UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

DISEÑO AD HOC DE SISTEMAS OPERATIVOS MODULARES PARA ENTORNOS DE ALTO RENDIMIENTO

Autor: Jon Larrea Martínez
Tutor: Eloy Anguiano Rey

JUNIO 2019

Dedicado a mi madre…

"I do not think there is any thrill that can go through the human heart like that felt by the inventor as he sees some creation of the brain unfolding to success . . . Such emotions make a man forget food, sleep, friends, love, everything."

Nikola Tesla

Abstract

The world of the high-performance embedded operating system has been historically dominated by Linux/Unix systems. This project presents an alternative that offers higher performances to the operating systems traditionally used based in the implementation of a new architecture called *Operating System as Server* that results from the integration of some popular architectures with the attend petition idea offered by iterative servers.

This system has been designed completely from scratch allowing the implementation of low level optimizations in function and algorithms used frequently in higher layers decrementing the execution time of the task which the system has been designed for. Also, it has been organised in a particular way that prioritizes versatility and facilitates the modification of the system's functionality thanks to an efficient remote configuration system.

The objective of this project is to offer a viable alternative with an easy deploy that could replace Linux in systems that use generic boards such as Raspberry Pi.

Key words: Operating System, Embedded System, High Performance System, Raspberry Pi, ARM, Lightweight Network Stack

Resumen

El mundo de los sistemas operativos empotrados de alto rendimiento ha estado históricamente dominado por los sistemas Linux/Unix. Este proyecto presenta una alternativa que ofrece rendimientos superiores a los sistemas operativos utilizados tradicionalmente basándose en la utilización de una nueva arquitectura denominada *Sistema operativo como Servidor*, la cual resulta de integrar varias arquitecturas conocidas con la idea de atención de peticiones ofrecida por los servidores iterativos.

Este sistema operativo ha sido diseñado íntegramente desde cero, permitiendo realizar optimizaciones a muy bajo nivel en funciones y algoritmos muy utilizados en las capas superiores, permitiendo reducir el tiempo de ejecución de la tarea para la que ha sido diseñado el sistema. Además, se ha organizado de manera que prime la versatilidad y la facilidad para cambiar la funcionalidad del sistema gracias a un sistema de configuración remota muy eficiente.

El objetivo de este proyecto es ofrecer una alternativa viable y fácil de desplegar que pueda sustituir a Linux en sistemas que utilicen placas genéricas como es el caso de la Raspberry Pi.

Palabras clave: Sistema Operativo, Sistema Empotrado, Sistema de Alto Rendimiento, Raspberry Pi, ARM, Pila de Red Ligera

Agradecimientos

En primer lugar quiero agradecer a mi tutor, Eloy Anguiano Rey, la confianza depositada en mí desde el principio del proyecto y la libertad que me ha dado para realizar la implementación del sistema en base a las ideas que en su momento le planteé. Especial mención a las incontables veces que me he presentado en su despacho con dudas de implementación, problemas aparentemente sin solución y actualizaciones del estado del sistema y en las que siempre he sido atendido.

A todos los profesores de la escuela a los que he ido a preguntar todo tipo de dudas y que en ningún momento han dejado de ofrecerme soluciones.

A mis padres y mi hermana Leire por educarme en valores como esfuerzo y responsabilidad, por apoyarme en todo momento en todas las decisiones que he tomado y por haberme dado tantas oportunidades.

A mis compañeros de la carrera por los grandes momentos que hemos vivido dentro y fuera de la escuela, así como por haberme servido de apoyo en los momentos difíciles.

Para finalizar, dar las gracias a todas las personas que en algún momento dudaron de mí y de mi idea de crear un sistema operativo desde cero que fuera capaz de mejorar algo tan grande como Linux. Vosotros habéis sido la motivación más fuerte para llevar a cabo este proyecto.

Índice general

Índice de figuras

Capítulo 1

Introducción

jonOS (nombre que se le ha dado al sistema operativo implementado) es un sistema operativo para arquitecturas ARM diseñado para ser ejecutado en la placa Raspberry Pi 1 y que implementa la arquitectura presentada en este documento, la cual resulta de mezclar las características de los sistemas operativos monousuario, monotarea, empotrados, de tiempo real, implementados como librería y la estructura de un servidor iterativo con el objetivo de ofrecer una alternativa más rápida y utilizable a la hora de implementar un sistema dedicado de altas prestaciones.

Este trabajo surge de un proyecto personal que consistía en construir un sistema operativo completamente desde cero para sistemas con arquitectura x86-64 [1] (versión de 64 bits de los procesadores de la familia x86) que corría en máquina virtual. Pero no fue hasta que se le dio una vuelta de tuerca a la idea principal, centrándose en el nicho de los sistemas operativos para entornos empotrados y de alto rendimiento, que fue planteado al tutor como idea para trabajo de fin de grado. El diseño y construcción de un sistema operativo completamente desde cero siempre fue una de mis metas personales y es por eso que se han dedicado más horas de las 300 necesarias para un TFG, para diseñar y programar una versión funcional basada en la arquitectura propuesta en este trabajo.

La idea de desarrollar un nuevo sistema operativo no fue aceptada por el tutor hasta que no se definió un objetivo que supusiese crear algo con un diseño nuevo y alternativo a lo utilizado convencionalmente en sistemas empotrados (Linux).

El objetivo planteado era diseñar una nueva arquitectura de sistema operativo y realizar una implementación en un dispositivo muy utilizado en los entornos empotrados cuyo rendimiento fuese superior a los sistemas operativos tradicionalmente utilizados en ese tipo de placas (Raspberry Pi 1). La hipótesis planteada para lograr este objetivo fue que el utilizar un sistema operativo monotarea como forma de ahorrar el tiempo dedicado a otras tareas no necesarias, de cara a un sistema operativo empotrado (como sí ocurre con Linux al tratarse de un sistema operativo multitarea), para dedicar todo el tiempo de CPU a la tarea para la que ha sido diseñado el sistema. Además, se quería diseñar una arquitectura que permitiese versatilidad y, que en cualquier momento y de una forma sencilla, se pudiera cambiar la funcionalidad del sistema. Es por eso que se optó por realizar una estructura similar a la ofrecida por los sistemas operativos organizados como librerías mezclada con la idea de un servidor iterativo que atiende a clientes. De manera que el resultado de esto es una arquitectura de sistema operativo que dedica el 100 % de su tiempo de CPU a la tarea que le ha sido enviada por red y que utiliza recursos del sistema como la memoria dinámica, la pantalla, librerías estándar o el acceso a la red u otros dispositivos.

La decisión por mi parte de implementar el sistema operativo completo sin utilizar

ningún módulo o función externa desarrollada por terceros, me ha permitido poder optimizar una gran cantidad de funciones básicas de los sistemas operativos de cara a lograr más velocidad de ejecución. Este es el caso de la pila de red, que ha sido diseñada para ofrecer altas velocidades [2], [3] gracias a implementar una versión *naive* de los protocolos IP y UDP. Otro ejemplo de funciones presentes en otros sistemas que han sido rediseñadas para este sistema son las funciones relacionadas con la memoria dinámica, las cuales se han simplificado de manera que la reserva y asignación de memoria se haga más rápido que con el sistema de lista doblemente enlazada usado por Linux.

La estructura del documento comienza con el capítulo Estado del arte, el cual contiene un resumen general del estado actual de las principales arquitecturas de sistemas operativos junto con un pequeño análisis de los más utilizados en la placa Raspberry Pi Model B. A continuación, tenemos el capítulo de Hardware en el que se explican las características y limitaciones de los componentes que están presentes en la placa. Después encontramos uno de los capítulos principales del documento, el Sistema operativo. En este capítulo se describe a fondo la estructura, topología y funcionamiento del sistema operativo implementado así como una explicación detallada de los módulos, drivers, librerías estándar, herramientas, algoritmos y optimizaciones implementadas. Seguido a este encontramos el otro capítulo principal, el análisis de resultados, en el que se analizan los resultados obtenidos en base a una comparación con Raspbian. Finalmente encontramos el capítulo de Conclusión seguido del Trabajo futuro y las referencias.

Capítulo 2

Estado del arte

Un sistema operativo es el programa del sistema que controla todos los recursos del computador y ofrece soporte básico sobre el cual pueden escribirse los programas de aplicación [4]. También puede concebirse un sistema operativo como la capa de software encargada de aislar los programas de aplicación del hardware proporcionando a estos una interfaz para que puedan acceder los diferentes recursos (partes físicas, disco duro, pantalla, teclado, etc). Además, en general, el sistema operativo se ejecuta con un nivel de privilegios superior al de los programas de aplicación y de usuario.

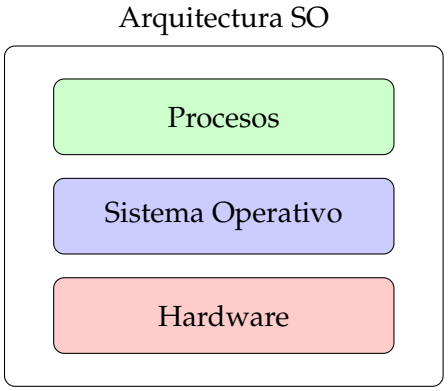

FIGURA 2.1: Situación del sistema operativo respecto al resto de componentes de un sistema informático

Como es de esperar, dependiendo del tipo de sistema sistema informático requerido para resolver un cierto problema, encontramos diferentes arquitecturas de sistemas operativos debido a que cada una de ellas está adaptada a resolver un conjunto de problemas concretos.

Los principales tipos [5] de arquitectura que encontramos son los siguientes:

- **Monotarea**: Arquitectura que solo permite ejecutar una tarea en cada momento, de manera que esta tiene que terminar para que otra pueda comenzar a ejecutarse.
- **Multitarea**: Arquitecturas que permiten ejecutar varias tareas de manera concurrente gracias a la asignación, por parte de un planificador, de cortos periodos de tiempo a cada una de ellas, dando la impresión al usuario de que se estén ejecutando simultáneamente.
- **Monousuario**: Arquitectura la cual no proporciona facilidades para distinguir entre usuarios, pero que no por ello tiene que restringirse a ser un sistema monotarea.

- **Multiusuario**: Arquitectura que, de manera similar a como ocurre en los sistemas multitarea, permite a través de la asignación de cortos periodos de tiempo ejecutar tareas de diferentes usuarios de manera que estos perciban que son los únicos usuarios utilizando el sistema.
- **Distribuido**: Arquitectura diseñada para controlar un grupo de ordenadores para que actúen como si de uno solo se tratase.
- **Empotrados**: Arquitectura diseñada para ser ejecutada en sistemas informáticos de ámbito específico, es decir, diseñados para realizar un número pequeño de funciones de manera eficaz.
- **Tiempo real**: Arquitecturas que garantizan el procesado de eventos y de información en un periodo de tiempo específico.
- **Librería**: Arquitectura que proporciona una interfaz a la capa de aplicación con las funcionalidades típicas que un sistema operativo ofrece, pero a través de un sistema similar al de una librería.

El sistema operativo desarrollado en este proyecto está inspirado en varias de las arquitecturas previamente mencionadas, definiendo una arquitectura mixta en la que se pretende reunir todas las características principales de dichas arquitecturas con el objetivo de crear un sistema empotrado para entornos de alto rendimiento (pudiendo considerarse sistema de tiempo real) y con una estructura similar a un sistema operativo como librería.

2.1. Sistema operativos empotrados

Un sistema operativo empotrado [6] se define como una combinación de hardware y software diseñada para mejorar la eficiencia de una función concreta. Al contrario que un ordenador personal, el cual está diseñado para resolver diversas tareas, un sistema empotrado está diseñado para realizar una única tarea concreta.

Los sistemas empotrados actuales suelen estar gobernados por un sistema operativo de tiempo real (RTOS ó Real-Time Operative System) el cual trabaja junto a un hardware dedicado a esa tarea específica, para garantizar un procesamiento de los datos en un tiempo determinado. Sin embargo, encontramos muchos otros entornos dedicados en los que, debido a las restricciones de presupuesto, no es posible utilizar hardware dedicado y es necesario utilizar un hardware más genérico, controlado por un sistema operativo más polivalente. Este es el caso de múltiples routers (domésticos y profesionales) los cuales utilizan una arquitectura similar a la de un ordenador convencional y están controlados por un sistema Linux al cual se le han eliminado diversos módulos innecesarios [7] en la tarea a desempeñar. La mayoría de routers domésticos proporcionados por las compañías de telecomunicaciones como por fabricantes, montan arquitecturas MIPS (Microprocessor without Interlocked Pipeline Stages) debido a que estas garantizan unos bajos consumos y buenas temperaturas en soluciones integradas. Estos sistemas suelen estar controlados por versiones reducidas de Linux las cuales cuentan con Busybox [8], un programa que agrupa múltiples utilidades de Unix que aumenta las funcionalidades del sistema.

2.2. Sistemas operativos en tiempo real

Un sistema operativo en tiempo real (Real-Time Operating System ó RTOS) [9] intenta servir en tiempo real aplicaciones que procesan los datos según son obtenidos sin esperas. Estos sistemas proporcionan soporte básico para planificación, gestión de recursos, sincronización, comunicación, precisión de tiempos y gestión de entrada/salida. Los sistemas operativos de tiempo real hacen énfasis en la predictividad de los resultados, eficiencia y en incluir características vinculadas a la medición de tiempo. El objetivo que se persigue con este tipo de sistemas es lograr ejecutar las subtareas en las que se divide una determinada tarea en unos tiempos definidos.

Existen múltiples tipos de sistemas operativos en tiempo real, desde sistemas especializados en una sola tarea hasta una gran variedad de sistemas operativos de un propósito más general (variantes de tiempo real de Linux).

Hoy en día, encontramos en el mercado muchos *kernels* propietarios, *kernels* compuestos y distribuciones modificadas de sistemas operativos populares como Linux o Windows-NT.

2.3. Sistemas operativos como librería

Los sistemas operativos organizados como librerías son aquellos que proveen los servicios típicos ofrecidos por un sistema operativo, como la red, de la misma forma en la que una librería (conjunto de implementaciones de funciones que ofrece una interfaz bien definida para la funcionalidad que se desea invocar) los proporcionaría a una aplicación.

Entre los sistemas operativos de este tipo más significativos encontramos dos: Unikernel [10] y Exokernel [11].

La idea tras esta arquitectura de sistema operativo es proporcionar a las aplicaciones una interfaz básica para el control y acceso al hardware, de manera que dicha interfaz no suponga una limitación en el rendimiento, flexibilidad y funcionalidad de las aplicaciones por culpa, por ejemplo, de un preestablecido sistema de intercomunicación de procesos o memoria virtual. En estos sistemas, el *kernel* confía toda la seguridad del sistema en las aplicaciones que hacen uso de la interfaz proporcionada.

2.4. Sistemas operativos de alto rendimiento

Se entiende por sistemas operativos de alto rendimiento aquellos destinados a funcionar en máquinas encargadas de realizar grandes cantidades de cálculos y tratar datos de manera masiva y eficiente. Este tipo de entornos están muy vinculados al ámbito académico así como al de la investigación y utilizan en su mayoría versiones de Linux [12] modificadas para que, junto a un hardware muchas veces específico, realicen de manera eficaz los cálculos y operaciones.

La lista TOP500 [13], la cual recopila cada 6 meses información sobre los 500 ordenadores más potentes del mundo, muestra cómo desde junio de 2017 el 100 % de los ordenadores de dicha lista utilizan un sistema operativo de la familia Linux.

2.5. Sistemas operativos de ámbito específico: Raspberry Pi

Existen múltiples alternativas de sistema operativo para ser ejecutadas en la placa Raspberry Pi 1. De manera oficial, Raspberry Pi Foundation (compañía creadora de la placa) únicamente da soporte a un sistema operativo llamado Raspbian aunque en su página web encontramos otras alternativas de diversos desarrolladores: Ubuntu Mate, Snappy Ubuntu Core, Windows IOT Core, Open Source Media Center (OSMC), LibreELEC, PiNet, RISC OS, Weather Station y IchigoJam RPi.

Además, encontramos muchos otros sistemas operativos de desarrolladores no tan conocidos que han cobrado importancia debido a su gran uso sobre todo en las áreas de los centros multimedia y de la emulación de videoconsolas (RetroPie).

De los anteriormente mencionados, el sistema operativo más utilizado para gestionar sistemas empotrados en esta plataforma es Raspbian, ya que está adaptado al hardware de la Raspberry Pi, por lo que al ser un software dedicado su nivel de eficiencia será mayor al del resto.

Raspbian es una distribución del conocido sistema operativo GNU/Linux basado en Debian originalmente enfocado en la enseñanza. Concretamente, Raspbian fue un adaptación de Debian armhf para el procesador montado en la placa Raspberry Pi (arquitectura ARM) con una optimización para la realización de operaciones en coma flotante mediante hardware. Este sistema operativo utiliza una versión ligera del entorno de escritorio X11 denominada LXDE (Lightweight X11 Desktop Environment).

Capítulo 3

Hardware

La plataforma seleccionada para este proyecto ha sido la Raspberry Pi 1 Model B, en parte, debido a la gran fama que han obtenido en los últimos años por ofrecer un hardware aceptable por un precio muy asequible (35 € en el momento de su lanzamiento en 2012), aunque a día de hoy con la versión 3 de esta placa en el mercado, los modelos antiguos se han devaluado considerablemente. Sin embargo, el motivo principal por el cual se decidió utilizar este hardware es porque únicamente posee un núcleo de ejecución y esto facilita la tarea de programación, al no necesitar tener en cuenta problemas de concurrencia, condiciones de carrera o la incorrecta escritura de datos entre otros.

La serie de placas Raspberry Pi son ordenadores de placa reducida o SBC (*Single Board Computer*) las cuales están impresas en un solo circuito de pequeñas dimensiones, como se muestra en la diagrama 3.1 cuyo diseño se centra en un solo microprocesador con un módulo de memoria RAM y los dispositivos de entrada salida de un ordenador funcional desarrollados por la compañía británica Raspberry Pi Foundation, con el objetivo de potenciar la enseñanza de informática en las escuelas.

Raspberry Pi es una plataforma de libre distribución aunque ciertos componentes hardware necesitan de código privativo para poder funcionar, como por ejemplo el *bootloader*. Sin embargo, cuentan con un sistema operativo libre específico para sus placas basado en Debian y denominado Raspbian.

3.1. Arquitectura y especificaciones

La versión utilizada para este proyecto ha sido la Raspberry Pi 1 Model B, la cual cuenta con un SoC (*system on a chip*) Broadcom BCM2835 que contiene el procesador ARM1176JZFS (perteneciente a la familia ARM11) con un núcleo con arquitectura ARMv6 y una unidad de procesamiento gráfico (GPU) Videocore 4, que utiliza la librería gráfica Open GL ES 2.0 y la aceleración por hardware OpenVG que proporcionan una resolución de 1080p30 y H.264 *high-profile encode/decode* a 250 MHz. El procesador es un RISC (*Reduced Instruction Set Computer*) de 32-bits con soporte de coma flotante a 700 MHz. En la parte superior del SoC se encuentra la memoria principal del sistema, una memoria SDRAM (*Synchronous Dynamic Random-Access Memory*) de 512 MiB.

Para el almacenamiento cuenta con un *slot* SD desde el cual cargará el sistema operativo.

En el aparatado de salida multimedia cuenta con una salida de vídeo compuesto y

una de HDMI para vídeo. En cuanto a audio cuenta con un jack de 3.5 mm y salida por HDMI.

La placa posee una entrada RJ45 con soporte 10/100 Ethernet, la cual adapta la velocidad en función de las posibilidades de la red pudiendo transmitir a una velocidad de 100 Mbps (Estándar *Fast Ethernet*). Además, cuenta con una gran cantidad de periféricos entre los cuales encontramos 2 puertos USB 2.0, 40 pines GPIO (*General Purpose Input/Output*), un bus SPI (*Serial Peripherical Interface*) y UART (*Universal asynchronous receiver/transmitter*). Todos estos componentes están conectados entre sí mediante buses I_2C e I_2S.

Uno de los apartados más importantes de la Raspberry Pi 1 Model B de cara al desarrollo del sistema operativo es que el SoC BCM2835 dispone de un puerto interno de USB por el cual se comunica con el concentrador LAN9512 al cual a su vez están conectados ambos puertos USB 2.0 y el puerto Ethernet. Dicho concentrador en determinadas situaciones puede suponer un cuello de botella a la hora de procesar los datos procedentes desde el controlador Ethernet o de los puertos USB.

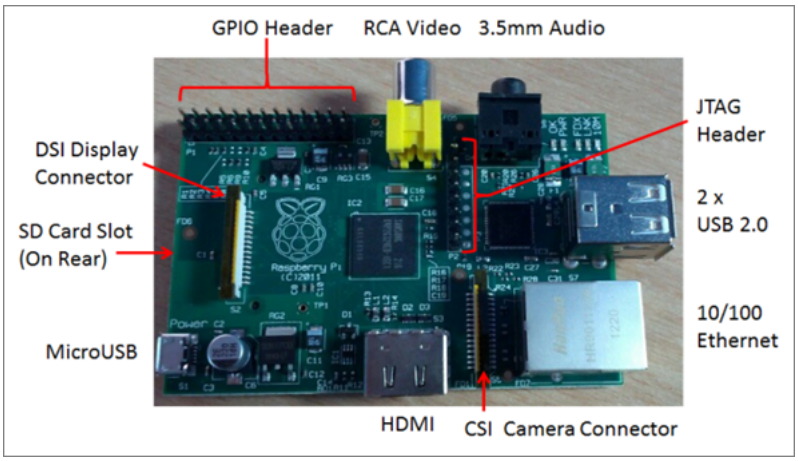

FIGURA 3.1: Disposición de los componentes principales de la placa Raspberry Pi Model B. Imagen obtenida de [14]

Capítulo 4

Sistema operativo

Como se ha mencionado previamente, el objetivo principal que se tenía en el desarrollo de este sistema era lograr, a través del concepto de monotarea, crear un sistema operativo que permitiese diseñar entornos dedicados de altas prestaciones con relativa facilidad, que fuesen capaces de ejecutar tareas con unos tiempos inferiores a los obtenidos con los sistemas operativos utilizados para este fin actualmente (Raspbian).

Se entiende como sistema operativo monotarea aquel que únicamente puede ejecutar un proceso a la vez. Esta característica propia de los primeros sistemas operativos como MS-DOS supone que el tiempo de CPU dedicado a dicho proceso será del 100 % y en caso de querer ejecutar un segundo proceso, este tendrá que esperar a que el primero termine para poder pasar a ser ejecutado. Teniendo en cuenta que la principal característica de un sistema dedicado/empotrado es que son sistemas especializados en una tarea, se pensó en combinar ambos conceptos para crear un sistema operativo monotarea cuya tarea a ejecutar sea determinada por un usuario remoto.

Como el objetivo final es un sistema operativo que ofrezca unos altos rendimientos en entornos dedicados, se ha decidido implementar las funcionalidades básicas de un sistema operativo (pantalla, memoria dinámica, librería estándar) intentando minimizar el tiempo de ejecución.

4.1. Estructura

La estructura que se ha seguido para el desarrollo de *jonOS* es la de un servidor tradicional [15]. El sistema operativo estará escuchando en un puerto determinado a que el usuario envíe la función que va a ser ejecutada previamente compilada en la máquina del usuario para la arquitectura de la Raspberry Pi.

El sistema operativo, en el arranque, inicializa todos los módulos y estructuras necesarias para que el proceso de carga remota por red se inicie y pueda recibir las funciones previamente compiladas denominadas *payloads*. Una vez el sistema ha acabado de ejecutar la tarea que ha recibido, al igual que haría un servidor iterativo, vuelve a bloquearse a la espera de recibir una nueva función que ejecutar.

Con esta arquitectura basada en la idea de un servidor, y sin la necesidad de definir lo que es un proceso, el sistema operativo no necesita un planificador que gestione el orden de ejecución. Esto permite ganar tiempo al evitarse tener que detener ejecuciones para dar paso a otras más prioritarias. En esta idea y en la de sistema operativo como servidor

son en las que se sustenta la hipótesis principal que se ha seguido para realizar un sistema que fuera capaz de ejecutar tareas en un tiempo significativamente inferior que los sistemas operativos más usados para realizar sistemas empotrados en estas plataformas (Raspbian).

Una característica destacable debida a esta estructura es la no existencia de roles de ejecución de manera que todo el código ejecutado en el sistema, ya sea *kernel* o *payload* de usuario, tienen permisos completos para acceder a cualquier zona de la memoria.

4.1.1. Topología del sistema

El sistema operativo por sí solo no realiza ninguna función, es por eso que necesita recibir la tarea a ejecutar desde otro ordenador denominado Cliente. Es el cliente el encargado de programar y de compilar la función para después enviarla a través de la red y que esta sea ejecutada por la Raspberry Pi. Una posible topología del sistema desplegado podría ser la mostrada en la figura 4.1, la cual muestra a la Raspberry Pi y a un cliente interconectados por Internet.

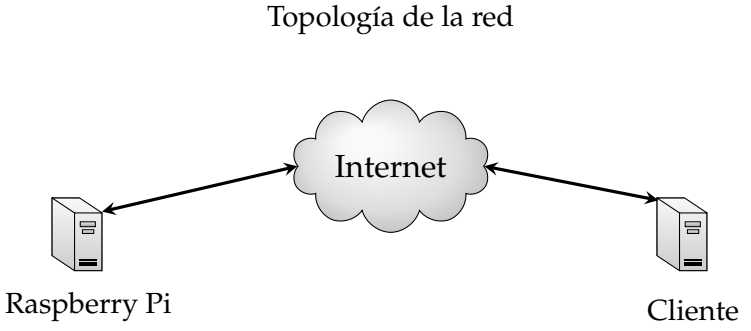

FIGURA 4.1: Descripción de la topología de un despliegue de jonOS en una Rasperry Pi conectada a través de internet con un cliente.

Esta topología de red no se limita únicamente a un solo cliente, sino que existe la posibilidad de que varios clientes deseen ejecutar diferentes tareas en el sistema, en cuyo caso estas serán ejecutadas en base al orden de solicitud.

4.2. Arranque y organización en memoria

Una vez la Raspberry Pi es conectada a la corriente, comienza el proceso de arranque o *boot process* [16]. Este proceso inicializa los componentes hardware y en última instancia el *kernel* del sistema operativo. El proceso de arranque de la Raspberry Pi no está muy documentado debido a que Broadcom (fabricante del SoC) únicamente ha publicado los binarios necesarios con todo el proceso de arranque y de inicialización de dispositivos.

El proceso de arranque comienza encendiendo la GPU manteniendo la CPU apagada. Es en este punto donde la GPU ejecuta el *Stage 1* que se encuentra en una ROM de la propia placa. Es el *Stage 1* el encargado de inicializar el módulo SD y de acceder a él para leer el *Stage 2* alojado en el fichero bootcode.bin y almacenándolo en la caché de nivel 2 (L2). El *Stage 2* activa la SDRAM y carga en memoria el fichero start.elf, ya que en este *Stage 2* se tiene conocimiento del formato ELF (*Executable and Linkable Format*).

Es el fichero start.elf el encargado de inicializar la CPU, cargar la imagen del *kernel* del fichero kernel.img en memoria con sus parámetros correspondientes determinados por los ficheros de configuración de la SD y de dar el control de la ejecución del *kernel* a la CPU. Una vez el *kernel* es cargado, este se coloca en memoria como se indica en el fichero linker.ld, el cual es utilizado en el proceso de enlazado para organizar las diferentes secciones que tendrá el ejecutable. Este proceso se puede expresar en forma de diagrama de estados, como se muestra en la figura 4.2.

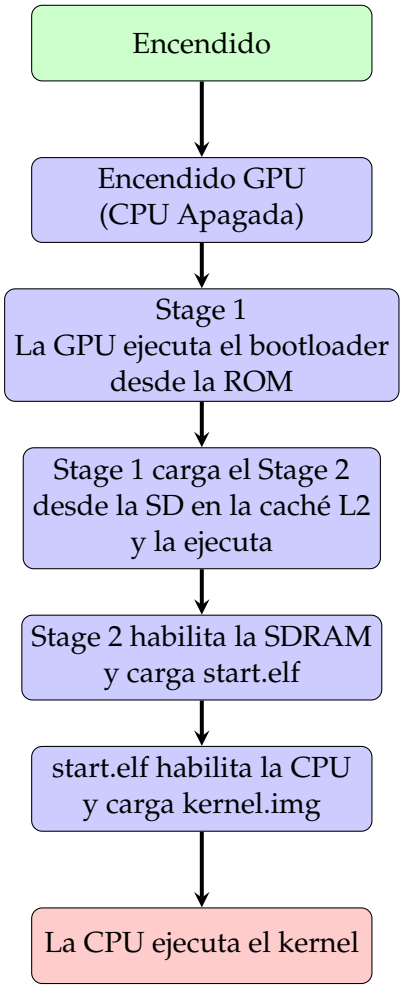

FIGURA 4.2: Descripción del proceso de arranque de la Raspberry Pi dividido en 2 Stages ejecutados en GPU para finalmente ejecutar el *kernel* en CPU.

El fichero linker.ld estructura el *kernel* en memoria de la siguiente manera en orden ascendente en memoria [17]:

- **Sección STACK**: sección encargada de almacenar los parámetros pasados a las funciones y las variables estáticas declaradas en cada función. Esta sección comienza en la dirección 0x8000 y crece hacia posiciones inferiores en memoria.
- **Sección .TEXT**: sección que contendrá el código y las funciones implementadas en el *kernel*. Esta sección se sitúa en la posición 0x8000 de memoria, que corresponde con la dirección que toma el registro PC (*Program Counter*) al pasar a ejecutar el *kernel*. Esta sección se encuentra a su vez formada por dos partes, la primera corresponde con el fichero boot.s el cual inicializa el puntero de pila también en la

posición 0x8000, limpia la sección .BSS y finalmente da paso a ejecutar la rutina principal del *kernel* (*main*).

- **Sección .RODATA**: contiene las variables estáticas de solo lectura.
- **Sección .DATA**: encargada de almacenar las variables utilizadas.
- **Sección .BSS**: contiene todas las variables que no han sido explícitamente inicializadas en el código y que por defecto su valor es cero.
- **Sección HEAP o montículo**: almacena la memoria que ha sido reservada dinámicamente por el sistema y que tiene un tamaño máximo de 4 MB.
- **Sección ADDRESS_SOLVER**: esta sección es la encargada de almacenar las estructuras necesarias para poder resolver las direcciones de las funciones en tiempo de ejecución por parte de los *payloads* del usuario.

Todas las secciones intermedias están alineadas a 4096 Bytes que corresponde con el tamaño de página de la Raspberry Pi quedando finalmente la memoria RAM con el aspecto mostrado en la figura 4.3.

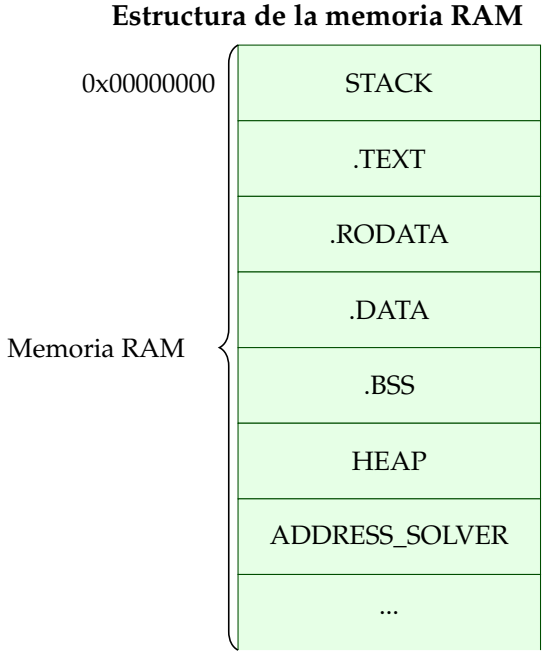

FIGURA 4.3: Representación de la memoria RAM cuando el *kernel* es cargado.

4.3. Funcionalidad

La idea principal en la que se basa el funcionamiento del sistema operativo desarrollado es la de un servidor iterativo que está a la espera de una solicitud de un usuario, ejecuta la tarea del usuario y vuelve a la posición de espera de una nueva tarea.

Para ello una vez la CPU pasa a ejecutar la rutina principal (*main*) del *kernel* tras ejecutar la rutina *boot* del mismo y se inicializan todos los drivers de dispositivos que el

usuario puede utilizar en sus *payloads* en un orden determinado debido a las interdependencias que existen entre ellos.

En primer lugar se configura la UART del sistema que será utilizada por el *kernel* para mostrar información interna del sistema a modo *debug*. A continuación se instancia el módulo ADDRESS_SOLVER, necesario para que los *payloads* enviados por el usuario puedan resolver en tiempo de ejecución la dirección de las funciones que utiliza. Tras esto, se inicializa el driver de red, el cual es requerido por el *kernel* para recibir los *payloads* del usuario. Finalmente, se configura la GPU para trabajar con la pantalla a la que esté conectada la Raspberry Pi.

4.3.1. Módulos

Se entiende por módulo o driver, la parte del software que proporciona las funciones de alto nivel para interactuar con los diferentes dispositivos conectados al sistema, abstrayendo al usuario del funcionamiento a bajo nivel.

Los módulos principales que están disponibles en este sistema operativo son la UART, el driver de red con la pila Ethernet/IP/UDP para enviar y recibir mensajes desde otras máquinas y el módulo de pantalla que permite escribir caracteres en la pantalla a través de la salida HDMI de la Raspberry Pi.

UART

El UART (*Universal Asynchronous Receiver-Transmitter*) [16] es el dispositivo que controla los puertos y dispositivos que trabajan con comunicaciones serie. Este toma los bytes de los datos que se quieren transmitir y envía los bits de estos de manera secuencial a una velocidad determinada, al igual que cuando recibe datos, los recibe bit a bit a dicha velocidad.

El sistema configura la UART en los puertos GPIO 14 para el envío de datos y el GPIO 15 para la recepción de datos a una velocidad de 115200 baudios (los baudios representan el número de símbolos por segundo en un medio de transmisión digital). Este driver ofrece funciones de alto nivel para enviar una cadena de caracteres a través del UART, así como enviar y recibir caracteres de manera individual.

Pila de red

En la Raspberry Pi, como se menciona en el capítulo dedicado al hardware, el puerto Ethernet está montado sobre un concentrador USB LAN9512 por lo que para acceder a él es necesario hacerlo a través de dicho concentrador, pudiendo provocar cuellos de botella que deriven en pérdidas de rendimiento.

Debido a la dificultad que suponía implementar dicho driver, la escasa documentación y sobre todo la falta de tiempo, se ha decidido utilizar dos funciones de un driver de código abierto llamado USPi [18] desarrollado por Rene Stange. De dicho driver únicamente se han utilizado dos funciones, las encargadas de escribir y leer datos del *buffer* del puerto Ethernet. Por encima de las funciones que dan acceso al *buffer* Ethernet se ha construido una pila de protocolos Ethernet/IP/UDP bidireccional ligera de alto rendimiento, como la mostrada en la figura 4.4. Se ha optado por implementar UDP en lugar

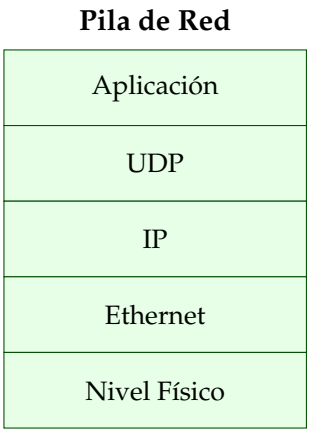

FIGURA 4.4: Representación de la pila de red implementada siguiendo el modelo OSI (*Open System Interconnection*).

de TCP ya que en el entorno que se iba desplegar, era un entorno sin pérdidas y porque las características de control de la congestión, control de pérdidas y el orden de los paquetes que ofrece TCP, se traducen en pérdida de rendimiento de CPU y de red.

La parte optimizada de la pila respecto a otras pilas de red ha sido el protocolo IP. Esta optimización [2], [3] consiste en establecer ciertos campos de la cabecera a valores por defecto tanto en sentido ascendente como descendente en la pila, como el tipo de servicio, el tiempo de vida del paquete o la longitud del paquete fija en 20 Bytes de manera que las opciones no se utilicen. Esto nos permite realizar menos comprobaciones de las que se harían en una pila completa permitiéndonos ganar tiempo.

Por otro lado se ha decidido implementar la fragmentación IP para poder soportar *payloads* de un tamaño superior al de la mínima MTU de la ruta entre la Raspberry Pi y el usuario.

Se ha implementado el protocolo ARP [15], [19], tanto la parte de descubrimiento como la de respuesta, junto con una implementación ligera de tabla ARP de 16 entradas para poder cachear las direcciones físicas de las últimas comunicaciones.

El mensaje ARP Request es utilizado cuando la pila se recorre de manera descendente (envío de mensajes) para obtener la dirección física de la máquina con la IP de destino en caso de que dicha IP se encuentre en la misma subred que la Raspberry Pi. Si la IP destino está en una subred diferente, se pregunta por la dirección física de la puerta de enlace predeterminada (*gateway*). Internamente, cuando se realiza el ARP Request, la pila de red comprueba en la tabla ARP si esa dirección IP está cacheada y en caso de estarlo, se evita tener que enviar un *frame* a la red.

El mensaje ARP Replay se utiliza cuando se realiza la llamada a la función recv, utilizada para recibir mensajes UDP, (bloqueante) de manera transparente al usuario. Cuando recv se desbloquea al recibir datos en el buffer de entrada, se detecta el tipo de mensaje de los datos y en caso de ser un mensaje ARP Request, se responde con un ARP Response y se vuelve a la situación de bloqueo a la espera del mensaje UDP. En caso de no ser un mensaje ARP o UDP, este se desecha volviendo a la situación de bloqueo. Únicamente se asciende por la pila en caso de que sea un mensaje UDP bien formado (esto implica que

tanto la cabecera IP como Ethernet son correctas).

Pantalla

El módulo de pantalla permite al sistema configurar la GPU para que sea capaz de modificar los valores de los píxeles de la pantalla que esté conectada a través del puerto HDMI a la Raspberry Pi y que a su vez proporciona al usuario una interfaz que abstrae de todo protocolo de bajo nivel necesario para la comunicación CPU-GPU.

El código de este módulo está dividido en dos partes principales, la configuración del *framebuffer* y la escritura sobre este.

El *framebuffer* es la abstracción del hardware que representa los píxeles de una pantalla. Concretamente en la Raspberry Pi, este *framebuffer* es una región de memoria que está compartida por la CPU y la GPU de manera que la CPU pueda escribir datos en ella mientras que la GPU los lee y se encarga de representarlos en la pantalla.

Para la configuración del *framebuffer* [20], [21] es necesario realizarla a través de la GPU utilizando un protocolo llamado mailbox, el cual, a través de los diferentes canales de comunicación existentes entre CPU y GPU, permite enviar y recibir datos.

El proceso de configuración del *framebuffer* de la pantalla pasa por los siguientes pasos para finalmente obtener por parte de la GPU el puntero a la base de este:

1. Obtener el tamaño real de la pantalla a la que se encuentra conectada la Raspberry Pi.
2. Establecer el ancho y alto en píxeles tanto físico como virtual especificando con cuántos bits queremos representar un píxel. El sistema establece que un píxel está representado por 16 bits.
3. Hacer que la CPU reserve memoria en función de los valores establecidos en el paso anterior (número de píxeles por tamaño de cada píxel) y obtener el puntero al inicio del *framebuffer*.
4. Obtener el número de bytes por línea de pantalla (*Pitch*).

Cabe destacar que todos los valores devueltos a través del *mailbox* por la GPU (ancho, alto, *pitch*, inicio del *framebuffer*, etc) es necesario almacenarlos en una estructura que definirá la pantalla, la cual será utilizada por la parte del módulo encargado de la escritura de caracteres y píxeles en la pantalla.

Esta parte proporciona al usuario una interfaz para configurar el tamaño de los caracteres que se van a imprimir (pequeño, mediano y grande), el color del fondo y el de los caracteres, así como una función similar a la función *printf* de los sistemas Unix, la cual permite imprimir en pantalla una cadena de caracteres siguiendo las especificaciones de color y tamaño de letra establecidas y con soporte a realizar *scroll* automático en la pantalla en caso de desbordarla con texto.

4.3.2. Librerías estándar y auxiliares

Se ha querido realizar una implementación de algunas de las librerías estándar necesarias por los módulos del sistema siguiendo la estructura que presentan en los sistemas

Unix/Linux. Entre estas funciones están las relacionadas con el manejo de memoria dinámica (*malloc, calloc y free*), algunas de las funciones presentes en la librería *stdlib* y que son requeridas por varias partes del *kernel* (*memset, memcpy, memcmp*, etc), funciones para realizar manejo de cadenas de caracteres y de conversión de variables a texto para poder imprimirlas en la pantalla (*strlen, strcpy, strcmp*, etc) y funciones secundarias requeridas por el sistema para configurar los diferentes dispositivos (gestor de interrupciones, manejo de la caché y contadores de tiempo).

Además de las librerías implementadas, se han creado nuevas definiciones de tipos de datos que por defecto no se encuentran soportadas por el compilador GCC (NULL, size_t, uint, etc).

Memoria dinámica

Para la implementación de la memoria dinámica se decidió usar como referencia la base del sistema de manejo de memoria utilizado en sistemas Linux/Unix [17] pero utilizando una estructura de datos más sencilla.

En los sistemas convencionales, al igual que en este sistema operativo, hay una región de memoria llamada HEAP encargada de almacenar toda la memoria que se ha reservado de manera dinámica. Para ello se hace uso de la función *sbrk*, la cual internamente incrementa o decrementa el puntero *brk* situado en el comienzo del HEAP tras el arranque. La función *sbrk* desplaza el puntero *brk* hacia direcciones más altas de memoria en caso de recibir un número positivo o hacia direcciones más bajas en caso de ser negativo. De esta forma podemos reservar una porción del HEAP introduciendo el tamaño del fragmento en la función *sbrk*.

Este sistema únicamente nos permite liberar el último fragmento de memoria reservado, pero para resolverlo, al igual que se hace en otras implementaciones, se utiliza una estructura de datos para encapsular los bloques de memoria reservados. La estructura de datos utilizada para la encapsulación es una lista simplemente enlazada debido, a su facilidad de implementación y a la vista de los numerosos problemas que históricamente ha tenido la implementación de lista doblemente enlazada utilizada en la Glibc.

Los nodos de la lista implementada están representados por una estructura formada por una variable con el tamaño del fragmento, un *flag* que indica si está libre y un puntero a la cabecera de la siguiente estructura. De esta forma, cuando se quiere reservar un fragmento, el puntero *brk* se incrementa en un valor correspondiente al tamaño de la estructura más la memoria que se quiere reservar y en la lista se añade un nodo con la información de tamaño correspondiente a lo reservado, el *flag* a falso y el puntero al siguiente fragmento con valor NULL.

A la hora de liberar un fragmento nos encontramos dos escenarios, el primero de ellos corresponde con el caso de querer liberar el último fragmento reservado, en cuyo caso solo hay que decrementar el puntero *brk* en tamaño de la cabecera más tamaño del fragmento posiciones y eliminar el último nodo de la lista. Por otro lado existe la situación de querer liberar un fragmento que no ha sido el último en reservarse. En este caso, solo es necesario marcarlo en la cabecera como fragmento libre y, en caso de que el fragmento posterior o anterior estén también marcados como libres, fusionarlos. En caso de querer liberar el último fragmento reservado y que el anterior esté libre, se deberá decrementar el puntero *brk* lo correspondiente a la suma de los dos fragmentos y sus cabeceras. En caso de tener fragmentos libres en la mitad de la lista, se deberá comprobar

antes de reservar si el fragmento cabe en alguno de estos espacios libres. Esto se hará siguiendo una estrategia *First-Fit*.

En ciertos casos, al disponer de un HEAP limitado a 4MB, puede ocurrir que la memoria quede muy fragmentada con huecos libres muy pequeños y no consecutivos, por lo que será necesario realizar una compactación de la memoria y dejar el espacio libre en la zona alta del HEAP.

Librería estándar

Debido a la necesidad de leer, escribir y comparar bloques de memoria, se optó por implementar ciertas funciones siguiendo el estándar establecido por los sistemas Linux/Unix [17] en la librería *stdlib*. La referencia utilizada para la implementación ha sido la Glibc de GCC, aunque se han realizado ciertas modificaciones para aumentar la eficiencia de algunas de ellas debido a las necesidades del sistema.

Las funciones que ha sido necesario implementar debido a su recurrente uso por parte del *kernel* han sido:

- **memset**: Permite establecer el valor de los bytes de un bloque de memoria a uno determinado.
- **bzero**: Caso particular de *memset* tomando como valor el cero.
- **memcpy**: Copia un bloque de un tamaño determinado a otra zona de memoria.
- **memcmp**: Determina si todos los bytes de dos bloques de memoria del mismo tamaño son iguales.
- **atoi**: Transforma la representación en cadena de caracteres de un número entero a una variable de tipo entero.

Manejo de cadenas

Debido principalmente a la necesidad de imprimir cadenas de caracteres y variables con diferentes representaciones en la pantalla, se han tenido que implementar funciones similares a las encontradas en la librería *string* de los sistemas Linux/Unix [17] junto con algunas funciones particulares requeridas por el sistema.

Entre las funciones estándar de manejo de cadenas encontramos las siguientes:

- **strlen**: Determina la longitud de una cadena de caracteres.
- **strcpy**: Copia una cadena de caracteres en otra dirección de memoria.
- **strcmp**: Determina si todos los caracteres de una cadena son iguales a los de una segunda cadena.

En cuanto a las funciones no estándares implementadas, todas ellas convierten un tipo de dato determinado a su representación en caracteres:

- **uint2hex**: Representa un entero sin signo en hexadecimal.
- **char2hex**: Representa un carácter en hexadecimal.
- **uint2dec**: Representa un entero sin signo en decimal.

Funciones auxiliares de bajo nivel

Este grupo de funciones comprende el conjunto de utilidades (interrupciones, caché y aritmética) [16], [17], [22] necesarias que han sido implementadas para facilitar una interfaz a los módulos a través de la cual puedan ejecutar pequeñas rutinas eficientemente.

El gestor de interrupciones es una pieza fundamental de cualquier sistema operativo responsable de identificar las señales procedentes desde el exterior de la CPU (interrupciones de dispositivos) como desde dentro (excepciones) e iniciar las rutinas encargadas de controlar dichas señales.

La interrupciones configuradas son: *Reset*, *Prefetch*, *Data Abort*, *Undefined Instruction*, *Software y Fast Interrupt Request*.

Las funciones de manejo de memoria caché han sido determinantes para el correcto funcionamiento del sistema debido a que en muchas situaciones, como en el protocolo de mailbox utilizado entre la CPU y la GPU, es necesario hacer esperas bloqueantes hasta que las memorias caché presentes en la Raspberry Pi se actualicen. También se han implementado funciones que fuerzan a las memorias de datos e instrucciones a invalidarse de manera que se actualicen con los valores reales de memoria.

Un dato curioso del procesador ARM1176JZFS es que a pesar de contar con soporte para coma flotante no dispone de una instrucción para realizar división entre números enteros, por lo que fue necesario implementar una rutina que permitiera al sistema realizar la división de dos números enteros positivos. Esta rutina es utilizada en el módulo de pantalla a la hora de calcular el número de píxeles necesarios para representar un carácter con un tamaño determinado.

4.4. Cliente

Esta arquitectura de sistema operativo requiere que el usuario/cliente envíe los *payloads* que van a ser ejecutados por red. Para esto se ha implementado un protocolo simple para recibir *payloads* de tamaños no conocidos. Este protocolo consiste en lo siguiente:

1. El sistema operativo se bloquea en la función *recv* esperando a recibir el tamaño del *payload* que el cliente quiere ejecutar.
2. Una vez se ha procesado el mensaje obteniendo el tamaño, el sistema operativo reserva memoria para un bloque del tamaño especificado por el usuario.
3. Seguidamente, el sistema se vuelve a bloquear en la función *recv* esperando a que el cliente envíe el *payload*, el cual será almacenado en la memoria previamente reservada.
4. Por último, el sistema operativo realiza un salto a función con la dirección donde se encuentra almacenado el *payload* [23].

4.4.1. Payload

El *payload* es el nombre que recibe la función compilada que el usuario ejecuta (ejecutable que el sistema implementado es capaz de interpretar). Para que este *payload* funcione correctamente tiene que ser compilado utilizando compilación cruzada en caso de

estar en una máquina de otra arquitectura diferente a ARM. Se ha desarrollado una serie de herramientas que funcionan utilizando GNU Embedded Toolchain para ARM (serie de herramientas que permiten realizar compilación cruzada para ARM) las cuales permiten compilar una función escrita en C [24], [25] del tamaño mínimo y enviarlo, siguiendo el protocolo descrito anteriormente, a la Raspberry Pi.

El *payload* se compila utilizando diferentes *flags* de GCC [25] que permiten realizar optimizaciones y estructurar el código máquina generado:

- **Flag -fpic**: *Position Independent Code* o Código Independiente de Posición significa que el código máquina generado no sea dependiente de la dirección de memoria en la que se encuentre almacenado durante su ejecución. A efectos prácticos se traduce en que el compilador establecerá direcciones relativas en lugar de absolutas en las llamadas a función.
- **Flag -Os**: GCC permite realizar optimizaciones en el código máquina generado (O0 por defecto, O1, O2, O3 y Os). Os funciona a efectos prácticos como una optimización de nivel 2 (02) pero añadiendo una optimización extra centrada en la reducción del código máquina.
- **Flag -r**: Este *flag* permite situar las variables locales de la función en una sección diferente a la sección .DATA (donde se sitúan por defecto) y referenciarlas de una manera relativa al registro IP/PC (*Instruction Pointer* o *Program Counter*). Esto permite que podamos ejecutar una función que tenga variables locales.
- **Flag -ffreestanding**: Hace que el compilador no asuma nombres por defecto del estándar ISO C, permitiendo que, por ejemplo, la función que se compila se llame *main*.

Tras la compilación de la función GCC nos genera un objeto ELF a partir del cual se generará el *payload* en el formato de fichero aceptado por el sistema.

4.4.2. Generación del payload

Cuando la función se compila con las opciones previamente enumeradas, genera un objeto el cual es pasado como parámetro a la herramienta desarrollada en Python, que permite generar el *payload* en el formato aceptado por el sistema y que será enviado a la Raspberry Pi donde se ejecutará. Esta herramienta hace uso del módulo de Python *pyelftools* el cual permite analizar y extraer información de ficheros ELF.

Con el objetivo de comprimir el tamaño ocupado por el *payload* se extraen del objeto ELF generado tras la compilación ciertas secciones y se eliminan los *headers* con información del objeto con las tablas de símbolos. Un objeto ELF puede ser ejecutado únicamente con la sección .TEXT en caso de no tener variables, pero en caso de tenerlas, estas se encuentran en la sección .TEXT referenciadas de forma relativa al PC. Únicamente es necesario agregar una subsección de la sección .RODATA denominada .rodata.str1.4 donde se almacenan las cadenas que han sido *hardcodeadas* en el código. Como se muestra en la figura 4.5, la estructura de los ejecutables con los que trabaja el sistema resulta de concatenar las secciones previamente mencionadas.

Con esto se consigue generar un tipo de fichero ejecutable que es aceptado por el sistema como código ejecutable bien formado y de un tamaño considerablemente inferior. A modo de ejemplo se ha compilado, sin realizar ninguna optimización de código a

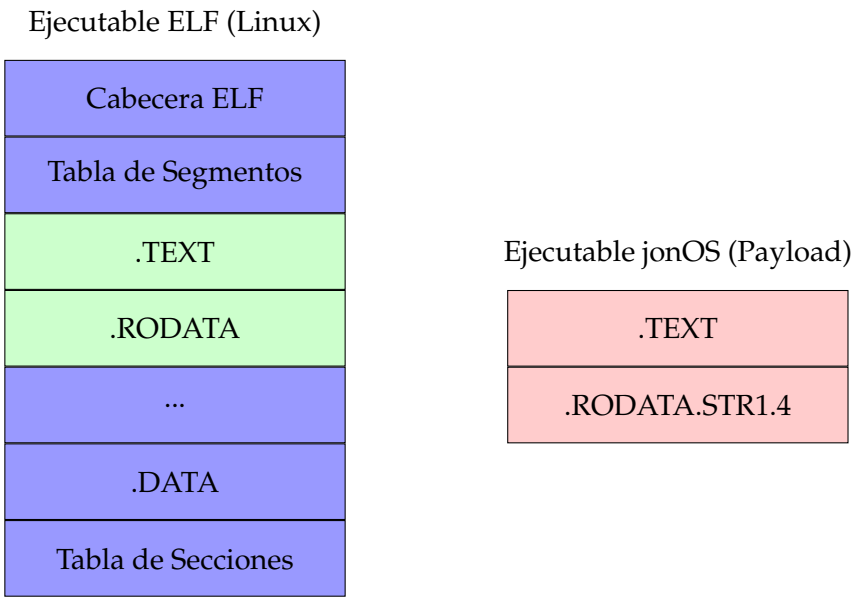

FIGURA 4.5: Comparación de los formatos de objetos ejecutables de Linux y de jonOS (en base a lo extraido del ELF).

través de alguna opción del compilador, la función mostrada en 4.6, la cual únicamente bloquea a la CPU en el bucle infinito definido en la línea 4. El resultado de esto es un objeto de 612 Bytes de tamaño en disco. Si aplicamos el proceso de generación del ejecutable para el sistema implementado se obtiene un objeto de 14 Bytes de tamaño en disco. Esto supone una reducción del tamaño del ejecutable en casi 44 veces el tamaño de un ejecutable en formato ELF.

```
1 // Funcion con un bucle
      infinito
2 void f()
3 {
4   while(1);
5 }
```

FIGURA 4.6: Posible función definida por el usuario para ser ejecutada en el sistema que provoca que la CPU se quede bloqueada en un bucle infinito.

4.4.3. Llamadas al sistema desde el payload

Hay que destacar que el proceso de generación del *payload* no incluye el enlazado de los objetos (parte del proceso de generación del ejecutable donde se resuelven las direcciones de las funciones) y las funciones internamente pueden llamar a métodos que se encuentran en el *kernel*. Esto plantea una gran problemática porque el usuario únicamente puede generar *payloads* con llamadas a funciones propias y no podría acceder a dispositivos como la red o la pantalla a través de los métodos implementados por el sistema.

Para solucionar este problema se optó por resolver la dirección de las funciones en tiempo de ejecución en lugar de en compilación. Esto es así porque, a pesar de que en

un primer momento se decidió por resolver las direcciones de memoria en tiempo de ejecución con macros que definen funciones cuyo valor era la dirección de memoria en la que se encontraban en el *kernel* (kernel.map), se producía un cambio de direcciones en el momento que se compilaba en otra máquina o se introducía una nueva función o variable global en el *kernel*. En su lugar, se ha optado por resolver las llamadas a función en tiempo de ejecución con el módulo ADDRESS_SOLVER que permite, únicamente conociendo la dirección de una función de este módulo por parte del cliente, resolver en ejecución el resto de direcciones de funciones del sistema.

Este módulo es una implementación simple de lo que en lenguajes de más alto nivel se conoce como mapa. Concretamente este módulo relaciona un número entero con una dirección de memoria. Esta relación número-dirección depende del orden establecido en la función *init_address_solver* en la cual el diseñador del sistema decide cuáles de las funciones implementadas podrán ser ejecutadas por el cliente. Este módulo dispone de una función llamada *solveFunction* que, dado un número, te devuelve la dirección de la función vinculada a dicho número. El cliente dispone de un fichero con cabeceras en el que se definen todas las relaciones número-función en forma de macros además de tipos de datos utilizados por el *kernel* y necesariamente la dirección de memoria en la que se encuentra la función *solveFunctión*. Este fichero tiene que ser proporcionado por el administrador del sistema a los usuarios en caso de que se decida realizar alguna modificación.

Utilizando este método, el usuario puede al inicio de la función, definir punteros a funciones y obtener su dirección en tiempo de ejecución igualando dichos punteros a la salida de la función *solveFunction* utilizando como parámetro la macro que tiene el mismo nombre que la función de *kernel* que se quiere resolver. Además, este método ayuda a reducir aún más el tamaño del *payload*, ya que no es necesario guardar ninguna información previa en variables globales sobre las funciones a las que se quiere llamar puesto que al utilizar punteros a funciones en forma de variables locales a la función, estas son reservadas en pila y no en el propio *payload*.

Capítulo 5

Pruebas de rendimiento

El objetivo perseguido con este proyecto era lograr ejecutar tareas más rápido de lo que se logra con un sistema operativo adaptado a esta máquina, como lo es Raspbian, logrando crear sistemas empotrados más veloces que los que se pueden crear con este último.

Tomando como base que los sistemas empotrados son sistemas cuyo objetivo es ejecutar una o algunas pocas funciones de manera repetitiva, se llegó a la hipótesis de que un sistema operativo con la arquitectura previamente descrita sería capaz de ejecutar tareas aisladas más rápido que los tiempos obtenidos en los sistemas operativos utilizados para crear sistemas dedicados sobre Raspberry Pi.

Se ha decidido medir los elementos más significativos de un sistema empotrado, la velocidad de cómputo y la velocidad en las comunicaciones. Se han seleccionado estas dos características porque son las características más importantes dentro de un sistema de este tipo. Con una velocidad de cómputo alta, las tareas a realizar por el sistema se ejecutarán en menor tiempo, que es lo deseable en un sistema de estas características. Por otro lado, la otra parte en la que es deseable aumentar el rendimiento es en las comunicaciones, ya que muchas de las implementaciones de los sistemas empotrados requieren la obtención de información del medio para procesarla y enviar una respuesta.

5.1. CPU

Antes de comparar la velocidad de ejecución en ambos sistemas es conveniente definir los dos tipos de tiempos que pueden medirse en pruebas de rendimiento como las que se han realizado.

Entendemos por tiempo de CPU la cantidad de tiempo que necesita la CPU para ejecutar todas las instrucciones de la tarea, sin tener en cuenta la demora producida por los tiempos de entrada/salida.

Por otro lado, se entiende por tiempo real el tiempo transcurrido desde que se comienza una tarea hasta que esta termina medido con un reloj ordinario. Cabe destacar que este tiempo incluye el tiempo de entrada/salida.

En un sistema multitarea, como en el caso de Raspbian (Linux), el tiempo real siempre es mayor o igual al tiempo de CPU, ya que es el sistema operativo el encargado de realizar los accesos a los dispositivos y, como es otra tarea diferente, los ciclos de CPU

que tarda en realizar los accesos no se tienen en cuenta en los ciclos de CPU de la tarea en cuestión. Además un sistema multitarea dispone de un planificador encargado de decidir qué tarea se ejecuta en qué momento, pudiendo hacer que la tarea sea reemplazada por otra en la CPU provocando que el tiempo real aumente. Por otro lado, en los sistemas operativos como el desarrollado (monotarea), en los que el sistema operativo es la única tarea que se ejecuta, el tiempo de CPU coincide con el tiempo real de la tarea.

Se han realizado mediciones de ambos tiempos aunque el verdaderamente significativo es el tiempo real, ya que es el indicador real de la velocidad con la que el sistema empotrado resuelve las tareas.

Para realizar las mediciones se han utilizado dos métodos: el primero para realizar medición de tiempo real y el segundo para medir el tiempo de CPU únicamente en Raspbian, ya que en el sistema implementado, el tiempo de CPU y el tiempo real es el mismo:

- Función *clock()*: Este método ha sido utilizado para medir el tiempo de CPU utilizado en Raspbian. Se obtiene el valor del reloj antes y después de la tarea que se desea cronometrar. Se obtiene el número de ciclos que ha durado la tarea y se divide por el número de ciclos por segundo obteniendo así el tiempo de CPU.
- Utilizando un convertidor PL-2303 conectado a la UART de la Raspberry notificamos a un ordenador externo que contará el tiempo real cuando comienza y cuando se termina de ejecutar la tarea.

Como prueba de rendimiento se ha utilizado una variante de la prueba de trabajo *HashCash*, utilizada en cadenas de bloques (Blockchain). Este tipo de prueba de trabajo consiste en realizar cálculos de *hashes* hasta que el resultado de uno de ellos comience por unos caracteres determinados. Cuanto más extensa sea la cadena de caracteres definida, más tiempo se tardará en encontrar una entrada de la función *hash* que satisfaga que su salida empiece por dicha cadena.

La variante implementada consiste en el cálculo de *hashes* MD5 (*Message-Digest Algorithm 5*) un número definido de veces. Aumentando el número de veces que se calcula el *hash* logramos que la tarea tarde más tiempo en ejecutarse. Esta tarea ha sido implementada de igual manera para ambos sistemas y compilada con la opción -O2 que realiza una optimización de código de nivel 2.

En cuanto a la ejecución sobre Raspbian, se ha realizado de manera que el proceso sobre el que se realizaban las mediciones fuese el único proceso de usuario que se está ejecutando.

Comparación de tiempo de CPU

Se han realizado mediciones con los siguientes valores de N (número de veces que se ejecuta el cálculo del *hash MD5*): 1, 10, 100, 1000, 10000, 100000, 200000, 400000, 600000, 800000 y 1000000.

A la vista de los resultados mostrados en la figura 5.1, se aprecia que el sistema implementado consigue reducir el tiempo de CPU de Raspbian para la misma tarea en un 30 %.

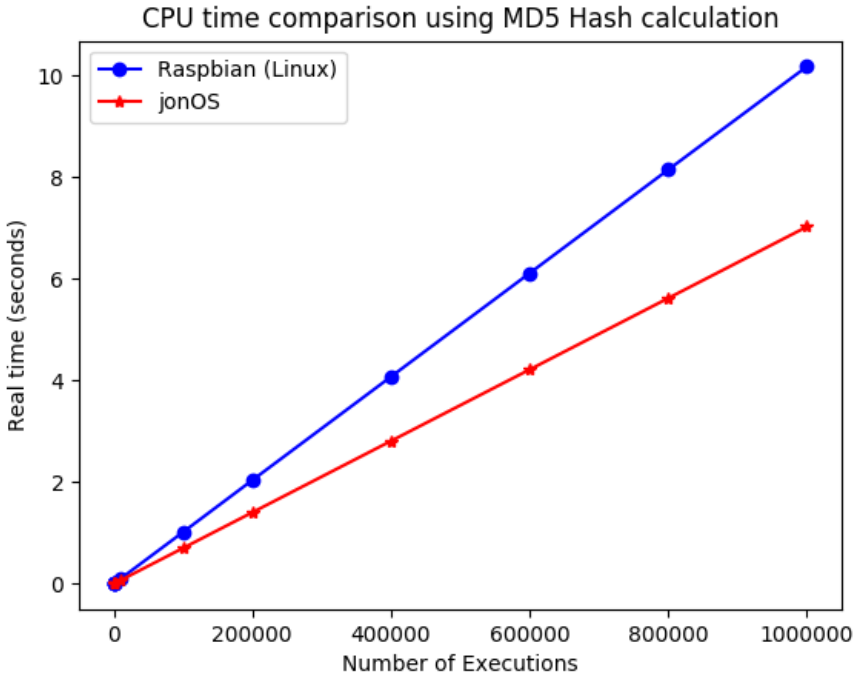

FIGURA 5.1: Comparación de tiempo de CPU consumido utilizando como prueba de trabajo el cálculo de *hash MD5*

Esto es debido a que los sistemas monotarea como el implementado disponen de toda la memoria caché para la única tarea existente. Sin embargo los sistemas operativos que soportan multitarea necesitan de un planificador que decida qué proceso se ejecuta en cada momento. Esto implica que cuando el sistema operativo realiza un cambio de contexto y el nuevo proceso comienza a ejecutarse, este se encuentra tanto con la caché de datos como con la de instrucciones con datos inválidos y es necesario actualizar las cachés L1 y L2 de datos y la caché L1 de instrucciones. Eso supone un incremento del tiempo de CPU ya que se tiene que esperar a que las cachés se actualicen correctamente con los datos e instrucciones de memoria.

Comparación de tiempo real

El tiempo real es la medida de tiempo de ejecución más importante en un sistema empotrado, ya que determina cómo de rápido este será capaz de resolver pequeños problemas y actuar en consecuencia en un entorno.

Para realizar la medición del tiempo real se ha conectado a los pines 14 y 15 de la Raspberry Pi un conversor de TTL a USB para poder, desde un segundo ordenador, leer los mensajes que el sistema operativo saca por la UART. Desde la Raspberry Pi, con ambos sistemas operativos, se envía la cadena 'S' (*Start*) antes de comenzar a ejecutar la tarea a cronometrar. Cuando la tarea se termina de ejecutar ambos sistemas envían por la UART la cadena 'E' (*End*) como indicador de fin de tarea.

Desde el ordenador que cuenta el tiempo transcurrido se realiza *polling* (mantenerse leyendo hasta que la cadena recibida sea la deseada) para esperar ambas cadenas y para dar comienzo y fin a la cuenta del tiempo. Dicha cuenta del tiempo se ha realizado con la función de Python time() del módulo time, la cual devuelve el tiempo en ese instante.

Cabe destacar que la parte del proceso encargada de recibir los mensajes de la UART es la misma para ambos sistemas operativos, por lo que el tiempo perdido en el *polling* no se tiene en cuenta.

Para las mediciones se han utilizado los siguientes valores de N (numero de veces que se ejecuta el cálculo del *hash MD5*): 1, 10, 100, 1000, 10000, 100000, 200000, 400000, 600000, 800000 y 1000000.

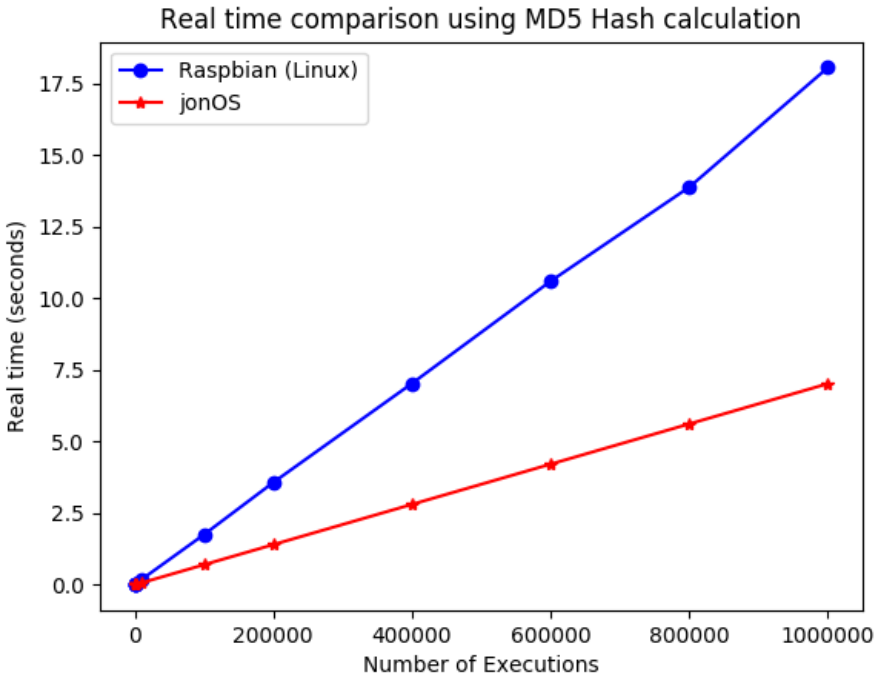

FIGURA 5.2: Comparación de tiempo real consumido utilizando como prueba de trabajo el cálculo de *hash MD5*

Como se puede apreciar en la gráfica 5.2, el rendimiento del sistema operativo implementado es notablemente superior al de Raspbian llegando a mejorarlo un 62 %.

Estos datos corroboran que la hipótesis en la que se basó la investigación era correcta. Esto es debido a que todos los recursos del sistema son utilizados exclusivamente para la tarea a ejecutar en sistemas operativos monotarea, mientras que en sistemas operativos con multitarea, estos recursos son compartidos entre todos los procesos que se estén ejecutando según ordene el planificador. Incluso en el caso probado de que el único proceso de usuario sea la tarea a cronometrar, como caso mejor nos encontramos con otro proceso correspondiente al sistema operativo. Además debido a la arquitectura modular que implementan los sistemas Linux internamente, cuando un proceso de usuario realiza una llamada a sistema, el *kernel* tiene que resolver esta petición y devolverle el puntero a la función solicitada, suponiendo esto un cambio de contexto en el proceso que se está ejecutando.

Hay que tener en cuenta que en un sistema como Raspbian, tras el arranque de este existe un número aproximado de 20 procesos en ejecución. Esto supone que los cambios de contexto se realizarán con una mayor frecuencia al existir más procesos en el sistema, lo que provoca que el tiempo real que tarda en ejecutarse la tarea aumente considerablemente. Los sistemas operativos multitarea implementan algoritmos de planificación

basados en prioridades para conseguir mejorar lo que en principio sería un factor 20, ya que se asigna el mismo tiempo de CPU a cada proceso (algoritmo *Round-robin*), y lograr reducir dichos tiempos considerablemente debido al factor de la prioridad que establece tiempos de CPU superiores a los procesos con más prioridad.

Además, el tiempo de CPU está incluido dentro del tiempo real, por lo que todos los factores que hacían que el tiempo de CPU aumentase, también afectarán al tiempo real de la misma manera.

5.2. Red

Una de las formas más sencillas de comprobar la velocidad de red de un sistema es utilizando un servidor echo, el cual únicamente devuelve al origen todos los paquetes que recibe (con el mismo contenido). Concretamente este servidor funciona bajo el protocolo UDP (*User Datagram Protocol*), el cual es un protocolo del nivel de transporte que funciona por encima de IP y que está basado en el intercambio de datagramas. Este protocolo, al contrario que TCP (*Transmission Control Protocol*), no tiene ninguna garantía. Los paquetes pueden no llegar al destino, pueden llegar desordenados y no se tiene control sobre la congestión de la red.

Para ello se ha implementado el mismo servidor de echo con la única diferencia de que las funciones de red utilizadas son las propias de cada sistema. En el caso de *Raspbian*, se utiliza la librería de red que Linux proporciona. Para el sistema operativo propuesto se ha utilizado la pila UDP/IP implementada para dicho sistema.

En el lado del cliente, al igual que se ha hecho para medir el tiempo real de ejecución, se ha utilizado el mismo cliente Python para medir los tiempos en ambos servidores. El funcionamiento es muy similar al utilizado para el tiempo real. Se toma una referencia de tiempo antes de comenzar con las N peticiones al servidor y se detiene cuando este responde a la última. El modelo de servidor utilizado es un servidor iterativo, el cual se caracteriza por no poder atender dos peticiones simultáneamente, de manera que hasta que el servidor no responde a una petición, el cliente no envía la siguiente.

Por último, la otra cosa importante a tener en cuenta en este tipo de pruebas es la arquitectura de la red que conectará la máquina que va a ser medida a la que va a realizar las mediciones. En el caso particular de este proyecto, se utilizo una subred creada por un router doméstico cuyas especificaciones indican que tiene soporte Gigabit Ethernet (1 gigabit por segundo). Esta arquitectura en principio no debería suponer ninguna limitación para realizar las mediciones porque el estándar de la puerta ethernet que utiliza la *Raspberry Pi* es 10/100 Ethernet (hasta 100 megabits por segundo). Tras realizar numerosas pruebas, se observó que el router estaba provocando un cuello de botella al sistema operativo implementado en la trasmisión de los paquetes por lo que se optó por realizar una conexión punto a punto.

Al utilizar una arquitectura punto a punto se observó que el tiempo medio de las peticiones era considerablemente inferior al obtenido utilizando la subred (más de un segundo de diferencia para el caso mayor de N).

A continuación se muestra la comparación de tiempos con una arquitectura punto a punto haciendo 1, 10, 100, 1000, 10000, 20000, 40000, 60000, 80000 y 100000 peticiones.

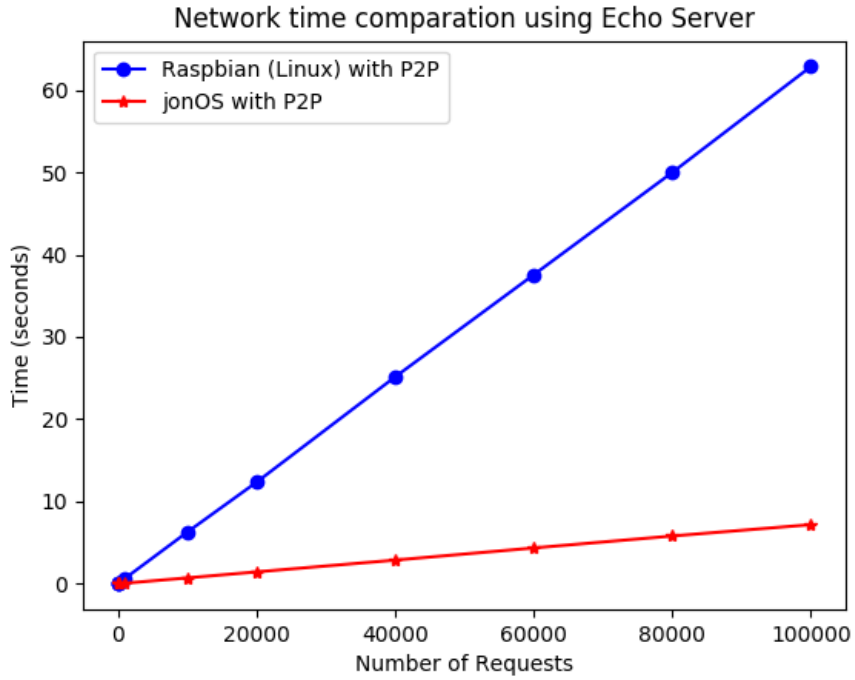

FIGURA 5.3: Comparación de tiempo de red utilizando como prueba de trabajo un servidor de echo

Como se observa en los resultados mostrados en 5.3, la pila de red implementada junto con el sistema operativo propuesto obtienen unos resultados mucho mejores a los ofrecidos por *Raspbian*. El factor de mejora respecto a Linux es aproximadamente 9.

Esta sustancial mejora se debe a varios factores de diseño y de implementación entre los que se destacan los siguientes:

1. **Monotarea**: Al igual que en el cálculo de *hashes MD5*, un sistema operativo tardará más en ejecutar N recepciones y respuestas de paquetes en un sistema con soporte multitarea debido a que los recursos, CPU, red y otros dispositivos se tienen que compartir entre los procesos que lo soliciten.

2. **Puertos**: En un sistema operativo monotarea no existe el concepto de puerto. Los puertos son usados por los sistemas operativos multitarea para poder diferenciar a qué proceso corresponde cada paquete, ya que en un puerto únicamente puede estar escuchando y enviando un proceso. En un sistema operativo monotarea el concepto de puerto no es necesario, por lo que el *kernel* no invierte tiempo en identificar el puerto y notificar al proceso correspondiente que puede leer el *buffer* de entrada.

3. **Lectura y escritura del buffer**: En los sistemas Linux el *kernel* permite al proceso leer del *buffer* de entrada cuando este considera que está suficientemente lleno y, al igual que ocurre con el *buffer* de salida, es el sistema operativo el que decide cuándo los paquetes encolados se mandan a la red. En el sistema implementado, cuando la tarea (*kernel*) desea enviar un paquete, este desciende por la pila UDP/IP y, sin esperas, se envía a la red. Ocurre lo mismo a la hora de leer un paquete del *buffer* de entrada, el paquete asciende por la pila de red cuando en el *buffer* hay la cantidad de bytes especificada.

4. **Pila UPD/IP**: La pila UDP/IP implementada ha sido específicamente diseñada para, a pesar de perder ciertas funcionalidades debido al establecimiento de valores por defecto en ciertos campos de la cabecera IP, centrarse en la velocidad para componer y descomponer paquetes.

Con la configuración establecida para realizar las pruebas de velocidad, no es posible determinar si la velocidad obtenida es el límite del sistema operativo implementado, ya que también se está teniendo en cuenta la velocidad con la que el ordenador del cliente envía y recibe paquetes. Esto significa que el valor medido puede deberse a que el ordenador del cliente que realiza las mediciones ha alcanzado su límite de red y provoca un cuello de botella que satura la velocidad máxima de la conexión a su velocidad máxima.

Dado esto, se puede garantizar que el tiempo máximo del sistema operativo implementado es el que se muestra en la gráfica, pudiendo esta velocidad encontrarse por debajo de la gráfica mostrada en caso de que ese límite de velocidad esté establecido por el ordenador del cliente.

La forma correcta de medir la velocidad de este sistema es utilizar el mismo sistema operativo como cliente en el otro extremo de la comunicación, de manera que podamos garantizar que ambas partes tienen la misma velocidad. Esta prueba no ha podido ser realizada debido a que no se disponía de otra placa Raspberry Pi Model B para usarse como cliente en el momento de realizar de este documento.

Capítulo 6

Conclusiones

Este proyecto pretende demostrar que la arquitectura presentada de sistema operativo como servidor es muy superior a las arquitecturas y sistemas operativos tradicionalmente utilizados en entornos empotrados y sistemas de alto rendimiento que utilizan hardware genérico como es el caso de la *Raspberry Pi* y de *Raspbian*.

El objetivo perseguido desde el principio con este proyecto era lograr desarrollar un sistema operativo completamente desde cero y que este tuviera un mejor rendimiento que los sistemas que se han venido utilizado tradicionalmente en entornos empotrados y de alto rendimiento.

Es por eso que, gracias al diseño dedicado a una tarea y a una implementación ligera y eficiente de los módulos básicos requeridos por un sistema operativo de estas características, como el módulo de red o las funciones para el manejo de memoria (funciones de la librería estándar y), ha sido posible demostrar la hipótesis de que un sistema como el desarrollado ofrece un rendimiento superior de CPU y sobre todo de tiempo real de ejecución en comparación al ofrecido por sistemas operativos más genéricos como es el caso de los basados en *Linux*. Además, debido a la mejora realizada en algunas funcionalidades de la librería estándar y a las realizadas en la pila de comunicaciones de cara a la optimización de tiempos, se ha logrado desarrollar un sistema que permite la interacción con la red a una velocidad muy superior a la ofrecida en otras alternativas. Esto abre un abanico de posibilidades de cara al desarrollo de sistemas enfocados al análisis de red analizando e inyectando paquetes en la red. La aplicación más directa de este tipo de sistemas empotrados es la implementación de sondas enfocadas para el análisis del tráfico IP o de sistemas IDS (*Intrusion Detection System*), los cuales requieren altos rendimientos de CPU para realizar cálculos sobre el tráfico capturado y una pila de comunicación que permita extraer paquetes de la red a altas velocidades.

La aplicación ideal de este tipo de enfoque es la de ser utilizado en sistemas de red como routers o conmutadores de paquetes. En estos últimos la aplicación es aún más clara ya que su funcionalidad principal es interconectar dos o más *host* y dirigir los paquetes que estos envíen en base a la dirección MAC. Por esto, se considera que el sistema operativo desarrollado es una alternativa que ofrecería mayor velocidad de conmutación que las soluciones actuales que utilizan derivados ligeros de *Linux*, debido a las características que ofrece en cuanto al tiempo de CPU y la velocidad en la pila de red.

Otro de los objetivos satisfechos con este proyecto es el de diseñar una arquitectura que, dada su versatilidad, permitiera ser desplegada en múltiples escenarios dejando la implementación de la funcionalidad específica en manos del programador. Esto se logra gracias a la inclusión de la idea de servidor que atiende las solicitudes de los clientes

junto a la posibilidad de diseñar y programar la funcionalidad del sistema dedicado en una máquina diferente, permitiendo que el sistema empotrado pueda cambiar su funcionalidad de una manera sencilla.

En conclusión, la arquitectura de sistema operativo materializada en la implementación del sistema *jonOS* ha conseguido demostrar que, para ciertos entornos como es el caso de los sistemas empotrados, de tiempo real o de alto rendimiento implementados sobre hardware genérico como la placa *Raspberry Pi*, es una mejor opción que los sistemas operativos tradicionales, ya que consigue unas velocidades de CPU tres veces superiores y una velocidad de red nueve veces superior.

Todo el código fuente se ha publicado de manera gratuita siguiendo la filosofía *Open Source* y bajo la licencia MIT (*Massachusetts Institute of Technology*), la cual permite su uso, copia, modificación, integración con otro *Software*, publicar, sublicenciar o vender. El código de ***jonOS*** puede ser accedido a través del siguiente enlace: jonOS.

Capítulo 7

Trabajo futuro

Como se ha mencionado en la introducción, este trabajo surgió de un proyecto personal del alumno y se va a seguir trabajando en él para mejorar la velocidad tanto de CPU como de red e incluir nuevas funcionalidades así como generar versiones más estables y fáciles de utilizar. A continuación se listan las partes del sistema en las que se va a trabajar o que van a ser incluidas por orden de importancia:

1. **Implementación de un sistema modular**: Implementar la funcionalidad para cargar módulos de manera similar a la forma que tiene el *kernel* de Linux de cargar módulos pero de una forma remota al igual que se ejecutan *payloads*.

2. **Implementación del driver USB**: Debido al uso de un driver genérico de USB que tiene soporte para gran cantidad de periféricos además del acceso a red, se producen excepciones no controladas que derivan en *crashes* del sistema. Es por esto que se pretende trabajar en un controlador propio del concentrador USB sobre el que va montado el puerto *Ethernet* de manera que, al ser dedicado al envío y recibo de paquetes de la red, se aumente la velocidad y se pueda tener un mayor control de las excepciones.

3. **Ampliar el soporte de placas**: Realizar las modificaciones en el *kernel* de manera que pueda ser ejecutado en sistemas de varios núcleos como es el caso de los modelos siguientes al utilizado (Modelo 2, 3 e intermedios). Además también se quiere migrar a otro tipo de arquitectura de procesadores como la x86 y MIPS.

4. **Desarrollar una variante para realizar sistemas distribuidos**: Una forma de ampliar el potencial de esta arquitectura de sistema operativo es añadir soporte para realizar computación distribuid,a de manera que el uso de varios sistemas simultáneamente se lleve a cabo de una forma coordinada y eficiente.

5. **Implementación de librería gráfica**: Ampliar la funcionalidad de la librería gráfica actual, la cual solo permite imprimir texto en pantalla, para incluir soporte a nivel de píxel permitiendo dibujar líneas y diversas formas geométricas en la pantalla. Además se incluirá soporte para mostrar imágenes en formatos conocidos (jpeg, png y bmp).

6. **Mejorar el sistema de compilación cruzada de *payloads***: Modificar el sistema de compilación cruzada de manera que permita compilar correctamente códigos repartidos en varios ficheros y solucionar los problemas con las referencias a cadenas de caracteres declaradas estáticamente en ámbito local.

7. **Adaptar motor gráfico 3D**: Realizar una adaptación del motor gráfico *Jongine* desarrollado en un proyecto personal del alumno para poder mostrar entornos de videojuegos en 3D sobre *jonOS* y disponible en el siguiente enlace: Jongine.

Todas estas mejoras y aumentos de la funcionalidad se pretenden realizar bajo una filosofía *Open Source* y colaborativa siguiendo el enfoque de propio de los sistemas empotrados y de alto rendimiento. Es por esto que se ha adjuntado al código una licencia MIT para que cualquier persona pueda usar, modificar y distribuir el código y posibles variantes. Además con está licencia se permite sublicenciar el código facilitando la venta de un posible producto basado en esta arquitectura.

Bibliografía

[1] E. Helin y A. Renberg. (ene. de 2015). The little book about OS development, dirección: https://littleosbook.github.io/.

[2] J. Jang, J. Jung e Y. Cho, "Design of a Lightweight TCP/IP Protocol Stack with an Event-Driven Scheduler", *JOURNAL OF INFORMATION SCIENCE AND ENGINEERING*, 2012.

[3] S. Goldschmidt y D. Ziegelmeier. (nov. de 2018). lwIP - A Lightweight TCP/IP stack, dirección: https://savannah.nongnu.org/projects/lwip/.

[4] W. Stallings, *Operative Systems: Internals and Design Principles*, fourth. Pearson, 2001.

[5] Various. (2018). Operative Systems types, dirección: https://en.wikipedia.org/wiki/Operating_system.

[6] M. Barr y A. Massa, *Programming Embedded Systems*, second. O'Reilly, feb. de 2009.

[7] J. H. L. III, C. Segura y N. Dauchy, "A Minimal Linux Environment for High Performance Computing Systems", *Sandia National Laboratories*,

[8] M. T. Jones. (ago. de 2006). BusyBox simplifies embedded Linux systems, dirección: https://www.ibm.com/developerworks/linux/library/l-busybox/.

[9] J. A. Stankovic y R. Rajkumar, "Real-Time Operating System", *Kluwer Academic Publishers*, 2004.

[10] A. Madhavapeddy y R. Mortier, "Unikernels: Library Operating Systems for the Cloud", 2013.

[11] D. R. Engler, M. F. Kaashoek y J. O. Jr, "Exokernel: An Operating System Architecture for Application-Level Resource Management", 1995.

[12] A. Maccabe, P. Bridges, R. Brightwell y R. Riesen, "Recent Trends in Operating Systems and their Applicability to HPC", *Cray User Group*, 2006.

[13] (Oct. de 2018). TOP500 list, dirección: https://www.top500.org/.

[14] D. Mulfari, A. Celesti, M. Fazio, M. Villari y A. Puliafito, "Using Embedded Systems to Spread Assistive Technology on Multiple Devices in Smart Environments", *Proceedings - 2014 IEEE International Conference on Bioinformatics and Biomedicine, IEEE BIBM 2014*, nov. de 2014. DOI: 10.1109/BIBM.2014.6999234.

[15] J. Kurose y K. Ross, *Computer Networking: A Top-Down Approach*, seventh. Pearson, 2016.

[16] K. Yaghmour y o. Jon Masters, *Building Embedded Linux Systems*, second. O'Reilly, ago. de 2008.

[17] D. P. Bovet y M. Cesati, *Understanding the Linux Kernel*, first. O'Reilly, dic. de 2002.

[18] R. Stange. (dic. de 2018). USPi Driver, dirección: https://github.com/rsta2/uspi.

[19] C. Benvenuti, *Understanding Linux Networks Internals*, first. O'Reilly, dic. de 2005.

[20] B. Corporation, *BCM2835 ARM Peripherals*, Broadcom Corporation, Broadcom Europe Ltd. 406 Science Park Milton Road Cambridge CB4 0WW, 2012.

[21] A. Chadwick. (). Baking Pi – Operating Systems Development, dirección: `https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/index.html`.

[22] A. Holdings, *ARM Instruction Set | ARM7TDMI-S Data Sheet*, ARM Holdings.

[23] D. Fu y F. Shi, "Buffer Overflow exploit and defensive techniques", *Fourth International Conference on Multimedia Information Networking and Security*, 2012.

[24] G. Project. (2017). GNU make, dirección: `https://www.gnu.org/software/make/manual/make.html`.

[25] F. S. Foundation. (2017). Using the GNU Compiler Collection (GCC), dirección: `https://gcc.gnu.org/onlinedocs/gcc-6.5.0/gcc/`.