. Lograr la persistencia en el ambiente descripto implica resolver varios problemas interesantes que no cuentan aún con demasiada exploración, como el problema del rendimiento de manejadores de dispositivos en alto nivel y la simulación de la atomicidad en caso de que esta sea imposible de lograr directamente. Los problemas son de gran interés técnico y es imprescindible su resolución para el futuro del proyecto SqueakNOS. Por otro lado es interesante comprobar si alguno de los mecanismos de hardware modelados, como ser el de paginación, puede utilizarse para probar soluciones novedosas a distintos problemas, valiéndose del hecho de que estos mecanismos están modelados en alto nivel. De no ser así, por lo menos verificar si ayuda a recorrer otros caminos impensados no teniendo modelado los mismos y tan accesibles a los diversos objetos del ambiente.

Volviendo a la pregunta original de esta sección, la simulación de la atomicidad para el grabado de la imagen es central en esta tesis. La idea no es sólo lograr esa atomicidad, sino hacerlo sin quebrar la idea de trabajo en SqueakNOS que requiere desarrollar lo máximo posible del lado de modelos de alto nivel dejando al mínimo indispensable el desarrollo en bajo nivel. ¿Cómo lograrlo? ¿Dónde situar ese límite?. Hay diferentes soluciones con distintos grados de compromiso entre el simulado de la atomicidad y el valor donde se ubicarían respecto al límite. Al no tener respuesta

aún la pregunta que se refiere a la factibilidad en Smalltalk, esta es un área no explorada donde los primeros resultados serán los presentados por este trabajo.

Capítulo 5

Solución

Para intentar resolver los problemas enunciados en el capítulo 4 se siguió un camino incremental. Las próximas secciones siguen el orden cronológico que tuvieron durante el trabajo. En primer lugar se presenta una implementación escrita completamente en Smalltalk de *FAT32* y de un controlador *ATA*. Mediante estas implementaciones, se logra tener acceso para lectura y escritura a dispositivos de almacenamiento cuyo hardware implemente la interfaz *ATA*. A partir de estas implementaciones se puede trabajar en SqueakNOS con los fuentes de los métodos, requerimiento acuciante para poder realmente tener una plataforma usable. Luego se exhibe una solución de tipo *naive* para la persistencia de la imagen, la cual no cumple con todas las propiedades deseadas en cuanto al orden espacial y al nivel de abstracción donde se sitúa. Sin embargo, la misma es una solución completa que sirve tanto para responder parte de los interrogantes como para ser fuente de comparación con otras soluciones más avanzadas. Luego se presenta la solución más completa, basada en el aprovechamiento de la *paginación*, con la cual se llega al objetivo deseado. Esta segunda solución recorrió un camino bastante complejo hasta hacerse realidad y en las secciones 6.3 y 6.4 se muestran todos los obstáculos que fueron sorteados. Las diferentes dificultades devinieron en diversas herramientas analizadas y/o desarrolladas para superarlas y estas serán descritas en este capítulo también.

5.1. Persistencia en dispositivos de almacenamiento

La solución clásica para atacar el problema de la persistencia por los sistemas operativos está basada casi exclusivamente en una única metáfora: el concepto de archivo. Estos están pensados y fueron creados principalmente como un medio donde almacenar datos. Además, los sistemas operativos manejan estructuras de control sobre los discos rígidos las cuales son llamadas *sistemas de archivos*. Estos logran abstraer la organización de los datos en los medios físicos, presentándole al usuario un esquema simple de archivos y carpetas. Sin embargo, con la aparición de los lenguajes orientados a objetos y su principal concepto, el de objeto, aparece la necesidad de persistir objetos en lugar de datos. La solución que se usa en general para resolver esta divergencia esencial entre los objetos y los archivos, dado que las herramientas que los sistemas operativos ofrecen son los

archivos, es la de serializar manualmente los objetos para luego guardar esta versión serial en un archivo. Diversos autores están convencidos de que en un ambiente puro de objetos la unidad de persistencia debería ser directamente el objeto, sin la necesidad de un previo proceso engorroso para obtener una versión serial de los mismos [Kay]. El modelo a seguir debería ser el usado en las bases de datos de objetos [Vis], y particularmente *Gemstone* [Gem] es la solución más completa conocida para los desarrollos en Smalltalk.

Sin embargo hay varios motivos importantes para fundamentar por qué se decidió implementar un sistema de archivos en SqueakNOS aunque el archivo sea un concepto antagónico al de objeto. En primer lugar SqueakNOS está pensado para usarse en lugar de un sistema operativo clásico, por lo cual es de vital importancia para el proyecto poder comunicarse con otros sistemas y ambientes. Sería necio no aceptar que en la actualidad la mayoría del software soporta y se comunica mediante el concepto de archivo. La comunicación con dispositivos como cámaras, memorias SD, USB y otros requieren de la comprensión e interacción con los *sistemas de archivos* sobre los cuales están generalmente montados. Por otro lado, SqueakNOS está basado en la máquina virtual de Squeak. Esta hace un importante uso del concepto de archivo, siempre basándose en las bibliotecas del sistema operativo correspondiente para interactuar con los mismos. El ejemplo más claro, es que la imagen misma, más los cambios que se generan en el ambiente, más los fuentes de los métodos compilados son persistidos en archivos. Como la idea para el trabajo de tesis es cambiar lo menos posible la arquitectura del proyecto, ya que este es un proyecto de gran envergadura y cambiar sus bases implica importantes cambios, fue obligatorio implementar el soporte para archivos para lograr que el ambiente posea las mismas características que tiene al ser usado mediante las interacciones con la *ABI* de un sistema operativo clásico.

Pero en SqueakNOS, el acceso a archivos tiene que ser manejado completamente con código Smalltalk, ya que lo que suele ser la *ABI* de un sistema operativo clásico, en SqueakNOS está desarrollado en la imagen. Implementar el manejo de archivos con código de alto nivel exclusivamente es menos complicado que hacerlo en código de bajo nivel, pero no por esa razón deja de ser trabajoso. Son varias las etapas del acceso que deben ser desarrolladas y muchos los factores a tener en cuenta hasta lograr la interacción a alto nivel con un archivo. En primer lugar está la necesidad de comunicación con un dispositivo de almacenamiento para transferir datos crudos desde o hacia posiciones lógicas del mismo. En segundo lugar, los sistemas operativos organizan los dispositivos en particiones por lo que hay que leer los sectores correspondientes e interpretar esos datos para lograr acceder a la particiones requeridas. Luego se debe acceder a los archivos en las particiones correspondientes mediante la localización de los mismos en el sistema de archivos bajo el cual están persistidos. Y finalmente se debe poseer un modelo para mantener los archivos abiertos para ser usados en los momentos requeridos por el usuario. Todos estos problemas fueron resueltos a alto nivel, con código Smalltalk y modelos de objetos los cuales pueden ser dinámicamente modificados y depurados al igual que cualquier mensaje y objeto del ambiente.

Por lo tanto, para dar el primer paso respecto a la persistencia se implementaron, completamente en Smalltalk mediante un modelo de objetos, un manejador de dispositivos *ATA* y un sistema de archivos *FAT32*. Una vez implementados, se adaptó el modelo de *FileStreams* de Squeak para que

usen estos objetos como intermediarios para recibir o dar los datos correspondientes. Esto hizo que se cumpla uno de los milestones más importantes que adeudaba SqueakNOS, poder ver los fuentes de los métodos compilados dentro del ambiente así como también poder persistir los cambios que se van realizando en la imagen mientras se trabaja sobre la misma. En las siguientes secciones se detalla la implementación de cada una de las partes.

5.1.1. ATA

El modelo de programación ATA es un modelo relativamente sencillo, aunque no cuenta con más documentación que la de referencia, en sus versiones de la 1 a la 7, y los múltiples ejemplos de código que pueden encontrarse disponibles. Debido a que el modelo de ATA es incremental, se decidió implementar inicialmente la versión original del protocolo, lo cual finalmente resultó más que suficiente para alcanzar los objetivos planteados.

Como se explicó anteriormente en la sección 2.5, en este modelo cada dispositivo se ve como un arreglo de sectores de tamaño fijo que se accede de manera aleatoria, pudiéndose acceder de a ráfagas para aumentar el desempeño. Las PCs comunes cuentan en general con dos controladores físicos ATA, cada uno de los cuales puede tener hasta dos dispositivos de almacenamiento, uno maestro y uno esclavo, alcanzando un límite de hasta cuatro dispositivos de almacenamiento simultáneos. Cada uno de estos controladores se accede mediante 8 puertos de entrada/salida ubicados en direcciones fijadas por el estándar. A su vez, cada uno de los puertos cumple una tarea específica, como brindar información de estado, o establecer la posición a leer o escribir en el próximo acceso. Desde el punto de vista de la imagen a los controladores se los modela con la clase ATAController, que subclasifica a PCIDevice. El comportamiento heredado le brinda, entre otras cosas, facilidades de detección de los puertos de entrada/salida y acceso a los mismos. La implementación realizada utiliza solamente polling, lo cual si bien es una solución más ineficiente y lenta que vía DMA, es de desarrollo mas simple. Además, en ningún momento se presentaron problemas de desempeño que hicieran necesario cambiar de técnica. De todas maneras, la jerarquía de PCIDevice brinda también acceso a los registros de configuración del puerto PCI, que servirá para implementar en un futuro acceso por vía DMA, mediante el uso de interrupciones.

Cada instancia de ATAController se encarga de brindar a alto nivel un servicio de lectura y escritura de sectores, el cual realiza mediante la manipulación de los registros del controlador ATA que representa, según exige el estándar. Debido a que controla tanto al dispositivo maestro como al esclavo, no es accedido directamente, sino que de él se suelen obtener instancias de ATADevice, mediante el envío de los mensajes master o slave según se desee. De esta manera, se obtiene un modelo uno a uno con el hardware subyacente, una instancia por cada chip controlador, y otra por cada dispositivo físico de almacenamiento.

La clase ATADevice subclasifica de BlockDevice, la cual brinda una interfaz que realiza la traducción entre direccionamiento a sectores -utilizado por el dispositivo-, y direccionamiento a bytes -utilizado por los clientes de nivel más alto-.

Esta separación permite además aplicaciones interesantes mediante el uso de objetos polimórfi-

cos a los dispositivos ATA, pero de otra naturaleza. Por ejemplo, si uno posee en un archivo una imagen de una partición FAT, podría testear la implementación de FAT, explicada en la próxima sección, pasándole a ésta un objeto que se comporte como un BlockDevice, pero que subyacentemente utilice un stream del archivo de imagen.

Por último, las instancias de BlockDevice poseen un caché que intercede en todas las lecturas y escrituras, y es capaz de mejorar el desempeño. En la implementación actual se implementa un caché simple que solamente guarda en memoria el contenido del último sector leído, y que sirve para evitar que lecturas sucesivas de bytes de un mismo sector desemboquen en múltiples accesos al bus y al dispositivo.

5.1.2. FAT32

Una vez que la lectura a disco está lista, el siguiente paso es detectar las particiones que contiene el dispositivo, las cuales están descritas en el primer sector del mismo. Una vez detectadas las particiones, se elige con cual se va a trabajar y todo el resto del trabajo queda del lado del sistema de archivos bajo el cual está organizada la partición.

En este trabajo se decidió implementar como sistema de archivos a *FAT32* por varios motivos. Principalmente por su simplicidad, la cual hace que sea el menos costoso de implementar. Pero además, sigue siendo usado y soportado por la gran mayoría de los sistemas actualmente y es de uso masivo en los dispositivos de almacenamiento portables del estilo *pendrive*. Sin embargo, se pone un importante énfasis en lograr una implementación genérica para que sea lo más simple posible la extensión de los modelos y la adaptación a otros nuevos. Esto es debido a que la idea es que el ambiente soporte la mayor cantidad posible de sistemas de archivos.

El modelo presentado se divide en dos importantes submodelos. El primero de ellos se aboca esencialmente a los entes exclusivos para el control y la administración de *FAT32*. El segundo de ellos, en cambio, se relaciona con los componentes compartidos por los diversos sistemas de archivos.

5.1.3. Conceptos particulares

Los primeros 512 bytes de *FAT32* están dedicados a la información del sistema de archivos. El concepto que modela esta información es FAT32BootRecord. Este objeto responde sobre los datos necesarios para la inicialización del sistema de archivos como por ejemplo la cantidad de sectores de la partición, el cluster donde comienza el directorio raíz y la cantidad de sectores que ocupa cada cluster. El objeto abstrae la dificultad de encontrar en que lugar o lugares del conjunto de bytes se encuentra la información requerida y de decodificar los valores que se leen de esos bytes a partir de la especificación formal del contenido de esos datos.

Una vez que se accede a los datos de la partición se puede descifrar en que posición física del disco se encuentra la *FAT (File Allocation Table)* de la partición. Este es otro concepto importante del modelo y fue nombrado como FAT32FileAllocationTable. Su responsabilidad es la de manejar la tabla de alocación, el elemento esencial de este *sistema de archivos*. Este objeto le presenta una interfaz clara a cualquier objeto que deba usar la tabla. Por ejemplo, este objeto es el responsable de

responder cuáles son todos los clusters que poseen datos para cierto numero inicial de cluster. Como la tabla es usada muy frecuentemente, ya que es accedida cada vez que se modifica o accede un archivo, la idea es tener una copia de la misma en memoria para mejorar los tiempos de respuesta. En la misma dirección de intenciones, también se agrego una caché donde se guarda la cadena de clusters de los archivos ya accedidos, para lograr que los siguientes accesos impliquen costos temporales mucho menores.

Sabiendo cuales son todos los clusters de un archivo, es necesario acceder a cada uno de ellos para poder juntar los datos de los mismos y formar el contenido del archivo. Esto lo modela la clase FAT32Cluster. En principio, en el caso que el cluster pertenezca a un archivo, este concepto no es más que una lista ordenada de bytes. En este caso no parece muy interesante modelar un concepto especial para este ente. Sin embargo, la cuestión cambia en el caso de que el cluster pertenezca a un directorio. En este caso, el cluster tiene la responsabilidad de responder sobre el contenido del mismo. Por ejemplo, si cierto archivo está referenciado por este cluster. También es la responsabilidad del objeto la de administrar el espacio en el cluster, agregar y borrar registros.

En un nivel más profundo de granularidad están modelados los registros de los clusters que son pertenecientes a directorios. FAT32ClusterRecord es la denominación en el modelo de este concepto el cual abstrae la complejidad de acceder y decodificar los campos del registro para acceder a los datos importantes como el nombre referenciado, si es un archivo o un directorio, atributos de seguridad, tamaño, etc. Su responsabilidad es muy similar a la de FAT32BootRecord y al igual que ese objeto, también está basado fundamentalmente en la información que provee la documentación de *FAT32*.

El último objeto es la interfaz mediante la cual los objetos del ambiente pueden interactuar con el *sistema de archivos*. A este objeto se lo clasificó con el nombre de FAT32FileSystem. Este es el único objeto que interactúa con los objetos de bajo nivel que están totalmente acoplados a el sistema de archivos particular. Ofrece los servicios comunes a cualquier sistema de archivos como la creación de archivos y directorios, la búsqueda y el acceso a los mismos.

5.1.4. Problema de acceso a los datos

Al trabajar a alto nivel y obtener una reificación completa del modelo, aparece un inconveniente a resolver. Cada una de las clases que han sido descritas en la sección previa, posee un colaborador interno con la porción de bytes correspondiente a su función. El objeto que reifica la FAT posee un colaborador con los bytes correspondientes desde el inicio al fin de la misma. A su vez, el objeto que representa el sistema de archivos mismo, posee los bytes que van desde el primer sector en disco hasta el último que este ocupa. Sin lugar a dudas, los bytes que representan a la FAT están incluidos dentro de los que representan al sistema de archivos. Como no es viable ni deseable tener todo el sistema de archivos en memoria o todo el contenido de cada uno de estos conceptos en memoria (salvo el caso de la FAT por su costo-beneficio entre tamaño y cantidad de accesos) lo que se decidió usar para acceder a los datos son *Streams*.

El mayor problema que surge con este modelo es fundamentalmente uno de sincronización. No

es aceptable para el correcto funcionamiento del sistema que el objeto que representa el sistema de archivos tenga en su porción de bytes correspondientes a la FAT datos diferentes a los que tiene el objeto que representa a ésta. Este problema es extrapolable también a otros objetos del modelo, ya que existe el mismo problema solamente cambiando el objeto que representa a la FAT por el que representa a un cluster. Sin embargo, si cada objeto posee un stream diferente sobre la porción de disco correspondiente a su información podrían quedar desincronizados. Esto es debido a que los streams sobre archivos, tienen un buffer interno donde van guardando los datos que van leyendo, ya que el costo de leer a disco es muy alto. Por lo tanto esta opción no es viable. Trabajar sobre el mismo stream tampoco lo es, ya que si un objeto modifica la posición interna del stream esto afectaría a todos los demás objetos.

A pesar de los problemas mencionados para este contexto, los streams tienen propiedades muy interesantes que se querían conservar. Estos abstraen el acceso a archivos, disco, etc. y presentan al usuario simplemente un flujo de bytes lo cual los hace realmente importantes en cualquier entorno de alto nivel. Por lo tanto, se buscó una solución que mantenga las propiedades de los streams, y que además asegure la sincronización entre ellos cuando distintos objetos quieren acceder a la misma porción de datos. Para este fin se implementó el concepto de *StreamOnStream*.

Este concepto representa esencialmente la idea de tener un flujo de datos sobre un flujo de datos previo. Entonces mediante esta solución, el objeto que representa a la FAT puede tener un stream sobre el objeto que representa al sistema de archivos, y ambos pueden interactuar con sus datos independientemente sin modificar el estado del otro aunque finalmente estén interactuando con el mismo flujo. Esencialmente, tienen como colaborador interno al flujo original y delegan en él la responsabilidad de conseguir o guardar los datos. El objeto que usa a estos *StreamOnStream*, los ve como un flujo normal, quedando completamente encapsulado en su implementación que delegan sus responsabilidades en otro flujo.

Para no modificar el estado del flujo original, no sólo se quedan como colaborador interno con el mismo, sino que generan una copia del mismo cuando son inicializados. La delegación de las responsabilidad sobre los datos se hacen sobre la copia justamente para no modificar el estado del original. En casos de streams especiales, como es el caso de los archivos, donde se guardan datos cacheados, la modificación de estos datos debería sincronizarse con el stream original y es por esto que el *StreamOnStream* tiene como colaborador interno al flujo original.

5.1.5. Conceptos generales

El núcleo de los conceptos generales modelados para los *sistemas de archivos* es el concepto de FilesystemContent, el cual está modelado en su respectiva clase y provee el protocolo abstracto para las operaciones comunes de creación, borrado y renombrado. Las dos subclases concretas de este concepto son File y Directory las cuales poseen la implementación particular de estas operaciones para un archivo y para un directorio respectivamente. Además File es un enlace para abrir y cerrar flujos de datos sobre archivos.

Las instancias de FilesystemContent consisten básicamente de un identificador dentro de un

FAT32Filesystem al cual también referencian. Entonces, al recibir una petición (un mensaje), estos objetos no lo resuelven ellos. Eso haría que la implementación de estos objetos esté fuertemente acoplada a una implementación de sistema de archivo particular. Sin embargo, en la implementación presentada estos objetos delegan la tarea en el sistema de archivos al cual referencian (una instancia de FAT32Filesystem en este caso) pasándole como argumento el identificador correspondiente. Específicamente, en FAT32 el identificador es el primer cluster en la cadena de clusters de la FAT que referencia a si mismo. En otro tipo de sistema de archivos, como por ejemplo EXT2, el identificador sería el número de *inodo*.

La última abstracción que enmarca el modelo de *sistema de archivos* son los streams. Una vez que se obtiene un objeto que es instancia de la clase File, lo ideal es poder leer y escribir del mismo de manera simple. Para esto, estas instancias responden a mensajes del estilo *readStream* o *readWriteStream*, los cuales responden con un objeto instancia de la clase MultiByteFilesystem-FileStream. Para evitar el mismo acoplamiento ya explicado respecto a la implementación en los archivos de las operaciones básicas, estos stream no realizan la tarea ellos mismos sino que la delegan también en el *sistema de archivos* correspondiente. Al ser subclase de FileStream, se asegura el polimorfismo con los streams usados por un ambiente standard, y por lo tanto acceder al archivo de fuentes es tan simple como conseguir el stream en el archivo donde estos se encuentran. La figura 5.1 muestra un ejemplo de colaboración entre los distintos objetos explicados para leer caracteres de un MultiByteFilesystemFileStream.

Presentados estos conceptos, se puede observar que la arquitectura sobre la cual se implementa FAT32, es lo suficientemente genérica y extensible como para ser usada por otras implementaciones de sistemas de archivos. Se eligió implementar FAT32 en este trabajo fundamentalmente debido a la simplicidad de sus conceptos y a la baja cantidad de los mismos. Pero además de esto, FAT es un aceptable sistema de archivos para las particiones no demasiado grandes ya que posee una sobrecarga bastante baja en estos casos. Con 200mb de disco se tiene una partición lo suficientemente grande de SqueakNOS para realizar las tareas de este trabajo y este requerimiento de tamaño es compatible con los casos en los que FAT presenta un rendimiento no sólo aceptable sino que hasta mejor que otros sistemas de archivos más completos y complejos. Las grandes desventajas de este tipo de sistema de archivos son justamente en los casos en los que se necesita persistir muchos datos. FAT32, tiene un límite bastante bajo de tamaño de partición respecto a los dispositivos de almacenamiento que se observan en la actualidad. Por otra parte, a medida que aumenta el tamaño de la partición, FAT tiene la propiedad de que disminuye su rendimiento. Por último, los sistemas de archivos más modernos (como NTFS, EXTx y otros), poseen mecanismos de seguridad mucho más completos. Por estas desventajas, fue importante desarrollar una arquitectura extensible para dejar las bases sólidas previendo el caso de que SqueakNOS requiera, en su evolución, de las propiedades que presentan los demas sistemas.

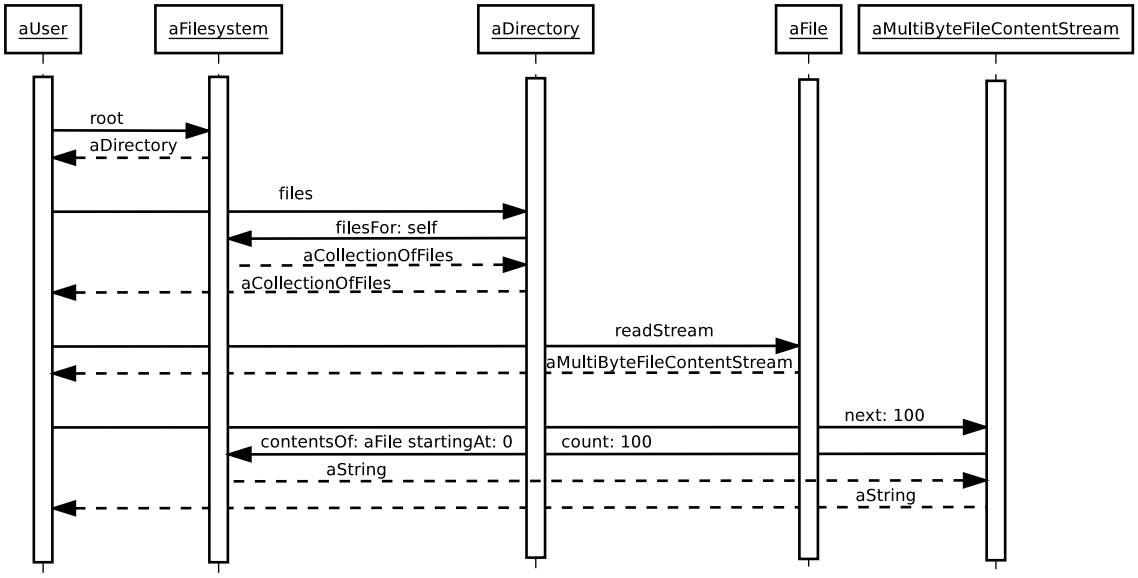


Figura 5.1: Leyendo 100 caracteres de un archivo en el directorio raíz de aFilesystem

5.2. Manejo del mecanismo de paginación

La puesta en funcionamiento de la paginación en SqueakNOS requirió dos trabajos por separado: por un lado la creación de un modelo que permita la administración de las estructuras que utiliza el procesador para manejar las páginas, y por otro la modificación del mecanismo de manejo de interrupciones para lograr que sea sincrónico.

En la primer parte del trabajo se reificaron objetos que representan al procesador, en este caso uno de arquitectura X86, así como también objetos que representan los directorios de páginas, las tablas de páginas, y las entradas de las mismas. Esto hizo posible administrar las tablas y directorios de páginas directamente desde Smalltalk. Pero el objetivo era que la totalidad de la lógica involucrada en la resolución de los fallos estuviera escrita en Smalltalk, así que se modelaron también otros elementos del dominio. Para manejar los fallos de página se utilizan objetos de la clase PageFaultHandler. Ellos permiten utilizar bloques de código ad hoc para resolver los fallos. El objeto que representa a la computadora tiene un PageManager, que se encarga entre otras cosas de llevar la cuenta de cuál handler debe utilizarse para cada rango de direcciones. Cada vez que un fallo de página llega, este objeto busca en un mapa al handler que fue establecido para la dirección que falló y lo ejecuta.

La segunda parte del trabajo relacionado con paginación fue sensiblemente más complicada, ya que involucraba la modificación del mecanismo de atención de interrupciones para hacerlo sincrónico. Lo que se deseaba era poder manejar los fallos de página desde Smalltalk, y para eso era necesario idear una nueva estrategia de manejo de interrupciones. La solución consistió esta vez en la utilización del mecanismo de *callbacks* disponible en Squeak. La rutina de atención de interrupciones por fallo de página resultante es muy similar a las demás rutinas de atención de interrupciones explicadas en la sección 2.7.1, en lo que hace al guardado y restauración del contexto

de ejecución, pero difiere en el hecho de que el manejo de la interrupción es, en este caso, sincrónico. En esta rutina, se libera un semáforo sobre el cual espera un proceso que va a manejar el fallo, y luego mediante el mecanismo de callbacks se reentra a la unidad de ejecución directamente. De la misma manera que con las interrupciones asincrónicas, al reentrar el motor descubre que hay un proceso de muy alta prioridad listo para ser ejecutado, y lo pone a trabajar. Una vez que este proceso termina, el mismo envía un mensaje que permite la salida del callback, volviendo así de la interrupción original.

El mecanismo implementado sólo requiere el agregado de unas pocas líneas de C, ya que incluso la creación del callback se hace desde smalltalk, y sólo debe pasarse al motor de objetos la dirección del mismo.

5.3. Persistencia de la imagen

La persistencia de los objetos de la imagen puede ser lograda de diferentes maneras. Una de ellas es por ejemplo la descarga de subconjuntos del grafo de objetos a disco. Una solución más sencilla sin embargo es la descarga del contenido entero de la imagen a un sólo archivo, técnica conocida como *snapshotting*. A continuación nos referimos a las dos técnicas implementadas para guardar snapshots de la imagen en disco.

Para comprender los caminos seguidos para resolver el problema de la *atomicidad* al grabar la imagen, es importante comprender qué ocurre cuando el motor de objetos intenta persistir la misma en el ambiente *Squeak*. El proceso se llama *snapshotting* ya que uno podría entenderlo como que se está tomando una foto instantánea del sistema en ese momento. Como ya fue introducido en la sección 4.3, la función *primitiveSnapshot* (el componente esencial para la persistencia de la imagen) prepara el *object heap* para ser serializado y luego ejecuta una llamada (*sqMemoryFileWrite*) al sistema la cual se encarga de escribir el contenido serializado en el archivo correspondiente.

Más específicamente, el objetivo principal de *primitiveSnapshot* consiste en asegurar la consistencia de la memoria a persistir. En primer lugar se asegura que el proceso en ejecución en el motor de objetos esté en la *rootTable* y que éste tenga un enlace (en el colaborador interno donde referencia al contexto de ejecución donde quedo suspendido) al contexto actual de ejecución para que al reiniciar el sistema este se ejecute desde el mismo lugar en el que fue pausado. Luego se hace una recolección de basura incremental seguida de una completa. Esto deja la imagen como un bloque de memoria contigua donde todos los objetos tienen alguna referencia hacia ellos. Finalmente, se escribe ese bloque de memoria compacto, que resulta de haber pasado por todos los procesos mencionados, escribiendo antes en el archivo un encabezado donde entre otras cosas se especifica el tamaño de la imagen.

Al no haber un sistema operativo en el cual el motor de objetos pueda delegar la responsabilidad de escribir el archivo mediante la llamada al sistema que se ejecuta en el ambiente *Squeak* tradicional, tampoco existe una función nativa implementada para tal requerimiento. Por esa razón, en la versión de SqueakNOS previa a el desarrollo de este trabajo, la función *sqMemoryFileWrite* (la que finalmente llama al sistema para la escritura de los bytes correspondientes) está implementada

como una función vacía, simplemente para mantener la compatibilidad con el motor de objetos clásico. Esto significa, ni más ni menos, que siempre que el usuario intente guardar un *snapshot* de la imagen, el motor de objetos va a preparar todo lo necesario para la serialización del *object heap*, pero sin embargo, nada va a grabarse en ningún archivo y ningún mensaje será observado por el usuario al respecto.

Este era el comportamiento observado debido a que en SqueakNOS, por su implementación, su filosofía y sus objetivos, la única manera de escribir o leer de algún archivo es mediante el modelo de objetos. Además, en la versión referenciada, el modelo de objetos no ofrecía servicios para interactuar con archivos ya que no existía una implementación de un modelo que pueda interactuar con dispositivos de almacenamiento ni tampoco con los sistemas de archivos en los cuales se organizan los mismos. Sin embargo, ya con el desarrollo de estos modelos, los cuales fueron presentados en las secciones previas de este capítulo, escribir en archivos implica una comunicación mediante mensajes entre los diversos objetos que modelan los conceptos correspondientes. Estos mensajes que se computan durante la ejecución de la primitiva de *snapshotting* rompen con la atomicidad requerida para esta tarea.

5.3.1. Copia de memoria (solución *naive*)

Como primera solución, en lugar de enviar mensajes o de hacer lo mismo que se hace en otras plataformas (delegar la escritura del archivo en un sistema operativo), se decidió implementar la función *sqMemoryFileWrite* como una simple copia de memoria, lo que se conoce en *lenguaje C* como un *memcpy()*. Esta función ya estaba implementada en SqueakNOS en código de bajo nivel por lo cual esta copia de memoria se hace atómicamente respecto al motor de objetos. Con esta implementación, la primitiva de *snapshotting* hace una copia atómica de la memoria en un lugar vacío de la misma y luego el motor de objetos escribe esa porción de memoria en el archivo correspondiente usando el modelo de archivos de SqueakNOS. En la figura 5.2 se puede ver como queda el estado de la memoria luego de haber ejecutado la primitiva.

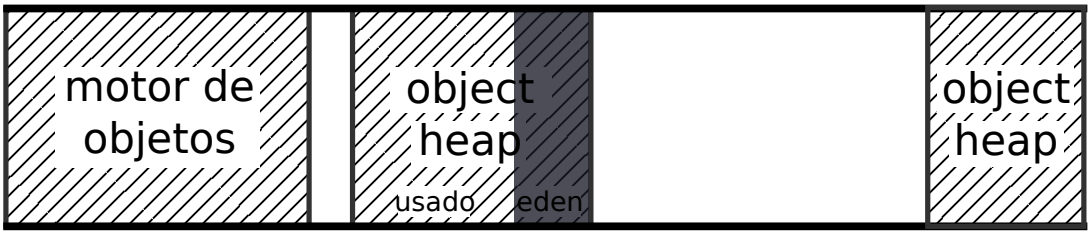


Figura 5.2: Mapa de la memoria cuando hace la copia del object heap.

Es necesario aclarar que en la explicación quedó implícito que existe un lugar de la memoria con el tamaño adecuado y que este está libre para alojar la copia del *object heap*. Por supuesto, este espacio no se encuentra de forma aleatoria, sino que se implementó un modelo de la memoria. En este modelo se reifica el concepto de la misma con lo cual en cualquier momento se le puede pedir al objeto que representa la memoria principal la alocación de cierta cantidad de memoria.

Este objeto es el encargado de administrar los espacios libres y ocupados de la memoria. Es creado cuando se inicializa el objeto que representa la computadora misma y la información necesaria para su correcta inicialización es obtenida de la información que deja el *cargador de arranque* sobre el uso de la memoria. Este modelo es esencialmente usado para encontrar un bloque de memoria libre lo suficientemente grande como para alojar al *object heap* antes de ejecutarse la primitiva de *snapshotting*.

Esta forma de encarar el problema tiene la ventaja de su simplicidad y su implementación clara y directa. Pero a la vez, posee una gran desventaja que la hace casi inaceptable para uso frecuente: requiere duplicar todos los objetos del ambiente para poder persistirlos. Para ponerlo en otras palabras, no se puede trabajar en el ambiente con imágenes mayores a la mitad de la memoria disponible en el sistema ya que para poder persistirse, el ambiente necesita duplicar su espacio de memoria. Esto podría ser aceptable quizás, en una imagen de unos pocos megabytes, pero sin lugar a dudas no es una solución escalable. En la próxima sección se presenta la solución que ataca el problema del exceso de consumo de memoria, la cual está basada en la técnica de *copy on write* mediante la paginación.

5.3.2. Copia lazy de memoria mediante paginado

Una de las grandes ventajas de trabajar en Smalltalk para lidiar con componentes de hardware básicos es la posibilidad de realizar prototipos rápidamente, y la de probar el código de manera dinámica. De esta manera, es mucho más sencillo buscar aplicaciones novedosas del hardware con el que cuenta la computadora. La solución aquí presentada es un caso de ejemplo.

El problema de la solución anterior es que es muy ineficiente en relación a la cantidad de memoria usada. Para mantener la atomicidad mientras se realiza el snapshot sería más eficiente -en relación al uso de memoria- llevar la cuenta de cuáles son los objetos que van siendo modificados, e ir guardando sus estados originales antes de que sean cambiados. Una posible manera de hacerlo sería modificar el motor de objetos para que se encargue de ello cada vez que se escribe una variable de instancia. Otra posibilidad, la descrita a continuación, es llevar la cuenta de las páginas de memoria que se van modificando, guardando su estado original antes de que el cambio ocurra.

El mecanismo utilizado para realizar el guardado de la imagen en forma más eficiente es muy similar al anterior. En este caso, se modifica la primitiva de snapshot para que marque las páginas de la memoria de objetos como de sólo lectura, y se coloca una rutina de atención de interrupciones por fallo de protección, que lo único que hace es copiar la página a un buffer especial antes de que sea escrita para luego marcarla como de lectura y escritura, saliendo de la interrupción. De esta manera, para cualquier página sobre la cual se haya escrito, se deja guardado en el buffer su valor original. Luego, el mecanismo de guardado de la imagen, que continúa su ejecución una vez que se vuelve de la primitiva, debe ir escribiendo a disco página por página, tomando el contendido de la memoria, a menos que ésta se haya modificado en cuyo caso se obtiene el contenido original del buffer. Una vez que la escritura en disco termina, se marcan todas las páginas como escritura y lectura otra vez, y se limpia el contenido del buffer que ya no es necesario.

Si bien el método es en la teoría sencillo, no fue así llevarlo a la práctica. Para entender el porqué es necesario entender que pasa en los momentos iniciales. Una vez que la primitiva de guardado termina, toda la memoria de objetos está marcada como sólo lectura. En ese momento la unidad de ejecución debe buscar un proceso listo para ejecutar y activar su contexto de ejecución. Ese contexto y ese proceso son a su vez objetos, y al ser activados son modificados. Esto causa que instantáneamente se genere un fallo de página, lo cual no sería un problema si no fuera porque el fallo de página a su vez intentará reentrar en la unidad de ejecución, activando otra vez otro proceso y contexto, continuando infinitamente. Por esta razón fue necesario guardar ciertas páginas especiales, en las que se almacenan objetos que involucran la activación de procesos, dejándolas luego marcadas como de lectura y escritura. De todas maneras, no es posible garantizar que la ejecución del código que maneja los fallos de página, estando toda la imagen marcada como de sólo lectura, no generará a su vez más fallos de página, terminando en una recursión infinita. Debido a esto, se implementó un page fault handler nativo escrito en C para estos casos. Este handler sólo entra en acción en el momento en que se entra a un page fault recursivo, es decir, cuando la rutina de atención de fallos de página misma genere otro fallo de página.

Capítulo 6

Problemas durante la implementación

El núcleo de este trabajo consiste en lograr superar algunos desafíos técnicos mediante una prueba empírica implementada en Smalltalk. El proceso por el cual se lograron los objetivos se llevó a cabo mediante sucesivas iteraciones de ciclos del tipo problema → investigación → implementación de solución. Gran parte de las iteraciones correspondieron a la resolución de problemas expuestos por la implementación del ciclo previo. En este capítulo se explican los problemas más desafiantes y más importantes conceptual y técnicamente. También los que consumieron mayor cantidad de recursos.

6.1. Ambiente de desarrollo

El primer objetivo dentro del trabajo realizado fue el posibilitar la lectura del código fuente de los métodos. Esto permitiría dejar de trabajar con código decompilado, el cual no contaba con comentarios ni nombres para las variables temporales. Ante este problema se entendió que sería necesario como primer medida implementar un driver para dispositivos de almacenamiento de tipo ATA, lo cual no resultó para nada sencillo. A la problemática de tener que trabajar con código decompilado (ya que el soporte para lectura de fuentes sólo sería posible después de implementar este controlador) se sumaba además el problema de no contar con ningún tipo de persistencia ni comunicación entre SqueakNOS y el mundo exterior. Esto implicaba que el código escrito mientras SqueakNOS corría debía ser reescrito luego fuera del mismo, en una imagen de Squeak estándar, para que pudiera ser guardado. Por otro lado, el controlador de teclado original era muy básico, por lo cual no resultaba posible realizar operaciones elementales como copiar y pegar texto. Por esta razón, resultaba más práctico escribir el código de manera offline, pero luego era necesario reiniciar SqueakNOS para probar los cambios. A esto se le sumaba la baja calidad de la documentación de ATA, que consistía básicamente en seis manuales de referencia, uno para cada versión, en los cuales a la interfaz de programación se le dedica muy poco espacio.

Una vez que el controlador estuvo funcional, se trabajó en posibilitar la lectura de los sources, pero sin que fuera necesaria la implementación de un sistema de archivos, ya que esto último era una etapa que llevaría bastante trabajo, y más aún en el estado en que el ambiente de desarrollo

se encontraba en ese momento. Para dar este paso, se creó un disco rígido virtual de VMWare, en el que se escribió en formato raw el contenido completo del archivo de sources. Luego, desde SqueakNOS se utilizó un stream que leía el contenido completo del disco rígido desde el sector 0 en adelante. Es decir, el contenido del disco rígido era en realidad el contenido de un único archivo, el de sources. De esta manera no era necesario contar con un filesystem que ubicara donde comenzaba el archivo de sources, y que buscara la lista de sectores en donde éste se encontraba almacenado, pues este empezaba en el sector 0 y continuaba linealmente.

Con estos avances, fue posible poner los sources en su lugar, y así se volvió mucho más sencillo investigar, entre otras cosas, que causaba los problemas en el teclado. De la misma manera, fue posible desarrollar de manera más eficiente el modelo de FAT. Recién cuando el acceso a disco y a archivos adquirió una estabilidad suficiente fue que se comenzó a pensar en el problema del grabado de la imagen. Sin embargo, esta versión no escalaba cuando había que interactuar con distintos archivos (archivos de cambios, archivo de la imagen) y eso suscitó la necesidad de desarrollar el sistema de archivos presentado en el capítulo anterior.

6.2. Depuración del motor de objetos

Los primeros planes respecto al proyecto de este trabajo tenían como premisa modificar la menor cantidad posible de código del motor de objetos. Esto traería como beneficio que no fuera necesario entender en detalle todo el funcionamiento del mismo para alcanzar los objetivos principales del trabajo, lo cual era importante ya que el motor de objetos es un software de gran complejidad. Sin embargo, con el desarrollo del trabajo y los problemas encontrados durante el mismo, la unión del conocimiento que requirió adquirir el total de las etapas de investigación concluyó en que se haya estudiado una porción importante del motor de objetos. En este capítulo serán explicados, entre otras cuestiones, los problemas que fueron surgiendo, los cuales pusieron una mayor obligación en adquirir nuevos conocimientos para poder ser resueltos.

Como primer punto es importante aclarar que existe una dificultad inherente al tipo de software con el cual se trabajó en el transcurso de este proyecto. Los sistemas operativos son un elemento muy difícil de depurar ya que trabajan a un nivel muy bajo, interactuando frecuentemente con el hardware. Además, generalmente para depurar una aplicación se necesita una ayuda importante del sistema operativo, pero sin embargo estos no suelen ofrecer mucha ayuda para la depuración de si mismos.

Al tener la mayor parte de las funcionalidades del mismo implementadas en alto nivel, SqueakNOS permite una depuración con las herramientas de Smalltalk, logrando llevar el dinamismo del desarrollo de aplicaciones bajo la plataforma, al área de los sistemas operativos. Pero SqueakNOS corre sobre un motor de objetos, el cual está escrito en lenguajes de bajo nivel y, en ciertos momentos en que se trabaja con elementos de ese nivel, es un requerimiento indispensable el poder entender y hacer un seguimiento de qué está ocurriendo. Aquí se llega a un obstáculo importante ya que no es trivial depurar el motor de objetos de SqueakNOS con las herramientas tradicionales y en ciertos momentos, los cuales se detallarán en las próximas secciones, sin esta posibilidad es

demasiado complejo continuar.

No es simple depurar el motor de objetos de por sí, pues este está escrito en lenguajes de bajo nivel. Esto implica que los modelos de programación son muy pobres si se comparan con los que se pueden lograr en Smalltalk, y que además las herramientas con las que se cuenta son mucho más básicas también. A diferencia de cuando se trabaja a nivel Smalltalk, es imposible en el motor de objetos cambiar líneas de código y ejecutarlas sin reiniciar, y el sólo hecho de cambiar una línea implica apagar el sistema, volver a compilarlo, cargarlo y probarlo desde cero otra vez. Además, durante la depuración del motor de objetos muchas veces es imposible correr código Smalltalk, ya que el proceso de depurado requiere que el motor de objetos esté detenido, siendo este el único capaz de ejecutar código Smalltalk.

En diversos momentos se fueron necesitando diferentes capacidades de depuración con lo cual se fueron resolviendo las distintas dificultades mediante la implementación o acondicionamiento de distintas herramientas para lograr obtener la información necesaria del sistema durante su ejecución. Por ejemplo, a veces puede llegar a alcanzar con ver un simple mensaje de debug por consola, mientras que otras veces puede ser necesario observar una porción importante del estado de ejecución.

Lo primero y más simple que uno se puede imaginar que ayudaría para obtener algo de información del programa es que por pantalla se vayan imprimiendo, de alguna manera, mensajes con información pertinente. Pero en la versión de SqueakNOS con la cual se comenzó a trabajar esto no era posible si se quería depurar el motor de objetos, ya que la única manera de cambiar el contenido de la pantalla era mediante la ejecución de código Smalltalk. Las primeras versiones de SqueakNOS no sufrían este problema, pues utilizaban al adaptador de video en modo texto, lo cual permitía mostrar mensajes de texto con sólo escribirlos en una posición especial de la memoria de video. De esta manera, bastaba con implementar la función *printf* para que escribiera en la memoria de video, y poner un *printf* en el lugar donde fuera necesario mostrar un mensaje para que éste apareciera en pantalla. En las nuevas iteraciones de SqueakNOS se pasó a correr en modo gráfico para poder renderizar correctamente la interfaz de ventanas. Esto trajo como contrapartida la ruptura de la implementación de *printf* heredada, quitando la posibilidad de emitir mensajes de depuración por pantalla.

El primer paso fue entonces lograr restaurar la funcionalidad de *printf* para ver por pantalla los mensajes de depuración. El camino elegido fue pasar directamente los caracteres al *framebuffer*, previo mapeo entre cada caracter y la imagen del mismo. A partir de desarrollar este mapeo fue posible observar por pantalla toda la información necesaria del motor de objetos, siempre y cuando se agregue el printf correspondiente en el lugar indicado, se recompile todo el código y se reinicie. Pero el framebuffer se reescribe todo el tiempo para refrescar la pantalla, con lo cual una vez que el proceso de la imagen de smalltalk refresque la pantalla, se perderá la información de debug. En muchos casos este paso se da a velocidades que hacen que el mensaje escrito en pantalla sea casi imposible de divisar para el ojo humano con lo cual esta solución podía ser válida para algunos casos, pero de ningún modo sería implacable.

Una vez que fue posible ver por pantalla la información necesaria, el siguiente paso fue lograr

que esos importantes datos pudieran persistir por un período de tiempo lo suficientemente extenso, primero para lograr descifrarlos y luego para analizarlos. Lo que se requería era poder tomar los datos y grabarlos en archivos. Pero una vez más, esto requería la ejecución de código implementado en la imagen (ATA, FAT32, etc.), con lo cual no era posible si se quería depurar el motor de objetos. Una forma de resolver este problema hubiera sido implementar todas las funcionalidades de grabado de archivos en bajo nivel, pero esto contradecía totalmente la filosofía de SqueakNOS y este era un precio que no se estaba dispuesto a pagar.

Lo que se decidió entonces fue que los mensajes mostrados por *printf* no sólo fueran al framebuffer sino que también fueran enviados por el puerto serie de la máquina. Se eligió el puerto serie debido a lo simple de su funcionamiento y al poco código de bajo nivel que había que agregar para lograr lo buscado. Entonces conectando cualquier dispositivo al puerto serie y decodificando en este los caracteres que se envían desde la máquina se lograría tener el ambiente de depuración buscado en ese momento, con la persistencia adecuada de los mensajes. Sin embargo, no es para nada simple conectar un equipo al puerto serie que haga las funciones requeridas y no era factible invertir en el tampoco.

Aquí es uno de los momentos del trabajo donde aparecen las bondades de las máquinas virtuales de sistema. Éstas fueron explicadas en la sección 2.2 y con la capacidad de procesamiento y almacenamiento primario de las máquinas en la actualidad, son una herramienta indispensable para todo tipo de desarrollos en el área de sistemas operativos. También son importantes en otras áreas donde es indispensable lograr la compatibilidad con diversas plataformas y no se tiene la posibilidad económica o física de adquirir cada una de estas. Con este tipo de software, siendo *VMWare* el más utilizado en este trabajo, se puede ejecutar SqueakNOS directamente sobre una máquina virtual, esto es, sin la necesidad de suspender el sistema operativo sobre el cual se esta desarrollando.

Entonces, desde la máquina sobre la cual se está desarrollando se puede abrir una conexión al puerto serie de la máquina que esta corriendo SqueakNOS. De esta manera, se puede recibir la información del estado de ejecución de SqueakNOS, en algún otro proceso del sistema operativo host. La manera de lograrlo en este trabajo fue mediante la ejecución de un script, el cual se encarga de leer los datos del puerto serie y enviarlos a la salida estándar. Redireccionando luego esta salida a un archivo se tiene toda la información persistida. Usando herramientas como *cat* o *tail* de la plataforma *Unix*, se pueden visualizar estos datos en tiempo real.

Sin dudas que este fue uno de los avances más importantes en cuanto a la depuración. Con el método previo, no sólo a veces era imposible lograr la visualización adecuada, sino que a veces la máquina se colgaba debido a algún error grave y directamente era imposible obtener el mensaje de error. De esta manera todos los mensajes quedan persistidos en el archivo y es posible, teniendo los mensajes correspondientes en el código, intentar seguir el comportamiento ejecutado por el motor de objetos para rastrear al generador del error.

Con este tipo de depuración la realidad es que se logró avanzar y resolver una gran cantidad de errores que fueron surgiendo durante la etapa de implementación. Sin embargo, se estaba bastante lejos de lo necesario para tener una herramienta real de depuración donde sentirse cómodo para re-

solver los errores y realizar los seguimientos correspondientes de modelos tan complejos. La primera complejidad es la de encontrar el lugar exacto donde imprimir los mensajes en el código. Otra de las importantes dificultades tiene que ver con el mantenimiento de estos mensajes. El motor de objetos sobre el que se trabaja también es usado en otros entornos y la línea de desarrollo oficial del mismo es mantenida por gente ajena a SqueakNOS. Por lo tanto, cada vez que se lanzaba una nueva versión del motor de objetos, los mensajes de error tenían que ser portados desde la versión sobre la cual se estaba trabajando hacia la nueva versión. Este problema retrasó muchas de las puestas a punto de las actualizaciones del motor de objetos en SqueakNOS.

Por lo tanto, llegados los problemas más intensos y complejos de resolver fue necesario mejorar la capacidad de depuración para resolverlos. Estos problemas serán explicados en las siguientes secciones donde quedará claro el por qué esta mejora era necesaria para lograr superarlos. Por parte del depurador, se encontró que *VMWare* ofrece una potente herramienta (muy poco documentada y conocida) mediante la cual es posible depurar, con programas como *GDB*, el código fuente ejecutado sobre la misma. Una vez lograda la configuración correspondiente, fue posible la depuración del motor de objetos con las mismas herramientas con que es depurado en un entorno estándar. Con lo cual, mediante estas herramientas, se logró depurar el código de bajo nivel del motor de objetos de SqueakNOS, o en otras palabras, el sistema operativo de la misma manera que es depurado el motor de objetos en un entorno clásico, lo cual permitió obtener la información necesaria para resolver los problemas más complejos que fueron surgiendo sobre el final del trabajo.

6.3. Grabado de la imagen en disco

Como se explicó con anterioridad, la primera versión del grabado de la imagen, la *naive*, tenía como gran ventaja la simplicidad. Cabe destacar que no se plantearon las dos soluciones de antemano sino que, como suele suceder, fue mientras se avanzaba con el desarrollo de la primera solución, una vez que se contó con un conocimiento más certero del dominio, que surgió la siguiente idea, más avanzada, de utilizar el mecanismo de paginación.

La primera solución requirió una importante investigación, ya que fue necesario encontrar una zona libre de memoria, fuera del object heap, en donde ubicar la copia temporal. Para esto hubo que realizar un mapa de la memoria. Tanto el BIOS como el bootloader utilizan un espacio de direcciones en una zona baja de memoria, luego del cual se encuentra el motor de objetos, que ocupa una cantidad de memoria fija que incluye tanto el código del mismo, como los datos y el lugar para el *stack* y un buffer utilizado por *malloc*. Luego de esta zona, se pudo ver que GRUB cargaba el object heap, y que más allá se encuentra la memoria libre.

La cuestión no era sólo encontrar un espacio libre de memoria, sino saber cuanta era la memoria que el motor de objetos iba a necesitar exactamente, lo cual sólo se puede conocer luego de que éste haga el proceso de compactación correspondiente. Para esto fue necesaria cierta comunicación entre el motor de objetos y la imagen y esto fue logrado mediante la capa de *FFI*. Esta capa es la encargada de abstraer la complejidad de la comunicación entre distintos lenguajes o entornos, y fue usada para comunicar al motor de objetos (escrito en C), con el Smalltalk de la imagen.

Al motor de objetos se le agregó una estructura llamada *Computer* la cual posee ciertas variables con los datos que deben ser compartidos por él y por la imagen. Datos como el tamaño de la imagen a persistir, o la posición donde comienza la memoria de los objetos misma, o los datos que el cargador de arranque le provee al sistema son guardados en estas posiciones de memoria las cuales son accedidas y modificadas tanto por el motor de objetos como por la imagen. El objeto que desde la imagen es el encargado de abstraer todo el acceso a esta estructura es denotado como *VMBridge* y es inicializado cuando se inicializa la misma para que apunte al sector correspondiente de memoria. Mediante este simple mecanismo de comunicación se logró resolver todos los casos donde era absolutamente necesario una comunicación entre el motor de objetos y la imagen.

6.4. Habilitación de la paginación

El mecanismo de paginación fue explicado en la sección 2.7.1. En esta sección serán explicadas las dificultades encontradas al intentar hacer un manejo de la misma completamente desde SqueakNOS. El desarrollo se realizó sobre procesadores (reales o virtuales) de la arquitectura *X86*. Estos tienen dos modos de trabajo respecto a la interpretación de las direcciones de memoria. El primero es el modo plano, donde se interpreta cada dirección como la posición física de la memoria, el cual era el tipo de direccionamiento usado por SqueakNOS en un principio. En realidad las posiciones son relativas al segmento correspondiente, pero SqueakNOS y muchos sistemas operativos (como Linux y Windows) tienen una configuración inicial de los segmentos que asegura que las direcciones se mantengan iguales luego de ser calculadas respecto a ellos. El segundo es el modo virtual donde las direcciones son traducidas mediante un *directorio de páginas* que es administrado por el sistema operativo.

Para habilitar este segundo modo, lo cual era un objetivo mismo del trabajo, alcanza con simplemente setear el bit número 32 (el más significativo) del registro *CR0* en 1. Sin embargo, en el momento en que este bit es seteado, todas las direcciones pasan a ser virtuales y traducidas en tiempo de ejecución, con lo cual es absolutamente necesario tener el *directorio de páginas* correctamente configurado de antemano. También es necesario comunicarle al procesador en que posición de memoria (posición física) se encuentra esta estructura. Esto se logra, poniendo la posición de memoria correspondiente en el registro *CR3*.

La escritura de estos registros debe ser obligatoriamente hecha con código especial y teniendo el procesador en *modo protegido*. El modo ya es una característica de SqueakNOS, pero sin embargo, desde Smalltalk no se puede leer ni escribir directamente el valor de los registros del procesador, ya que se requiere un código especial en *assembler*. Para lograr esto desde Smalltalk, lo que se hizo fue crear la clase *MicroProcessor*, de la cual hereda el comportamiento general de los procesadores la subclase *X86Processor*. En esta última clase están definidos los mensajes que permiten modificar el contenido de los registros *CR0* y *CR3*. Simplemente con mensajes del estilo *cr0:* se puede cambiar el valor de este registro. Esto se logra llamando a una primitiva (escrita en el plugin de SqueakNOS) la cual se encarga a bajo nivel de hacer el cómputo correspondiente. Esta clase también tiene mensajes de más alto nivel como por ejemplo *enablePagingWithDirectory:* que se encarga de hacer todo

la configuración correspondiente para dejar la paginación habilitada con el *directorio de páginas* recibido como colaborador externo. A continuación se muestra un ejemplo de como son los mensajes de alto nivel que permiten este comportamiento.

```
X86Processor >> enablePagingWithDirectory: memoryAddress
self pageDirectory: memoryAddress.
self enablePaging.

X86Processor >> enablePaging
| pagingEnabled |
pagingEnabled := self cr0 bitAt: 32 put: 1. "Enable paging"
pagingEnabled := pagingEnabled bitAt: 17 put: 1. "Enable write protect"
self cr0: pagingEnabled
```

Explicado como fue modelado el mecanismo de paginación en SqueakNOS, queda por explicar cuales fueron los inconvenientes importantes que han surgido en el transcurso de su puesta a punto. En primer lugar, cuando se hicieron las primeras pruebas se notó que se escribía el registro *CR0* pero que nada sucedía. Esto era porque el bit que hay que modificar es justamente el bit más significativo. Como Smalltalk usa un rango de 31 bits para representar enteros (el último bit de todo objeto representa la diferencia entre un objeto y un entero) al interpretar desde el plugin el valor a escribir en el registro *CR0*, este era distinto al valor que se intentaba enviar realmente y nunca tenía el bit más significativo seteado. Sin embargo, una vez detectado esto, fue bastante simple lograr que desde el plugin se haga la traducción correcta (a este proceso se lo conoce como *marshaling*) y este problema fue rápidamente solucionado.

El segundo problema fue mucho más complejo de solucionar y requirió del uso y el análisis de gran cantidad de información de depuración. Las pruebas siempre fueron sobre un VMWare, el cual tenía asignados 256mb de memoria virtual. Dada esta cantidad de memoria, la estructura de directorio de traducción que se creó para la misma poseía la traducción de las direcciones lógicas desde la cero hasta la última correspondiente ya que se sabía que el archivo objeto no usaba otro tipo de direcciones. Sin embargo, apenas habilitada la paginación, la máquina se apagaba anunciando un error de triple fault. Este tipo de errores ocurren cuando surgen errores en cascada en la atención de interrupciones, y al tercero de estos la máquina se resetea.

La primer sospecha estuvo sobre la estructura de directorios creada. Cualquier falla de traducción, era plausible que genere este tipo de errores. El mismo seteo erróneo del registro CR3 hubiera llevado a lo mismo. Sin embargo, en primer lugar, el registro CR3 estaba bien seteado, y en segundo lugar, se comprobó mediante tests de unidad que las traducciones también estaban correctamente configuradas. Por lo tanto otro tipo de información de depuración era necesaria. La herramienta VMWare ofrece un archivo de log, con distintos niveles de logueo, donde se va mostrando el estado de la máquina virtual durante su ejecución. Mediante el análisis de este archivo en su modo más completo de logueo, donde se puede ver el contenido de cada registro del procesador por cada

instrucción ejecutada, se encontró que por algún motivo, el software de virtualización estaba haciendo referencia a una posición alta de memoria la cual no era correctamente traducida en la tabla presentada el procesador lo que generaba el error que derivaba en el triple fault. Es interesante destacar que para observar este tipo de comportamiento hubo que analizar un archivo de texto (el de log) de mas de 500mb. Además, la ejecución de la máquina virtual con estos niveles de logueo, es mucho más lenta.

6.5. Callbacks

Los fallos de página son una interrupción de hardware especial en lo que hace a SqueakNOS. Esto es así si se las compara con las interrupciones de hardware que este ya manejaba para atender a los dispositivos que generan eventos como el mouse y el teclado. En estos últimos casos, SqueakNOS atiende a bajo nivel la interrupción (ya que no hay otra posibilidad), pero lo único que hace es despertar un semáforo del lado de la imagen y volver de la misma. Esto genera que el procesador devuelva el control al motor de objetos, que al continuar ejecutando bytecodes rápidamente detecta que hay un proceso de alta prioridad esperando, listo para ejecutarse. En ese instante desaloja al proceso que estuviera corriendo y empieza a ejecutar el de alta prioridad, el cual es el que termina realmente atendiendo la interrupción. Se verifica empíricamente que los tiempos de atención son los adecuados.

Sin embargo, en los casos donde la interrupción es debido a un fallo de página, el comportamiento descripto no es posible. Si se mantuviera el comportamiento exhibido, la máquina se reiniciaría cada vez que se genere un fallo de página debido a un error de triple fault. Esto pasa porque cuando el procesador expone un fallo de página, quiere decir que el mismo está en un lugar de la ejecución del programa donde necesita una parte de la memoria la cual no está presente (o no es escribible). Si al volver de la interrupción, esto no es resuelto, sigue existiendo el mismo inconveniente y el procesador vuelve a exponer la misma interrupción. Por lo tanto los fallos de página deben resolverse inmediatamente, o por lo menos, no se puede volver al mismo punto de ejecución previo al fallo de no mediar la resolución del mismo.

La idea que se implementó para sortear esta dificultad fue la del uso de *callbacks*. Estos son un mecanismo mediante el cual el motor de objetos o cualquier otro código de bajo nivel, puede ejecutar directamente un método de Smalltalk. Es un mecanismo bastante complejo de implementar y la única librería que lo hacía al momento de comenzado el trabajo era *Alien*. El uso que se le daba a los callbacks de Alien era un poco distinto al que era necesario en SqueakNOS. En el uso estándar, sirve para desde Smalltalk llamar a funciones de bajo nivel que reciben como parámetro punteros a función. En estos casos, el objetivo es que la función pasada como parámetro termine llamando a un método Smalltalk que resuelva parte de su problema. La salida del motor de objetos y el reingreso ocurren en momentos de ejecución bastante controlados. El uso más clásico de este tipo de mecanismos es el de llamar a una función de C del estilo *Quicksort* la cual recibe como parámetro la función de comparación.

En el caso de SqueakNOS y los fallos de página, debía haber una comunicación con el motor de

objetos, para que cada vez que se genere un fallo de página se llame al mismo método Smalltalk, que sea el que resuelva el mismo. Además, hay tener en cuenta que debido a la recolección de basura los objetos se encuentran en constante movimiento en la imagen, lo cual puede causar problemas, y también que el código y los datos que se acceden para atender la interrupción no incluyan a la dirección que falló originalmente, ya que de lo contrario se caería en una recursión infinita. Finalmente, se logro instalar la librería en SqueakNOS y crear los objetos con las propiedades necesarias para lograr el objetivo. De esta manera, se logró la resolución de los fallos de página con código completamente de alto nivel. A continuación se exhibe el código de bajo nivel que atiende la interrupción.

```
void pageFaultISR(unsigned long errorCode) {
   extern Computer computer;
   extern t_IRQSemaphores IRQSemaphores;
   unsigned long virtualAddressFailure;
   computer.inPageFault++;
   computer.totalPageFaults++;
   asm volatile("movl %%cr2, %0" : "=a" (virtualAddressFailure));
   computer.pageFaultAddress = virtualAddressFailure;
   sti();
   if ((computer.inPageFault > 1) || (computer.inGC)){
      saveSnapshotPage(virtualAddressFailure);
   } else {
      VmStatus status;
      signalSemaphoreWithIndex(IRQSemaphores[15]);
      saveStatus(&status);
      computer.pageFaultHandler(virtualAddressFailure);
      releaseStatus(&status);
      }
   tabs-=1;
   computer.inPageFault--;
}
```

Como se puede observar, mas allá de las variables que se usan para la comunicación de ciertos parámetros entre el motor de objetos y la imagen, el código multiplexa entre dos caminos. Si se está resolviendo un fallo de página y surge otro, este segundo se resuelve nativamente para no caer en una recursión infinita. En el otro caso, se despierta al semáforo correspondiente y se llama a la función *computer.pageFaultHandler* la cual es la que llama a la función de Alien correspondiente que prepara toda la máquina para reingresar al motor de objetos en el lugar provisto para la atención de la interrupción. Una vez atendida ésta, el motor de objetos sigue en el estado previo al fallo de página, pero con este resuelto desde Smalltalk. Las funciones *saveStatus* y *releaseStatus* serán

explicadas en la próxima sección. A continuación, el bloque de código Smalltalk que se ejecuta para resolver los fallos de página por error de protección.

```
copyOnWriteHandlingBlock
   ^[ :anAddress :handler |
      | directoryEntry pageEntry table newTable contents start manager |
      manager := computer memoryManager.
      directoryEntry := manager pageDirectoryEntryFor: anAddress.
      directoryEntry isPresentAndAllowed ifFalse: [self error: 'Not present!'].
      pageEntry := manager pageTableEntryFor: anAddress.
      pageEntry isPresentAndForbidden ifFalse:
         [pageEntry isPresent ifTrue: [self]
                    ifFalse: [ handler doDefaultHandlingFor: anAddress ]]
         ifTrue:[
            contents := self pageContentsOf: anAddress.
            start := manager pageStartAddressOf: anAddress.
            savedPages at: start put: contents.
            pageEntry setPresentAndWritable
      ]
   ]
```

6.6. Callbacks seguros

El inconveniente más complejo que ha tenido que ser sorteado durante este trabajo apareció cuando el mecanismo de callbacks explicado en la sección anterior fue llevado a la práctica para resolver los fallos de página que se generan en el medio de un grabado de la imagen. En este caso, los callbacks pueden ser realizados en cualquier momento, mismo en el medio de la ejecución de un bytecode, de una recolección de basura o de cualquier otro proceso de bajo nivel del motor de objetos. Esto genera que no sea tan controlado el momento donde estos surgen como era la hipótesis laxa que tenía la implementación de Alien. Al hacer las pruebas, se comprobó que en ciertos momentos los callbacks fallaban dejando el motor de objetos en un estado inconsistente. Este fue el caso particular donde de no mediar la ayuda del conjunto de los mecanismos de depuración analizados, no hubiera sido posible continuar.

Una explicación exhaustiva del problema requeriría la comprensión casi total del motor de objetos de Squeak y del mecanismo de callbacks implementado por la librería de Alien. Sin embargo se puede ofrecer una conclusión resumida de lo obtenido a partir del análisis de la información de depuración, y de los pasos mediante los cuales se logró superar esta anomalía. Esencialmente, al poder generarse los fallos de página en cualquier situación, aparecieron lugares donde al generarse estos era necesario guardar una porción mayor del estado del motor de objetos que el que se estaba guardando en los casos donde el ingreso a los callbacks estaba circunscrito a ciertos casos. Además,

se encontró que en ciertos casos, se generaba un fallo de página mientras se estaba ejecutando una recolección de basura y esto dejaba al motor de objetos inconsistente también. La solución en este caso fue aprovechar la implementación de atención de la interrupción nativa, ya que ingresar a ejecutar bytecodes en el medio de una recolección de basura parece extremadamente peligroso para la consistencia de los objetos.

Capítulo 7

Conclusiones

En el presente trabajo se describió la implementación de diversos componentes en su mayoría relacionados con el manejo de memoria, desde la memoria de acceso aleatorio hasta la de almacenamiento (primaria y secundaria respectivamente). Se desarrolló un controlador para dispositivos ATA, un filesystem FAT32 con las conexiones necesarias para ser usado desde una imagen de Squeak estándar, un modelo de la memoria física, un mecanismo para el manejo de la paginación y dos métodos alternativos de persistencia de la imagen, uno de los cuales se basa en la utilización del mecanismo de paginación.

Si bien la sola existencia de SqueakNOS es un comienzo de respuesta afirmativa a la pregunta sobre la factibilidad de escribir un sistema operativo en un lenguaje de alto nivel, todavía quedan muchos interrogantes por responder para poder dar una respuesta completa. En este sentido, los resultados obtenidos durante este trabajo han sido un paso más, al determinar que es posible escribir controladores de discos, y mecanismos de administración de páginas de memoria completamente en Smalltalk.

La implementación del acceso al disco permitió la lectura del código fuente de los métodos, lo cual facilita enormemente el trabajo de desarrollo directamente dentro de SqueakNOS mientras el mismo está siendo ejecutado. En la misma dirección, es un gran avance la posibilidad de persistir la imagen. Todos estos logros colaboran en pos del desarrollo de un sistema completamente auto-sustentable, el cual puede ser escrito y evolucionado desde si mismo. Un sistema completamente reflexivo, pero ya sin la barrera del sistema operativo, como se acostumbra a ver en las implementaciones de Smalltalk tradicionales.

A su vez, para cada uno de los problemas resueltos fue posible encontrar un diseño que cumpla con la filosofía de SqueakNOS, es decir que no recurra a código ensamblador o C, salvo en casos contados y muy precisos. Particular importancia tiene este punto en el caso de la paginación, ya que las dificultades técnicas del problema generaban dudas sobre la posibilidad de su resolución completamente en Smalltalk. Además, esta resolución permitió alcanzar un conocimiento muy alto del funcionamiento de los sistemas operativos, tanto de las tareas que realiza, cuales son los módulos que posee y como realiza la conexión con los dispositivos, lo cual era uno de los objetivos iniciales del trabajo. Asimismo, los modelos desarrollados tanto para discos rígidos, como para el filesystem o

para la memoria y el procesador pueden ser accedidos directamente por cualquier persona interesada en el área, y sirven de documentación y de ejemplo para el claro entendimiento del dominio que abarcan.

El caso de la paginación nos lleva también a una de las conclusiones más interesantes de este trabajo. El uso de la paginación para implementar un mecanismo copy-on-write eficiente no es algo nuevo. Sin embargo, los sistemas operativos de uso general no suelen ofrecer acceso libre al mecanismo de paginación, por lo que en ellos resulta muy difícil, si no imposible, poner esta utilidad a disposición de las aplicaciones comunes. En SqueakNOS, en cambio, todo el acceso al hardware está en principio disponible para las aplicaciones. A pesar de que esto es criticable desde el punto de vista de la seguridad, gracias a esta característica, a la maleabilidad del ambiente Smalltalk, y a la simplicidad que brindan los modelos de alto nivel, es posible experimentar con nuevos mecanismos y realizar todo tipo de pruebas, que en los otros casos serían imposibles, como los que llevamos a cabo al implementar la persistencia de memoria de manera lazy. Esto da lugar a pensar en otras posibilidades, como por ejemplo, de que manera es posible unificar y sincronizar los mecanismos de administración de memoria y de recolección de basura, o si es posible la utilización del mecanismo de protección para implementar barreras de lectura y/o escritura en el caso del multiprocesamiento.

Otro de los objetivos logrados durante el trabajo fue mantener un alto grado de independencia respecto del motor de objetos existente. Sólo fue necesaria una pequeña modificación al recolector de basura, y alguna que otra optimización que no resulta indispensable. Básicamente, al recolector de basura hubo que agregarle un mecanismo simple, el cual consiste en dejar registrado en algún flag que se está ejecutando el mismo para que en caso de generarse fallos de página en el interín, estos se resuelvan nativamente. El resto del desarrollo se produjo en módulos complementarios y la mayor parte en la imagen misma. De este modo, se mantiene la propiedad que poseía el proyecto respecto a poder ser enlazado, sin mayores obstáculos, a diferentes versiones del motor de objetos. Además, debería ser posible, ya con un poco más de trabajo, pero en principio manejable, portar el proyecto a otros dialectos Smalltalk que no tengan un acoplamiento demasiado fuerte con algún sistema operativo.

Si bien una finalidad de SqueakNOS es poder trabajar completamente en Smalltalk, esto aún no es posible en la actualidad en determinadas ocasiones. Como pudo observarse, a lo largo del trabajo fue necesario conocer en detalle el funcionamiento del motor de objetos, el cual si bien está escrito en slang, termina siendo traducido a C. Esto hizo que en varias etapas la mayor parte del trabajo se desarrolle leyendo código fuente C, y depurando con herramientas como GDB y por medio de mensajes en la consola. Esto está muy lejos del dinamismo buscado. Este tipo de problemas podría ser resuelto total o parcialmente si se contara con un motor de objetos completamente implementado en Smalltalk, que genere código nativo on-the-fly en vez de requerir una previa traducción a C.

SqueakNOS permite trabajar en un ambiente de complejidad muy reducida comparado con los sistemas operativos comunes. Los modelos de objetos y las herramientas para el desarrollo diario son extremadamente poderosas y facilitan el trabajo enormemente. Por otro lado, se observa que en determinadas ocasiones es necesario contar con un conocimiento amplio sobre el funcionamiento del motor de objetos sobre el cual se corre, para poder modificarlo y extenderlo. Si bien esto disminuye

en parte algunas de las ventajas originales, forzando a trabajar en bajo nivel en ciertos momentos, finalmente prevalece el hecho de trabajar con modelos de alto nivel, ya que permite una mucho mayor comprensión del dominio. Además, esta necesidad de entendimiento del motor de objetos se da en pocas ocasiones, y en la mayoría de los casos se utiliza para el desarrollo de funcionalidades genéricas, que puedan ser reutilizadas luego con otros objetivos, distintos de los originales, sin requerir modificaciones.

Además, se comprueba con el transcurrir del trabajo el carácter y las posibilidades educativas que abre el proyecto al lograr modelar en niveles de abstracción bastante altos y con una sintaxis relativamente cercana a la de los lenguajes humanos, elementos tan complejos de *hardware* y *software*. En algunos casos no es tanta la complejidad inherente de estos sino la dificultad que genera comprenderlos a partir de las herramientas usadas para modelarlos. La posibilidad dinámica de interactuar con los objetos que representan los entes correspondientes ofrece en todo momento la posibilidad de darle nuevos usos (como el caso de este trabajo donde se usan las herramientas del procesador para lograr implementar la técnica de *copy on write*), o simplemente de explorarlos, entenderlos y mejorarlos

Capítulo 8

Trabajo futuro

El proyecto actualmente queda en un estado de funcionalidad suficiente como para ser usado y evolucionado desde sí mismo. De todas maneras quedan abiertas las posibilidades de mejorar los desarrollos presentados en este trabajo, sobre todo respecto al desempeño. Pero fundamentalmente, se abre la posibilidad de desarrollar en Smalltalk una vasta cantidad de manejadores de dispositivos y herramientas para lograr que las funcionalidades soportadas por SqueakNOS sean cada vez más abarcativas respecto al desarrollo de la ciencia misma.

Las plataformas Squeak y Pharo poseen en la actualidad modelos de objetos de un nivel cercano al de los sistemas operativos, con lo cual el agregado de manejadores de dispositivos y un modelo que sirva de enlace entre los objetos actuales y éstos, haría que instantáneamente una gran cantidad de funcionalidades queden disponibles para el usuario. Por ejemplo, la implementación de un manejador de placas de red, más algunos de los algoritmos más importantes de la pila tcp/ip, generarían la posibilidad de usar desde SqueakNOS el manejador de paquetes *Metacello* [Met]. También, quedarían disponibles para el uso otros tantos protocolos de aplicación ya implementados, como http, pop3, smtp, ssh, ftp, o ssl.

Asimismo, SqueakNOS podría servir como servidor para cualquiera de estos servicios mencionados. Queda por explorar que ventajas y desventajas tendría en cuanto a desempeño y propiedades demostrables, tener el control total de la máquina subyacente desde Smalltalk. Seguro que se encontrarían importantes diferencias corriendo un servidor de mails sobre Smalltalk y un sistema operativo, que corriéndolo todo sobre el mismo ambiente.

Además de la evolución de SqueakNOS, desde su creación también han progresado muchos otros proyectos totalmente ortogonales al mismo. Una de las áreas donde la comunidad está intentando converger es con respecto al uso de *FFI*, donde existen muchas implementaciones con similares prestaciones. SqueakNOS hace un uso intensivo de esta funcionalidad y se puede afirmar que sin éstas, el proyecto no sería viable. Las nuevas implementaciones buscan mantener la compatibilidad con las versiones anteriores, mientras que al mismo tiempo añaden funcionalidades importantes de alto nivel. Esto permite esencialmente interactuar de manera más limpia y fácil con componentes de más bajo nivel como es el hardware mismo.

NativeBoost [Nat] es una de estas nuevas implementaciones y presenta una novedad respecto a

las demás. Permite la ejecución de código de bajo nivel sin haberlo escrito en bajo nivel, sino dentro de un método escrito en Smalltalk. Esto tiene implicancias fuertes para el proyecto SqueakNOS, ya que en caso de usar esta implementación de *FFI* podría evitarse el paso de compilación necesario para ejecutar algunos cambios al motor de objetos, logrando que el desarrollo sea aún más dinámico que el modelo actual y mucho más acorde al proceso de desarrollo en Smalltalk.

Otra área que ha tenido una gran evolución es el de las máquinas virtuales. Más concretamente, los proyectos Squeak y Pharo están migrando hacia un nuevo motor de objetos, siempre preservando la compatibilidad con el anterior. Este nuevo motor de objetos presenta ventajas significativas con respecto al desempeño comparándolo con el motor de objetos estándar, el cual es usado por SqueakNOS. El nuevo motor de objetos, llamado *Cog* [Cog] y desarrollado por *Eliot Miranda*, usa la pila nativa directamente en lugar de usar la pila de Smalltalk lo cual repercute en una mayor eficiencia. Además, cuenta con un compilador en tiempo de ejecución, con lo cual la primera vez que se ejecuta cada método este es compilado a código nativo y luego cada nueva ejecución del mismo es ejecutada a nivel nativo logrando una velocidad de ejecución considerablemente más rápida.

La transición del motor de objetos actual a *Cog* no es un proceso trivial, debido a varias razones. Por un lado, Cog cuenta con un *heartbeat*, es decir un timer periódico, que le permite realizar un scheduling más adecuado de los distintos procesos de smalltalk. Este timer depende directamente de la implementación que haga cada sistema operativo, por lo que para utilizar Cog sería necesario primero implementar el timer usado por el heartbeat. Además, SqueakNOS tiene sus propios *makefiles* los cuales habría que modificar, así como también algunos *plugins* para que sean compatibles con el nuevo motor de objetos. Sin embargo, tampoco debería ser un paso demasiado costoso y sin dudas es uno de los siguientes pasos a realizar que lograrían una mejora ostentosa en la experiencia del usuario aunque no necesariamente en los temas abordados por este trabajo.

Otra cuestión interesante que sería importante abordar es la utilización de callbacks para lograr portar la mayor cantidad posible de código que se este haciendo a bajo nivel hacia el lado de la imagen. Por ejemplo, una idea interesante que surgió sobre el final del trabajo fue la de implementar el malloc en la imagen y que cada vez que se lo llame desde bajo nivel se ejecute en Smalltalk mediante un callback. Esto permitiría tener un manejo casi total de la memoria de la máquina desde código de alto nivel.

Un problema de la implementación de callbacks de *Alien*, utilizada en este trabajo, es que estos se ejecutan dentro del proceso que se encarga de regular la interfaz gráfica dentro de Smalltalk. Si dentro del callback surge algún error o en su defecto se quiere hacer una depuración de la ejecución esto hace que se paralice la imagen ya que el proceso de la interfaz gráfica queda parado y no permite el refresco de la pantalla. La solución que se propone es hacer que el mecanismo de callbacks sea configurable respecto a en que proceso interno de la imagen va a ser ejecutado, para lograr evitar este posible defecto.

Finalmente, con una relación más estrecha a lo presentado en este trabajo, hay tres cuestiones que sería interesante explorar. La primera es la implementación de diversos *sistemas de archivos* como *NTFS*, y *EXT* (en sus diferentes versiones). La segunda es la mejora en los mecanismos de cache del controlador ATA, y del filesystem a partir de los cuales se podría mejorar considerable-

mente el rendimiento. Y finalmente, explorar la posibilidad de usar dos *motores de objetos*, uno de los cuales posea la responsabilidad de controlar al otro. Logrando una correcta sincronización (tarea nada sencilla), se podría persistir la imagen desde Smalltalk sin tener que utilizar otros mecanismos como los presentados en este trabajo en el capítulo 5.

Un problema interesante a analizar en el futuro es el de como lograr compatibilidad entre SqueakNOS y el software existente, y como posibilitar la ejecución de código escrito en otros lenguajes de programación. Actualmente SqueakNOS sólo permite la ejecución de programas escritos en Smalltalk. Si bien esta característica es útil, dadas las ventajas de desarrollo con las que se cuenta, también es a veces un problema, ya que se corta el acceso a la infinidad de programas que fueron escritos en otros lenguajes y pensados para otros sistemas operativos. Una posibilidad para salvar este problema podría ser la creación de un modelo de objetos que representen al formato ejecutable ELF y la implementación de ciertas secciones de la interfaz POSIX, como ser los syscalls. De esta manera sería posible ejecutar programas compilados para otros sistemas.

Finalmente, un terreno donde todavía no se ha trabajado en el proyecto, pero en el cual seguro hay mucho por investigar y proponer es el de seguridad. Esta es una área que ha venido teniendo una participación importante en el software debido a la cantidad cada vez mayor de equipos de diversa índole interconectados y expuestos a todo tipo de ataques. En este caso, mas allá de que todavía el sistema no posee capacidades de conexión de red, hay un modelo de objetos donde todos estos son accesibles. Esto significa que cualquier objeto podría acceder a la memoria y modificarla, al procesador y cambiarle algún bit a un registro, etc. No existe en el sistema un modelo de seguridad que regule el acceso a los objetos. Esto es una limitación inherente al dialecto de Smalltalk sobre el cual se trabaja. Lo más interesante observado al respecto es el trabajo plasmado en el proyecto *Newspeak* [GB08].

Bibliografía

[ACE] Benjamin Chelf Seth Hallem Andy Chou, Junfeng Yang and Dawson Engler. An empirical study of operating systems errors.

[AG] David Robson Adele Goldberg. *Smalltalk-80, The language and its implementation.* Addison-Wesley.

[ATA] Ata. http://www.t13.org.

[AWA] Kai Li Andrew W. Appel, John R. Ellis. Real-time concurrent collection on stock multiprocessors.

[Boc] Bochs: An ia-32 emulator. http://bochs.sourceforge.net/.

[Cog] Cog. http://www.mirandabanda.org/cog/.

[FAT] Fat32. http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx.

[GB08] Vassili Bykov Eliot Miranda Yaron Kashai Gilad Bracha, Peter Ahe. The newspeak programming platform. May 2008.

[Gem] Sitio oficial de gemstone. http://www.gemstone.com.

[Gru] Gnu grub. http://www.gnu.org/software/grub/.

[Han01] Brinch Hansen. *Classic operating systems.* Springer-Verlag New York, Inc., 2001.

[Hou] House: Haskell user's operating system and environment. http://programatica.cs.pdx.edu/House/.

[Ing81] Dan Ingalls. Design principles behind smalltalk. *BYTE*, 6(8):286–298, August 1981.

[Jav] Sitio oficial de java. http://www.java.com/es/about/.

[JNo] Jnode: a new java operating system. http://jnode.sourceforge.net/index.html.

[JX] Jx: The fast and flexible java os. http://www.jxos.org/.

[Kay] Alan Kay. *The Computer Revolution hasn't happend yet. Keynote OOPSLA 1997.* http://video.google.com/videoplay?docid=-2950949730059754521.

[Lil] Lilo. http://www.acm.uiuc.edu/workshops/linux_install/lilo.html.

[lsa] Jnode creator blog. http://lsantha.blogspot.com/.

[Met] Metacello. https://gforge.inria.fr/frs/download.php/28462/Metacello.pdf.

[MG] Christian Wawersich Jurgen Kleinoder Michael Golm, Meik Felser. The jx operating system.

[Mul] Especificación multiboot. http://www.gnu.org/software/grub/manual/multiboot/multiboot.html.

[Nat] Nativeboost. http://code.google.com/p/nativeboost/wiki/NativeBoost.

[NL06] Linda Null and Julia Lobur. *The Essentials of Computer Organization And Architecture.* Jones and Bartlett Publishers, Inc., USA, 2006.

[Row01] Tim Rowledge. *A Tour of the Squeak Object Engine. In M. Guzdial and K. Rose, editors, Squeak: Open Personal Computing and Multimedia.* Prentice Hall, 2001.

[SGG08] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts.* Wiley Publishing, 2008.

[SN05] James E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38:32–38, 2005.

[Squa] Squeak. http://www.squeak.org.

[Squb] Squeaknos. http://www.squeak.org/squeak/1762.

[Sta97] William Stallings. *Operating Systems - Second edition.* Prentince Hall, 1997.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[TH] Rebekah Leslie Andrew Tolmach Thomas Hallgren, Mark P Jones. A principled approach to operating system construction in haskell.

[Tio] Tiobe index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[Ung] David Ungar. Generation scavenge: A non-disruptive high performance storage reclamation algorithm.

[Vbo] Virtualbox virtual machine. http://www.virtualbox.org/.

[Vis] German Viscuso. Bases de objetos.

[Vmw] Vmware virtual machine. http://www.vmware.com/.

[Wik] Wikimaquinavirtual.

Agradecimientos

Este trabajo se desarrolló en parte durante gracias al aporte de la Universidad Nacional de General Sarmiento.

Gracias a:

- A Hernán y a Gera, por el apoyo, la paciencia, las ideas y especialmente por todo el tiempo y esfuerzo que le dedicaron a lo largo de este trabajo.
- A los jurados, Gabriela Arévalo y Máximo Prieto.
- Al grupo de imágenes, por recibirnos cuando no teníamos donde ir y por la continua buena onda para con nosotros.
- A la Universidad de General Sarmiento.
- A Caesar Systems en general y todos sus integrantes en particular.
- A Jime y Aní por bancarnos incodicionalmente.
- A nuestras familias.
- Al cuerpo de docentes y no docentes del Departamento, y de la Facultad.
- A nuestros amigos de la generación: Cisquito, Pachi, Gennaro, Alfa, Maniche, el Marine, Henri, Marta, Arti, Eze, Nigel, Cesaro, Gaboto, Sol, Lichu, Tiger, Tevi, Facu, Matías L., Jose, Daniel, Lucio, David.
- Al equipo campeón Tu amiga te tiene ganas, y al no tan glorioso pero digno Kaiserlautern.
- A los amigos del barrio y de toda la vida.
- A los grandes amigos de Las Lajas.
- A River por irse al descenso.