1 Introducción

¡Bienvenidos!

Este es el libro de **Redox**, que analiza -casi- todo sobre **Redox**: diseño, filosofía, funcionamiento, cómo contribuir, cómo implementar **Redox** y muchos aspectos más.

Te notificamos que este libro se está actualizando permanentemente.

Si deseas directamente probar Redox, consulta: **primeros pasos**.

1.1 ¿Qué es Redox?

Redox es un sistema operativo de propósito general escrito en **Rust**. Nuestro objetivo es proporcionar un microkernel similar a **Unix** que funcione completamente, que sea seguro y gratuito.

Tenemos una compatibilidad moderada con **POSIX**, lo que permite a **Redox** ejecutar muchos programas sin portarlos.

Nos inspiramos en **Plan9**, **Minix**, **Linux** y **BSD**. Redox tiene como objetivo sintetizar años de investigación y experiencia ganada con esfuerzo en un sistema que es moderno y familiar.

En este momento, **Redox** admite:

- **Todas las CPU x86-64.**
- **Tarjetas gráficas con soporte VBE (todas las tarjetas Nvidia, Intel y AMD de la última década tienen esto).**
- **Discos AHCI.**
- **Tarjetas de red E1000 o RTL8168.**
- **Controladores de audio Intel HDA.**
- **Ratón y teclado con emulación PS/2. TODO: actualizado**

Este libro se divide en los siguientes capítulos: **Ver tabla de Contenidos**

TODO

Está escrito de manera que no necesitas ningún conocimiento previo en desarrollo de **Rust** y/o **Sistemas Operativos**.

1.2 Nuestras metas

Redox intenta crear un sistema operativo de propósito general completo y de pleno funcionamiento con un enfoque en la seguridad, la libertad, la confiabilidad, la corrección y el pragmatismo.

Queremos poder usarlo, sin obstáculos, como una alternativa a Linux en nuestras computadoras. Debería poder ejecutar la mayoría de los programas de Linux con sólo modificaciones mínimas.

Nuestro objetivo es lograr un ecosistema de Rust completo y seguro. Esta es una elección de diseño que mejora la exactitud y la seguridad (**Por qué Rust**).

Queremos mejorar el diseño de seguridad en comparación con otros núcleos similares a **Unix** mediante el uso de valores predeterminados seguros y la prohibición de configuraciones inseguras cuando sea posible.

No son objetivos de Redox

No somos un clon de **Linux**, ni compatible con POSIX, ni somos científicos locos que desean rediseñar todo. En general, nos ceñimos a diseños correctos y bien probados. Si no está roto, no lo arregles.

Esto significa que una gran cantidad de programas y librerías estándar serán compatibles con **Redox**. Algunas otras que no sean compatibles con nuestras decisiones de diseño tendrán que ser portadas.

La clave aquí es el compromiso entre corrección y compatibilidad. Idealmente, deberíamos obtener ambos, pero desafortunadamente, no siempre se puede lograr.

1.3 Nuestra filosofía

Creemos en el **software libre**.

Redox OS se empaquetará solo con software gratuito y compatible, para garantizar que toda la distribución predeterminada pueda inspeccionarse, modificarse y redistribuirse. El software que no permite estas características, es decir, el software propietario, va en contra de los objetivos de seguridad y libertad y no será respaldado por **Redox OS**. Respaldamos las pautas de distribución del sistema libre de **GNU**.

Para ver una lista de licencias compatibles, consulte la **Lista de licencias de GNU**.

El sistema operativo **Redox** tiene predominantemente una licencia de estilo MIT y X11, incluido todo el software, la documentación y las fuentes. Solo hay algunas excepciones a esto:

- GNU Unifont, que es GPLv2
- Fira font, que es SIL Open Font License 1.1
- Íconos de Faba y Moka, que son GPLv3
- Librería Newlib C, **que es una serie de licencias de software libre, en su mayoría BSD**
- NASM, qué es BSD 2-cláusula *TODO: actualizar*

La licencia estilo MIT X11 tiene las siguientes propiedades:

- Le brinda al usuario de software, acceso completo y sin restricciones al software, de modo que pueda **inspeccionar, modificar y redistribuir** sus cambios.
    - **Inspección:** Cualquiera puede verificar el software en busca de vulnerabilidades de seguridad.
    - **Modificación:** Cualquiera puede modificar el software para corregir vulnerabilidades de seguridad.
    - **Redistribución:** Cualquiera puede redistribuir el software para parchear las vulnerabilidades de seguridad

- Es compatible con licencias GPL - Los proyectos con licencia GPL se pueden distribuir con Redox OS
- Permite la incorporación de software libre no compatible con GPL, como OpenZFS, que tiene licencia CDDL

Sin embargo, la licencia no restringe el software que puede ejecutarse en Redox, y gracias a la arquitectura de microkernel, incluso los componentes tradicionalmente y estrechamente acoplados, como los controladores, pueden distribuirse por separado, por lo que los mantenedores pueden elegir la licencia que deseen para sus proyectos.

1.4 ¿Por qué un nuevo sistema operativo?

¿Por que Redox?

Hay un montón de sistemas operativos activos. Es natural preguntarse ¿por qué deberíamos construir un nuevo sistema operativo? ¿no sería mejor contribuir a un proyecto existente?

La comunidad **Redox** cree que los proyectos existentes se quedan “cortos” y que nuestros objetivos se cumplen mejor con un nuevo proyecto construido desde cero.

Consideremos 3 proyectos existentes.

Linux

Linux "gobierna" el mundo y arranca en todo tipo de dispositivos, desde servidores de alto rendimiento hasta pequeños dispositivos integrados. De hecho, muchos miembros de la comunidad Redox ejecutan Linux como sus principales estaciones de trabajo. Sin embargo, Linux no es una plataforma ideal para nuevas innovaciones en el desarrollo de sistemas operativos.

- **Legado hasta el infinito:** las viejas llamadas al sistema permanecen para siempre, los controladores para el hardware obsoleto permanecen en el núcleo como partes obligatorias. Si bien se pueden deshabilitar, no es

necesario ejecutarlos en el espacio del kernel ya que pueden ser una fuente de fallas del sistema, problemas de seguridad y errores inesperados.
- **Enorme base de código:** para contribuir, debes encontrar un lugar para encajar en casi 30 millones de líneas de código, solo en el kernel. Esto se debe a la arquitectura monolítica de Linux.
- **Licencia no permisiva:** Linux tiene licencia GPL2, lo que impide el uso de otras licencias de software libre dentro del kernel. Para obtener más información sobre por qué, consulta **nuestra filosofía**.
- **Problemas de seguridad con la memoria:** Linux ha tenido numerosos problemas con la seguridad de la memoria a lo largo del tiempo. C es un buen lenguaje, pero para un sistema muy crítico en materia seguridad, C es muy difícil de usar de manera segura.

BSD

No es ningún secreto que preferimos BSD. La comunidad BSD ha liderado el camino en muchas innovaciones en las últimas 2 décadas. Cosas como **Jails**, **ZFS** y **Hammer** producen sistemas más confiables, mientras que otros sistemas operativos recién se están actualizando.

Dicho esto, BSD tampoco satisface nuestras necesidades:

- Todavía tiene un núcleo monolítico. Esto significa que un solo error en el controlador puede colapsar, bloquearse, fallar o en el peor de los casos, dañar el sistema.
- El uso de C en el kernel hace que sea probable escribir código con problemas de seguridad en la memoria.

MINIX

¿Acerca de MINIX? Su diseño de microkernel es una gran influencia en el proyecto Redox, especialmente por razones como la **confiabilidad**. MINIX es el más acorde con la filosofía de Redox. Tiene un diseño similar y una licencia similar.

- **Uso de C:** nuevamente, nos gustaría que los controladores y el kernel se escribieran en Rust, para mejorar la legibilidad y la organización, y para detectar mayor cantidad de posibles errores de seguridad. En comparación

con los núcleos monolíticos, Minix es en realidad una base de código muy bien escrita y manejable, pero por ejemplo aún es propensa a errores de seguridad de la memoria. Desafortunadamente, estas clases de errores pueden ser bastante fatales debido a su naturaleza inesperada.

- **Falta de compatibilidad con controladores:** MINIX no funciona bien en hardware real, en parte debido a que se centra menos en el hardware real.
- **Menos enfocado en "Todo es un archivo":** MINIX se enfoca menos en "Todo es un archivo" que otros sistemas operativos como Plan9. Estamos particularmente enfocados en este concepto, para crear una infraestructura de programa más uniforme.

La necesidad de algo nuevo

Debemos reconocer que nos gusta la idea de escribir algo que es nuestro (síndrome de Not Invented Here). Hay numerosos lugares en el código fuente de MINIX 3 adónde nos gustaría realizar cambios, tantos que tal vez una reescritura en Rust tenga más sentido.

- Modelo VFS diferente, basado en URL, donde un programa puede controlar un completo filesystem segmentado
- Modelo de controlador diferente, donde los controladores interactúan con sistemas de archivos como `red:` y `audio:` para proporcionar funciones
- Un filesystem diferente, Redox FS, es una implementación de **TFS** en curso
- Espacio de usuario escrito principalmente en Rust
- **Orbital**, una nueva GUI

1.5 Redox comparado con otros sistemas operativos

Compartimos bastante con otros sistemas operativos.

System call

La interfaz de Redox syscall es Unix-y. Por ejemplo, tenemos `open`, `pipe`, `pipe2`, `lseek`, `read`, `write`, `brk`, `execv`, etc. Actualmente, soportamos las 31 llamadas al sistema de Linux más comunes.

En comparación con Linux, nuestra interfaz **syscall** es minimalista. Esto no se debe a una etapa de desarrollo, sino a un diseño minimalista.

"Todo es una URL"

Esta es una generalización de "todo es un archivo", inspirada en gran medida por Plan 9. En Redox, los "recursos" (*TODO: link*) pueden ser tanto de tipo socket como de tipo archivo, lo que los hace lo suficientemente rápidos para usarlos virtualmente para todo.

De esta forma, obtenemos una API de sistema más unificada. Explicaremos esto más adelante, en **URL, esquemas y recursos**.

El kernel

El kernel de Redox es un microkernel. La arquitectura está inspirada en gran medida en MINIX.

A diferencia de Linux o BSD, Redox tiene solo 16000 líneas de código en el kernel, un número que a menudo está disminuyendo. La mayoría de los servicios se proporcionan en el espacio del usuario.

Cantidades mucho más pequeñas de líneas de código en el kernel hace que sea más fácil encontrar y corregir errores/problemas de seguridad de manera más eficiente. Andrew Tanenbaum (autor de MINIX) afirmó que por cada 1000 líneas de código correctamente escritas, hay un error. Esto significa que para un núcleo monolítico con casi 30.000.000 de líneas de código, habría casi 30.000 errores. En un microkernel con solo 16.000 líneas de código existen alrededor de 16 errores.

Cabe señalar que las líneas adicionales basadas fuera del espacio del kernel son menos peligrosas, no necesariamente un número menor de ellas.

La idea principal es tener los componentes y controladores que están dentro de un kernel monolítico en el espacio del usuario y seguir el Principio de Mínima Autoridad (Principle of Least Authority -POLA-). Aquí es donde se encuentra cada componente individual:

- Completamente aislado en la memoria y como un proceso de usuario separado
    - La falla de un componente no bloquea otros componentes
    - El código extraño y no confiable no expone todo el sistema
    - Los errores y el malware no pueden propagarse a otros componentes
- Tiene comunicación restringida con otros componentes
- No tiene privilegios de administrador/superusuario
    - Los errores se mueven al espacio del usuario, lo que reduce su poder.

Todo esto incrementa significativamente la fiabilidad del sistema. Esto es útil para aplicaciones de misión crítica y para usuarios que desean minimizar los problemas en sus sistemas informáticos.

1.6 ¿Por qué Rust?

¿Por qué escribir un sistema operativo en Rust? ¿Por qué incluso utilizar Rust? Rust tiene enormes ventajas, dado que para los sistemas operativos, la seguridad es importante y esencial, en realidad.

Consideremos que los sistemas operativos son una parte muy integrada de la informática, son un componente muy crítico para la seguridad.

Ha habido numerosos errores y vulnerabilidades en Linux, BSD, Glibc, Bash, X, etc. a lo largo del tiempo, simplemente debido a fallas en la memoria y seguridad de tipos. Rust hace esto muy bien, al lograr cumplir la seguridad de la memoria de forma estática.
El diseño importa, pero también su implementación. Rust intenta evitar estas condiciones inseguras e inesperadas en la memoria (que son una fuente importante de errores críticos de seguridad). El diseño es una fuente de problemas muy transparente. Sabes lo que está pasando, sabes lo que se pretendía y lo que no.

El diseño básico de la separación del espacio kernel/user es bastante similar a los sistemas Unix-like, en este punto. La idea es más o menos la misma: se separa el

kernel y el espacio del usuario, a través de una aplicación estricta por parte del kernel, que administra la memoria y otros recursos críticos.

Sin embargo, tenemos una ventaja: memoria reforzada y seguridad de tipos. Este es el lado fuerte de Rust: una gran cantidad de "errores inesperados" (por ejemplo, undefined behavior) se eliminan en el momento de la compilación.

El diseño de Linux y BSD es seguro, no así su implementación:

- **Vulnerabilidades del kernel de Linux**
- **Vulnerabilidades de Glibc**
- **Vulnerabilidades de bash**
- **Vulnerabilidades de X**

Selecciona los enlaces de arriba. Probablemente notarás que muchos son errores que se originan en “unsafe conditions” (que Rust elimina de manera efectiva) como desbordamientos de búfer, no el diseño general.

Esperamos que el uso de Rust produzca finalmente un sistema operativo más seguro.

Unsafes

`unsafe` es una forma de decirle a Rust que "¡Sé lo que estoy haciendo!", lo que a menudo es necesario cuando se escribe código de bajo nivel, proporcionando abstracciones seguras. No se puede escribir un núcleo sin `unsafe` s.

En ese sentido, un kernel no puede ser 100% seguro, sin embargo, las partes no seguras deben marcarse como un código `unsafe`, lo que mantiene las partes no seguras separadas del código seguro. Buscamos eliminar los `unsafe` s donde podemos, y cuando utilizamos `unsafe` s, somos extremadamente cuidadosos.

Un **“grep”** rápido nos da algunas estadísticas: el núcleo tiene unas 300 invocaciones `unsafe` en unas 16.000 líneas de código en general. Cada uno de estos es cuidadosamente auditado para garantizar su corrección.

Esto contrasta con los núcleos escritos en C, que no pueden garantizar la seguridad sin un análisis formal costoso.

Puedes obtener más información sobre cómo funciona `unsafe` **en la sección correspondiente del libro Rust**.

1.7 Proyectos paralelos

Redox es un sistema operativo completamente escrito en Rust. Además del núcleo, estamos desarrollando varios proyectos paralelos, que incluyen:

- **TFS**: filesystem inspirado en ZFS.
- **Ion**: el shell de Redox.
- **Orbital**: El servidor de visualización de Redox.
- **OrbTK**: un conjunto de herramientas (widget).
- **pkgutils**: la biblioteca de gestión de paquetes de Redox y su interfaz de línea de comandos.
- **Sodium**: un editor tipo Vi.
- **ralloc**: Un asignador de memoria.
- **libextra**: Suplemento para libstd, utilizado en toda la base de código Redox.
- **games-for-redox**: una colección de minijuegos para Redox (como los juegos en BSD).
- y algunos otros proyectos emocionantes que puedes explorar **aquí**.

También tenemos tres distribuciones de utilidades, que son colecciones de pequeños y útiles programas de línea de comandos:

- **Coreutils**: un conjunto mínimo de utilidades esenciales para un sistema usable.
- **Extrautils**: utilidades adicionales como recordatorios, calendarios, corrector ortográfico, etc.
- **Binutils**: Utilidades para trabajar con archivos binarios.

También contribuimos activamente a proyectos de terceros que se utilizan en Redox.

- **uutils/coreutils**: reescritura de Rust multiplataforma de GNU coreutils.
- **m-labs/smoltcp**: la pila de red utilizada por Redox.

¿Qué herramientas son adecuadas para la distribución Redox?

Algunas de estas herramientas en el futuro se moverán de la distribución predeterminada a paquetes opcionales separados. Ejemplos de estos son Orbital, OrbTK, Sodium, etc.

Las herramientas enumeradas se dividen en tres categorías:

- **Críticos**, que son necesarios para un sistema completamente funcional y utilizable.
- **Amigables con el ecosistema**, que están ahí para establecer la consistencia dentro del ecosistema.
- **Diversión**, que son "agradables" de tener y son inherentemente simples.

La primera categoría debería ser obvia: un sistema operativo sin ciertas herramientas básicas es un sistema operativo inútil. La segunda categoría contiene las herramientas que probablemente no sean predeterminadas en el futuro, pero que, sin embargo, están en la distribución oficial en este momento, por el encanto. La tercera categoría está ahí por conveniencia: es decir, para asegurarse de que la infraestructura Redox sea consistente e integrada (por ejemplo, pkgutils, OrbTK y libextra).

Es importante tener en cuenta que buscamos evitar herramientas que no sean codificadas en Rust, por seguridad y consistencia (**¿Por qué Rust?**).

2 Comenzando

En el punto de preparar **la compilación** hay información sobre cómo configurar tu sistema para compilar Redox, lo cual es necesario si deseas contribuir al desarrollo de Redox.

Si actualmente no estás interesado en pasar la molestia de construir Redox, puedes descargar la última versión. Consulta las instrucciones para ejecutarlo en una **máquina virtual** o en **hardware real**.

2.1 Probando Redox en una máquina virtual

La imagen ISO no es la forma preferida de ejecutar Redox en una máquina virtual. Actualmente, la imagen ISO carga toda la imagen del disco duro (incluido el espacio no utilizado) en la memoria. En el futuro, el disco "live" debería mejorar este aspecto para que eso no suceda.

En su lugar, debes utilizar la imagen del disco duro, que puedes encontrar **aquí (versión 0.7.0)** como un archivo .bin.gz. Descarga y extrae este archivo.

Luego puedes ejecutarlo en tu emulador preferido; este comando ejecutará qemu con varias características que Redox puede utilizar:

qemu-system-x86_64 -serial mon:stdio -d cpu_reset -d guest_errors -smp 4 -m 1024 -s -machine q35 -device ich9-intel-hda -device hda-duplex -net nic,model=e1000 -net user - device nec-usb-xhci,id=xhci -device usb-tablet,bus=xhci.0 -enable-kvm -cpu host -drive file=harddrive.bin,format=raw

Cambia **harddrive.bin** a la ruta del archivo **.bin** que acabas de extraer.

Una vez que el sistema se haya iniciado por completo, aparecerá la pantalla de inicio de sesión de RedoxOS. Para iniciar sesión, ingresa la siguiente información:
Usuario = root
contraseña = password

2.2 Ejecutar Redox en hardware real

Actualmente, Redox solo admite de forma nativa el arranque desde un disco duro sin tabla de particiones. Por lo tanto, la imagen ISO actual usa un gestor de arranque para cargar el filesystem en la memoria y emula uno. Esto es ineficiente y

requiere una gran cantidad de memoria, que se solucionará una vez que se implemente el soporte adecuado para varias cosas (como un controlador de almacenamiento masivo USB).

A pesar de su forma incómoda de funcionar, la imagen ISO es la forma recomendada de probar Redox en hardware real (en un emulador, un disco duro virtual es mejor). Puedes obtener una imagen ISO descargando la **última versión** o creando una con `make iso` de **Redox source tree**.

Puedes crear un CD de arranque o una unidad USB desde la ISO como con otras imágenes del disco de arranque.

El soporte de hardware es limitado en este momento, por lo que su alcance puede variar. No hay controlador USB HID, por lo que un teclado o mouse USB no funcionarán. Hay un controlador PS/2, que funciona con los teclados y paneles táctiles de muchas computadoras portátiles. Para redes, actualmente se admiten los controladores Ethernet rtl8168d y e1000d.

Redox actualmente no va a reemplazar tu sistema operativo existente, pero es algo divertido de probar; inicia Redox en tu computadora y observa qué y cómo funciona.

2.3 Preparando la construcción

¡Guau! Has llegado hasta aquí, recorrido todo el camino hasta aquí. ¡Felicitaciones! Ahora tenemos que construir Redox.

PRINCIPIANTES POR PRIMERA VEZ

Requisitos previos de Bootstrap y fuentes de búsqueda

Si estás en una computadora Linux o macOS, simplemente puedes ejecutar el script de arranque, que realiza la preparación de la compilación por ti. Ejecuta los siguientes comandos:

```
$ mkdir -p ~/tryredox
$ cd ~/tryredox
```

```
$ curl -sf
https://gitlab.redox-os.org/redox-os/redox/raw/master/bootstrap.sh -o
bootstrap.sh
$ bash -e bootstrap.sh
```

Lo anterior hace lo siguiente:

- Crea una carpeta principal llamada **"tryredox"**. Dentro de esa carpeta, creará otra carpeta llamada **"redox"** donde residirán todas las fuentes.
- Instala los paquetes de requisitos previos mediante el administrador de paquetes de tu sistema operativo (pop os/ubuntu/debian: apt, redhat/centos/fedora: dnf, archlinux: pacman).
- Clona el código Redox de GitLab y verifica una versión etiquetada del equipo redox de los diferentes subproyectos destinados a que la comunidad pruebe y envíe reportes de éxito/errores.

Ten paciencia, esto puede demorar desde 5 minutos a una hora, según el hardware y la red en la que lo estés ejecutando.

Ajustando el tamaño del filesystem

El tamaño del filesystem se especifica en megabytes. El valor predeterminado es 256 MB.

Probablemente desee un tamaño mayor, como 20GB (20480 MB).

Abre con tu editor de texto favorito (vim o emacs) **redox/mk/config.mk**

```
$ cd ~/tryredox/
$ gedit redox/mk/config.mk &
```

Busca **FILESYSTEM_SIZE** y cambia el valor en **MegaBytes**

```
FILESYSTEM_SIZE?=20480
```

Personaliza la configuración en config.mk antes de comenzar a construirlo

WIP esta sección necesita más trabajo.

Abre con tu editor de texto favorito (vim o emacs) **redox/mk/config.mk**

¿Cuáles son las configuraciones interesantes que los usuarios podrían querer cambiar?

Usuarios avanzados

Los usuarios avanzados pueden lograr lo mismo que el script **bootstrap.sh** anterior con los siguientes pasos.

Ten cuidado, la documentación no puede mantenerse al día con el script **bootstrap.sh** recuerda que hay muchas distribuciones desde las cuales construir Redox-OS: MacOS, PopOS, Arch Linux, Redhat/Centos/Fedora.

NOTA: Los principales desarrolladores de Redox usan PopOS para construir Redox-OS. Recomendamos utilizar PopOS para las compilaciones repetibles de Redox-os con “cero dolor”.

Clone el repositorio

Cambia a la carpeta donde deseas que se almacene tu copia de Redox y escribe el siguiente comando:

```
$ mkdir -p ~/tryredox
$ cd ~/tryredox
$ git clone https://gitlab.redox-os.org/ redox-os/redox.git --origin upstream --recursive
$ cd redox
$ git submodule update --recursive --init
```

Ten paciencia, esto puede demorar desde 5 minutos a una hora, según el hardware y la red en la que estés ejecutando el proceso

Instalar paquetes de requisitos previos

Pop OS Usuarios de Linux:

```
$ sudo apt-get install cmake make nasm qemu pkg-config libfuse-dev wget gperf libhtml-parser-perl autoconf flex autogen po4a expat openssl automake aclocal
```

Usuarios de ArchLinux:

```
$ sudo pacman -S cmake make nasm qemu pkg-config libfuse-dev wget gperf libhtml-parser-perl autoconf flex autogen po4a expat openssl automake aclocal
```

Usuários de Fedora/Redhat/Centos Linux:

```
$ sudo dnf install cmake make nasm qemu pkg-config libfuse-dev wget gperf libhtml- parser-perl autoconf flex autogen po4a expat openssl automake aclocal
$ sudo dnf install gettext-devel bison flex perl-HTML-Parser libtool perl-Pod-Xhtml gperf libpng-devel patch
$ sudo dnf install perl-Pod-Html
```

Usuarios de MacOS que usan MacPorts:

```
$ sudo port install make nasm qemu gcc7 pkg-config osxfuse x86_64-elf-gcc
```

Usuarios de MacOS que usan Homebrew:

```
$ brew install automake bison gettext libtool make nasm qemu gcc@7 pkg-config Caskroom/cask/osxfuse
$ brew install redox-os/ gcc_cross_compilers/x86_64-elf-gcc
```

Instalar Rust Stable And Nightly

Para instalar Rust, debes lograr que la versión nocturna sea tu toolchain predeterminado, la lista de toolchain instaladas:

```
$ curl https://sh.rustup.rs -sSf | sh
$ rustup default nightly
$ rustup toolchain list
$ cargo install --force --version 0.3.20 xargo
```

NOTA: xargo permite que redox-os tenga una libstd personalizada
NOTA: ~/.cargo/bin se ha agregado a su RUTA para la sesión en ejecución.

Agregue la siguiente línea a su archivo de inicio de shell, como ".bashrc" o ".bash_profile" para futuras sesiones de rust:

```
export PATH=${PATH}:~/.cargo/bin
```

Actualizando fuentes

Cómo actualizar submódulos usando make pull

```
cd ~/tryredox/redox
make pull
```

Cómo actualizar las fuentes del paquete usando make fetch

```
cd ~/tryredox/redox
make fetch
```

Cambiar las fuentes para compilar la imagen ARM Redox OS de 64 bits

WIP esta sección necesita más trabajo

```
cd ~/tryredox/redox
make distclean # important to remove x86_64 stuff
git remote update
git fetch
git restore mk/config.mk
git checkout aarch64-rebase
git pull
git submodule update --init --recursive
git rev-parse HEAD
git pull
```

Instalar herramientas adicionales para compilar y ejecutar la imagen ARM 64 –bit Redox OS

```
sudo apt-get install u-boot-tools
sudo apt-get install qemu-system-arm qemu-efi
```

Siguientes pasos

Una vez que todo esté configurado, finalmente podemos **compilar Redox**.

2.4 Construyendo/compilando todo el proyecto Redox

Ahora hemos logrado:

- obtener las fuentes
- ajustar la configuración a nuestro gusto
- posiblemente agregar nuestro propio paquete fuente/binario a Filesystem.toml

Estamos listos para construir la imagen completa del sistema operativo Redox.

Construir la imagen de Redox

Construyendo Redox-OS para x86_64:

```
$ cd ~/tryredox/redox/
$ time make all
```

Construyendo Redox-OS para aarch64/arm64:

```
$ cd ~/tryredox/redox/
time ./aarch64.sh
```

Espera un momento. Redox es bastante grande.

- "make all" obtiene las fuentes de las herramientas principales de los servidores fuentes de Redox-os y luego las compila.
- crea algunos archivos vacíos que contienen diferentes partes del sistema de archivos de la imagen final.

- Utilizando las herramientas principales recién creadas, crea los paquetes no principales en una de esas partes del filesystem
- llena las partes restantes del filesystem de manera adecuada con cosas creadas por las herramientas principales para ayudar a iniciar Redox.
- fusiona las diferentes partes del filesystem en una imagen final del disco duro del sistema operativo Redox lista para ejecutarse en Qemu.

Limpieza de ciclos de compilación anteriores

La limpieza destinada a reconstruir paquetes principales y todo el sistema.

Cuando necesites reconstruir paquetes centrales como relibc, gcc y herramientas relacionadas, limpia todo el ciclo de compilación anterior con:

```
cd ~/tryredox/redox/
rm -rf prefix/x86_64 -unknown-redox/relibc-install/
cookbook/recipes/gcc/{build,sysroot,stage*} build/filesystem.bin
```

o intenta lo siguiente:

```
cd ~/tryredox/redox/
touch initfs.toml
touch filesystem.toml
```

Limpieza prevista solamente para la reconstrucción de paquetes no centrales

Si solo estás reconstruyendo un paquete no central, limpia parcialmente el ciclo de compilación anterior lo suficiente como para forzar la reconstrucción del paquete no central:

```
cd ~/tryredox/redox/
rm build/filesystem.bin
```

o intenta:

```
cd ~/tryredox/redox/
touch filesystem.toml
```

Ejecutando Redox

Ejecutando el escritorio Redox

Para ejecutar Redox, haz lo siguiente:

```
$ make qemu
```

Esto debería abrir una ventana de Qemu, arrancando en Redox.

Si no funciona, intenta:
```
$ make qemu kvm=no # deshabilitamos KVM
```

o también:
```
$ make qemu iommu=no
```

Si esto tampoco funciona, debes abrir una incidencia.

Ejecución de la consola Redox únicamente

Deshabilitamos el escritorio de la GUI pasando "vga=no". Lo siguiente deshabilita el soporte de gráficos y le da la bienvenida con la consola Redox:
```
$ make qemu vga=no
```
Es ventajoso ejecutar en la consola para capturar la salida de las aplicaciones que no son GUI. Es de mucha ayuda depurar aplicaciones y compartir los registros capturados de la consola con otros desarrolladores de la comunidad de Redox.

Ejecución de la consola Redox con un Qemu Tap para pruebas de red

Expone Redox a otras computadoras dentro de una LAN. Configura Qemu con un "TAP" que permitirá que otras computadoras prueben las capacidades de Redox client/server/net.

Estos son los pasos para configurar Qemu Tap: WIP

Nota

Si encuentra errores, obstrucciones u otras cosas molestas, informa del problema al **Repositorio Redox**. ¡Gracias!

2.5 Probando Redox

Utiliza la tecla **F2** para acceder a un shell de inicio de sesión. El usuario `user` puede iniciar sesión sin contraseña. Para `root`, la contraseña es `password` por ahora. `help`

enumera los comandos incorporados para la shell (ion). `ls /bin` mostrará una lista de aplicaciones que puedes ejecutar.

Usa la tecla **F3** para cambiar a una interfaz gráfica de usuario (orbital). Inicia sesión con las mismas combinaciones de nombre de usuario/contraseña mostrado anteriormente.

Usa la tecla **F1** para volver a la salida del kernel.

Sodium

Sodium es el editor tipo Vi de Redox. Para probarlo:

1. Abre el terminal haciendo clic en el ícono en la barra de botones
2. Escribe `sudo pkg install sodium` para instalar **Sodium**. Necesitarás una conexión de red activa.
3. Escribe `sodium`. Esto debería abrir una ventana de edición separada.

Una breve lista de los valores predeterminados de **Sodium**:

- **`hjkl`: Navigation keys.**
- **`ia`: Go to insert/append mode.**
- **`;`: Go to command-line mode.**
- **`shift-space`: Go to normal mode.**

Para obtener una lista más extensa, escribe **`;help`**.

Configurando un reminder/countdown

Para demostrar el soporte de ANSI, jugaremos con recordatorios elegantes.
Abre el emulador del terminal. Ahora, escribe `rem -s 10 -b`. Esto establecerá una cuenta regresiva de 10 seg. con una barra de progreso.

Jugando con Rusthello

Rusthello es una IA reversa avanzada, creada por **HenryTheCat**. Es altamente concurrente, por lo que se demuestra las capacidades de subprocesos múltiples de

Redox. Admite varias IA, como fuerza bruta, minimax, optimizaciones locales e IA híbridas.
¡Vamos a intentarlo!

```
# instale rusthello escribiendo el comando
$ sudo pkg install games
# inícielo con el comando
$ rusthello
```

Luego se solicitarán varias cosas, como dificultad, configuración de la IA, etc. Cuando esto se realiza, Rushhello inicia de forma interactiva la batalla entre tu y una IA o una IA y otra IA.

Explorando OrbTK

Click en la aplicación demo de OrbTK, localizada en la barra de menú. Esto abrirá una interfaz gráfica del usuario que muestra los diferentes **widgets** que OrbTK admite actualmente.

2.6 Preguntas, comentarios, informes de problemas

Únete al **Redox-os Chat Server**. Es el mejor método para chatear con el equipo de Redox. ¡Pasarás el rato con otros fans de Redox! El servidor de chat Redox tiene muchos canales que se centran en diferentes aspectos del desarrollo y uso de redox. Envía un correo electrónico a **info@redox-os.org** y simplemente nos avisas que te gustaría unirte al chat.

Alternativamente, puedes utilizar our **Discourse Redox-os Forum**.

Si deseas informar problemas de Redox, dirigete a **Gitlab Redox Project Issues** y haz clic en el botón **New Issue**.

3 Explorando Redox

Este capítulo se dedicará a explorar todos los aspectos de un sistema Redox en funcionamiento, con horribles detalles.

Comenzaremos con el sistema de arranque, continuando con las utilidades de shell y línea de comandos, pasando a la GUI, mientras explicamos dónde suceden las cosas y cómo cambiarlas.

Redox está destinado a ser un **increíble y personalizable sistema**, que permite al usuario reducirlo a una distribución de línea de comandos muy pequeña, o construirlo en un entorno de escritorio completo con facilidad.

3.1 Boot Process

Bootloader

El primer código que se ejecutará es el sector de arranque en `bootloader/${ARCH}/bootsector.asm`. Esto carga el gestor de arranque desde la primera partición. En Redox, el cargador de arranque localiza el kernel y lo carga por completo en la dirección 0x100000. También inicializa el mapa de memoria y el modo de visualización VESA, ya que estos se basan en funciones del BIOS a las que no se puede acceder fácilmente una vez que el control se cambia al kernel.

Kernel

El kernel se ingresa a través de la tabla de interrupciones, con una interrupción 0xFF. Esta interrupción sólo está disponible en el gestor de arranque. Al utilizar este método, todas las entradas del núcleo pueden estar contenidas en una sola función, la función del núcleo, que se encuentra en `kernel/main.rs`, que sirve como punto de entrada en el archivo ejecutable `kernel.bin`.

En esta etapa, el kernel copia el mapa de memoria de la memoria baja, configura un mapeo de la página inicial, asigna el objeto de entorno, definido en `kernel/env/mod.rs`, y comienza a inicializar los controladores y esquemas que están integrados en el kernel. Este proceso imprimirá la siguiente información del kernel:

```
Redox 32 bits
  * text=101000:151000 rodata=151000:1A4000
  * data=1A4000:1A5000 bss=1A5000:1A6000
 + PS/2
   + Keyboard
      - Reset FA, AA
      - Set defaults FA
      - Enable streaming FA
   + PS/2 Mouse
      - Reset FA, AA
      - Set defaults FA
      - Enable streaming FA
 + IDE en 0, 0, 0, 0, C120, IRQ: 0
   + Primary on: C120, 1F0, 3F4, IRQ E
      + Master: Status: 58 Serial: QM00001 Firmware: 2.0.0 Model:
QEMUHARDDISK 48 bits LBA Size: 128 MB
      + Slave: Status: 0
   + Secondary on: C128, 170, 374, IRQ F
      + Master: Status: 41 Error: 2
      + Slave: Status: 0
```

Después de inicializar las estructuras, los controladores y los esquemas del kernel, el primer proceso del espacio de usuario generado por el kernel es el proceso `init`, más específicamente el proceso `initfs:/bin/init`.

Init

Redox tiene un proceso de inicio de varias etapas, diseñado para permitir la carga de controladores de disco de forma modular y configurable. Esto se conoce comúnmente como ramdisk de inicio.

Ramdisk Init

El ramdisk init tiene el trabajo de cargar los controladores necesarios para acceder al **root filesystem** y luego transferir el control al espacio **userspace init**. Este es un **filesystem** que está vinculado con el kernel y cargado por el gestor de arranque como parte de la imagen del kernel. Puedes ver el código asociado con el `init process` en `crates/init/main.rs`.

El ramdisk init carga, por defecto, el archivo `/etc/init.rc`, localizado en `initfs/etc/init.rc`. Este archivo actualmente tiene el contenido:

```
echo ##############################
echo ##   Redox OS is booting    ##
echo ##############################
echo

# Load the filesystem driver
initfs:/bin/redoxfsd disk:/0

# Start the filesystem init
cd file:/
init
```

Realmente es muy fácil modificar Redox para cargar un filesystem diferente como root, o para mover procesos y controladores dentro y fuera del ramdisk.

Filesystem init

Como se ve arriba, la inicialización del disco RAM tiene el trabajo de cargar e iniciar la inicialización del filesystem. De forma predeterminada, esto significa que se generará un nuevo proceso de inicio que carga un nuevo archivo de configuración, ahora en el root filesystem en `filesystem/etc/init.rc`. Este archivo actualmente tiene el siguiente contenido:

```
echo ############################
echo ## Redox OS has booted  ##
echo ## Press enter to login ##
echo ############################
echo

# Login process, handles debug console
login
```

La modificación de este archivo permite iniciar directamente la GUI. Por ejemplo, podríamos reemplazar el login de sesión con orbital.

Login

Después de que los procesos de inicio hayan configurado los controladores y los demonios, es posible que el usuario inicie la sesión en el sistema. Actualmente se usa un programa de inicio de sesión simple, su fuente se puede encontrar en `crates/login/main.rs`

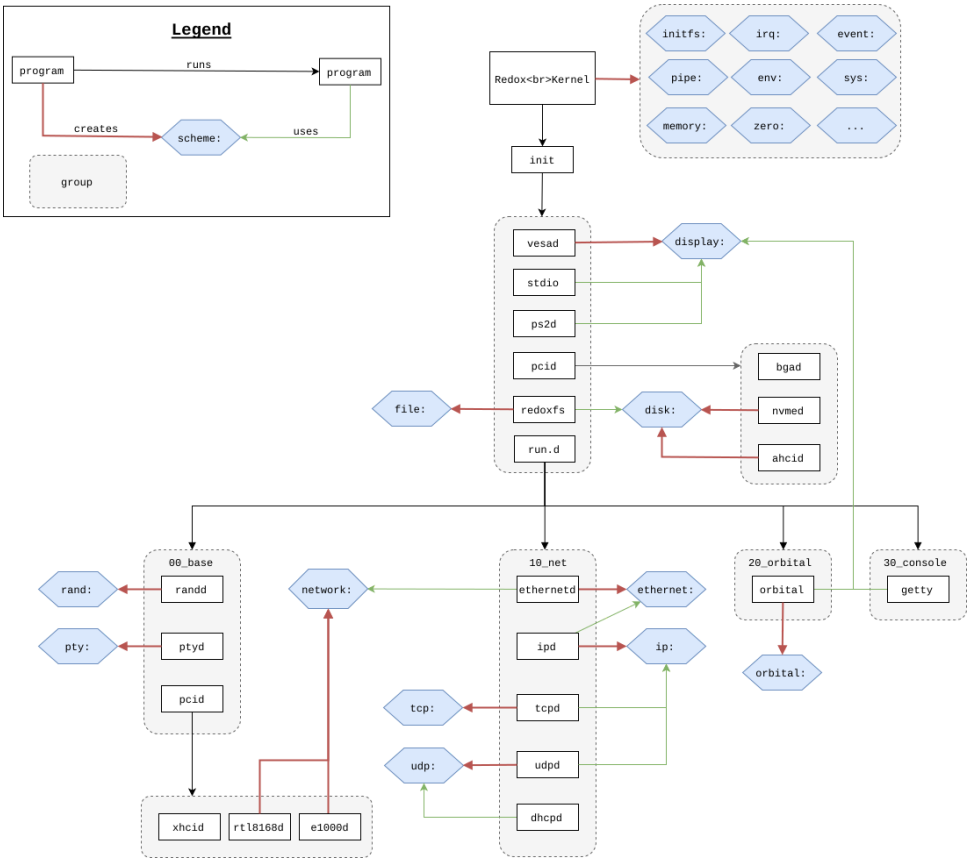
El programa de inicio de sesión acepta un nombre de usuario, actualmente se puede usar cualquier nombre de usuario, imprime el archivo `/etc/motd` y luego ejecuta `sh`. El archivo `motd` se puede configurar para imprimir cualquier mensaje, está en `filesystem/etc/motd` y actualmente tiene el contenido:

```
##########################
## Welcome to Redox OS  ##
## For GUI: Run orbital ##
##########################
```

En este punto, el usuario ahora podrá acceder al **Shell**

Vista general gráfica

Aquí hay una vista general del proceso de inicialización con la creación y el uso del esquema. En aras de la simplicidad, no representamos todas las interacciones del esquema, pero al menos las más importantes.



3.2 Shell

El shell utilizado en Redox es ion.

Cuando se llama ion sin **"-c"**, inicia un ciclo principal, que se puede encontrar dentro de `Shell.execute()`.

```
self.print_prompt();
while let Some(command) = readln() {
	let command = command.trim();
	if !command.is_empty() {
		self.on_command(command, &commands);
	}
	self.update_variables();
	self.print_prompt();
}
```

`self.print_prompt();` se utiliza para imprimir el indicador de shell.

La función `readln()` es el lector de entrada. El código está localizado en `crates/ion/src/input_editor`.

La documentación sobre `trim()` se puede encontrar **aquí**. Si el comando no está vacío, se llamará al método `on_command`. Luego, el shell actualizará las variables y volverá a imprimir el aviso.

```
fn on_command(&mut self, command_string: &str, commands:
&HashMap<&str, Command>) {
      self.history.add(command_string.to_string(), &self.variables);

      let mut pipelines = parse(command_string);

      // Executes commands
      for pipeline in pipelines.drain(..) {
      if self.flow_control.collecting_block {
            // TODO move this logic into "end" command
            if pipeline.jobs[0].command == "end" {
                  self. flow_control.collecting_block = false;
                  let block_jobs: Vec<Pipeline> = self.flow_control
                                                .current_block
                                                .pipelines
                                                .drain(..)
                                                .collect();
                  match self.flow_control.current_statement.clone() {
                  Statement::For(ref var, ref vals) => {
                        let variable = var.clone();
                        let values = vals.clone();
                        for value in values {
                              self.variables.set_var(&variable, &value);
                              for pipeline in &block_jobs {
                                    self.run_pipeline(&pipeline,
commands);
                              }
                        }
                  },
                  Statement::Function(ref name, ref args) => {
```

-
```
                        self.functions.insert(name.clone(),
    Function { name: name.clone(), pipelines: block_jobs.clone(),
    args: args.clon() });

                },
                _ => {}
                }
                self.flow_control.current_statement =
Statement::Default;
           } else {

self.flow_control.current_block.pipelines.push(pipeline);
           }
     } else {
           if self.flow_control.skipping() &&
!is_flow_control_command(&pipeline.jobs[0].command) {
                continue;
           }
           self.run_pipeline(&pipelines, commands);
     }
     }
}
```

Primero, `on_command` agrega los comandos al historial de shell con `self.history.add(command_string.to_string(), &self.variables);`.
Luego se analizará el script. El código del analizador está en `crates/ion/src/peg.rs`. El análisis devolverá un conjunto de pipelines, y cada pipeline contiene un conjunto de trabajos. Cada trabajo representa un solo comando con sus argumentos. Puedes dar un vistazo en `crates/ion/src/peg.rs`.

```
pub struct Pipeline {
     pub jobs: Vec<Job>,
     pub stdout: Option<Redirection>,
     pub stdin: Option<Redirection>,
}
pub struct Work {
     pub command: String,
     pub args: Vec<String>,
     pub background: bool ,
}
```

Qué sucede después:

- Si el bloque actual es un bloque de recopilación (un bucle for o una declaración de función) y el comando actual finaliza, cerramos el bloque:
    - Si el bloque es un bucle for, ejecutamos el bucle.
    - Si el bloque es una declaración de función, empujamos la función a la lista de funciones.
- Si el bloque actual es un bloque de recopilación pero el comando actual no finaliza, agregamos el comando actual al bloque.
- Si el bloque actual no es un bloque colector, simplemente ejecutamos el comando actual.

Los bloques de código se definen en `crates/ion/src/flow_control.rs`.

```
pub struct CodeBlock {
     pub pipelines: Vec<Pipeline>,
}
```

El código de la función se puede encontrar en `crates/ion/src/functions.rs`.
La ejecución del contenido del pipeline se ejecutará en run_pipeline().
La clase Command dentro de `crates/ion/src/main.rs` mapea cada comando con una descripción y un método para ser ejecutado. Por ejemplo:

```
commands.insert("cd",
                Command {
                nombre: "cd",
                help: "Change the current directory\n cd <path>",
                main: box |args: &[String], shell: &mut Shell| -> i32
{
                     shell.directory_stack.cd(args, &shell.variables)
                },
                });
```

`cd` se describe mediante `"Change the current directory\n cd <paths>",` y cuando se llama utiliza el método `shell.directory_stack.cd(args, &shell.variables)`. Puedes ver su código en **`crates/ion/src/directory_stack.rs`**.

3.3 GUI

El **entorno de escritorio en Redox**,es conocido como **Orbital**, es provisto por un conjunto de programas que se ejecutan en el espacio del usuario:

Programas

Las siguientes son utilidades de línea de comandos que brindan servicios de GUI

orbital

El administrador de pantalla y ventana orbital configura el orbital: scheme, administra la pantalla y maneja las solicitudes de creación de ventanas, redibujados y encuestas de eventos

launcher

El programa multipropósito lanzador que escanea las aplicaciones en el directorio `/apps/` y proporciona los siguientes servicios:

Llamado sin argumentos

Una barra de tareas que muestra íconos para cada aplicación

Llamado con argumentos

Un selector de aplicaciones que abre un archivo en un programa coincidente

- Si encuentra una aplicación que coincide, se abrirá automáticamente
- Si encuentra más de una aplicación, se mostrará un selector

Aplicaciones

Las siguientes son utilidades GUI que se pueden encontrar en el directorio `/apps/`.

Calculadora

Una calculadora que proporciona una funcionalidad similar al programa `calc`

Editor

Un editor simple que es similar a notepad

File browser

Un explorador de archivos que muestra iconos, nombres, tamaños y detalles de los archivos. Utiliza el comando launcher para abrir archivos cuando se hace click en ellos

Visor de imágenes

Un visor de imágenes simple

Pixelcannon

Un renderizador 3D que se puede utilizar para benchmarking del escritorio Orbital.

Sodium

Un editor similar a **vi** que proporciona resaltado de sintaxis

Terminal Emulator

Un emulador de terminal ANSI que inicia `sh` de forma predeterminada.

4 El diseño de Redox

Este capítulo repasa el diseño de Redox: el kernel, el espacio de usuario, el ecosistema, el balance y mucho más.

4.1 Componentes de Redox

Redox se compone de varios componentes discretos.

- **ion - shell**
- **TFS/RedoxFS - filesystem**
- **kernel**
- **drivers (controladores)**
- **orbital - DE/WM/Display Server**

Orbital subcomponents

- **orbterm - terminal**
- **orbdata - imágenes, fuentes, etc.**
- **orbaudio - audio**
- **orbutils - montón de aplicaciones**
- **orblogin - indicador de inicio de sesión**
- **orbtk - como gtk pero orb**
- **orbfont - biblioteca de renderizado de fuentes**
- **orbclient - cliente de visualización**
- **orbimage - biblioteca de renderizado de imágenes**

Aplicaciones principales

- **Sodium - editor de texto**
- **orbutils**
    - **background**
    - **navegador**
    - **calculadora**
    - **mapas de caracteres**
    - **editor**
    - **administrador de archivos**
    - **launcher**
    - **visor**

4.2 URL, esquemas y recursos

Esta es una de las opciones de diseño más importantes que realiza Redox. Estos tres conceptos esenciales están muy enredados.

¿Qué significa "Todo es una URL"?

"Todo es una URL" es una generalización de **"todo es un archivo"**, lo que permite un uso más amplio de esta interfaz unificada para esquemas.

Estos pueden usarse para modular efectivamente el sistema de una manera "sin parches".

El término es bastante engañoso, ya que una URL es solo el identificador de un esquema y un descriptor de recursos. Entonces, en ese sentido, **"todo es un esquema, identificado por una URL"** es más preciso, pero no muy pegadizo.

Entonces, ¿en qué se diferencia de los archivos?

Podemos pensar en las URL como filesystem virtuales segregados, que pueden estructurarse arbitrariamente (no tienen que ser como un árbol) y definirse arbitrariamente por un programa. Además, ¡Los "archivos" no tienen que comportarse como archivos! Más adelante volveremos sobre este tema.

Abre muchas posibilidades.

[... TODO]

La idea de filesystem virtuales no es nueva. Si estás en una computadora con Linux, debes intentar hacer un `cd` to `/proc` y ver qué sucede allí.

Redox extiende este concepto a uno mucho más poderoso.

TODO

4.2.1 URLs

URL (Uniform Resource Locator) en sí es una noción relativamente poco interesante (pero esencial) para el diseño de Redox. Lo importante es lo que representa.

La URL

En resumen, una URL es un identificador de un recurso. Contiene dos partes:

1. La parte del esquema. Esto representa el "receptor", es decir, qué esquema maneja la llamada (F)OPEN. Puede ser cualquier cadena UTF-8 arbitraria y, a menudo, será simplemente el nombre de su protocolo.

2. La pieza de referencia. Esto representa la "carga útil" de la URL, es decir, a qué se refiere la URL. Considera al `file`, como un ejemplo. Una URL que comienza con un `file`: simplemente tiene una referencia que es la ruta a un archivo. La referencia puede ser cualquier cadena de bytes arbitraria. El análisis, la interpretación y el almacenamiento de la referencia se dejan al esquema. Por esta razón, no se requiere que sea una estructura en forma de árbol.

Por lo tanto, la representación de cadena de una URL se ve así:
`[scheme]:[reference]`

Por ejemplo:
`file:/path/to/myfile`

NOTA **`//`** no es necesario, por conveniencia.

Abriendo una URL

Las URL pueden abrirse, generando esquemas, que pueden abrirse a recursos, que pueden leerse, escribirse y buscarse (hay operaciones que se describen más adelante).

Por razones de compatibilidad, usamos una API de archivo similar a la librería estándar de Rust para abrir URL:

```
use std::fs::OpenOptions;
use std::io::prelude::*;

fn main() {
      // Let's read from a TCP stream
      let tcp = OpenOptions::new()
                  .read(true) // readable
                  .write(true) //
                 writeable .open("tcp:0.0.0.0");
}
```

TODO: Tal vez hacer algo con el flujo tcp. ¿Ping pong?
TODO: La terminología puede resultar algo confusa para el lector.

4.2.1.1 ¿Cómo funcionan las URL bajo el capó?

Representación

Dado que es imposible pasar del espacio de usuario al anillo 0 de forma tipificada, tenemos que usar alguna representación débilmente tipada (es decir, no podemos usar una enumeración, a menos que queramos hacer transmutaciones y amigos). Por lo tanto, usamos una representación similar a un string cuando nos movemos al espacio del núcleo. Esto es básicamente sólo un puntero sin procesar a una cadena terminada en tipo C. Para evitar más sobrecarga, usamos representaciones más eficientes:

Url<'a>

La primera de las tres representaciones es la más simple. Consiste en una `struct` que contiene dos punteros grandes, que representan el esquema y la referencia respectivamente.

OwnedUrl

Esta es una `struct` que contiene `dos String S` (es decir, una cadena UTF-8 en crecimiento y asignada al montón), que son el esquema y la referencia, respectivamente.

CowUrl<'a>

Esta es una Copy-on-Write (CoW) URL, que, cuando muta, se clona en el montón. De esta manera, obtiene una asignación condicional y eficiente de la URL.

No hay mucha fantasía aquí.

Apertura de una URL

La apertura de una URL se realiza a través de `OPEN` de la llamada al sistema. `OPEN` toma una cadena tipo C, terminada en nulo, y dos enteros del tamaño de un puntero, manteniendo las banderas abiertas y el modo de archivo, respectivamente.

El argumento de ruta de `OPEN` no tiene que ser una URL. Por razones de compatibilidad, se establecerá de forma predeterminada en el `file:` scheme. Si se especifica lo contrario, el esquema será resuelto por el registrador (ver **root scheme**), y luego se abre.

TODO

4.2.2 Schemes

Los esquemas son la contraparte natural de las URL. Las direcciones URL se abren a los esquemas, que luego se pueden abrir para producir un recurso.

Los esquemas se nombran de tal manera que el kernel pueda identificarlos de manera única. Este nombre se utiliza en la parte del esquema de la URL.

Los esquemas son una generalización de los sistemas de archivos. Cabe señalar que los esquemas no representan necesariamente archivos normales; a menudo son un "archivo virtual" (es decir, una unidad abstracta con ciertas operaciones definidas en él).

A lo largo de todo el ecosistema de Redox, los esquemas se utilizan como la principal primitiva comunicación porque son una poderosa abstracción. Con los esquemas, Redox puede tener una interfaz de I/O unificada.

Los esquemas se pueden definir tanto en el espacio del usuario como en el espacio del núcleo, aunque de ser posible, se prefiere el espacio del usuario.

4.2.2.1 Esquemas del kernel

El kernel proporciona una pequeña cantidad de esquemas para soportar el espacio de usuario.

Nombre	Descripción	Links
`:`	**Root scheme - permite la creación de schemes en el user space**	**Docs**
`debug:`	**Proporciona acceso a la consola en serie**	**Docs**
`event:`	**Permite la lectura de Eventos que son registrados usando `fevent`**	**Docs**
`env:`	**Acceder y modificar variables de entorno**	**Docs**
`initfs:`	**Lectura solamente del filesystem utilizado para inicializar el sistema**	**Docs**
`irq:`	**Permite el manejo del espacio de usuario de IRQs**	**Docs**
`pipe:`	**Usado internamente por el kernel para implementar `pipe`**	**Docs**
`sys:`	**Información del sistema, como la lista de contexto y de esquema**	**Docs**

Userspace Schemes

El espacio de **usuario de Redox**, comenzando con **initfs:bin/ init**, creará esquemas durante la inicialización. Una vez que el usuario puede iniciar sesión, se debe establecer lo siguiente:

Nombre	Daemon	Description
`disk:`	`ahcid`	**Acceso sin formato a discos**
`display:`	`vesad`	**Multiplexación de pantalla, proporciona pantallas de texto y gráficas, utilizadas por orbital:**
`ethernet:`	`ethernetd`	**Envío/recepción del marco de Ethernet sin procesar, utilizada por ip:**
`file:`	`redoxfs`	**Root filesystem**
`ip:`	`ipd`	**Envío/recepción de paquetes IP sin formato**

network:	**e1000d rtl8168d**	**Envío/recepción del nivel de enlace de red, utilizado por ethernet:**
null:	**nulld**	**El Esquema que descarta todas las escrituras y no leerá bytes**
orbital:	**orbital**	**Sistema de ventanas**
pty:	**ptyd**	**Pseudo terminales, usados por emuladores de la terminal**
rand:	**randd**	**Generador de números pseudoaleatorios**
tcp:	**tcpd**	**Enchufes TCP**
udp:	**udpd**	**Enchufes UDP**
zero:	**zerod**	**El esquema que descarta todas las escrituras y siempre llenará los búferes de lectura con ceros**

Schemes Operations

¿Qué hace que un esquema sea un esquema? ¡Schemes Operations!

Un esquema es solo una estructura de datos con ciertas funciones definidas en él:

1. abierto - abre el esquema. abierto se utiliza para iniciar inicialmente la comunicación con un esquema; es un método opcional y devolverá ENOENT de manera predeterminada.
2. mkdir - crea una nueva subestructura. Tenga en cuenta que el nombre es un poco engañoso (e incluso podría cambiarse de nombre en el futuro), ya que en muchos esquemas mkdir no creará un directorio, sino que realizará algún tipo de creación de subestructura.

Los métodos opcionales incluyen:

1. desvincular: eliminar un enlace (es decir, un enlace de una subestructura a otra).
2. enlace - añadir un enlace.

El esquema root

Root scheme es el esquema del núcleo que se utiliza para registrar y recuperar información sobre los esquemas. El nombre de root scheme es simplemente una cadena vacía ("").

Registro de un esquema

El registro de un esquema se realiza abriendo el nombre del esquema con el indicador `CREATE`, en el root scheme.

TODO

4.2.3 Recursos

Los recursos son esquemas abiertos. Puedes pensar en ellos como una conexión establecida entre el proveedor del esquema y el cliente.

Los recursos están estrechamente relacionados con los esquemas y, a veces, se entrelazan con los esquemas. La diferencia entre esquemas y recursos es sutil pero importante.

Operaciones de recursos

Un recurso se puede definir como un tipo de dato con los siguientes métodos definidos en él:

1. **`read`** - leer N bytes en un búfer proporcionado como argumento. El valor predeterminado es EBADF
2. **`write`** - un búfer en el recurso. El valor predeterminado es `EBADF`.
3. **`seek`** - buscar el recurso. Es decir, mover el "cursor" sin escribir. Muchos recursos no admiten esta operación. El valor predeterminado es `EBADF`.
4. **`close`** - cierra el recurso, liberando potencialmente un bloqueo. El valor predeterminado es `EBADF`.

TODO añadir operaciones F.

El tipo de recurso

Hay dos tipos de recursos:

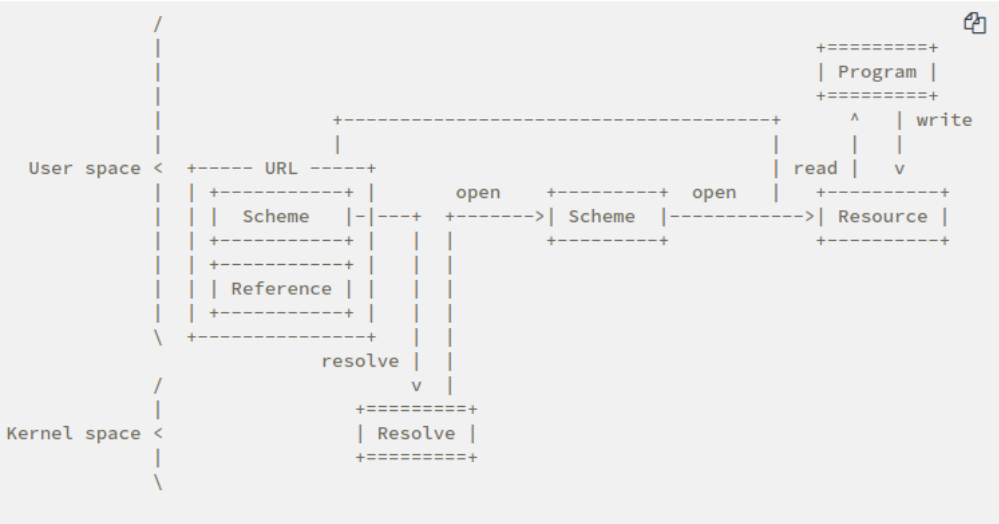
1. **Recursos similares a archivos**. Estos se comportan como archivos. Actúan como bloqueadores; las lecturas y escrituras son "similares a un búfer".
2. **Recursos tipo socket.** Estos se comportan como enchufes. No actúan de manera bloqueante; las lecturas y escrituras son más "similares a un flujo".

TODO Amplíe esto.

4.2.4 Cosiéndolo todo junto

El modelo "URL, esquema y recurso" es simplemente una interfaz unificada para una comunicación eficiente entre procesos. Las URL son simplemente descriptores de recursos. Los esquemas son simplemente "entradas" de recursos, que se pueden abrir. Puedes pensar en un esquema como un libro cerrado. No puede leerse ni escribirse en sí mismo, pero puedes abrirlo como un libro abierto: un recurso. Los recursos son simplemente elementos primitivos para las comunicaciones. Pueden comportarse como un socket (un flujo de bytes, por ejemplo, TCP y Orbital) o como un archivo (un buffer de bytes bajo demanda, por ejemplo, sistemas de archivos y stdin).

Un diagrama rápido y feo se vería así:



TODO

4.2.4.1 "Todo es una URL"

"Todo es una URL" es un principio importante en el diseño de Redox. En términos generales, significa que la API, el diseño y el ecosistema se centran en las URL, los esquemas y los recursos como la principal comunicación primitiva. Las aplicaciones se comunican entre sí, el sistema, los demonios, etc., mediante URL. Como tal, los programas no tienen que crear sus propias construcciones para la comunicación. Al unificar la API de esta manera, podemos tener una interfaz coherente, limpia y flexible.

Realmente no podemos reclamar créditos de este concepto (más allá de nuestro diseño e implementación exactos). La idea no es nueva y es muy similar a *9P* de *Plan 9* (Bell Labs); Se ha adoptado un enfoque similar en Unix y sus sucesores.

En qué se diferencia de "Todo es un archivo"

Con "todo es un archivo" se puede acceder a todo tipo de dispositivos, procesos y parámetros del kernel como archivos en un filesystem normal. Esto lleva a situaciones absurdas como que el disco duro que contiene el root filesystem `/` contiene una carpeta llamada `dev` con archivos de dispositivo que incluyen **sda** que contiene el root filesystem. Situaciones como ésta carecen de toda lógica.

Además, muchas propiedades de un archivo no tienen sentido en estos "archivos especiales" ¿Cuál es el tamaño de `/dev/null` o una opción de configuración en **sysfs**?

A diferencia de "todo es un archivo", Redox no impone un nodo de árbol común para todo tipo de recursos. En cambio, los recursos se distinguen por protocolo. De esta manera, los dispositivos USB no terminan en un "file system", sino en un esquema basado en protocolos como `EHCI`. Los archivos reales son accesibles a través de un esquema llamado `file`, que es ampliamente utilizado y especificado en **RFC 1630** y **RFC 1738**.

4.2.4.2 Un ejemplo.

¡Basta de teoría! Es tiempo de un buen ejemplo.

Implementaremos un esquema que contiene un vector. El esquema empujará elementos al vector cuando reciba escrituras y los abrirá cuando se lea. Llamémoslo **`vec:`**.

La fuente completa de este ejemplo se puede encontrar en **redox-os/vec_scheme_example**.

Configuración

Para compilar y ejecutar este ejemplo en un entorno Redox, deberá estar configurado para compilar el sistema operativo desde el origen. El proceso para incluir un programa en una compilación Redox local se describe en el **capítulo 5**. Haz una pausa aquí y sigue esta guía si deseas ejecutar este ejemplo.

Este ejemplo asume que `vec` se usó como el nombre de la caja en lugar de helloworld. Por lo tanto, la caja debe ubicarse en cookbook/recipes/vec/source.

Modifica `Cargo.toml` para la caja `vec` para que se vea así:

```
[package]
name = "vec"
version = "0.1.0"
edition = "2018"

[[bin]]
name = "vec_scheme"
path = "src/scheme.rs"

[[bin]]
name = "vec"
path = "src/client.rs"

[dependencies]
redox_syscall = "^0.2.6"
```

Considera que aquí hay dos binarios. Necesitaremos otro programa para interactuar con nuestro esquema, ya que las herramientas CLI como `cat` usan más operaciones de las que estrictamente necesitamos implementar para nuestro esquema. El cliente usa solo la biblioteca estándar.

El Scheme Daemon

Cree `src/scheme.rs` en la caja. Comienza con **`use`** en un par de símbolos.

```
use std::cmp::min;
use std::fs::File;
use std::io::{Read, Write};

use syscall::Package;
use syscall::scheme::SchemeMut;
use syscall::error::Result;
```

Comenzamos definiendo nuestra estructura de esquema mutable, que implementará el rasgo `SchemeMut` y mantendrá el estado del esquema.

```
struct VecScheme {
      vec: Vec<u8>,
}

impl VecScheme {
      fn new() -> VecScheme {
      VecScheme {
            vec: Vec::new(),
      }
      }
}
```

Antes de implementar las operaciones de esquema en nuestra estructura de esquema, analicemos brevemente la forma en que se utilizará esta estructura. Nuestro programa (`vec_scheme`) creará el esquema `vec` abriendo el controlador de esquema correspondiente en el esquema root (`:vec`). Implementemos un `main()` que inicialice nuestra estructura de esquema y registre el nuevo esquema:

```
fn main() {
	let mut scheme = VecScheme::new();

	let mut handler = File::create(":vec")
	.expect("Failed to create the vec scheme");
}
```

Cuando otros programas open/read/write/etc contra nuestro esquema, el kernel Redox hará que esas solicitudes estén disponibles para nuestro programa a través de este controlador de esquema. Nuestro esquema leerá esos datos, maneja las solicitudes y enviará respuestas al kernel escribiendo en el controlador del esquema. Luego, el núcleo pasará los resultados de las operaciones a la persona que llama.

```
fn main() {
	// ...

	let mut packet = Packet::default();

	loop {
	// Wait for the kernel to send us requests
	let read_bytes = handler.read(&mut packet)
		.expect("vec: failed to read event from vec scheme
handler");

	if read_bytes == 0 {
		// Exit cleanly
		break;
	}

	// Scheme::handle passes off the info from the packet to the
individual
	// scheme methods and writes back to it any information returned
by
	// those methods.
	scheme.handle(&mut packet);

	handler.write(&packet)
```

```
            .expect("vec: failed to write response to vec scheme
handler");
      }
}
```

Ahora tratemos las operaciones específicas en nuestro esquema. La llamada `Scheme.handle(...)` envía solicitudes a estos métodos, por lo que no tenemos que preocuparnos por los detalles de la estructura del paquete.

En la mayoría de los sistemas Unix (¡incluido Redox!), un programa necesita abrir un archivo antes de que pueda hacer algo con él. Dado que nuestro esquema es solo un "virtual filesystem", los programas llaman a `open` con la ruta al "archivo" con el que quieren interactuar cuando quieren iniciar una conversación con nuestro esquema.

Para nuestro esquema `vec`, empujemos cualquier ruta que nos lleve a `vec`:

```
impl SchemeMut for VecScheme {
      fn open(&mut self, path: &str, _flags: usize, _uid: u32, _gid:
u32) -> Result<usize> {
      self.vec.extend_from_slice(path.as_bytes());
      Ok(0)
      }
}
```

Digamos que un programa llama a `open("vec:/hello")`. Esa llamada funcionará a través del kernel y terminará siendo enviada a esta función a través de nuestra llamada `Scheme::handle`.

El tamaño que devolvemos aquí nos devolverá como el parámetro `id` de las otras operaciones del esquema. De esta manera podemos realizar un seguimiento de los diferentes archivos abiertos. En este caso, no haremos distinción entre dos programas diferentes que nos hablan y simplemente devolveremos cero.

De manera similar, cuando un proceso abre un archivo, el kernel devuelve un número (el descriptor del archivo) que el proceso puede usar para leer y escribir en

ese archivo. Ahora implementemos las operaciones de lectura y escritura para `VecScheme`.

```
impl SchemeMut for VecScheme {
	// ...

	// Fill up buf with the contents of self.vec, starting from
self.buf[0].
	// Note that this reverses the contents of the Vec.
	fn read(&mut self, _id: usize, buf: &mut [u8]) -> Result<usize> {
	let num_written = min(buf.len(), self.vec.len());

	for b in buf {
		if let Some(x) = self.vec.pop() {
			*b = x;
		} else {
			break;
		}
	}

	Ok(num_written)
	}

	// Simply push any bytes we are given to self.vec
	fn write(&mut self, _id: usize, buf: &[u8]) -> Result<usize> {
	for i in buf {
		self.vec.push(*i);
	}

	Ok(buf.len())
	}
}
```

Considera que cada uno de los métodos del rasgo `SchemeMut` proporciona una implementación predeterminada. Todos estos devolverán errores ya que esencialmente no están implementados. Hay un método más que debemos implementar para evitar errores a los usuarios de nuestro esquema:

```
impl SchemeMut for VecScheme {
      // ...

      fn close(&mut self, _id: usize) -> Result<usize> {
      Ok(0)
      }
}
```

Las librerías estándar de la mayoría de los lenguajes se cierran automáticamente cuando se destruye un objeto de archivo, y Rust no es una excepción.
Para ver todas las operaciones posibles en los esquemas, consulte los documentos **API**.

Un cliente simple

Como se mencionó anteriormente, necesitamos crear un cliente muy simple para usar nuestro esquema, ya que algunas herramientas de línea de comandos (como `cat`) usan operaciones distintas de abrir, leer, escribir y cerrar. Coloca este código

```
src/client.rs:
use std::fs::File;
use std::io::{Read, Write};

fn main() {
      let mut vec_file = File::open("vec:/hi")
      .expect("Failed to open vec file");

      vec_file.write(b" Hello")
      .expect("Failed to write to vec:");

      let mut read_into = String::new();
      vec_file.read_to_string(&mut read_into)
      .expect("Failed to read from vec:");

      println!("{}", read_into); // olleH ih/
}
```

Simplemente abrimos algún "archivo" en nuestro “scheme”, escribimos algunos bytes en él, leemos algunos bytes y luego lanzamos esos bytes en la salida estándar (stdout). Recuerda que no importa qué ruta usemos, ya que todo lo que hace nuestro “scheme” es agregar esa ruta al `vec`. En este sentido, el “scheme” `vec` implementa un vector global.

Ejecutar el esquema

Dado que ya configuramos el programa para compilar y ejecutar en la VM redox, simplemente ejecuta:

- **`touch filesystem.toml`**
- **`make qemu`**

Necesitaremos varias ventanas del terminal abiertas en la ventana de QEMU para este paso. Tenga en cuenta que los dos binarios que definimos en nuestro `Cargo.toml` ahora se pueden encontrar en el `file:/bin (vec_scheme y vec)`. En una ventana de terminal, ejecuta sudo `vec_scheme`. El programa necesita ejecutarse como root para poder registrar un nuevo esquema. En otro terminal, ejecuta `vec` y observa la salida.

Ejercicios para el lector

- **Logra que el esquema `vec` imprima algo cada vez que tenga eventos. Por ejemplo, imprima las identificaciones de usuario y grupo del usuario que intenta abrir un archivo en el esquema.**
- **Crea un `vec` único para cada archivo abierto en su esquema. Puede encontrar un mapa hash útil para esto.**
- **Escribe un esquema que pueda ejecutar código para su lenguaje de programación esotérico favorito.**

4.2 El kernel de Redox

El kernel Redox se deriva en gran medida del concepto de micronúcleos, con una inspiración particular en **MINIX**. En este capítulo analizaremos el diseño del núcleo Redox.

4.2.1 Microkernels

El kernel de Redox es un microkernel. Los micronúcleos se destacan en su diseño al proporcionar abstracciones mínimas en el espacio del núcleo. Los micronúcleos tienen un énfasis en el espacio del usuario, a diferencia de los núcleos monolíticos que tienen un énfasis en el espacio del núcleo.

La filosofía básica de los micronúcleos es que cualquier componente que pueda ejecutarse en el espacio del usuario debe ejecutarse en el espacio del usuario. El espacio del kernel solo debe utilizarse para los componentes más esenciales (ej., llamadas al sistema, separación de procesos, gestión de recursos, IPC, gestión de subprocesos, etc.).

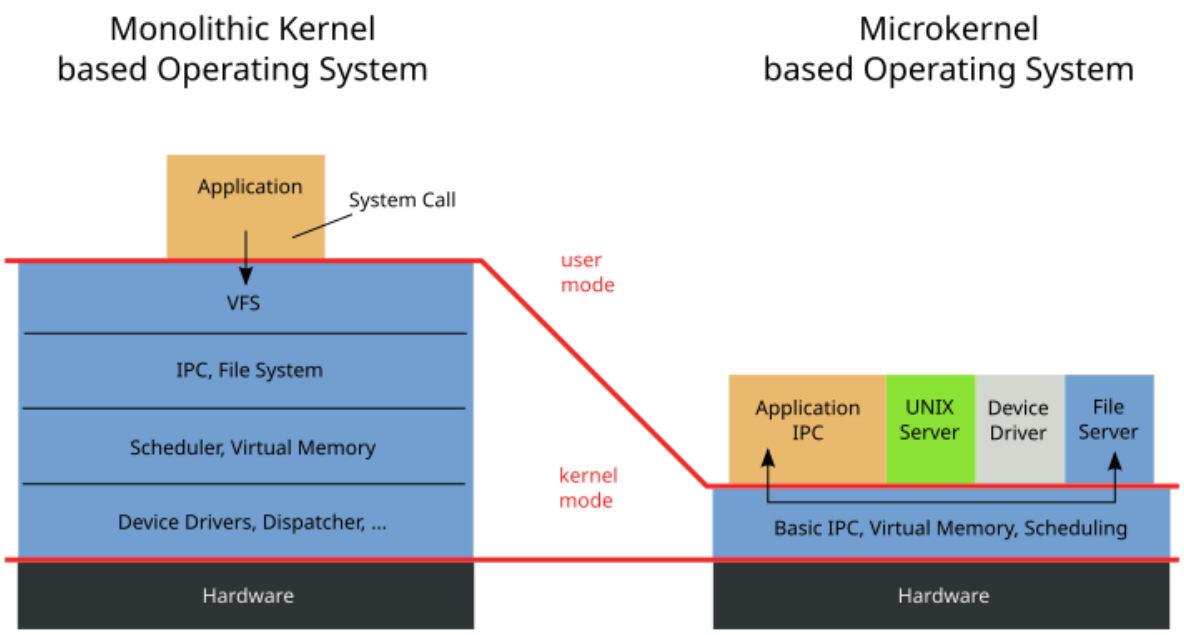
La tarea principal del núcleo es actuar como medio de comunicación y segregación de procesos. El kernel debe proporcionar una abstracción mínima sobre el hardware (es decir, los controladores pueden y deben ejecutarse en modo de usuario).

Los micronúcleos son más seguros y menos propensos a fallas que los núcleos monolíticos. Esto se debe a que los controladores y otras abstracciones tienen menos privilegios y, por lo tanto, no pueden dañar el sistema. Además, los micronúcleos son extremadamente fáciles de mantener, debido a su pequeño tamaño de código, esto puede reducir potencialmente la cantidad de errores en el núcleo.

Como cualquier otra cosa, los micronúcleos también tienen desventajas. Hablaremos de esto más adelante.

Microkernel VS núcleos monolíticos

Los núcleos monolíticos proporcionan muchas más abstracciones que los micronúcleos.



La ilustración anterior ([de Wikimedia], por Wooptoo, Licencia: Dominio público) muestra en qué se diferencian.

TODO

Una nota sobre el estado actual

Redox tiene menos de 9000 líneas de código en el núcleo. A modo de comparación, Minix tiene 6000 líneas de código en el kernel.

Nos gustaría mover partes de Redox al espacio del usuario para obtener un núcleo aún más pequeño.

4.3.1.1 Ventajas de los micronúcleos

Hay bastantes ventajas (¡y desventajas!) en los micronúcleos, algunas de las cuales se tratarán aquí.

Modularidad y capacidad de personalización

Los núcleos monolíticos son, al fin, monolíticos. No permiten un control tan detallado como los micronúcleos. Esto se debe a que muchos componentes

esenciales están "codificados" en el núcleo y, por lo tanto, requieren modificaciones en el propio núcleo (por ejemplo, controladores de dispositivos).

Los micronúcleos son muy modulares por naturaleza. Puedes reemplazar, recargar, modificar, cambiar y eliminar módulos, en tiempo de ejecución, sin siquiera tocar el kernel.

Los núcleos monolíticos modernos intentan resolver este problema utilizando módulos del núcleo, pero a menudo requieren que el sistema se reinicie.

Seguridad

Los microkernels son sin duda más seguros que los kernels monolíticos. El principio de minimalidad de los micronúcleos es una consecuencia directa del **principio de privilegio mínimo**, **según el cual todos los componentes deben tener solo los privilegios absolutamente necesarios para proporcionar la funcionalidad necesaria**.

Muchos errores críticos para la seguridad en los núcleos monolíticos se derivan de servicios y controladores que se ejecutan sin restricciones en el modo de núcleo, sin ningún tipo de protección.

En otras palabras: **en los núcleos monolíticos, los controladores pueden hacer cualquier cosa, sin restricciones, cuando se ejecutan en el anillo 0**.

Menos bloqueos

En comparación con los micronúcleos, los núcleos monolíticos tienden a ser propensos a “crashes”. ***Un controlador bloqueado en un kernel monolítico puede bloquear todo el sistema***, mientras en un microkernel hay una separación de los **"cuidados"**, lo cual, permite que el sistema maneje, de manera segura, cualquier bloqueo.

En Linux, a menudo vemos errores con los controladores que eliminan las referencias a los punteros incorrectos, lo que en última instancia resulta en **pánico del kernel**.

Hay muy buena documentación en MINIX sobre cómo un microkernel puede abordar esto.

4.3.1.2 Desventajas de los microkernels

Rendimiento

Cualquier sistema operativo moderno necesita mecanismos básicos de seguridad como la virtualización y la segmentación de la memoria. Además, cualquier proceso (incluido el kernel) tiene su propia pila y variables almacenadas en registros. En el **cambio de contexto**, es decir, cada vez que se invoca una llamada al sistema o se realiza cualquier otra comunicación entre procesos (IPC), se deben realizar algunas tareas, que incluyen:

- **Guardar los registros de llamadas, especialmente el contador del programa (persona que llama: proceso que invoca la llamada al sistema o IPC)**
- **Reprogramando el MMU (también conocida como TLB)**
- **Poner la CPU en otro modo (modo kernel, modo usuario)**
- **Restaurar registros de llamadas (llamadas: proceso invocado por syscall o IPC)**

Esto inherentemente no convierte en más lentos a los microkernels, pero los microkernels sufren por tener que realizar estas operaciones con más frecuencia. Gran parte de la funcionalidad del sistema la realizan **los procesos del espacio del usuario**, lo que requiere cambios **de contexto adicionales**.

La diferencia de rendimiento entre el kernel monolítico y los micronúcleos está siendo marginal con el tiempo, lo que hace que su rendimiento sea comparable. Esto se debe en parte a una superficie más pequeña que puede ser más fácil de optimizar.

Desafortunadamente en Redox aún no ha llegado. Todavía tenemos un núcleo relativamente lento ya que no se ha dedicado mucho tiempo a optimizarlo.

4.3.2 Planificación en Redox

El núcleo Redox utiliza un algoritmo de planificación llamado **Round Robin Scheduling**.

El kernel utiliza una función llamada **controlador de interrupciones** que la CPU llama periódicamente. Esta función realiza un seguimiento de cuántas veces se llama y planificará el próximo proceso listo para programar cada 10 **"ticks"**.

4.3.3 Administración de memoria

TODO

4.3.4 Controladores

TODO

4.4 Programas y librerías

Redox puede ejecutar programas. Algunos programas son interpretados por el **runtime** para el lenguaje de programación, como un script que se ejecuta en el shell Ion o un programa Python. Otros se compilan en instrucciones de máquina que se ejecutan en un sistema operativo particular (Redox) y hardware específico (por ejemplo, CPU compatible con x86-64 bits).

- **En los binarios compilados de Redox, utiliza el estándar ELF ("Executable and Linkable Format").**

Los programas podrían invocar directamente las llamadas al sistema Redox, pero la mayoría de las funciones de la librería de llamadas son de nivel superior y más cómodas de usar. Debes vincular tu programa con las librerías que necesitas.

- Redox aún no admite librerías de enlaces dinámicos **(problema n.º 927)**, por lo que las librerías que utilizan un programa están vinculadas estáticamente a su binario compilado.

- La mayoría de los programas C y C++ llaman funciones en un **librería estándar C** ("libc"), como **`fopen`**
- Redox, incluye un “port” (**portar un programa a otra plataforma**) de la librería C estándar **newlib**. Así es como se pueden ejecutar programas como git en Redox. **newlib** tiene cierta compatibilidad con **POSIX**
- Los programas de Rust llaman implícita o explícitamente a funciones en la librería estándar de Rust **(libstd)**.
- ?? ~~Redox implements a subset of this in libredox~~
- Rust **libstd** ahora incluye una implementación de sus partes dependientes del sistema (como el acceso a archivos y la configuración de variables de entorno) para Redox, en **`src/libstd/sys/redox`**. ?? La mayoría de libstd funciona en Redox, por lo que en Rust muchos programas de línea de comandos se pueden compilar para Redox.

El **proyecto del libro "Cookbook"** de Redox incluye recetas para compilar proyectos C y Rust en binarios Redox.

4.5 Coreutils

Coreutils es una colección de utilidades básicas de línea de comandos incluidas en Redox (como Linux, BSD, etc.). Esto incluye programas como **`ls`**, **`cp`**, **`cat`** y varias otras herramientas necesarias para la interacción básica de la línea de comandos.

Los coreutils de Redox pretenden ser más minimalistas que, por ejemplo, las coreutils de GNU incluidas en la mayoría de los sistemas Linux.

4.5.1 Suplemento de utilidades

4.5.1.1 Binutils

Binutils contiene utilidades para manipular archivos binarios. Actualmente, los binutils de Redox incluyen **`strings`, `disasm`, `hex` y `hexdump`**

4.5.1.2 Extrautils

Algunas herramientas de la línea de comandos adicionales se incluyen en extrautils cómo `less`, `grep` y `dmesg`.

5 Desarrollo en espacio de usuario

5.1 Incluir un programa en una compilación Redox

El cookbook de Redox hace que el empaquetado sea un programa bastante sencillo para incluirlo en una compilación. Este ejemplo explica cómo agregar el programa "hola mundo" que `cargo new` genera automáticamente a una compilación local del sistema operativo.

Este proceso es en gran medida el mismo para otras “crates” de Rust e incluso para programas que no están escritos en Rust.

Paso uno: configurando la receta

El cookbook solo creará programas que tengan una receta definida en `cookbook/recipes`. Para crear una receta para hello world, primero crea un directorio `cookbook/recipes/helloworld`. Dentro de este directorio, crea un archivo `recipe.toml` y escribe estas líneas:

```
[build]
template = "cargo"
```

La sección `[build]` define cómo el cookbook debe construir nuestro proyecto. Hay varias plantillas, pero se debe usar `"cargo"` para proyectos de Rust.

El Cookbook obtendrá las fuentes de un programa desde una URL de git o tarball especificada en la sección `[source]` de este archivo si `cookbook/recipes/program_name/source` no existe, y también obtendrás actualizaciones cuando ejecutes `make fetch`.

Para este ejemplo, no hay una URL ascendente para obtener las fuentes, por lo tanto, no hay una sección `[source]`. En cambio, simplemente lo desarrollamos en el directorio `source`.

Paso dos: escribir el programa

Dado que este es un ejemplo de "hello world", este paso es muy sencillo. Simplemente crea un libro de `cookbook/recipes/helloworld/source`. En ese directorio, ejecuta `cargo init --name="helloworld"`.

Para los proyectos de “cargo” que ya existen, incluye una URL al repositorio de git en el “recipe” y deja que el cookbook extraiga las fuentes automáticamente durante la primera compilación, o simplemente copia las fuentes en el directorio de `sources`.

Paso tres: agregue el programa a la compilación redox

Para poder acceder a un programa desde Redox, debe agregarse al filesystem. Abre `redox/filesystem.toml` y busca la tabla `[packages]`. Durante la (re)construcción del filesystem, el sistema de compilación usa el cookbook para empaquetar todas las aplicaciones en esta tabla y luego instala estos paquetes en el nuevo filesystem. Simplemente agrega `helloworld = {}` en cualquier lugar de esta tabla.

```
[packages]
#acid = {}
#binutils = {}
contain = {}
coreutils = {}
#dash = {}
extrautils = {}
#
# 100+ omitted for brevity
#
pkgutils = {}
ptyd = {}
randd = {}
redoxfs = {}
#rust = {}
smith = {}
#sodium = {}
```

```
userutils = {}
# Add this line:
helloworld = {}
```

Para reconstruir la imagen del filesystem y reflejar los cambios en el directorio de origen, es necesario ejecutar `touch filesystem.toml` antes de ejecutar `make`.

Paso cuatro: ejecutar tu programa

Ve a su directorio `redox/` y ejecuta `make all`. Una vez finalizada la reconstrucción, ejecuta `make qemu`, inicia sesión en Redox, abre la terminal y ejecuta `helloworld`. Debería imprimir:

```
Hello World!
```

Considera que el binario `helloworld` se puede encontrar en el `file:/bin` en la VM (`ls file:/bin`).

5.2 Ion Shell

Ion es la librería subyacente para shells y ejecución de comandos en Redox, así como el shell predeterminado.

¿Qué es Ion?

1. El shell predeterminado en Redox

¿Qué es shell?

Shell es una capa alrededor del kernel y las librerías del sistema operativo que permite a los usuarios interactuar con el sistema operativo. Eso significa que se puede usar shell en cualquier sistema operativo (Ion se ejecuta tanto en Linux como en Redox) o la implementación de una librería estándar siempre que la API proporcionada sea la misma. Las shells pueden ser gráficos (GUI) o de línea de comandos (CLI).

Shells de texto

Los shells de texto son programas que proporcionan una interfaz de usuario interactiva con un sistema operativo. Un shell lee de los usuarios a medida que escriben y realiza operaciones de acuerdo con la entrada. Esto es similar al ciclo de

read-eval-print (REPL) que se encuentra en muchos lenguajes de programación (por ejemplo, Python).

Típicos *nix shells

El shell más conocido es **Bash**, que se puede encontrar en la gran mayoría de las distribuciones de Linux, y también en macOS (anteriormente conocido como Mac OS X). Por otro lado, FreeBSD usa **tcsh** por defecto.

Hay muchas más implementaciones de shell, pero estas dos forman la base de dos conjuntos fundamentalmente diferentes:

- **Sintaxis de shell Bourne (bash, sh, zsh)**
- **Sintaxis de shell C (csh, tcsh)**

Por supuesto, estos dos grupos no son exhaustivos; cabe mencionar al menos la **shell** de **fish** y el **xonsh**. Estos shells están tratando de abandonar algunas características del shell de la vieja escuela para hacer que el lenguaje sea más seguro y consistente.

Funciones sofisticadas

Escribir comandos sin la ayuda del shell sería muy agotador e imposible de usar para el trabajo diario. Por lo tanto, la mayoría de los shells (¡incluido Ion, por supuesto!) Incluyen funciones sofisticadas como el historial de comandos, el autocompletado basado en el historial o las páginas de manual, accesos directos para acelerar la escritura, etc.

2. Un lenguaje de scripting

Ion también se puede utilizar para escribir scripts simples para tareas comunes o configuración del sistema después del inicio. No pretende ser un lenguaje de programación con todas las funciones, sino más bien un pegamento para conectar otros programas.

Relación con los terminales

Los **primeros terminales** eran dispositivos que se usaban para comunicarse con grandes sistemas informáticos como los **mainframes de IBM**. Hoy en día, los

sistemas operativos Unix suelen implementar los llamados terminales virtuales **(tty significa teletypewriter... ¡vaya!)** y emuladores de terminal **(ej., xterm, gnome-terminal)**.

Los **terminales** se utilizan para leer la entrada de un teclado y mostrar la salida textual del shell y otros programas que se ejecutan dentro de él. Esto significa que un terminal convierte las pulsaciones de teclas en códigos de control que luego utiliza el shell. El shell proporciona al usuario una línea de comando (por ejemplo: nombre de usuario y directorio de trabajo), capacidades de edición de línea **(Ctrl + a,e,u,k...)**, historial y la capacidad de ejecutar otros programas **(ls , uname, vim, etc.)** de acuerdo con la entrada del usuario.

TODO: En Linux tenemos archivos de dispositivo como `/dev/tty`, ¿cómo se maneja este concepto en Redox?

5.2.2 Manual de Ion

Ion tiene su propio manual, que puedes **encontrar aquí**.

6 Contribuyendo

¡Te gustaría contribuir a mejorar Redox! Excelente. Este capítulo puede ayudarte a contribuir con Redox.

6.1 Comunicación

6.1.1 Chat

La forma más rápida y abierta de comunicarse con el equipo de Redox es en nuestro servidor de chat **Mattermost**. Actualmente, la única forma de unirse es enviando un correo electrónico a **info@redox-os.org**, lo que puede demorar un poco, ya que no está automatizado. Actualmente estamos trabajando en una manera más fácil de hacer esto, pero esta es la forma vigente de momento.

6.1.2 Reddit

Puedes localizar Redox en Reddit en **/r/rust/** y **/r/redox/**. Las noticias de actualización semanal se publican en el primero.Contribuciones directas

6.2 Contribuciones directas

6.2.1 Fruta madura

También conocido como objetivos fáciles para novatos y triunfos rápidos para profesionales

Si no dominas Rust:

- **Escribir documentación**
- **Usar/probar Redox, archivar problemas de errores y funciones necesarias**
- **Desarrollo web (sitio web Redox, repositorio separado)**
- **Escritura de pruebas unitarias (puede requerir un conocimiento mínimo de Rust)**

Si dominas Rust, pero no el desarrollo de sistemas operativos:

- **Desarrollo de aplicaciones**
- **Desarrollo de shell (Ion)**
- **Desarrollo del administrador de paquetes (pkgutils)**
- **Otras tareas de código en alto nivel**

Si dominas Rust y tienes experiencia con OS Dev:

- **Familiarizarte con el repositorio y el código base**
- **Grep para TODO, FIXME, BUG, UNOPTIMIZED, REWRITEME, DOCME y PRETTYFYME y corrige el código que encuentres.**
- **Mejorar y optimizar el código, especialmente en el kernel.**

6.2.2 Discusiones de GitLab

Las discusiones de GitLab son un modo formal de comunicarse con otros desarrolladores de Redox, más lento y conveniente que el chat. Las discusiones son una buena manera de discutir temas específicos, pero si deseas una respuesta rápida, usa el chat que probablemente sea mejor.

Si aún no has solicitado unirte al chat, ¡deberías (si estás interesado en contribuir)!

6.2.2.1 Crear un informe adecuado sobre errores

1. **Verifica que el código con el que está viendo el problema esté actualizado con `upstream/master`. Esto ayuda a eliminar de los informes los errores que ya se han solucionado.**
2. **Confirma de que el problema sea reproducible (actívalo varias veces). Si el problema ocurre de manera inconsistente, puede valer la pena presentar un informe de error, pero informa aproximadamente con qué frecuencia ocurre el error**
3. **Registra la información de compilación como:**

- **La cadena de herramientas de Rust que usaste para construir Redox**
    - `rustc -V` y/o `rustup show` **desde la carpeta de tu proyecto Redox**
- **El hash de confirmación del código que usaste**
    - `git rev-parse HEAD`
- **El entorno en el que está ejecutando Redox (el "objetivo")**
    - `qemu-system-x86_64 -versión` **o sus especificaciones de hardware reales, si corresponde**
- **El sistema operativo que usaste para construir Redox**
    - `uname -a` **o un formato alternativo**

4. **Verifica que tu error no tenga un problema en GitLab. Si envías un duplicado, debes aceptar que puedes ser ridiculizado por ello, aunque aún así recibirás ayuda. Eres libre de preguntar en el chat de Redox si no estas seguro de que tu problema es nuevo**
5. **Abre un discusión en GitLab siguiendo la plantilla**

    - **Los problemas que no sean informes de errores pueden ignorar esta plantilla**
6. **Mira el problema y debes estar disponible para las preguntas.**
7. **¡Éxito!**

Con la ayuda de compañeros Redoxers, los errores legítimos pueden mantenerse a fuego lento, no hervir de golpe.

6.2.3 Pull Requests

Es muy bueno enviar un pequeño pull request sin primero presentar una discusión, pero si se trata de un gran cambio que requiere mucha planificación y revisión, es mejor que comiences escribiendo el problema primero. Consulta también las **directrices de git**.

6.2.3.1 Creando un pull request adecuado

Los pasos que se indican a continuación son para el proyecto principal de Redox: los submódulos y otros proyectos pueden variar, aunque la mayor parte del enfoque es el mismo.

1. **Clona el repositorio original en tu PC local usando uno de los siguientes comandos según el protocolo que estés usando:**
    - **HTTPS:** `git clone https://gitlab.redox-os.org/redox-os/redox.git --origin upstream --recursive`
    - **SSH:** `git clone git@gitlab.redox-os.org:redox-os/redox.git --origin upstream --recursive`
    - **Usa HTTPS si no sabes cuál usar. (Recomendado: aprende sobre SSH si no quieres iniciar una sesión cada vez que realices un PUSH/PULL).**
2. **Luego "rebase" para asegurarte de que estas usando los últimos cambios:** `git rebase upstream master`
3. **Bifurcar el repositorio**
4. **Agrega tu bifurcación a tu lista de controles remotos de git con**

    - **HTTPS:** `git remote add origin https://gitlab.redox-os.org/your-username/redox.git`
    - **SSH:** `git remote add origin git@gitlab.redox-os.org:your-username/redox.git`
5. **Alternativamente, si ya tienes una bifurcación y una copia del repositorio, simplemente puedes verificar para asegurarse de que está actualizado**
    - **Obtener el upstream:** `git fetch upstream master`
    - **"Rebase" con confirmaciones locales:** `git rebase upstream/master`
    - **Actualiza los submódulos:** `git submodule update --recursive --init`
6. **Opcionalmente, crea una rama separada (recomendado si estás realizando varios cambios simultáneamente) (**`git checkout -b my-branch`**)**
7. **Realiza los cambios**
8. **Commit (**`git add . --all; git commit -m "my commit"`**)**
9. **Opcionalmente, ejecuta rustfmt en los archivos que cambiaste y confirma nuevamente si hiciste algo (verifica primero con** `git diff`**)**
10. **Prueba tus cambios con** `make qemu` **o** `make virtualbox` **(es posible que debas usar** `make qemu kvm=no`**, anteriormente** `make qemu_no_kvm`**) (consulte las mejores prácticas y pautas)**
11. **Pull from upstream (**`git fetch upstream; git rebase upstream/master`**) (Nota: intenta no usar** `git pull`**, es equivalente a hacer** `git fetch upstream; git merge master upstream/master`**, que generalmente no es preferido para repositorios local/fork, aunque está bien en algunos casos).**
12. **Repite el paso 9 para asegurarte de que "rebase" todavía se compila y se inicia**
13. **Empuja a tu bifurcación (**`git push origin my-branch`**)**
14. **Crea una solicitud de PULL, siguiendo la plantilla**
15. **Describe tus cambios**
16. **¡Enviarlo!**

6.2.4 Comunidad

La importancia comunitaria es una parte importante del éxito de Redox. Si más personas saben acerca de Redox, probablemente participen más colaboradores, y todos pueden beneficiarse de su experiencia adicional. Puedes marcar la diferencia escribiendo artículos, hablando con otros entusiastas del sistema operativo o buscando comunidades que estén interesadas en saber más sobre Redox.

6.3 Mejores prácticas y pautas

Estas son un conjunto de las mejores prácticas para tener en cuenta al realizar una contribución a Redox. Como siempre, las reglas están hechas para romperse, pero estas reglas en particular juegan un papel en la decisión de fusionar tu contribución (o no). Así que intenta seguirlas.

6.3.1 Estilo Rust

Dado que Rust es un lenguaje relativamente pequeño y nuevo en comparación con otros como C, en realidad solo hay un estándar. Simplemente sigue los estándares oficiales de Rust para formatear, y tal vez ejecuta rustfmt en tus cambios, hasta que configuremos el sistema CI para que lo haga automáticamente.

6.3.2 Rusteando adecuadamente

Algunas pautas generales:

- **Utiliza** `std::mem::replace` y `std::mem::swap` **cuando puedas.**
- Usa **`.into() y .to_owned()`** en lugar de **`.to_string()`.**
- **Es preferible pasar referencias a los datos sobre datos propios. (No utilizar** `String`**,** `tome &str`**. tampoco** `Vec<T> tome &[T]`**).**
- **Utiliza genéricos, "traits" y otras abstracciones que proporciona Rust.**
- **Evita el uso de conversiones con pérdida (por ejemplo:** no hagas **`my_u32 as u16 == my_u16`**, utiliza **`my_u32 == my_u16 as u32`).**
- **Es preferible en el lugar (**`box keyword`**) al realizar asignaciones de almacenamiento dinámico.**
- **Prefiere enteros de tamaño independiente de plataforma sobre enteros de tamaño de puntero (**`u32` **sobre** `usize`**, por ejemplo).**

- **Sigue los modismos habituales de la programación, como "composición sobre herencia", "deja que tu programa se divida en partes más pequeñas" y "la adquisición de recursos es inicialización".**
- **Cuando** `unsafe` **es innecesario, no lo uses. ¡Un código seguro de 10 líneas más largo es mejor que un código inseguro más compacto!**
- **Verifica de marcar las partes que necesitan trabajo con** `TODO, FIXME, BUG, UNOPTIMIZED, REWRITEME, DOCME` **y** `PRETTYFYME`**.**
- **Usa los atributos de sugerencia del compilador, como** `#[inline]`**,** `#[cold]`**, etc. cuando tenga sentido hacerlo.**
- **Intenta desterrar** `unwrap()` **y** `expect()` **de tu código para gestionar los errores correctamente. El pánico debe indicar un error en el programa (no un error que no deseas administrar). Si no puede recuperarse de un error, imprime un buen error en stderr y sal. Consulta el libro de Rust sobre unwrapping.**

6.3.2.1 Evitando pánicos del kernel

Cuando intentes acceder a un segmento, utiliza siempre el rasgo `common::GetSlice` y el método `.get_slice()` para obtener un segmento sin que el kernel entre en pánico.

El problema con el corte en Rust normal, `ejem. foo[a..b]`, es que si alguien intenta acceder con un rango que está fuera de los límites de una **array/string/slice**, causará pánico del kernel en tiempo de ejecución, como medida de seguridad. Lo mismo al acceder a un elemento.

Utiliza siempre `foo.get(n)` en lugar de `foo[n]` e intenta cubrir la posibilidad de `Option::None`. Hacerlo de la manera habitual puede funcionar bien para las aplicaciones, pero nunca en el kernel. Nunca deberían existir posibles pánicos en el espacio del kernel, porque entonces todo el sistema operativo dejaría de funcionar.

6.3.3 Pautas de Git

- Los mensajes de confirmación deben describir sus cambios en tiempo presente, por ejemplo, **`"Add stuff to file.ext"`** en lugar de **`"added stuff to file.ext"`**.
- Intenta eliminar confirmaciones inútiles duplicate/merge de los PR, ya que saturan el historial y pueden dificultar la lectura.
- Por lo general, al sincronizar tu copia local con la rama maestra, querrás volver a hacer un "rebase" en lugar de un "merge" (fusionar). Esto se debe a que creará confirmaciones duplicadas que en realidad no hacen nada cuando se fusionan con la rama maestra.
- Cuando comiences a realizar cambios, querrás crear una rama separada y mantener la rama `master` de tu bifurcación idéntica al repositorio principal, de modo que pueda comparar tus cambios con la rama principal y probar una compilación más estable si lo deseas.
- Debes tener una bifurcación del repositorio en GitLab y una copia local en tu computadora. La copia local debe tener dos controles remotos; `upstream` y `origin`, `upstream` debe establecerse en el repositorio principal y el `origen` debe ser tu bifurcación.

FIN

Nota Final

Este manual en lengua castellana fue traducido -del original y oficial en Inglés ubicado en el website de ***Redox****- por la comunidad de* ***HackMadrid%27****. Especial gratitud a los siguientes miembros de la comunidad que participaron en su traducción y corrección. Colaboraron en esta tarea:* ***Eduardo Fórneas, José Luis Esteban Aparicio, Francisco Arencibia y Danmery.***