UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

OPTIMIZACIÓN DEL RENDIMIENTO DE SOCKETS UDP EN APLICACIONES MULTITHREADS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

DIEGO ARTURO GUILLERMO ALEJANDRO RIVERA VILLAGRA

PROFESOR GUÍA:
JOSÉ MIGUEL PIQUER GARDNER

MIEMBROS DE LA COMISIÓN:
ALEXANDRE BERGEL
CÉSAR GUERRERO SALDIVIA

Este trabajo ha sido parcialmente financiado por NIC Chile Research Labs

SANTIAGO DE CHILE
SEPTIEMBRE 2013

Resumen

Los servidores DNS son máquinas que resuelven consultas sobre nombres de dominio y se caracterizan por atender grandes cantidades de pequeñas peticiones que usualmente caben en una única unidad de comunicación en Internet llamada "paquete". Para aumentar la cantidad de respuestas, el software encargado de esto explota las máquinas con múltiples procesadores, paralelizando las atenciones, lo cual ha mostrado no generar las ganancias esperadas.

Para estudiar este problema se replicó la operación usando núcleos de Sistemas Operativos modernos e intentando leer concurrentemente desde un socket, identificando así los posibles puntos de falla: la implementación de `read` en la *libc*, el mecanismo de atención de las llamadas al sistema, o alguna porción de código ejecutado al recibir un paquete desde Internet.

Los primeros dos posibles orígenes fueron descartados con pruebas directas mediante la comparación del rendimiento de `read` al leer desde distintas fuentes y determinando cómo se comporta la atención de las llamadas a sistema, mediante la creación de una de estas con fines de prueba. Así, el estudio se concentra en la tercera posible fuente del problema: el núcleo de Linux.

Para estudiar el comportamiento de cómo es recibido un paquete, se investigó cómo fluye la información a través del stack de red desde que ésta arriba al dispositivo. Se descubrió que la información era encolada en estructuras de datos compartidas, requiriendo sincronización, e identificando, por lo tanto, un posible punto de falla. Para corroborarlo, se modificó un núcleo con el fin de determinar cómo la sincronización afectaba en la serialización de los accesos a un socket.

Los resultados de las pruebas anteriores ejecutadas sobre el núcleo modificado arrojaron que el esquema de sincronización utilizado no permitía las lecturas concurrentes, por lo que se propuso cambiar el esquema de encolamiento de los paquetes en el socket, introduciendo estructuras que sí permitan la paralelización de `read`.

Para simplificar la situación, el esquema de encolamiento de paquetes fue modelado en una implementación en C en espacio usuario, replicando estructuras y sincronización presentes en el núcleo. Sobre este modelo fue implementada una solución con múltiples colas de recepción de paquetes, creando colas por cada lector concurrente desde el lado de la aplicación.

Finalmente, el modelo arrojó que esta solución permite efectivamente paralelizar los accesos, llegando a duplicar el throughput alcanzado actualmente por los sockets en determinadas configuraciones de threads.

Para mis hermanas; ya que sin ellas este documento no existiría.

Agradecimientos

Creo que esta es la página que más me ha costado escribir de este documento, y no es porque no tenga nada ni nada a nadie a quien agradecer, sino que todo lo contrario; deberé hacer una gran esfuerzo por comprimir las palabras en una página a todo quien merece estar acá.

Primero que nada, quiero agradecer a la vida, por haberme dado altos y bajos, alegrías y tristezas, y por sobre todo, haberme dado la entereza para llegar hasta donde estoy ahora.

También quiero agradecer a mi familia, porque ellos siempre han estado en los momentos difíciles con palabras de "fuerza" o "ánimo", tirando del carro con tanto esmero como yo lo he hecho.

Palabras especiales van dedicadas a mis hermanas; ellas han sido la inspiración para seguir adelante en los peores momentos, han sido la razón para no haberme rendido en las infinitas oportunidades que pude haberlo hecho. Ellas siempre serán la razón por la que sigo adelante e, inconscientemente, han sido la luz que me ha hecho ver el camino cuando lo he perdido.

A mis amigos, les debo palabras personales: Rodrigo, la distancia nos separa, pero siempre recuerdo los momentos que hemos compartido en Iquique; Felipito, desde la básica juntos, y aún nos juntamos a tomar cerveza y conversar de cartas o poker; Tomás, esas infinitas tardes de viernes jugando Smash en la pajarera, son geniales recuerdos tuyos; Felipe, siempre he criticado lo sometido que puedes llegar a ser, pero sé que jamás te has olvidado de mí; Fernanda, aunque no partimos muy bien, supimos entendernos de maravilla entre nosotros; y a los "Talentosos Campesinos" (Camilo, Camilo, Miguel y Mauricio), son todos unos grandes, y profesionales de excelencia. Todas estas personas han aportado de manera fundamental en mí, y les estaré eternamente agradecidos. Mención especial para Mauricio Quezada, quien se ofreció voluntariamente a revisar este documento.

Para finalizar, me gustaría agradecer al equipo de Niclabs, lugar en el cual pude ver la investigación como el área en donde me gustaría continuar mi carrera profesional.

Tabla de contenido

Introducción **1**

1. Antecedentes **5**
1.1. Modelo OSI y sockets . . . 5
1.2. Generalidades de Sistemas Operativos . . . 6
1.3. Sincronización dentro del núcleo . . . 7
1.3.1. Semáforos . . . 7
1.3.2. Operaciones atómicas . . . 8
1.3.3. Barreras de Memoria . . . 8
1.3.4. Spinlocks . . . 8
1.4. Interrupciones y controladores de interrupciones . . . 9
1.5. Sockets de Linux . . . 10
1.5.1. Introducción . . . 10
1.5.2. Anatomía . . . 11
1.5.3. Funcionamiento . . . 12

2. Medición del rendimiento actual de sockets UDP **19**
2.1. Estrategia de las pruebas . . . 19
2.2. Metodología de las pruebas . . . 20
2.2.1. Descripción de las pruebas . . . 20
2.3. Especificaciones técnicas del equipo . . . 22
2.4. Resultados de la ejecución . . . 22
2.4.1. Mediciones de lectura en dispositivo virtual . . . 23
2.4.2. Mediciones de lectura en cola FIFO . . . 23
2.4.3. Mediciones de lectura en socket UDP . . . 24
2.5. Discusión de resultados . . . 25
2.6. Conclusiones y desafíos . . . 26

3. Aislamiento del problema en el Kernel **27**
3.1. Antecedentes . . . 27
3.1.1. Análisis de la recepción de un paquete . . . 27
3.1.2. Posibles puntos de falla . . . 29
3.2. Metodología de pruebas . . . 30
3.2.1. Herramientas a utilizar . . . 30
3.2.2. Mecanismo de pruebas . . . 31
3.3. Resultados obtenidos . . . 32

3.3.1. Pruebas de stress . . . 33
3.3.2. Prueba de stress, Memory Accounting desactivado . . . 34
3.3.3. Prueba de stress, accesos sincronizados . . . 36
3.3.4. Prueba de stress, tiempos de ejecución de udp_recvmsg . . . 38
3.4. Discusión . . . 39
3.5. Conclusiones . . . 40

4. Modelamiento y Solución Propuesta **42**
4.1. Modelamiento actual del núcleo . . . 42
4.1.1. Arquitectura del modelo . . . 42
4.1.2. Sincronización de estructuras . . . 44
4.1.3. Emulación de las pruebas . . . 45
4.2. Esquema de encolamiento propuesto . . . 46
4.3. Metodología de las pruebas . . . 47
4.4. Resultados obtenidos . . . 48
4.4.1. Escalamiento de lectores: Esquema original . . . 49
4.4.2. Escalamiento de lectores: Esquema modificado . . . 52
4.4.3. Escalamiento simétrico . . . 55
4.4.4. Resumen esquema original versus múltiples colas . . . 55
4.5. Discusión . . . 58
4.6. Conclusiones . . . 60

Conclusión **61**

Bibliografía **64**

Índice de tablas

2.1. Tiempo medido en las pruebas de dispositivo virtual en Kernel 3.5.0–2 . . . 23
2.2. Tiempo medido en las pruebas de cola FIFO para distintas versiones del Kernel 23
2.3. Tiempo medido en las pruebas de socket UDP, para distintas versiones del Kernel . . . 24
2.4. Tiempo medido en las pruebas de cambio de modo en Kernel 3.5.2 . . . 25

3.1. Resultados lockdep para lock `AF_INET` usando protocolo IPv4 . . . 33
3.2. Resultados lockdep para lock `sk_buffer_head_lock` usando protocolo IPv4 33
3.3. Resultados lockdep para lock `AF_INET6` usando protocolo IPv6 . . . 33
3.4. Resultados lockdep para lock `sk_buffer_head_lock` usando protocolo IPv6 33
3.5. Resultados lockdep para lock `sk_buffer_head_lock` usando protocolo IPv4, sin Memory Accounting . . . 34
3.6. Resultados lockdep para lock `sk_buffer_head_lock` usando protocolo IPv6, sin Memory Accounting . . . 35
3.7. Resultados lockdep con accesos sincronizados para lock `AF_INET` usando protocolo IPv4 . . . 36
3.8. Resultados lockdep con accesos sincronizados para lock `sk_buffer_head_lock` usando protocolo IPv4 . . . 36
3.9. Resultados lockdep con accesos sincronizados para lock `AF_INET6` usando protocolo IPv6 . . . 37
3.10. Resultados lockdep con accesos sincronizados para lock `sk_buffer_head_lock` usando protocolo IPv6 . . . 37

4.1. Resultados esquema original, 1 thread Kernel . . . 49
4.2. Resultados esquema original, 2 threads Kernel . . . 50
4.3. Resultados esquema original, 3 threads Kernel . . . 50
4.4. Resultados esquema original, 4 threads Kernel . . . 51
4.5. Resultados múltiples colas, 1 thread Kernel . . . 52
4.6. Resultados múltiples colas, 2 threads Kernel . . . 53
4.7. Resultados múltiples colas, 3 threads Kernel . . . 54
4.8. Resultados múltiples colas, 4 threads Kernel . . . 54
4.9. Resultados de escalamiento simétrico, modelos con spinlock . . . 55
4.10. Resultados resumen modificación versus original, 1 thread Kernel . . . 56
4.11. Resultados resumen modificación versus original, 2 threads Kernel . . . 57
4.12. Resultados resumen modificación versus original, 3 threads Kernel . . . 57
4.13. Resultados resumen modificación versus original, 4 threads Kernel . . . 58

Índice de figuras

1.1. Definición de `struct socket` . . . 11
1.2. Representación de un socket Linux . . . 12
1.3. Flujo de datos desde y hacia la red . . . 13
1.4. Representación del envío de un paquete . . . 14
1.5. Representación de la recepción de un paquete . . . 16

2.1. Arquitectura de procesos, libc y Kernel . . . 20
2.2. Gráfico de los resultados para dispositivo virtual . . . 23
2.3. Gráfico de los resultados para cola FIFO . . . 24
2.4. Gráfico de los resultados para socket UDP . . . 24
2.5. Gráfico de los resultados para el cambio de modo . . . 25

3.1. Comparación de locks en IPv4 . . . 34
3.2. Comparación de locks en IPv6 . . . 34
3.3. Comparación de locks con y sin Memory Accounting para IPv4 . . . 35
3.4. Comparación de locks con y sin Memory Accounting para IPv6 . . . 36
3.5. Comparación de locks con accesos sincronizados en IPv4 . . . 37
3.6. Comparación de locks con accesos sincronizados en IPv6 . . . 38

4.1. Arquitectura del modelo implementado . . . 43
4.2. Modelamiento del socket en user space . . . 43
4.3. Código esquemático de un spinlock . . . 45
4.4. Modelo propuesto para el socket . . . 46
4.5. Rendimiento esquema original, 1 thread Kernel . . . 49
4.6. Rendimiento esquema original, 2 threads Kernel . . . 49
4.7. Rendimiento esquema original, 3 threads Kernel . . . 50
4.8. Rendimiento esquema original, 4 threads Kernel . . . 51
4.9. Rendimiento múltiples colas, 1 thread Kernel . . . 52
4.10. Rendimiento múltiples colas, 2 threads Kernel . . . 53
4.11. Rendimiento múltiples colas, 3 threads Kernel . . . 53
4.12. Rendimiento múltiples colas, 4 threads Kernel . . . 54
4.13. Rendimiento de escalamiento simétrico, modelos con spinlock . . . 55
4.14. Throughput de escalamiento simétrico, modelos con spinlock . . . 55
4.15. Resumen modificación versus original, 1 thread Kernel . . . 56
4.16. Throughput modificación versus original, 1 thread Kernel . . . 56
4.17. Resumen modificación versus original, 2 threads Kernel . . . 56
4.18. Throughput modificación versus original, 2 threads Kernel . . . 56

4.19. Resumen modificación versus original, 3 threads Kernel 57
4.20. Throughput modificación versus original, 3 threads Kernel 57
4.21. Resumen modificación versus original, 4 threads Kernel 58
4.22. Throughput modificación versus original, 4 threads Kernel 58

Introducción

Hoy en día, la era de las comunicaciones nos ha llevado a depender de más servicios como Internet. La masificación de dispositivos como los *smartphones* y *tablets* ha hecho que la cantidad de aparatos que consumen información disponible a través de Internet crezca sostenidamente. Esta información se compone principalmente de contenido alojado en servidores o servicios que son capaces de generar información nueva en función de datos del usuario o extraídos por el mismo dispositivo, caracterizando el contexto en el cual se accede a Internet. Entre los contenidos más populares podemos mencionar las imágenes y videos residentes en sitios como *YouTube* o *Facebook*, entradas en blogs, diarios digitales, entre otros. Entre los servicios disponibles podemos encontrar servicios de mapas en línea (como *Google Maps*, *Bing Maps*, etc.), sitios de compras a través de Internet (como *eBay* o *Amazon*) o incluso servicios gubernamentales ofrecidos a través de la red. Todo lo antes dicho no es de utilidad si no se establece una forma única y fija para acceder e identificar a cada uno de ellos.

Las URL (*Uniform Resource Locator*) presentan una solución para la identificación y ubicación para cada contenido o servicio ofrecido en la red, los cuales fueron introducidos en el año 1994 [5]. Estos identificadores únicos permiten localizar cada recurso en la red, definiendo completamente el host dentro del dominio en donde reside dicho recurso, y la ubicación local en dicha máquina en donde encontrar el contenido [6].

Sin embargo, la determinación del host dentro del dominio representa el mismo problema de antes pero a un nivel más global, ya que la red física sólo es capaz de enviar mensajes a una determinada dirección IP, ignorando el hecho de que esta máquina pueda pertenecer a un dominio. Para ello, el concepto de DNS fue introducido [12].

DNS (*Domain Name System*) nació de la necesidad de expandir el mecanismo que se estaba usando para traducir los nombres de host en sus respectivas direcciones IP. Este sistema consistía en mantener un archivo (`hosts.txt`) con la tabla de estas traducciones; cada vez que una nueva máquina entraba en la red, este archivo era actualizado y enviado a cada host para reflejar el cambio. Sin embargo, el mecanismo antes descrito no escala fácilmente ante escenarios en donde la cantidad de máquinas conectadas a la red crece.

Este servicio fue diseñado como una base de datos centralizada en la cual reside la misma información que se encontraba en el archivo, pero definiendo niveles organizacionales denominadas *dominios*. Esto busca agrupar la información que cambia frecuentemente y separarla de la que se mantiene relativamente estática. Paralelamente, fue concebido como un sistema distribuido, garantizando el acceso a la información y evitando centralizarla completamente, debido a que la totalidad de la base datos antes descrita es necesaria por toda la red para

realizar dichas traducciones.

Asimismo, el sistema de consulta a esta base de datos distribuida fue definido de manera delegativa entre zonas, donde éstas se organizan jerárquicamente desde la zona más general, cuya información no varía frecuentemente, hasta las zonas más específicas en donde el contenido de ella es muy dinámico. Esta organización se puede visualizar fácilmente mediante un árbol en donde los nodos internos son zonas DNS y las hojas representan a los hosts dentro de ellas. Siguiendo este esquema, las consultas son resueltas mediante la ubicación del camino desde la raíz del árbol hasta la hoja que se quiere determinar, resolviendo la traducción paso a paso mediante conexiones a cada servidor autoritativo de la zona por la cual responden.

Dada esta arquitectura, es claro observar que los servidores encargados de servir zonas localizadas más hacia la raíz del árbol tienen tráficos de consultas mayores. Por esto es necesario que las máquinas que responden a estas consultas sean capaces de procesar la mayor cantidad de requerimientos por unidad de tiempo. Esto introduce una restricción sobre la interfaz provista por el Sistema Operativo para intercambiar mensajes: ésta debe estar caracterizada por ser ágil en la recepción y envío de mensajes.

Esta interfaz es llamada *socket*, y es el objeto implementado en el núcleo del Sistema Operativo utilizado por cualquier aplicación para enviar mensajes a otros hosts a través de Internet. Estas estructuras son las encargadas de dividir y empaquetar la información que será enviada a través de la red, en conformidad a los protocolos de comunicación establecidos por el modelo OSI, los cuales están implementados en código residente en el núcleo del Sistema.

Una vez que el servidor recibe una petición (la cual está contenida en un paquete de red), ésta es procesada desde capas inferiores de la especificación OSI hacia arriba, siendo la capa 4 la última por la cual cada paquete debe pasar antes de dejar lista la información para que la aplicación pueda leerla.

En el caso de servidores de alto tráfico y con consultas pequeñas (como es el caso de los servidores DNS, como antes se dijo), es necesario que la máquina encargada de atender las peticiones sea capaz de atender la mayor cantidad de consultas en el menor tiempo posible. Esto, en conjunto con la introducción de máquinas multiprocesadores, lleva a la elaboración de aplicaciones capaces de aprovechar la computación paralela para atender peticiones. Sin embargo, el software cuya tarea es la de leer peticiones desde un socket, procesarlas y enviar respuestas, no logra ganancia alguna con la introducción del paralelismo.

Fenómenos similares ya han sido observados anteriormente por empresas como Facebook [18] o Toshiba [20], quienes han hecho esfuerzos por optimizar las aplicaciones que hacen uso paralelo de los sockets sin lograr mejoras sustanciales, delegando el problema al núcleo de Linux, específicamente, en la forma en que éste administra el acceso al socket.

Este problema es simple de ser observado en la siguiente situación: varios niños quieren extraer galletas desde un frasco que las contiene, inicialmente cerrado. En cuanto el frasco sea abierto, los niños intentarán sacar galletas todos al mismo tiempo, por lo que ocurrirán choques entre ellos, entorpeciendo la tarea y, muy probablemente, aumentando el tiempo que se demoran para obtener las galletas que quieren. Una forma fácil de solucionar este problema es *serializar* los accesos de los niños, es decir, coordinar (o *sincronizar*) a éstos

para que definan un orden en el cual cada uno pueda sacar una galleta de manera exclusiva.

Esquemas que subsanan este hecho en servidores han sido descubiertos experimentalmente por empresas como NIC Chile, que luego de observar el fenómeno y descartar una posible saturación de la máquina utilizada, lograron duplicar el rendimiento del servidor configurando dos máquinas virtuales en la misma máquina física, sin saber cómo solucionar de raíz este problema. En nuestro ejemplo, esta solución sería representada por la colocación de dos frascos de galletas.

Es por esto que en las siguientes páginas se intenta cuantificar y caracterizar el actual rendimiento de extraer información de un socket, presentar evidencia que permita establecer la fuente del problema en el núcleo del Sistema Operativo y establecer las razones dentro del Kernel que originan el comportamiento secuencial de las lecturas concurrentes, lo cual no logra ser explicado por el sobrecosto introducido por el hecho de empaquetar la información que es enviada a través de un socket.

Específicamente, la metodología de pruebas consistirá en:

- *Replicación de la situación,* consistente en la introducción de lecturas concurrentes de información de prueba desde un socket.
- *Pruebas sobre la libc,* las cuales buscan determinar la influencia de esta capa de abstracción entre el núcleo mismo y las aplicaciones que hacen uso de sus llamadas a sistema.
- *Pruebas de llamadas al sistema,* buscando el potencial cuello de botella en la forma en que el núcleo atiende las llamadas al sistema, como es la petición de datos desde un socket.
- *Aislamiento del problema en el núcleo,* que caracterizará y determinará la raíz del rendimiento observado dentro del código que recibe paquetes en un socket.
- *Emulación del núcleo y propuesta de solución,* creando una copia "a escala" de lo que ocurre en el Kernel en un programa en C e implementando sobre ella una posible solución para este problema.

Para este último paso, se buscará implementar una "maqueta a escala" del Kernel en una aplicación escrita en lenguaje C. Básicamente, esta maqueta consistirá en un modelo de productores y consumidores utilizando una lista enlazada como estructura de datos para permitir el intercambio de información entre los participantes, la cual será protegida mediante un spinlock ante modificaciones concurrentes. Asimismo, y para mantener la simplificación lo más cercana a la realidad posible, será necesario implementar la primitiva de sincronización en código ensamblador, por lo que este mecanismo no introducirá ningún tipo de sobrecosto como lo hacen otras primitivas de sincronización disponibles en librerías existentes.

Sobre este modelo se replicarán las pruebas de stress analizando los rendimientos obtenidos y comparándolos con las pruebas de esfuerzo realizadas sobre el núcleo real del Sistema Operativo. Con esto se buscará observar el mismo comportamiento secuencial que se experimenta en los sockets.

Como forma de solucionar el problema, se propuso crear una cola de paquetes por cada

consumidor presente, cada una de las cuales está protegida por un *spinlock* exclusivo de cada estructura. Este nuevo esquema pretende minimizar la cantidad de contenciones que se producen por modificaciones concurrentes en la única cola de paquetes disponible en el modelo anterior. Esto último se logra probar mediante la emulación de los procesos ejecutados por el núcleo, dentro de los cuales se introduce la modificación antes propuesta.

Esto demostró que la introducción de múltiples colas logra producir mejoras de rendimiento, principalmente en escenarios en los cuales existen tanto *threads* concurrentes por el lado del núcleo como *threads* de aplicación que extraen paquetes desde el socket. Sin embargo, los datos obtenidos no terminan de caracterizar completamente la solución propuesta, dada la cantidad de variables que influyen en el problema, entre las cuales se cuentan el número de procesadores de la máquina, número de colas creadas, entre otras.

Esto último delega como trabajo futuro:

- Concluir la caracterización de la solución propuesta, según se especifica en el Capítulo 3.
- Estudiar otras estructuras de datos para ser utilizadas como cola de recepción.
- Estudiar el comportamiento de todas las estructuras de datos candidatas.
- Implementar la solución definitiva en un Kernel de Linux actual.
- Ejecutar las pruebas requeridas sobre servidores con alto tráfico de consultas. Un servidor DNS representaría un muy buen caso de pruebas para la situación.

Finalmente, el trabajo presentado aquí tiene un alcance de investigación del problema, por lo que cualquier solución propuesta para el problema descrito en estas páginas debe obtener retroalimentación de la comunidad de desarrolladores en todo el proceso que resta de manera transversal. Esto último se debe a que Linux es una plataforma que es desarrollada por la comunidad, lo que obliga a que cualquier modificación semántica que se realice sobre el núcleo del Sistema deba ser discutida en la colectividad y aprobada por el respectivo mantenedor del código.

Capítulo 1

Antecedentes

1.1. Modelo OSI y sockets

Para poder establecer una comunicación entre 2 computadores o hosts, es necesario el uso de un protocolo de comunicación estándar para que los mensajes fluyan entre el emisor y el receptor. Este protocolo se conoce como modelo OSI (Open Systems Interconnection), el cual establece la existencia de 7 capas intermedias entre la aplicación y el medio físico de comunicación entre los hosts.

Dentro de estas capas, las concernientes al estudio que se explica en las siguientes páginas, son las capas 2 (Capa de Enlace), 3 (Capa de Red) y 4 (Capa de Transporte), siendo esta última la más estudiada. Estas capas permiten establecer niveles de abstracción sucesivos, permitiendo un mayor grado de modularidad en las implementaciones de los protocolos de red establecidos por los respectivos RFC en los años 80 [9, 10, 16].

La Capa de Enlace es la encargada de proveer los servicios de direccionamiento físico entre 2 hosts, con lo que se consigue que una serie de bytes (datagramas) sean enviados directamente desde un host hacia otro de manera directa y confiable, corrigiendo errores que la capa inferior (Capa Física) pudiera generar. Para lograr correctamente el intercambio de datos, esta capa provee el servicio de vecindarios (*neighbouring*), introduciendo las direcciones físicas presentes en cada adaptador de red (NIC). Sin embargo, en redes de más de 2 computadores, un sistema de direccionamiento más confiable y escalable es requerido.

La Capa de Red se encarga de proveer un sistema de direccionamiento más amplio, relajando la limitación de conectar sólo 2 hosts directamente, con lo que se logra conseguir una conexión virtual entre cualquier par de hosts pertenecientes a redes de computadores. Esto es provisto mediante el protocolo IP (Internet Protocol), el cual se encarga de determinar y agrupar de manera más amplia a los hosts en redes, identificándolos por una dirección única dentro de la red (y dentro del Universo de hosts). Asimismo, esta capa provee los mecanismos necesarios para el ruteo de mensajes (llamados paquetes en esta capa), con lo cual es posible alcanzar a cualquier host con un paquete simplemente delegando el envío de estos últimos a los routers de la red.

Finalmente, la Capa 4 establece los mecanismos de transporte de la información, garantizando la fiabilidad de la conexión a través de algoritmos de control de flujo. En esta capa, se puede localizar el sistema de multiplexación de conexiones, permitiendo a un host tener múltiples comunicaciones abiertas en paralelo con múltiples hosts. Esta característica es provista a través de la inclusión de los puertos de comunicación, con lo cual se logra identificar completamente cualquier flujo de datos que circula a través de la red.

Las funcionalidades de las capas antes descritas son provistas mediante implementaciones de cada Sistema Operativo, a través de entidades encargadas de implementar las características de los protocolos de red, y proveer al programador de interfaces para realizar la comunicación de procesos a través de la red. Estas interfaces son denominadas *sockets*, quienes están encargadas, entre otras cosas, de identificar unívocamente una conexión, protocolos de comunicación en juego, parámetros de red, etcétera, llegando incluso al procesamiento de los paquetes tanto entrantes como salientes del Sistema Operativo.

Antes de explicar su funcionamiento en detalle, es necesario exponer algunos conceptos de Sistemas Operativos en general.

1.2. Generalidades de Sistemas Operativos

La literatura define un Sistema Operativo como un conjunto de programas básicos incluidos en cada máquina [19]. Uno de estos programas es el Kernel o núcleo, concepto que tiende a ser tratado como sinónimo de Sistema Operativo (y lo será para efectos de este documento). Sin embargo, el Kernel es, como su nombre lo sugiere, la parte fundamental y central para que el resto de los programas funcionen correctamente en el sistema.

Específica y técnicamente, el Sistema Operativo es el encargado de las siguientes tareas (entre otras):

- Servir de interfaz para todos los componentes de hardware, incluyendo los componentes programables de bajo nivel.
- Proveer un ambiente de ejecución para todos los otros programas que correrán en el equipo.

Esto último fuerza a que el Sistema Operativo provea (entre otras cosas): administración de memoria, administración de procesos, gestión de usuarios y permisos y acceso a los archivos contenidos en el disco duro. Con esto, el sistema operativo termina por ser el programa raíz que permite la ejecución de cualquier otro software instalado, actuando como capa de abstracción del hardware. Esta última característica es importante hoy en día debido a la gran cantidad de fabricantes de hardware existentes, permitiendo a los programadores trabajar de manera indistinta de la máquina física en la cual su software va a ser ejecutado.

Sin ir más lejos, algunas de las principales características del Kernel de Linux se nombran a continuación:

- *Linux es un núcleo monolítico:* Linux es una gran y compleja porción de software

compilada, la cual está separada en algunas partes lógicas todas interrelacionadas entre sí.

- *Linux soporta threads de Kernel:* Los núcleos modernos, soportan hilos de ejecución exclusivos para tareas del Kernel (los cuales se reservan para realizar tareas asíncronas dentro del núcleo). Esto tiene la ventaja de aprovechar de mejor manera los procesadores de la máquina para ejecutar subrutinas pertenecientes al núcleo de manera paralela.
- *Linux es un núcleo "preemptive":* Esta característica permite que el mismo Kernel pueda desplazar tareas (incluso tareas de Kernel) menos importantes en pos de otras que tienen mayor prioridad y que, por lo tanto, deben ser ejecutadas inmediatamente.
- *Linux soporta aplicaciones con múltiples threads:* La mayor parte de las aplicaciones actuales delegan las acciones pesadas en hilos de ejecución paralelos, los cuales comparten el espacio de memoria con su padre. Linux soporta este tipo de aplicaciones mediante la creación de hilos de ejecución que son tratados de igual manera que cualquier otro proceso por el scheduler de procesos.
- *Linux soporta Procesamiento Simétrico de Multiprocesador (SMP):* Las máquinas con múltiples procesadores son cada vez más comunes, por lo que Linux (desde su versión 2.6) implementa el soporte de varios procesadores independientes, los cuales pueden atender cualquier tarea del sistema. Además, muy pocas partes del sistema continúan siendo serializadas con un gran lock, por lo que el soporte de SMP es cercano al óptimo.

Sin embargo, la última característica presentada introdujo el problema de sincronización de hilos de ejecución dentro del núcleo, lo cual representó un gran cambio desde la versión 2.6.

1.3. Sincronización dentro del núcleo

Con la introducción del soporte para máquinas SMP, Linux tuvo que resolver problemas de concurrencia dentro del mismo Sistema Operativo, ya que desde la versión 2.0 hasta la 2.5, el soporte de esta característica en el Kernel consistía en serializar los accesos simultáneos al sistema.

Desde la versión 2.6, un cambio en el scheduler de tareas llevó los hilos de ejecución paralelos al mismo núcleo del sistema, por lo que áreas de código (y principalmente estructuras), debían ser protegidas para evitar modificaciones concurrentes.

Para esto, Linux provee diversas primitivas de sincronización a disposición de los programadores, cuyos principales ejemplos serán explicados a continuación.

1.3.1. Semáforos

Los semáforos son primitivas dedicadas a garantizar exclusión mutua entre varios hilos de ejecución, garantizando que una tarea no podrá continuar la ejecución normal hasta que la

barrera del semáforo sea levantada completamente por la tarea que la posee al minuto de la contención.

Además, los semáforos garantizan que la tarea que deba esperar la barrera será suspendida hasta que sea levantada. Por esto, una tarea dormida en un semáforo no gastará mayores recursos de máquina (CPU principalmente) hasta que sea despertada asíncronamente, lo cual restringe el uso de esta primitiva de sincronización sólo a funciones que pueden dormir. Esto último implica que los semáforos no pueden ser usados por handlers de interrupciones o funciones que pueden ser pospuestas arbitrariamente por el núcleo.

1.3.2. Operaciones atómicas

Las operaciones atómicas se basan en instrucciones de máquina, las cuales realizan operaciones de "lectura, modificación y escritura" de manera atómica. En otras palabras, estas operaciones se realizan sin que otro hilo de ejecución pueda interrumpirlas o cambie el estado de la memoria, llevando a condiciones de *datarace*.

Para que estas operaciones funcionen correctamente, se debe asegurar que las instrucciones de máquina sean ejecutadas de manera atómica a nivel de chip, es decir, deben ser provistas e implementadas a nivel de CPU de esta manera.

1.3.3. Barreras de Memoria

Las barreras de memoria son primitivas de sincronización que permiten serializar los accesos a memoria. Debido a optimizaciones del compilador, las líneas de código escritas pueden ser reordenadas en las instrucciones de máquina finales, por lo que los accesos a memoria podrían no ejecutarse en el mismo orden en que se ven en el código que se escribió. El uso de una barrera de memoria permite que todos los accesos de memoria existentes hasta antes de la barrera sean efectivamente ejecutados, actuando como un punto de control en el cual se espera a que se completen todos los accesos a la memoria hasta antes de la barrera.

Esta primitiva permite eliminar errores de *datarace* producidos por comportamientos del tipo *set and test*, en donde el reordenamiento de las instrucciones podría hacer que la operación de set termine siendo ejecutada por el procesador después de la operación de test.

1.3.4. Spinlocks

Los spinlocks representan la primitiva de sincronización de más bajo nivel que garantiza exclusión mutua entre dos o más hilos de ejecución. Ellos permiten que uno o más hilos permanezcan esperando hasta que el hilo que posee el lock decida soltarlo.

El funcionamiento básico de estas primitivas se basa en las instrucciones de máquina *CAS (Compare And Swap)*, las cuales permiten, atómicamente, comparar e intercambiar valores

entre un registro de CPU y una dirección de memoria.

Sin embargo, estas primitivas están diseñadas para ser utilizadas al sincronizar bloques de código pequeños, ya que sus ciclos de espera se basan en *busy-waiting*, es decir, en ejecutar la misma instrucción una y otra vez en el procesador (en este caso, tratar de obtener el lock). Para mejorar un poco el rendimiento de los spinlocks se introducen instrucciones que "relajan" la CPU antes de intentar bloquear nuevamente el lock, logrando una mejor eficiencia energética.

Cabe destacar que no tiene sentido usar este tipo de primitivas en sistemas que no soportan SMP, debido a que no existen múltiples hilos de ejecución en el núcleo del sistema. En caso de que Linux sea compilado sin soporte para esta característica, se modificará el código de estos mecanismos para que no interfieran con el rendimiento del sistema en general.

1.4. Interrupciones y controladores de interrupciones

Todo dispositivo del computador es provisto de una dirección de memoria única en el espacio para su comunicación, por lo que escribir o leer datos desde o hacia ellos representa una operación de escritura o lectura en memoria. Sin embargo, los dispositivos de entrada están constantemente produciendo dicha información independientemente del sistema, por lo que es necesaria la existencia de un mecanismo asíncrono a través del cual el dispositivo informe al Sistema de la existencia de dichos datos, para luego ser rescatados a través de una operación de lectura en la dirección de memoria del periférico.

Este método de información es implementado mediante *interrupciones*. Éstas consisten en señales específicas que son enviadas desde el dispositivo de manera directa a la CPU, informando de la situación de disponibilidad de datos. Este envío se realiza mediante líneas de comunicación directas entre los dispositivos y el mismo procesador, con lo que se asegura que la señal enviada siempre llegará a la CPU, evitando la posibilidad de que la señal se pierda entre los datos de algún bus compartido.

Sin embargo, las interrupciones son por naturaleza asíncronas, por lo que pueden llegar en cualquier minuto al CPU, interrumpiendo cualquier tarea que el procesador esté ejecutando en ese minuto. Es por esto que el Sistema Operativo debe proveer de la funcionalidad de desplazar a la tarea que tiene ocupada la CPU en el instante en que llega la interrupción, guardar su estado de ejecución, atender la señal del dispositivo mediante la invocación del *controlador de interrupciones* (*handler*), y continuar la ejecución de la tarea que fue desplazada de manera transparente para esta última.

Es en este handler en donde se deben ejecutar las instrucciones necesarias para rescatar los datos disponibles en el dispositivo (leyendo desde su respectiva dirección de memoria, la cual es asignada por el Sistema Operativo), determinar qué es lo que se debe hacer con la información, y dejar el dispositivo en un estado normal, en caso de ser necesario. Dado que los handlers de interrupciones modifican el estado de los dispositivos del sistema, su ejecución no puede ser interrumpida por otro dispositivo que lo desplace con otro controlador. Es por esto

que, mientras se ejecuta un handler de interrupciones, éstas son inhibidas en el procesador que ejecuta dicho controlador, previniendo que el dispositivo que está siendo atendido quede en un estado inconsistente.

Para el caso específico de dispositivos de entrada (como lo son las tarjetas de red, discos duros, etc.), la información contenida en el dispositivo probablemente será almacenada en alguna estructura de datos manejada por el Kernel, la cual podría estar siendo ocupada paralelamente por algún otro procesador. En casos como estos, se requiere sincronizar el acceso a dichas estructuras incluso si son accedidas desde contextos de interrupciones.

Sin embargo, las interrupciones no son desencadenadas exclusivamente por dispositivos, sino que también pueden ser programadas por algún hilo de ejecución en algún determinado punto. Esto nos lleva a la siguiente clasificación de interrupciones:

- *Interrupciones de hardware (Hard IRQ):* Corresponden a las interrupciones generadas asíncronamente por algún dispositivo, como las antes descritas.
- *Interrupciones de software (Soft IRQ):* Se caracterizan por ser programadas por un hilo de ejecución del núcleo para ser ejecutadas en un futuro. Si bien no corresponden a interrupciones como tales, le deben su nombre a la capacidad de *reentrancia*[1] del código que ejecutan en contraste a otros mecanismos de ejecución programada del núcleo (como los *kworkers*), cuyo código no puede ser ejecutado simultáneamente por ningún CPU de la máquina.

Las interrupciones de software son utilizadas como el mecanismo para poder realizar una llamada a sistema en Linux, debido a que debe haber un cambio de modo de ejecución; elevar los privilegios de ejecución desde modo usuario hasta modo Kernel. Además, este tipo de interrupciones son usadas por el Kernel para programar tareas que se deberían hacer por un handler de interrupción de hardware, pero que, dada su complejidad, conviene posponerlas con el fin de agilizar la atención de la interrupción (ya que desactiva otras interrupciones en el procesador que atiende). Un ejemplo de esto último son los paquetes de red que entran al sistema; una explicación detallada de este proceso se verá más adelante.

1.5. Sockets de Linux

1.5.1. Introducción

Como antes se dijo, los sockets de Linux representan la interfaz que ofrece el Sistema Operativo para enviar mensajes a través de la red, mediante protocolos de comunicación previamente establecidos. Estos protocolos deben ser implementados en el núcleo del Sistema, para que los paquetes que sean recibidos o enviados puedan ser procesados en conformidad a su especificación.

[1]Del inglés *reentrancy*. Un código se dirá reentrante si tiene la capacidad de ser interrumpido en cualquier punto de ejecución, y ser ejecutado nuevamente de manera segura antes de completar la primera llamada.

Además de actuar de interfaz, los sockets son las estructuras del Kernel que mantienen todas las configuraciones de la conexión a la que sirven; parámetros como la dirección de destino, puerto de destino, protocolos en uso, cantidad de memoria máxima a usar, entre otros, deben estar almacenados en estas estructuras.

La organización de estos objetos y su funcionamiento, será analizado en las siguientes secciones.

1.5.2. Anatomía

Los sockets de Linux están implementados como estructuras dentro del Kernel definidas en el archivo `include/linux/net.h`. En esta estructura se definen campos como el estado, el tipo y algunos *flags* del socket (ver Figura 1.1). Sin embargo, los campos más relevantes de esta estructura son dos [4].

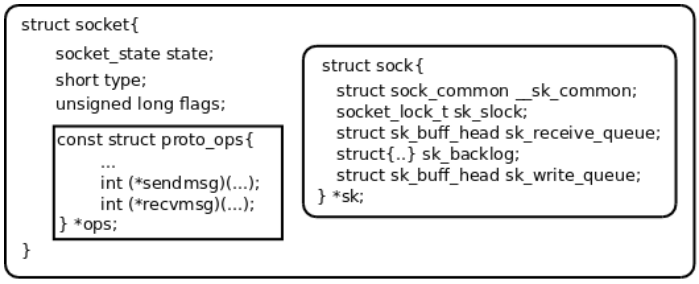

Figura 1.1: Definición de `struct socket`

- `struct proto_ops *ops`: Puntero a una estructura con funciones virtuales (puntero a los procedimientos reales) que actúa de interfaz, con el fin de adaptarse a las distintas implementaciones de lectura, escritura u otros procedimientos sobre el socket que dependan de las configuraciones de éste. En esta estructura se encuentran punteros a las implementaciones reales de `sendmsg`, `recvmsg`, entre otras.
- `struct sock *sk`: Puntero a una estructura de más bajo nivel, la cual contiene datos relativos a la conexión, las colas de paquetes, entre otras cosas.

Esta última es la que se estudiará a fondo, ya que contiene estructuras de datos que son modificadas constantemente por el Sistema Operativo al recibir o enviar paquetes desde o hacia la red. Asimismo, y para efectos prácticos, se ignorará la presencia de la estructura `socket`, debido a que presta una organización lógica más que funcional, por lo que nos referiremos al socket sólo considerando el contenido de la estructura `struct sock`.

El socket está conformado esquemáticamente por las siguientes componentes (ver Figura 1.2)

- Tres colas (`sk_receive_queue`, `sk_write_queue` y `sk_backlog`), implementadas a través de listas doble enlazadas con una cabecera. Estas colas están destinadas a almacenar paquetes de manera temporal dentro del socket. Sin ir más lejos, sus nombres sugieren la utilidad que presta cada una.

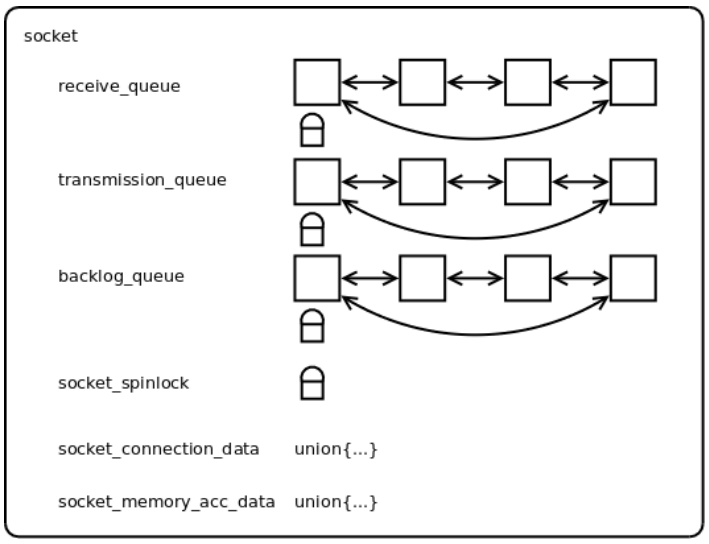

Figura 1.2: Representación de un socket Linux

- Una estructura de sincronización global (`sk_slock`), dentro de la cual se encuentra un spinlock global para el socket.
- Campos referidos a datos sobre la conexión que representa el socket, incluyendo información estadística de la conexión.
- Variables utilizadas para la característica de *Memory Accounting*.

Cada una de estas colas dentro del socket está protegida por un spinlock único para cada cola, residente en la cabecera de la lista enlazada (`struct sk_buff_head`). Esta primitiva de sincronización se debe utilizar para evitar modificaciones concurrentes de dos o más threads sobre la misma estructura de datos.

De la misma forma, el spinlock global del socket previene que las variables de configuración de la conexión e incluso los datos relacionados con el Memory Accounting, sean modificados de manera concurrente por dos o más hilos de ejecución.

Finalmente, la característica de Memory Accounting utilizada en el socket, será explicada más adelante con mayor detalle.

1.5.3. Funcionamiento

El funcionamiento general de los sockets de Linux se puede observar en la Figura 1.3; una imagen con mayor resolución, puede ser encontrada en [8]. Esta imagen muestra claramente el complejo funcionamiento de estos objetos en el núcleo de Sistema, mecanismo que será explicado en detalle en las próximas páginas.

De manera general, el socket funciona en dos sentidos: transmisión y recepción de paquetes. Para cada sentido de funcionamiento, el comportamiento de la estructura es ligeramente distinto, siendo la transmisión un poco más fácil de entender. Sin embargo, el trabajo expuesto en los siguientes capítulos se centra en la recepción de paquetes.

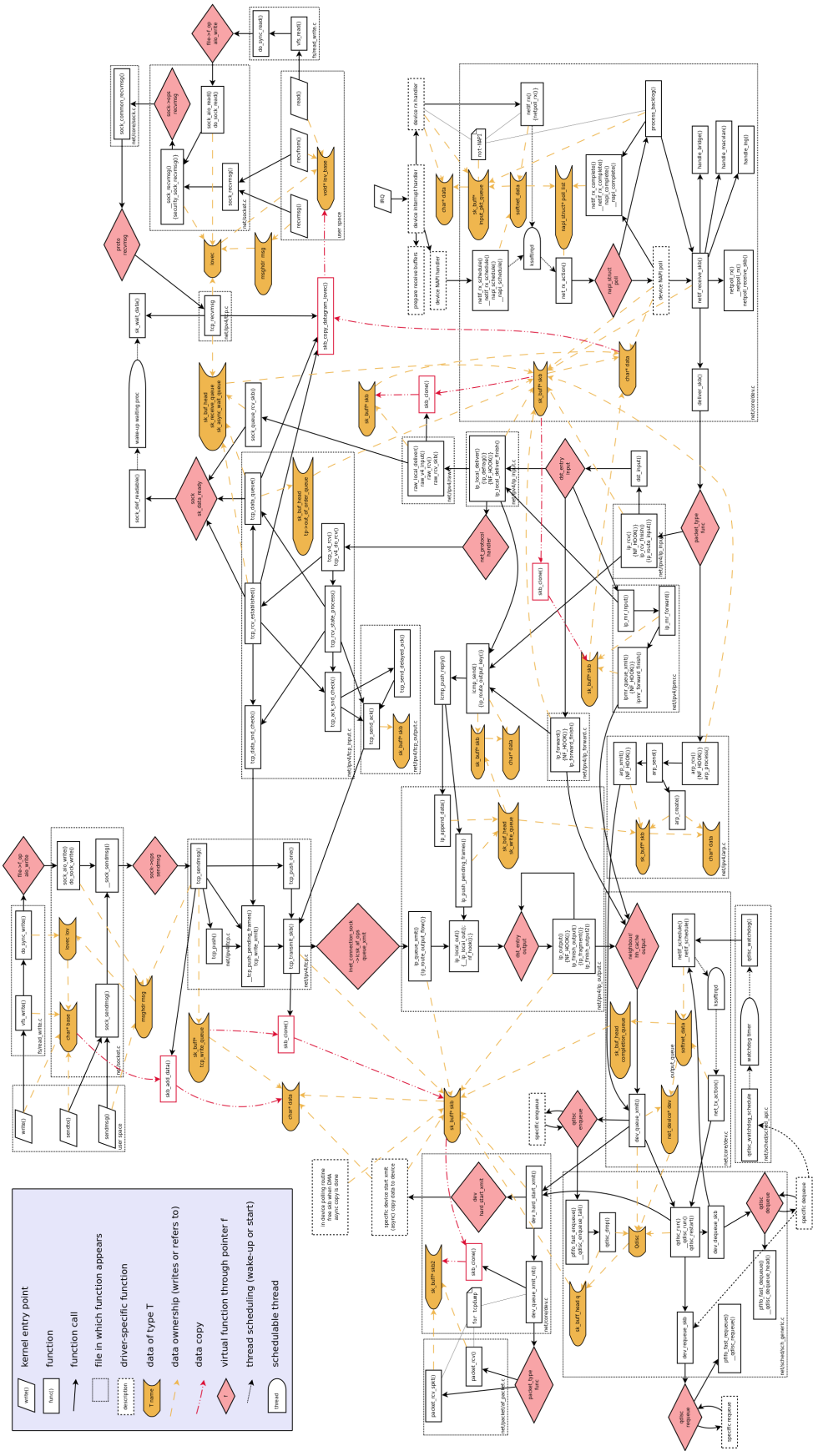

Figura 1.3: Flujo de datos desde y hacia la red

Transmisión de paquetes

Cada vez que una aplicación desea enviar un paquete a través de un socket (utilizando alguna llamada a sistema como `sendmsg` o `recvmsg`), la información deber ser llevada al espacio de memoria del Sistema Operativo para que sea procesada antes de ser enviado a la tarjeta de red.

Una representación del proceso completo se puede visualizar en la Figura 1.4 [7]

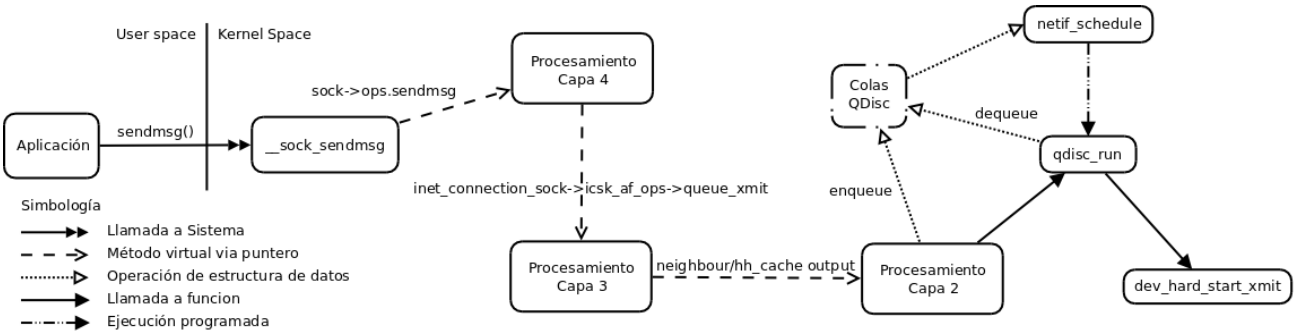

Figura 1.4: Representación del envío de un paquete

En esta figura se puede observar el siguiente flujo de información:

1. Cualquiera de las tres llamadas a sistema para enviar datos por un socket (`write`, `sendto` o `sendmsg`) terminará en una llamada a la función del sistema `__sock_sendmsg`, procedimiento que comprobará los permisos de envío, e invocará a la función virtual del socket `sock->ops->sendsmsg`, con el fin de que los datos sean procesados en las capas inferiores.
2. En el procesamiento de Capa 4, la información a enviar es copiada en memoria del Kernel; fragmentada en caso de que la información sea mayor al tamaño máximo de segmento; puesta en la cola de escritura del socket y ensamblados los headers de capa 4 de todos los fragmentos, si es que hay uno o más. Luego, cada fragmento se envía a la capa inferior a través de la llamada a función virtual `queue_xmit`.
3. En el procesamiento de Capa 3 se realiza el enrutamiento del paquete, el armado de los headers de Capa 3, el filtrado de paquetes (por firewall, NAT u otros mecanismos, los cuales podrían descartar el paquete), y el envío a la capa inferior a través de una llamada a una función virtual para que sea programada su transmisión mediante el uso de QDisc. Esta llamada virtual generalmente termina en la ejecución de la función `dev_queue_xmit`, la cual, a su vez, ejecuta `qdisc_run`, explicada en el paso siguiente.
4. La llamada a `dev_queue_xmit` inserta el paquete en alguna de las colas pertenecientes a QDisc, mecanismo por el cual el Kernel puede priorizar el envío de paquetes bajo determinadas condiciones de QoS (llamadas *Queue Disciplines*) con el fin de administrar correctamente el ancho de banda disponible. Una vez encolado el paquete, invoca a `qdisc_run`, función encargada de ser el punto de entrada para determinar el próximo paquete a enviar. Esta función desencola un paquete en caso de ser necesario, y lo

envía al driver de red para que sea enviado, a través de una invocación de la función `dev_hard_start_xmit`. Sin embargo, la acción de desencolamiento podría determinar que ningún paquete debe ser enviado por el minuto, debido a políticas de QoS. En este caso, se programa un reintento mediante una interrupción de software, a través de la ejecución de la función `netif_schedule`, la cual invocará nuevamente a `qdisc_run`. Finalmente, cuando el driver de red anuncia que ha terminado de enviar un paquete, se gatilla su destructor, liberando la memoria ocupada por él.

De esta descripción, podemos observar que el comportamiento del envío de un paquete es completamente síncrono desde el lado de la aplicación. Sin embargo, el envío efectivo a través de la tarjeta de red, podría ser retrasado por políticas de QoS, transformándolo en un proceso asíncrono en algunas ocasiones. Sin ir más lejos, este retraso del envío de la información es una decisión transparente para la aplicación que envía el mensaje, cuya llamada a sistema retorna en cuanto el núcleo encola el paquete en QDisc.

Recepción de paquetes

Como antes se dijo, la recepción de paquetes es un proceso que representa una mayor complejidad, por lo que será explicado con mayor profundidad en este apartado. Sin ir más lejos aún, debemos tener en cuenta que el procesamiento de un paquete entrante por parte del Sistema Operativo, es una tarea bastante pesada, la cual requiere de un tiempo de procesamiento considerable. Esto tiene ciertas repercusiones en el diseño del mecanismo de recepción de paquetes.

Este último se ilustra en la Figura 1.5, y consiste en los siguientes procedimientos, en concordancia con la tecnología NAPI de Linux (drivers más antiguos utilizan otros mecanismos de procesamiento de Capa 2):

1. La tarjeta de red genera una interrupción de hardware hacia la CPU, lo cual gatilla la ejecución del handler de interrupciones para el dispositivo. Este handler se encarga de llamar a la función `netif_rx_schedule`, la cual programa una nueva interrupción de software, agrega el dispositivo a la lista de aquellos que están listos para ser leídos, y termina la invocación del handler. La rápida ejecución de este método y la postergación del trabajo pesado del procesamiento de los paquetes, son decisiones que buscan aumentar el rendimiento del procedimiento completo, debido a que los métodos que atienden interrupciones de hardware no pueden ser desplazados de la CPU. Por este hecho, y sumado a que la recepción de un nuevo paquete implica mucho trabajo por parte del Kernel, se tomó la decisión de posponer la mayor cantidad de éste, dando lugar a lo que se conoce como *Controladores Bottom-Half.*
2. Una vez que el Kernel ha decidido ejecutar la interrupción de software programada en el paso anterior, se invoca el método `net_rx_action`, encargado de recorrer la lista de dispositivos listos para ser leídos y leer paquetes desde cada uno de ellos. Los métodos encargados de realizar esta lectura son implementados en el driver de la tarjeta de red instalada, y cumplen las funciones de: extraer los paquetes desde los buffers del dispositivo de red o cualquier espacio de memoria compartida entre éste y el núcleo; invocar a la función `netif_receive_skb`, encargada de preparar el paquete para ser

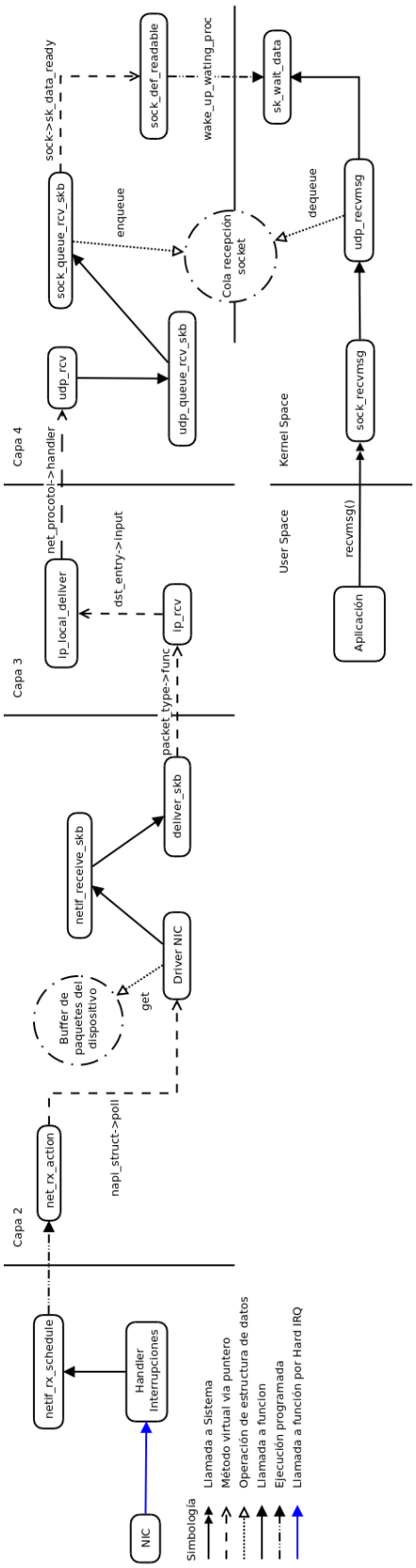

Figura 1.5: Representación de la recepción de un paquete

procesado por las capas superiores; y finalmente llamar a `deliver_skb` por cada uno de los paquetes extraídos, procedimiento que se encarga de invocar a través de una función virtual al handler del protocolo de capa 3 utilizado por el paquete. La determinación de qué protocolos utiliza el paquete y, en consecuencia, qué funciones virtuales serán llamadas para procesarlo correctamente, fueron determinadas durante la invocación de `netif_receive_skb`.

3. En el caso de la Figura 1.5, se trata de un paquete que utiliza el IP como protocolo de Capa 3, por lo que la función que procesa dichos headers es `ip_rcv`. En esta función se encarga de comprobar los headers L3, hacer circular el paquete por *netfilter*, y tomar las decisiones de enrutamiento del paquete en base a la información contenida en las cabeceras. Luego de estas acciones aplicadas por el núcleo, el paquete puede ser reenviado por la interfaz o despachado localmente, con lo cual se establece el valor de la función virtual `dst_entry->input`; en nuestro caso, el paquete fue despachado localmente, con lo que dicha función virtual apuntará y ejecutará `ip_local_deliver`, encargada de reensamblar un datagrama más grande en caso de que el paquete venga fragmentado, y de determinar el handler de Capa 4 que será invocado más adelante. Además, en esta función se establece si existe un socket de tipo *raw* abierto para este paquete, y, en caso de existir, el datagrama es encolado inmediatamente en las estructuras del socket respectivo.
4. Luego del procesamiento L3, se debe procesar la información contenida en las cabeceras de Capa 4, en este caso, con el protocolo UDP. El punto de entrada de este protocolo es la función `udp_rcv`, encargada de validar el paquete y encontrar el socket correspondiente al datagrama. Si no existe ningún socket para dicho paquete, éste es descartado. En caso contrario, se invoca a la función `udp_queue_rcv_skb`. Este método es el encargado de determinar el encolamiento del paquete: si es que irá a la cola *backlog*, en caso de estar bloqueado el socket por el lado del usuario, o a la cola de recepción como tal. En ambos casos, el socket es bloqueado completamente (mediante el uso del spinlock global de la estructura) antes de tomar esta determinación, siendo el encolamiento en la cola backlog el proceso más liviano de ambos. En el caso de que se haya decidido insertar el paquete en la cola de recepción, se invoca a la función `sock_queue_rcv_skb`, la cual se encarga de realizar chequeos de *netfilter*, actualizar las estadísticas de *Memory Accounting* del socket, insertar el paquete en la lista de recepción (tomando el spinlock respectivo de la estructura) y despertar a cualquier tarea que esté durmiendo a la espera de datos listos en el socket, mediante la invocación a la función virtual `sock->sk_data_ready`, realizando una llamada directa al scheduler de procesos de Linux. Finalmente, y luego de todos estos procesos, el bloqueo global del socket es liberado, y el procesamiento del paquete ha terminado.

Cabe destacar que en las siguientes páginas se analizará cómo se comporta el proceso de encolamiento de un paquete, estudiando principalmente el comportamiento de los spinlocks utilizados en el proceso de inserción un paquete en la cola de recepción.

Memory Accounting

Esta característica fue introducida (para los sockets UDP) en la versión 2.6.25 del Kernel de Linux [2], en conjunto con la nueva interfaz para hacer uso de esta característica.

Su principal función es la de mantener controlada y limitar, en caso de ser necesario, la memoria de la cual un socket puede llegar a hacer uso. Esto es requerido debido a que los paquetes (tanto para enviar como para recibir) son almacenados temporalmente en el espacio de memoria del Kernel, acotando la cantidad de memoria disponible para otras estructuras. Estos límites son configurados a través de `/proc/sys/net/ipv{4,6}/udp_mem` y representan el límite máximo de memoria que el Kernel utilizará (para sockets UDP) en almacenar paquetes; sobrepasado este límite, los paquetes serán descartados por falta de memoria.

Por otra parte, esta característica fue clonada desde la implementación existente en los sockets TCP, por lo que el sistema de bloqueo global de estos tuvo que ser replicado también. La decisión de incluir este mecanismo, junto con replicar el esquema de sincronización del protocolo TCP, introdujo un spinlock extra en las estructuras de los sockets, aumentando la latencia de los sockets UDP.

Evidencias de esto han sido observadas por la comunidad, llegando a producir parches para el núcleo que desactivan esta característica [3], sin que éstos hayan sido aceptados como modificaciones oficiales al stack de red.

Capítulo 2

Medición del rendimiento actual de sockets UDP

Con el fin de determinar la existencia del problema de rendimiento en los sockets UDP, fue aplicado un conjunto de pruebas a un Sistema Operativo moderno, el cual busca establecer el origen del problema en el núcleo de Linux.

2.1. Estrategia de las pruebas

Como antes se ha mencionado, existen antecedentes previos de que el problema puede tener su explicación en la implementación de los sockets en el Kernel de Linux. Evidencia de esto son los hechos de que al virtualizar en una misma máquina física y el uso de sockets distintos y paralelos para realizar conexiones, logran aumentar la cantidad de peticiones DNS que se logran atender.

Con estos antecedentes previos, y en conjunto con el hecho de que múltiples threads leyendo concurrentemente en un mismo socket no logran una ganancia de rendimiento, se puede inferir que el origen de este comportamiento se encuentra en el Kernel de Linux. Sin embargo, se deben descartar previamente otros posibles puntos de falla, como lo son la librería C y el mecanismo empleado por el núcleo para atender las llamadas al Sistema.

Cualquier aplicación que se pueda escribir en C para probar las llamadas a sistema (como por ejemplo `read`) debe tener en consideración la arquitectura presentada en la Figura 2.1.

Esta organización sugiere que los programas en C jamás generan llamadas al sistema de manera directa, sino que lo hacen a través de funciones *wrappers* disponibles en la librería estándar de C (*libc*). Un ejemplo de estas últimas son `read`, `open`, e incluso `socket`, las cuales terminan en ejecuciones de las funciones correspondientes de ellas dentro del núcleo.

Considerando esta arquitectura de capas de los sistemas Linux, se nos presenta el escenario en donde el origen del problema podría estar en la implementación de alguna primitiva para

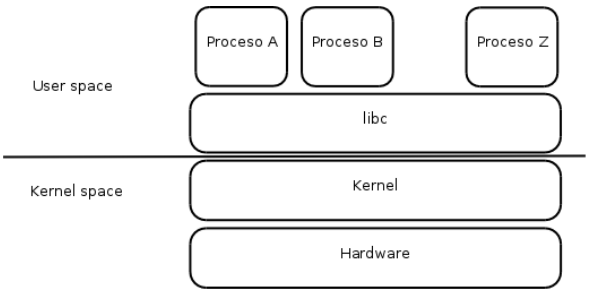

Figura 2.1: Arquitectura de procesos, libc y Kernel

leer sobre los sockets, como sería el caso de `recv`, `recvfrom`, `recvmsg` e incluso `read`.

Este hecho obliga a que el diseño de las pruebas tenga en consideración dicha arquitectura, con el fin de descartar la influencia de la capa intermedia introducida por la libc.

2.2. Metodología de las pruebas

Las mediciones realizadas persiguen tres objetivos principales:

- *Replicar los rendimientos observados,* mediante la lectura desde un socket con las primitivas de sistema más utilizadas.
- *Determinar la influencia de la libc en los rendimientos,* mediante la comparación del comportamiento de la lectura desde los sockets contra la lectura de otros orígenes dentro del sistema.
- *Determinar la influencia del mecanismo usado por el núcleo para atender llamadas al sistema,* con lo cual se termina de establecer la raíz del problema en el código de los sockets implementado en el núcleo del sistema.

2.2.1. Descripción de las pruebas

Con estos objetivos en mente, se diseñaron las siguientes pruebas que se aplicarán sobre una de las principales primitivas de sistema para recuperar datos:

- *Lectura desde un dispositivo virtual:* Con esto, se busca determinar la cantidad de llamadas a `read` que el Sistema Operativo es capaz de tolerar por unidad de tiempo, y cómo este rendimiento varía en función de la cantidad de threads concurrentes (sin ningún tipo de sincronización) intentando leer en el dispositivo.
- *Lectura desde un archivo FIFO:* Con la lectura desde una cola FIFO implementada en el Kernel, se intenta determinar el tiempo requerido por una aplicación para leer concurrentemente desde una estructura interna del núcleo sin ningún tipo de sincronización.
- *Lectura desde un socket UDP conectado en el loopback:* Esto busca replicar el comportamiento antes observado en los sockets, y obtener datos acerca de éstos para luego ser

comparados con los dos orígenes de datos anteriores.

- *Test de esfuerzo sobre una primitiva de sistema:* Sin lugar a dudas, el cambio de contexto que ocurre para atender una llamada a sistema por parte del núcleo puede representar la serialización de los accesos a esta porción de código. Esta prueba determina cómo se comporta este mecanismo ante usos concurrentes.

Todas estas pruebas consisten en tests de esfuerzo introduciendo threads a ejecutar accesos concurrentes en cada estructura o mecanismo, con el fin de sobrepasar la cantidad de procesadores disponibles en la máquina y producir escenarios en donde las contenciones de threads son esperables. Esto busca determinar el comportamiento de cada posible fuente del problema cuando el sistema se ve sobrepasado por ejecuciones concurrentes. En el caso del sistema utilizado, la emulación de esta situación corresponde a introducir ocho threads simultáneos, lo cual nos permite tener una aproximación de cómo se comportará cualquier sistema al sobrepasar sus límites físicos.

Sin ir más lejos, las tres primeras pruebas cuentan con las siguientes características comunes:

- Arquitectura global de *cliente-servidor*, consistente en un proceso escribiendo indefinidamente en la estructura.
- Unidades de información para ser leídas de 10 bytes cada una.
- Lectura de 500,000 unidades por cada hilo de ejecución.
- Registro del tiempo empleado por el programa para leer (sin procesar) esta cantidad de unidades de información.
- Medición del tiempo mediante el uso de la llamada a sistema `gettimeofday`.

Por el contrario, cada una de estas pruebas utiliza un mecanismo de transferencia de datos distinto, con el fin de comprar sus rendimientos en los mismos escenarios.

- Dispositivo virtual del Sistema Operativo. En esta prueba, se utilizó `/dev/zero` para obtener cada unidad de información.
- Cola FIFO de comunicación, la cual fue creada con la utilidad `mkfifo` de Linux. Esta herramienta instancia una cola FIFO en el núcleo del sistema, representándola por un archivo, la cual permite el paso de mensajes entre dos procesos mediante la escritura y lectura de dicho archivo.
- Socket UDP a través de la interfaz loopback. Esta estructura, similar a la anterior, permite el paso de mensajes entre dos o más procesos, presentando la sustancial diferencia que la información necesita ser empaquetada (cumpliendo todos los protocolos de red involucrados) al ser enviada y desempaquetada antes de ser leída por el consumidor.

Para esta última estructura, se utiliza la librería `jsockets` [15] con el fin de simplificar el código de los programas de test. Esta librería se caracteriza por ser una abstracción de todas las llamadas a primitivas de sistema necesarias para poder establecer correctamente el canal de comunicación entre los dos puntos.

Finalmente, la cuarta prueba caracterizada por el stress de una primitiva de sistema permite determinar el comportamiento del cambio de contexto al ingresar al núcleo para

atender una llamada al sistema. Para medir correctamente el tiempo empleado en los cambios de contexto, fue necesario implementar una nueva llamada al sistema en el Kernel de Linux 3.5.2.

Esta llamada consiste en retornar un valor fijo sin realizar más procesamiento, con lo que el tiempo de ejecución medido corresponde, principalmente, al tiempo que le toma al sistema cambiar el ambiente de ejecución desde modo usuario a modo privilegiado, y de vuelta a modo usuario.

Asimismo, y de modo de simplificar la forma de ejecutar dicho código introducido en el núcleo, se utiliza la función de la librería C `syscall`, la cual se encarga de actuar como *wrapper* del código ensamblador que realiza efectivamente el cambio de contexto y ejecuta el código en el Sistema.

Sin embargo, y dado que la primitiva implementada no realiza gran trabajo, el tiempo de ejecución de cada llamada es significativamente menor que el tiempo empleado en leer datos desde alguna fuente del Sistema Operativo, razón por la cual la prueba debe ser adaptada a este hecho. Con el fin de registrar el comportamiento asintótico del cambio de modo, es necesario medir lapsos de tiempo mayores, por lo que se decidió registrar el tiempo empleado por el programa para atender 15,000,000 de llamadas a dicha primitiva por thread de ejecución.

2.3. Especificaciones técnicas del equipo

Las características de la máquina en donde fueron ejecutados las pruebas antes descritas son:

- Sistema Operativo Fedora Core 17.
- Kernel Linux 3.5.2.
- Procesador Intel Core i5–2400 @ 3.10 GHz (4 núcleos).
- 16 GB memoria RAM DDR3 @ 1333 MHz.

Sin embargo, para las mediciones de lectura desde una cola FIFO y socket UDP, se utilizaron distintas versiones del Kernel, con el fin de detectar que el problema sigue presente incluso en versiones modernas.

2.4. Resultados de la ejecución

Una vez ejecutadas las pruebas se obtuvieron los siguientes resultados para las respectivas versiones del núcleo. En todos los casos, los tiempos expuestos están medidos en segundos.

2.4.1. Mediciones de lectura en dispositivo virtual

Threads	3.5.0–2
1	2.29
2	2.35
4	2.43
8	4.88

Tabla 2.1: Tiempo medido en las pruebas de dispositivo virtual en Kernel 3.5.0–2

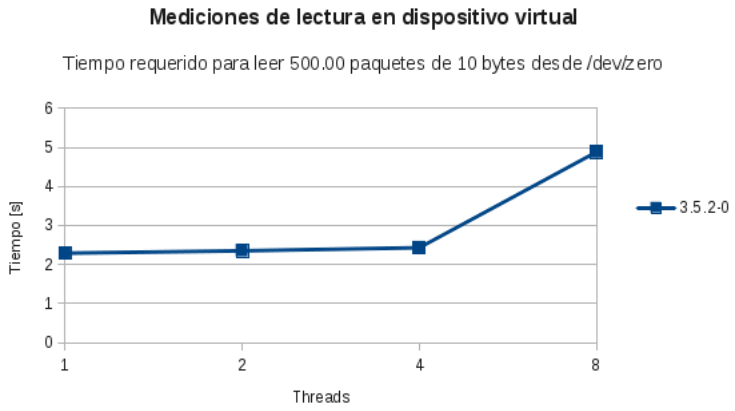

Figura 2.2: Gráfico de los resultados para dispositivo virtual

2.4.2. Mediciones de lectura en cola FIFO

Threads	2.6.24	2.6.35	2.6.38	3.4.6–2	3.5.0–2
1	0.2	0.17	0.17	0.58	0.55
2	0.4	0.30	0.53	1.93	1.80
4	3.3	0.98	2.20	5.47	5.70
8	15.1	6.50	11.10	16.00	14.30

Tabla 2.2: Tiempo medido en las pruebas de cola FIFO para distintas versiones del Kernel

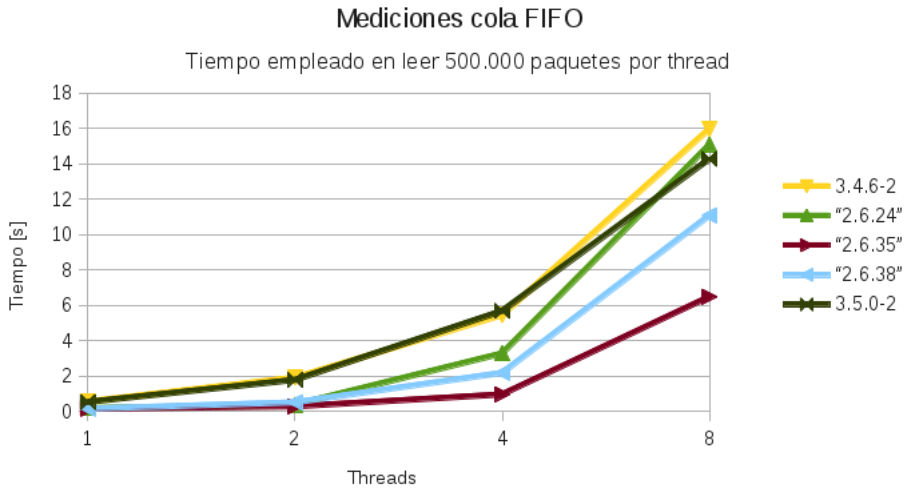

Figura 2.3: Gráfico de los resultados para cola FIFO

2.4.3. Mediciones de lectura en socket UDP

Threads	2.6.24	2.6.35	2.6.38	3.4.6–2	3.5.0–2
1	0.6	0.83	1.07	1.5	1.47
2	1.4	1.92	2.05	3.0	3.00
4	3.4	3.85	4.20	6.1	6.00
8	8.1	7.80	9.60	12.3	12.20

Tabla 2.3: Tiempo medido en las pruebas de socket UDP, para distintas versiones del Kernel

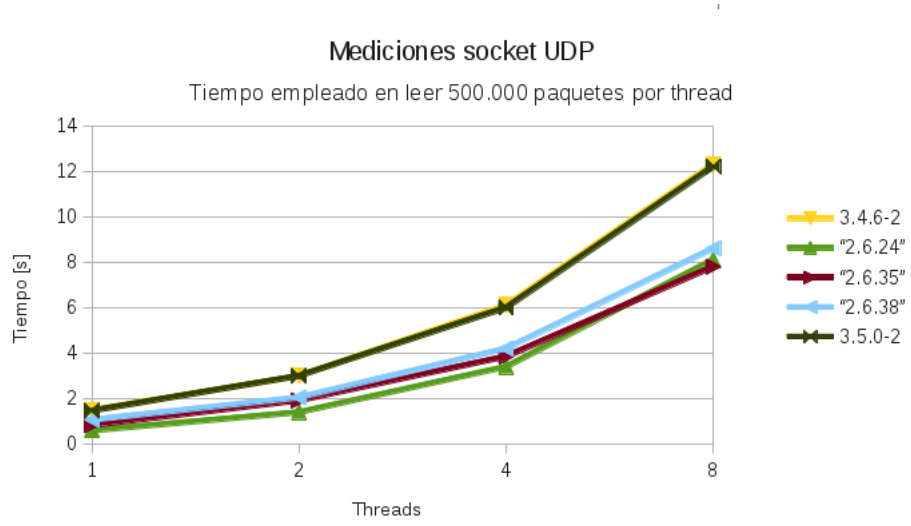

Figura 2.4: Gráfico de los resultados para socket UDP

Una vez modificado el Kernel 3.5.2, se procedió a medir la cantidad de cambios de modo que era capaz de soportar por unidad de tiempo, arrojando los resultados que se muestran

en la Tabla 2.4.

Threads	3.5.2
1	1.05
2	1.08
4	1.18
8	2.22

Tabla 2.4: Tiempo medido en las pruebas de cambio de modo en Kernel 3.5.2

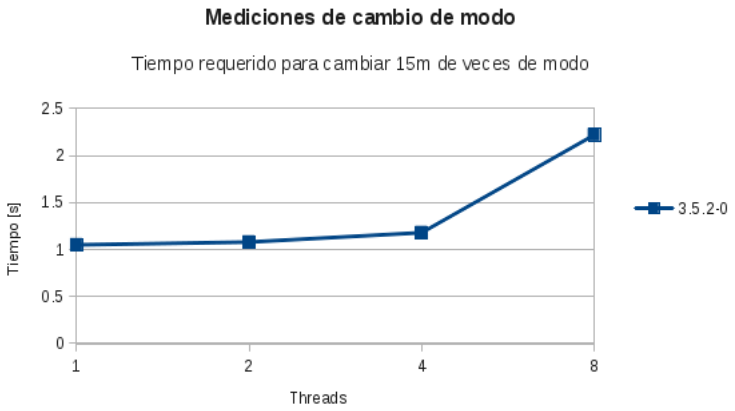

Figura 2.5: Gráfico de los resultados para el cambio de modo

2.5. Discusión de resultados

De la simple observación de los datos, podemos decir que los sockets UDP presentan un rendimiento muy por debajo de lo esperado, llegando a ser hasta un 529 % más costosos (en el Kernel 2.6.38) que una comunicación vía cola FIFO. Una de las posibles causas de este problema sería el sobrecosto de procesamiento que tiene el añadir los headers a los paquetes. Sin embargo, considerando que los headers de un paquete UDP son de 28 bytes (tomando en cuenta headers IP y UDP), en ningún caso se logra explicar este sobrecosto que multiplica por 6 los tiempos necesarios para procesar la misma cantidad de información.

Asimismo, núcleos más recientes (de la familia 3.X), mostraron rendimientos más bajos que los núcleos antiguos (familia 2.6.X). Este menor rendimiento se logra explicar mediante la introducción de *Memory Accounting* a los sockets UDP, característica que será analizada en mayor profundidad en las siguientes páginas. Sin ir más lejos, las observaciones más importantes se pueden extraer de las mediciones de lectura en dispositivo virtual y de la prueba de cambios de modo del sistema.

Con la lectura desde `/dev/zero` se puede observar que las llamadas a `read` se comportan como es de esperar ante la lectura concurrente de múltiples threads. Cada thread es capaz de leer paquetes de manera concurrente sin intervenir con otros, por lo que el tiempo necesario para leer 500,000 paquetes no varía considerablemente al aumentar la cantidad de hilos de

ejecución. Sin embargo, y como es de esperar, al aumentar la cantidad de threads al doble de la cantidad de procesadores disponibles, se estarán realizando el doble de lecturas paralelas al mismo tiempo, por lo que la mitad de los threads estará bloqueado esperando su turno para utilizar el procesador, decisión que pasa por el scheduler de Linux. Esto repercute en que el tiempo de lectura total para procesar la misma cantidad de paquetes por thread aumenta al doble: la mitad de ese tiempo están efectivamente leyendo y la otra mitad se encuentran bloqueados en alguna primitiva de sincronización del Kernel.

Esto nos permite deducir que por lo menos la implementación de `read` en la libc es correcta, y que es capaz de permitir que varios hilos de ejecución llamen concurrentemente a `read` sin interferirse mutuamente. Este es el comportamiento que las lecturas concurrentes en un socket UDP deberían tener para el caso de servidores DNS.

Asimismo, de las pruebas de cambio de modo usuario a modo sistema se puede desprender que la baja de rendimiento no se encuentra en la cantidad de ciclos de cambios de modo (user-Kernel-user), dado que el procesador es capaz de responder con, aproximadamente, 6,25 millones de llamadas al sistema por segundo por cada thread. Esto mismo nos dice que el Kernel es capaz de soportar llamadas a la misma primitiva de sistema (en este caso, una primitiva de prueba) de manera concurrente.

2.6. Conclusiones y desafíos

Con este análisis se puede establecer de manera concluyente que la serialización de accesos en los sockets UDP se encuentra dentro del Kernel de Linux, debido a que la librería es capaz de soportar llamadas concurrentes a `read` y se pudo establecer que el núcleo es capaz de procesar llamadas al sistema de manera concurrente. Para continuar aislando el problema se debe buscar, entonces, en la implementación de los sockets dentro del núcleo de Linux, analizando el uso de las estructuras de datos, los flujos de información y, principalmente, cuáles primitivas de sincronización se usan para mantener coherentes estas estructuras de datos y cómo éstas son utilizadas.

En las siguientes páginas se detalla cómo se procede la investigación dentro del núcleo de Linux, para continuar aislando la raíz de este comportamiento.

Capítulo 3

Aislamiento del problema en el Kernel

La continuación del estudio se debe realizar al interior del Kernel para identificar y aislar la fuente de la serialización de los accesos al socket. Sin embargo, esto no representa tarea sencilla dentro de un código conformado por quince millones de líneas.

3.1. Antecedentes

De lo observado en las mediciones anteriores, podemos deducir la existencia de la serialización dentro del Kernel del Sistema Operativo, comportamiento que debe ser aislado para su posterior análisis. Por otro lado, el preámbulo presentado en el Capítulo 1 permite entender cuál es el proceso que se ejecuta en la recepción de un paquete, así como los mecanismos de locking utilizados por el Kernel para serializar el acceso a las estructuras de datos compartidas.

Además, la naturaleza *multithreaded* de la situación antes expuesta nos lleva a pensar acerca del mal uso o funcionamiento de una primitiva de sincronización para estas estructuras, lo que puede requerir, en el peor de los casos, el cambio del tipo de primitiva utilizada para sincronizar dichos accesos, utilizando (y diseñando, en caso de ser necesario) una primitiva que se ajuste a los requerimientos del proceso de recepción de paquetes.

3.1.1. Análisis de la recepción de un paquete

De lo antes expuesto en el Capítulo 1, se puede determinar que el proceso de recepción es bastante independiente en sí mismo, en las cuales se reconocen claramente tres sub-procesos básicos:

1. Generación de la interrupción de hardware y programación de la interrupción de software.
2. Procesamiento del paquete a través del stack de red.

3. Encolamiento del paquete en alguna cola del socket Linux.

De estos sub-procesos, los primeros dos están ampliamente paralelizados y no requieren mayor optimización, llegando a ser el primero de ellos es un proceso asíncrono iniciado por la tarjeta de red en sí. Sin embargo, y dada la naturaleza consistente en modificar una estructura de datos compartida por ambos lados del socket, el encolamiento del paquete representa el proceso que debe ser llevado a cabo con mayor cautela de los tres, por lo que es necesario analizarlo en detalle.

De manera más refinada al análisis presentado en el Capítulo 1, el encolamiento de un paquete lo podemos desglosar en varias etapas:

1. Bloqueo completo de la estructura socket. Esto se hace mediante una llamada a la función `bh_lock_sock`, la cual es una macro C para bloquear el spinlock global del socket.
2. Invocar a `sock_owned_by_user`, determinando si el socket está bloqueado por alguna aplicación en espacio de usuario. En función de este último resultado, acciones distintas se toman en cada caso:
   (a) El socket está bloqueado por el usuario. En este caso, los paquetes son encolados directamente en la cola *backlog*, y serán movidos por el usuario a la cola de recepción en cuanto éste deje de utilizar el socket, es decir, después de la ejecución de cualquier método de lectura o escritura sobre el socket, pero antes de que dicho método retorne. Por cada paquete que el usuario procese, se aplicarán los mismos procesos descritos en el siguiente punto.
   (b) El socket no está bloqueado por el usuario. En este caso, se deben tomar las acciones pertinentes para este datagrama, las que incluyen:
      - Hacer pasar el paquete por netfilter, descartándolo en caso de ser necesario.
      - Actualizar las estadísticas mantenidas por la característica de *Memory Accounting* para sockets UDP.
      - Encolar el paquete en la cola de recepción, tomando los spinlocks correspondientes a dicha cola.
      - Invocar al scheduler de procesos del Sistema, con el fin de despertar a cualquier aplicación de usuario que esté esperando datos para ser leídos.
3. Finalmente, liberar el bloqueo global del socket, permitiendo que otro hilo de ejecución continúe procesando otro paquete correspondiente a la conexión.

Desde el otro lado, cuando una aplicación de usuario solicita datos al socket, deben realizar las siguientes acciones:

1. Obtener un paquete de la cola de recepción, tomando el lock de la cola de recepción de paquetes.
2. Copiar la información al espacio de memoria del usuario.
3. Liberar la memoria utilizada por el buffer de datos, mediante una llamada a la función `skb_free_datagram_locked`. Esta función se encarga de los siguientes procesos:
   (a) Bloquear el socket por parte del usuario, tomando el spinlock global sólo para cambiar el valor del estado como "bloqueado por el usuario", si es requerido. Esta

decisión es tomada por una llamada a `lock_sock_fast`.

(b) Invocar a la función `skb_orphan`, encargada de desvincular el paquete del socket correspondiente e invocar al destructor del buffer, en caso de existir.

(c) Actualizar las estadísticas de Memory Accounting para el socket, mediante una invocación a la función `sk_mem_reclaim_partial`.

(d) Tomar nuevamente el spinlock global, para volver el socket a su estado original, es decir, marcarlo como "no bloqueado por el usuario", en concordancia al bloqueo anterior. Este comportamiento se determina en `unlock_sock_fast`.

(e) Finalmente, se libera la memoria del paquete, mediante una llamada a la función `__kfree_skb`.

Este proceso de liberación del buffer de paquete es el que desactiva los controladores *bottom-half* en caso de ser necesario, existiendo dos tipos de "bloqueo" del socket posibles. La vía *rápida* se ejecuta cuando se produce la primera lectura por parte del usuario, marcando el socket como bloqueado del lado del usuario, y dejando bloqueado el lock global del socket, lo que no permite la ejecución de los controladores *botton-half*. Por otro lado, la vía *lenta* se produce cuando el socket ya ha sido bloqueado por otro thread desde el lado del usuario, habiendo al menos dos o más lecturas concurrentes en el mismo. Este proceso reprograma la ejecución de esta llamada (mediante una invocación a `schedule`), esperando a que el socket sea liberado por el otro hilo de ejecución. Cuando esto ocurre, el socket es marcado como bloqueado por el lado de la aplicación, y se activan los controladores *bottom-half*.

Esta serialización de los accesos se deben realizar con el fin de mantener controlada la cantidad de memoria que el socket está ocupando, clonando el sistema de bloqueo de los sockets TCP.

3.1.2. Posibles puntos de falla

El análisis de la recepción de un paquete antes expuesto, permite identificar tres posibles puntos que evitan la lectura concurrente de paquetes de un mismo socket:

- *Spinlock en la cabecera de la lista de paquetes:* Cada vez que se va a encolar un paquete, se debe proteger la estructura de datos de accesos concurrentes a ella. Esta sincronización se realiza mediante un spinlock residente en la cabecera de la lista doble enlazada, el cual es tomado sólo para modificar punteros dentro de la lista.
- *Spinlock global de la estructura del socket:* Cada vez que un paquete es encolado o desencolado de la lista de recepción, se deben actualizar las estadísticas para mantener acotada la cantidad de memoria utilizada por el socket. Esta sincronización se realiza mediante un spinlock global en el socket, el cual es utilizado, además, para contener los handlers *bottom-half*, limitando la cantidad de paquetes que están listos para ser leídos desde la aplicación por unidad de tiempo.
- *Esquema de bloqueo de sockets del lado de la aplicación:* Cuando un segundo thread intenta leer concurrentemente en el socket, esta operación se pospone hasta que la primera no ha terminado completamente su operación sobre la estructura, obligando a invocar al scheduler de procesos para que la tarea vaya a esperar a alguna cola.

De estas tres posibles fallas, la última de ellas es la que de manera más clara genera la serialización de los accesos al socket. Sin embargo, no es la única forma de solucionar el problema; en caso de cambiar el esquema de acceso al socket, el spinlock global del socket y el spinlock en la cabecera de la lista de recepción se convertirían en los principales puntos de contención de los hilos de ejecución. De este hecho se presentarán evidencias más adelante en este capítulo.

Es por esto, que el análisis se centrará en el rendimiento de estas primitivas de sincronización, posponiendo el cambio del esquema de acceso al socket para las futuras páginas.

3.2. Metodología de pruebas

Para determinar cual de los dos bloqueos representa la mayor barrera para el núcleo, se utilizará una herramienta que permita hacer *tracing* de cada una de las primitivas de sincronización del núcleo, en conjunto con otra que realice el correspondiente *profiling* del Kernel, con el fin de ubicar cuáles son los spinlocks más pesados y cuánto tiempo de CPU utilizan dichas primitivas.

3.2.1. Herramientas a utilizar

Lockdep

Lockdep es una herramienta imbuida en el mismo Kernel que permite agrupar spinlocks dentro de objetos denominados *clases*, las cuales pueden llegar a agrupar múltiples instancias del mismo spinlock. Estas clases de spinlocks permiten determinar el correcto del uso de estas primitivas de sincronización, mediante el seguimiento del estado de cada clase de lock, en conjunto con las dependencias entre clases de spinlocks que puedan existir.

Dentro de las estadísticas de estado de cada clase, esta herramienta es capaz de determinar la cantidad de veces que fue tomado un spinlock, tiempo total que estuvo tomado, número de veces que hubo contenciones, entre otros datos básicos. Esta información es complementada con el grafo de dependencia entre clases, el cual es primordial para que el programador evite deadlocks en el núcleo del Sistema Operativo, que podría conllevar a situaciones de bloqueo completo del sistema. Adicionalmente, lockdep es capaz de detectar la posibilidad de un deadlock, informando en tiempo real al programador a través de un mensaje en el log del sistema. Mayor información de esta herramienta, puede ser encontrada en [14, 21].

Sin embargo, para los fines de esta investigación, lockdep será utilizado para obtener estadísticas básicas acerca de los spinlocks antes mencionados, lo cual será útil para determinar el tiempo promedio que cada uno de los dos locks es tomado y el tiempo promedio de contención de un hilo de ejecución.

Para hacer esto posible fue necesario identificar los spinlocks dentro del Kernel y asignar

una clase a cada uno de ellos, con el fin de que sus estadísticas puedan ser recolectadas por lockdep y presentadas a través de las interfaces que ofrece la herramienta. En particular, los dos locks fueron asignados de la siguiente manera:

- *Spinlock global del socket:* Este lock ya poseía una clase lockdep, la cual se establece automáticamente cada vez que la estructura es inicializada. Esta clase es `AF_INET` para IPv4 y `AF_INET6` para IPv6.
- *Spinlock de la cola de paquetes:* Debido a que este lock no estaba considerado dentro de lockdep, fue necesario asignarle una clase dentro de su código e invocar al incializador de ella cuando la lista era creada, por lo que el núcleo del Sistema debió ser modificado con este fin. Se estableció la clase `skb_buffer_head_lock` para ser identificado en los registros finales.

Ftrace

Si bien lockdep provee estadísticas acerca de cómo son usados los spinlocks, no permite hacer *profiling* de las funciones del Kernel, información que será de gran utilidad para determinar si un trozo de código protegido por un spinlock es suficientemente atómico o no. Recordemos que estos mecanismos de sincronización se basan en *busy-waiting* para evitar accesos concurrentes, por lo que, por naturaleza, deben ser utilizados para sincronizar pequeñas porciones de código, debido a que el mecanismo de espera que utilizan mantiene la CPU ocupada ejecutando instrucciones que no son aprovechadas.

Es por esto que se determinará el tiempo que toma la ejecución del método `udp_recvmsg` mediante el uso de *ftrace* [17], con el fin de establecer el tiempo total que el socket permanece bloqueado desde el lado del usuario, evitando encolamientos en la lista de recepción de paquetes.

Si bien, el trabajo presentado aquí no tiene relación con la serialización de los accesos al socket por el lado del usuario, se intentará establecer que esta barrera es el primer punto por donde se debe comenzar al optimización. Sin embargo, relajar esta restricción será en vano en caso de mantener las estructuras de datos que manejan actualmente los sockets.

3.2.2. Mecanismo de pruebas

Con todo lo antes expuesto, es necesario diseñar pruebas que determinen cuál de las clases de lock está provocando el cuello de botella con los paquetes. Para esto, se realizarán los siguientes tests:

- *Prueba de stress del socket:* Para esto, se utilizó el mismo código de las pruebas hechas en el Capítulo anterior, pero esta vez se activó la funcionalidad de lockdep en el núcleo. Por esto último se espera que el tiempo de ejecución no sea el mismo que el anterior (debido al sobrecosto implícito de recolectar la información necesaria), a cambio de obtener estadísticas de cómo se comportan ambos locks.

- *Prueba de stress, desactivando Memory Accounting:* Dado que el lock general de la estructura del socket fue introducido por el Memory Accounting, se buscará eliminar dicho spinlock mediante la aplicación de un parche [3]. Al correr la prueba, se utilizará el mismo código anterior, activando lockdep en el Kernel. Estas estadísticas serán comparadas con aquellas extraídas del test previo para analizar ganancias o perdidas de rendimiento.
- *Prueba de stress, con Memory Accounting Activado, pero con accesos al socket "sincronizados":* El análisis del mecanismo de recepción de un paquete sugiere que el spinlock que protege el socket completo es un potencial cuello de botella, por lo cual se repetirá la primera prueba esta vez sincronizando los accesos al socket. Para esto se inducirá un retardo entre cada llamada a `read` desde el lado del usuario, el cual dependerá del tiempo promedio que permanece bloqueado el spinlock de la estructura. Este dato se obtendrá experimentalmente de las pruebas anteriores, y dependerá de la máquina en donde se ejecuten los tests. Además, estas pruebas se realizarán en un esquema en donde existe solo un thread leyendo desde el socket, con el fin de evitar accesos concurrentes a la estructura, lo que permitirá determinar con mayor certeza el punto de falla de la estructura.
- *Prueba de stress, midiendo tiempos de ejecución de* `udp_recvmsg`*:* La primera prueba se ejecutará nuevamente, esta vez utilizando *ftrace* en modalidad *dynamic function graph tracer*, configurando esta herramienta para que determine los tiempos de ejecución de `udp_recvmsg` y de todas las funciones de las cuales hace uso. Estas pruebas se realizarán sobre un Kernel sin haber sido modificado.

Las pruebas antes descritas fueron ejecutadas en el mismo sistema utilizado para las pruebas del capítulo anterior:

- Sistema Operativo Fedora Core 17.
- Kernel Linux 3.5.2.
- Procesador Intel Core i5–2400 @ 3.10 GHz (4 núcleos).
- 16 GB memoria RAM DDR3 @ 1333 MHz.

3.3. Resultados obtenidos

Luego de ejecutar las pruebas de stress nuevamente con el rastreo de lockdep activado, se extrajeron datos desde los registros, los cuales se presentan en las Tablas 3.1 a 3.10 y las Figuras 3.1 a 3.6(todos los tiempos están medidos en μs)[1,2]:

[1]Para las tablas 3.1 a 3.10, las abreviaciones son las siguientes:
Hldtime-tot: Tiempo total que el spinlock estuvo bloqueado.
Acq: Número de adquisiciones del spinlock.
Hldtime-avg: Tiempo promedio que el spinlock estuvo bloqueado.
Wttime-tot: Tiempo total que se debió esperar por un spinlock bloqueado.
Cont: Número de contenciones del spinlock.
Wttime-avg: Tiempo promedio que se esperó por un spinlock bloqueado.

[2]En las figuras 3.1 a 3.6, las series con sufijo *sk* representan los datos de spinlock global del socket. Asimismo, las series con sufijo *head* representan los datos del spinlock en la cabecera de la lista de paquetes.

3.3.1. Pruebas de stress

Threads	Hldtime-tot	Acq	Hldtime-avg	Wttime-tot	Cont	Wttime-avg
1	1727674.73	1000102	1.7274985252	122.21	222	0.5504954955
2	3546050.22	2000037	1.7729923096	1417.54	1235	1.1478056680
4	7253937.17	4000143	1.8134194628	3163.35	2576	1.2280085404
8	14511741.72	8000053	1.8139556975	5301.16	4281	1.2382994627

Tabla 3.1: Resultados lockdep para lock `AF_INET` usando protocolo IPv4

Threads	Hldtime-tot	Acq	Hldtime-avg	Wttime-tot	Cont	Wttime-avg
1	241739.56	1498456	0.1613257647	14.69	72	0.2040277778
2	491475.11	3000009	0.1638245452	83.76	427	0.1961592506
4	999715.69	6001903	0.1665664523	193.33	956	0.2022280335
8	2001099.30	12002304	0.1667262636	300.51	1479	0.2031845842

Tabla 3.2: Resultados lockdep para lock `sk_buffer_head_lock` usando protocolo IPv4

Threads	Hldtime-tot	Acq	Hldtime-avg	Wttime-tot	Cont	Wttime-avg
1	1738910.81	1000043	1.7388360401	387.39	623	0.6218138042
2	3509948.38	2000030	1.7549478658	1522.71	1404	1.0845512821
4	7320601.39	4000036	1.8301338763	3843.34	3049	1.2605247622
8	14378248.95	8000063	1.7972669653	5980.84	4766	1.2548971884

Tabla 3.3: Resultados lockdep para lock `AF_INET6` usando protocolo IPv6

Threads	Hldtime-tot	Acq	Hldtime-avg	Wttime-tot	Cont	Wttime-avg
1	248990.61	1496438	0.1663888581	36.09	188	0.1919680851
2	492992.92	3000372	0.1643105988	99.88	487	0.2050924025
4	1012625.99	6002174	0.1687098691	214.65	1041	0.2061959654
8	2120605.52	12003191	0.1766701471	346.04	1687	0.2051215175

Tabla 3.4: Resultados lockdep para lock `sk_buffer_head_lock` usando protocolo IPv6

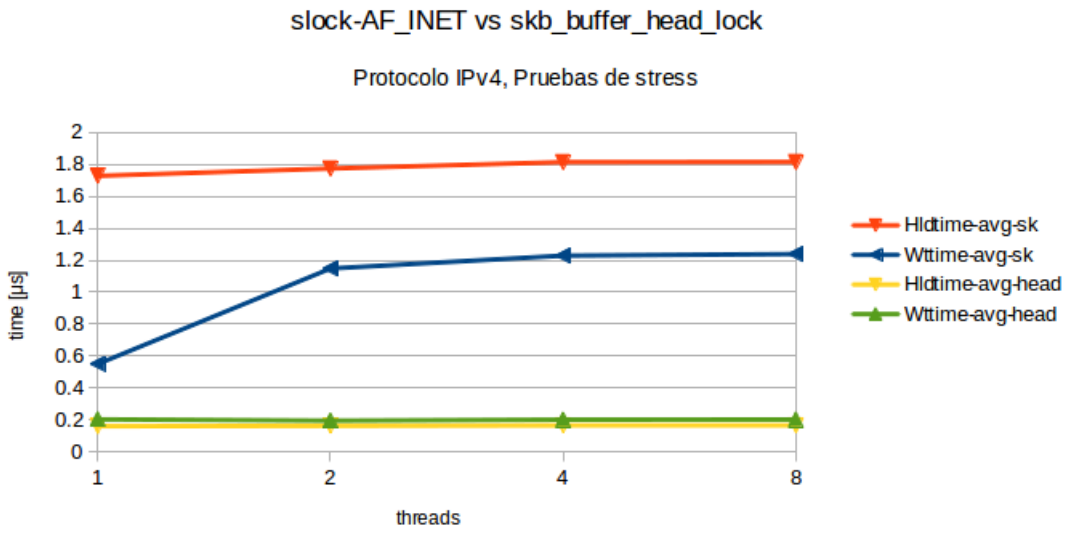

Figura 3.1: Comparación de locks en IPv4

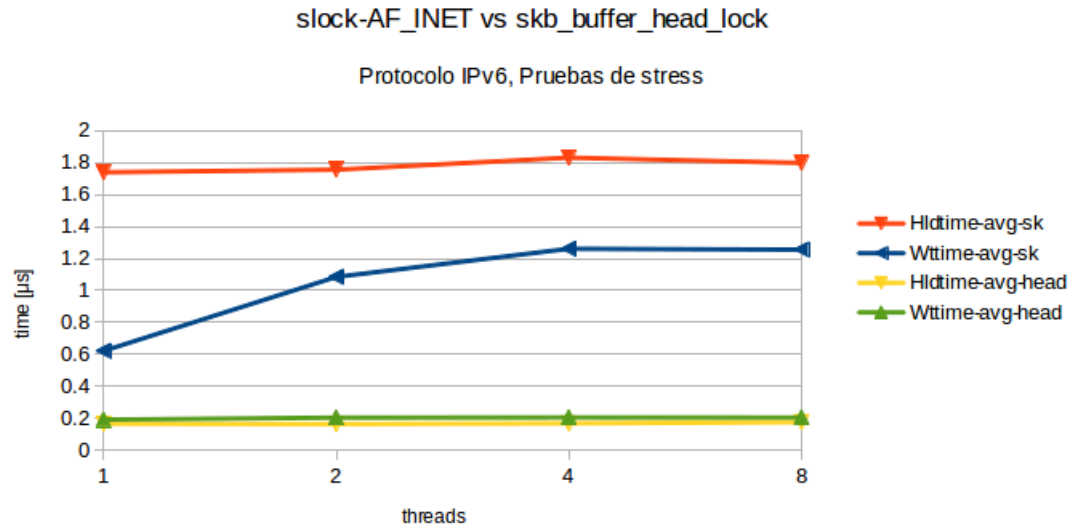

Figura 3.2: Comparación de locks en IPv6

3.3.2. Prueba de stress, Memory Accounting desactivado

Threads	Hldtime-tot	Acq	Hldtime-avg	Wttime-tot	Cont	Wttime-avg
1	475795.94	1995714	0.2384088802	173.13	543	0.3188397790
2	983877.62	4000453	0.2459415521	677.32	1743	0.3885943775
4	1988615.96	8004417	0.2484398252	1147.46	3679	0.3118945366
8	3976472.06	16007064	0.2484198264	2429.32	8650	0.2808462428

Tabla 3.5: Resultados lockdep para lock `sk_buffer_head_lock` usando protocolo IPv4, sin Memory Accounting

Threads	Hldtime-tot	Acq	Hldtime-avg	Wttime-tot	Cont	Wttime-avg
1	477019.75	1997968	0.2387524475	86.37	271	0.3187084871
2	980625.64	3999802	0.2451685458	705.58	1999	0.3529664832
4	1970749.60	8004050	0.2462190516	1070.58	3319	0.3225610124
8	3971759.58	16006516	0.2481339212	1485.62	5062	0.2934847886

Tabla 3.6: Resultados lockdep para lock `sk_buffer_head_lock` usando protocolo IPv6, sin Memory Accounting

Para extraer mejores conclusiones, estos datos se grafican en conjunto a los de la prueba anterior, comparando los rendimientos del lock global del socket con los de este spinlock bajo el escenario sin Memory Accounting (las series marcadas con el sufijo *sk* representan los resultados de las Tablas 3.1 y 3.3):

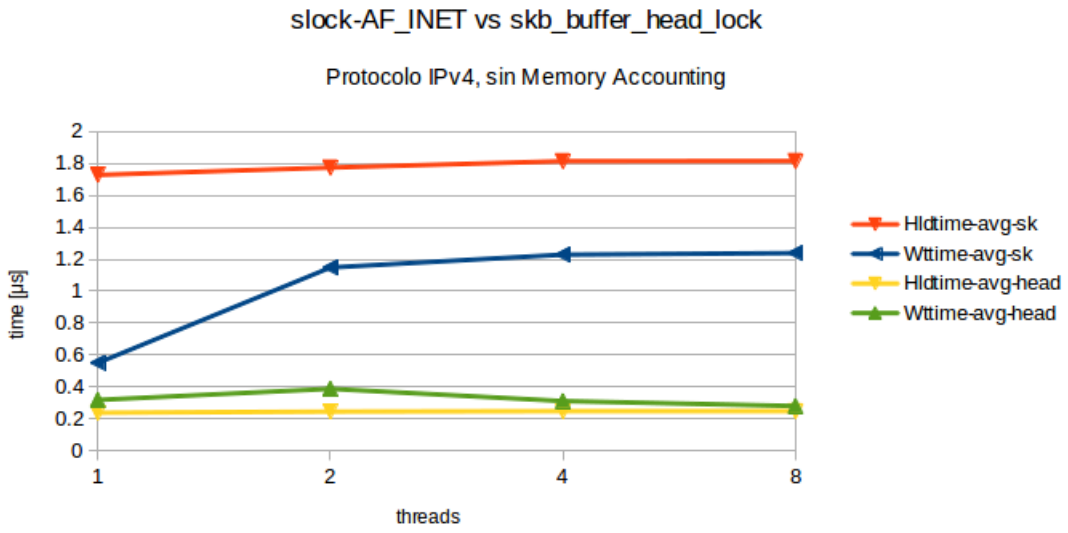

Figura 3.3: Comparación de locks con y sin Memory Accounting para IPv4

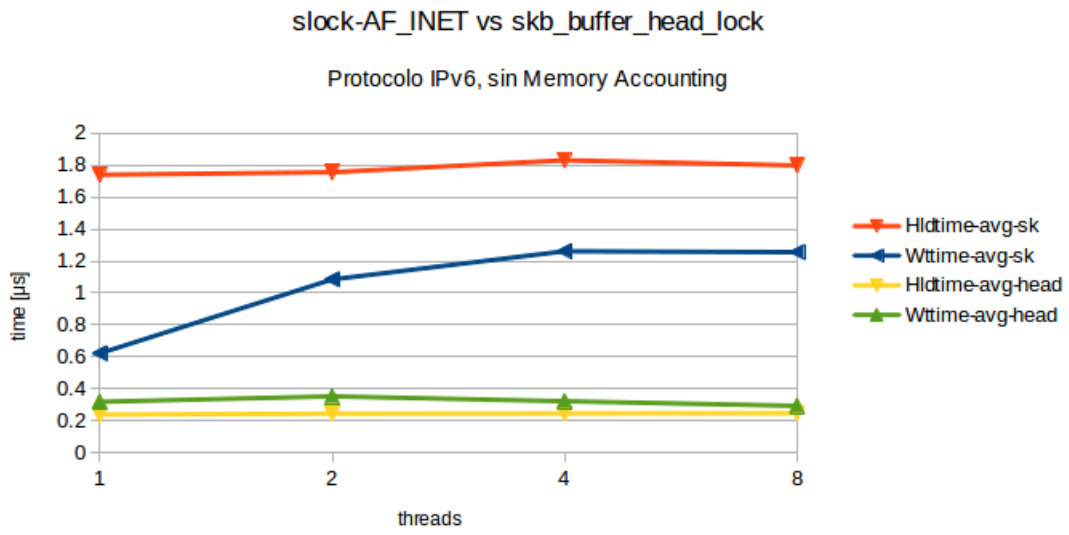

Figura 3.4: Comparación de locks con y sin Memory Accounting para IPv6

3.3.3. Prueba de stress, accesos sincronizados

Ejecución	Hldtime-tot	Acq	Hldtime-avg	Wttime-tot	Cont	Wttime-avg
1	667115.03	1000350	0.6668816214	40.79	42	0.9711904762
2	648883.52	1000349	0.6486571387	45.70	36	1.2694444444
3	682023.38	1000317	0.6818072471	25.08	23	1.0904347826
4	668093.97	1000333	0.6678715688	33.43	32	1.0446875000
5	663543.47	1000678	0.6630938923	25.05	28	0.8946428571

Tabla 3.7: Resultados lockdep con accesos sincronizados para lock `AF_INET` usando protocolo IPv4

Ejecución	Hldtime-tot	Acq	Hldtime-avg	Wttime-tot	Cont	Wttime-avg
1	164253.34	1000579	0.1641582923	0.38	1	0.3800000000
2	161873.55	1000607	0.1617753524	0.43	1	0.4300000000
3	162873.87	1000519	0.1627893823	0.70	1	0.7000000000
4	163454.24	1000567	0.1633616140	1.07	3	0.3566666667
5	163265.32	1000572	0.1631719856	1.55	4	0.3875000000

Tabla 3.8: Resultados lockdep con accesos sincronizados para lock `sk_buffer_head_lock` usando protocolo IPv4

Ejecución	Hldtime-tot	Acq	Hldtime-avg	Wttime-tot	Cont	Wttime-avg
1	663778.68	1000304	0.6635769526	31.22	39	0.8005128205
2	664589.32	1000282	0.6644019586	46.99	50	0.9398000000
3	671537.47	1000318	0.6713239890	11.73	22	0.5331818182
4	666654.67	1000294	0.6664587311	35.21	40	0.8802500000
5	665620.93	1000304	0.6654186427	32.31	33	0.9790909091

Tabla 3.9: Resultados lockdep con accesos sincronizados para lock `AF_INET6` usando protocolo IPv6

Ejecución	Hldtime-tot	Acq	Hldtime-avg	Wttime-tot	Cont	Wttime-avg
1	162530.00	1000642	0.1624257227	0.97	2	0.4850000000
2	162126.78	1000614	0.1620272952	1.02	3	0.3400000000
3	165162.01	1000763	0.1650360875	0.82	3	0.2733333333
4	163543.87	1000634	0.1634402489	1.97	5	0.3940000000
5	187914.85	1000754	0.1877732690	0.78	2	0.3900000000

Tabla 3.10: Resultados lockdep con accesos sincronizados para lock `sk_buffer_head_lock` usando protocolo IPv6

Estos datos se visualizan de la siguiente manera:

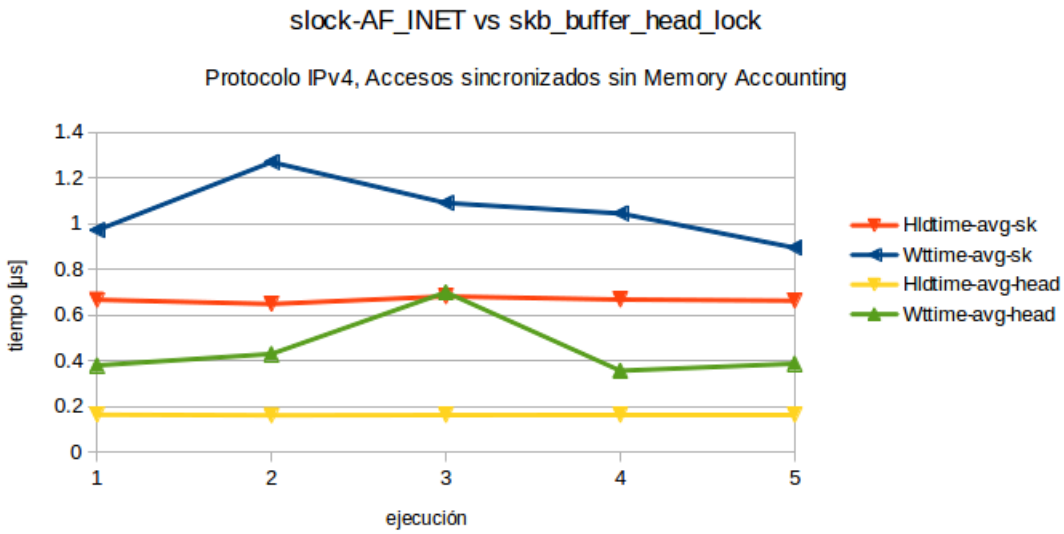

Figura 3.5: Comparación de locks con accesos sincronizados en IPv4

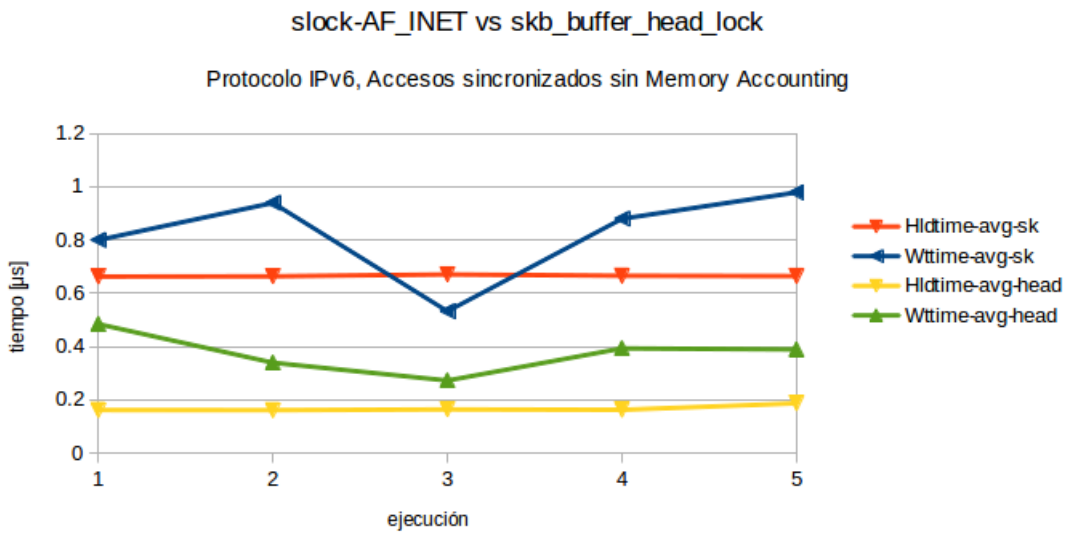

Figura 3.6: Comparación de locks con accesos sincronizados en IPv6

3.3.4. Prueba de stress, tiempos de ejecución de udp_recvmsg

En los logs de ftrace, se pueden encontrar entradas del tipo:

```
1)               |  udp_recvmsg() {
1)               |    __skb_recv_datagram() {
1)   0.267 us    |      _raw_spin_lock_irqsave();            /* (1) */
1)   0.132 us    |      _raw_spin_unlock_irqrestore();       /* (1) */
1)   0.986 us    |    }
1)               |    skb_copy_datagram_iovec() {            /* (2) */
1)               |      memcpy_toiovec() {
1)               |        might_fault() {
1)               |          __might_sleep() {
1)   0.033 us    |            debug_lockdep_rcu_enabled();
1)   0.030 us    |            debug_lockdep_rcu_enabled();
1)   0.031 us    |            debug_lockdep_rcu_enabled();
1)   0.962 us    |          }
1)   0.029 us    |          _cond_resched();
1)   1.764 us    |        } /* might_fault */
1)   2.082 us    |      } /* memcpy_toiovec */
1)   2.372 us    |    } /* skb_copy_datagram_iovec */
1)               |    skb_free_datagram_locked() {           /* (3) */
1)               |      lock_sock_fast() {                   /* (4) */
1)               |        __might_sleep() {
1)   0.026 us    |          debug_lockdep_rcu_enabled();
1)   0.031 us    |          debug_lockdep_rcu_enabled();
1)   0.033 us    |          debug_lockdep_rcu_enabled();
1)   1.085 us    |        }
```

```
1)   0.021 us    |        _cond_resched();
1)               |        _raw_spin_lock_bh() {              /* (5) */
1)               |          local_bh_disable() {
1)   0.050 us    |            __local_bh_disable();
1)   0.341 us    |          }
1)   0.871 us    |        }
1)   2.792 us    |      } /* lock_sock_fast */
1)   0.032 us    |      sock_rfree();
1)   0.047 us    |      __sk_mem_reclaim();
1)               |      _raw_spin_unlock_bh() {              /* (6) */
1)   0.080 us    |        local_bh_enable_ip();
1)   0.520 us    |      }
1)               |      __kfree_skb() {                      /* (7) */
1)   0.024 us    |        skb_release_head_state();
1)               |        skb_release_data() {
1)               |          kfree() {
1)   0.023 us    |            __phys_addr();
1)   0.032 us    |            kmemleak_free();
1)   0.045 us    |            __slab_free();
1)   1.017 us    |          }
1)   1.320 us    |        } /* skb_release_data */
1)               |        kmem_cache_free() {
1)   0.030 us    |          __phys_addr();
1)   0.030 us    |          kmemleak_free();
1)   0.041 us    |          __slab_free();
1)   0.963 us    |        }
1)   3.119 us    |      } /* __kfree_skb */
1)   7.865 us    |    } /* skb_free_datagram_locked */
1) + 12.069 us   |  } /* udp_recvmsg */
```

Estos registros entregan la siguiente información en cada columna, de izquierda a derecha:

1. Identificador del procesador que ejecuta la función.
2. Tiempo medido que tarda la ejecución de la función en cuestión. Si este valor sobrepasa los 10 μs, un signo **+** denota este hecho. Si esta valor sobrepasa los 100 μs, un signo **!** precede esta columna. Además, este valor se muestra en las llaves que cierran el bloque o en la mima línea si la ejecución es una hoja del grafo de funciones.
3. Nombre de la función que se ejecuta y mide, en notación estilo C.

3.4. Discusión

De la primera prueba de stress, se observa claramente que el lock global del socket representa un gran punto de overhead en comparación con el lock de la cola de paquetes, siendo 9 veces mayor el tiempo que está bloqueado por algún hilo de ejecución en comparación con el spinlock de la cola. Este resultado se repite tanto en el protocolo IPv4 como IPv6. De manera

adicional, con la simple observación de los registros en bruto de lockdep se logra identificar a este lock como uno de los más pesados de todo el núcleo de Linux (justo por debajo de un spinlock relacionado con el scheduler el Sistema).

Paralelamente, al aplicar el parche del Kernel que desactiva la característica de Memory Accounting, se puede observar que el spinlock de la lista toma toda la responsabilidad de mantener coherente la estructura de datos compartida en el socket, lo cual se observa como un leve aumento en el tiempo promedio que este lock permanece tomado por algún hilo de ejecución. Este fenómeno se visualiza en los logs de lockdep clasificando a este spinlock como uno de los más pesados en todo el núcleo. Asimismo, se puede observar cómo la introducción de lockdep aumentó los tiempos de latencia de todas las comunicaciones UDP; la introducción de esta característica requirió la clonación del mecanismo de sincronización de los sockets TCP, por lo que los tiempos que éstos permanecían bloqueados sin poder entregar mayor información crecieron. Esto se deduce al observar las figuras 3.3 y 3.4, en donde el lock introducido para este fin tiene un tiempo promedio de uso mayor que el de la lista, lo que hace que el proceso completo de lectura de un paquete desde el socket tarde más tiempo.

Por otra parte, con el simple hecho de sincronizar los requerimientos de información desde el socket se logra bajar el costo del spinlock global del socket, e incluso, bajar drásticamente el número de contenciones que esta barrera generó, logrando hacer comparables estas mediciones con las del spinlock perteneciente a la cabeza de la lista de paquetes.

De manera adicional los logs de ftrace midieron los tiempos de ejecución del método `udp_recvmsg`, encargado desencolar los paquetes del socket y copiarlos al buffer de memoria en el espacio de la aplicación. En `(1)` podemos observar cómo es utilizado, como es de esperar, por breves instantes el spinlock de la lista de paquetes, siendo liberado de forma casi instantánea. En `(2)` se observa como la información es copiada al espacio de memoria del usuario, mediante la invocación a `skb_copy_datagram_iovec`. Luego, en `(3)` se realiza una llamada para liberar la memoria en el espacio del núcleo asociada al paquete. Sin embargo, y dado el Memory Accounting, el socket debe ser bloqueado mediante `lock_sock_fast (4)`, invocación que podría dormir el hilo de ejecución, por lo que el bloqueo del spinlock se pospone hasta `(6)`, justo después de haber determinado si la aplicación debe esperar más paquetes o no. Más adelante, la invocación a `sock_rfree` y `__sk_mem_reclaim` permiten actualizar las estadísticas de memoria utilizada por el socket; estos métodos son los principales usados por Memory Accounting desde el lado de la aplicación, y es la razón por la cual debe existir el spinlock general del socket. Una vez finalizado este proceso, dicho lock es liberado en `(6)` para que luego, finalmente, el núcleo recupere la memoria invocando a `__kfree_skb`, lo cual se refleja en `(7)`.

3.5. Conclusiones

Todos estos resultados a la vista sugieren que el esquema de encolamiento de paquetes en una lista es el problema que se persigue. En otras palabras, la estructura de datos utilizada por el núcleo para encolar los datagramas provenientes de Internet (en este caso, una lista doble enlazada protegida por un spinlock), no es capaz de soportar eficientemente accesos

concurrentes para desencolar paquetes de información. La eliminación del lock global del socket no soluciona el problema, lo cual se prueba con la ejecución sobre un Kernel sin Memory Accounting, en donde el spinlock perteneciente a la lista de paquetes resultó ser el punto de serialización.

Por esta razón se requiere la implementación de una estructura de datos nueva para ser utilizada en el socket, esto último, sin perjuicio de la eliminación del Memory Accounting para sockets UDP.

Además, como antes se dijo, la función `udp_recvmsg` contiene un mecanismo de serialización de accesos concurrentes al socket, el cual debe ser relajado para soportar lecturas concurrentes desde el lado del usuario.

Capítulo 4

Modelamiento y Solución Propuesta

Una vez que se logró determinar la causa de la serialización de las lecturas desde el socket, se procedió a proponer una solución que mejora la situación actual. Esta solución debe ser implementada dentro del núcleo de Linux, el cual se caracteriza por ser un Kernel monolítico, por lo que el tiempo de compilación y depuración del código introducido hace que el proceso de implementación se alargue más de lo necesario.

Por esta razón, fue necesario elaborar un modelo del funcionamiento de los sockets de Linux, replicando las estructuras de datos utilizadas por el núcleo y los mecanismos de sincronización sobre éstas en un programa C.

Esta simplificación permite introducir cambios en el esquema de encolamiento de paquetes de manera simple, evitando los largos tiempos de compilación, ya que el modelo se encuentra implementado en un proceso Linux que emula las pruebas de stress.

4.1. Modelamiento actual del núcleo

El modelo elaborado se caracteriza por correr en un proceso Linux, e introduce la facilidad de compilación y modificación. Asimismo, está diseñado para emular las pruebas de stress sobre la estructura de socket emulada, por lo que se utiliza para determinar el comportamiento de otros esquemas de encolamiento de paquetes.

4.1.1. Arquitectura del modelo

El modelo presenta la arquitectura que se muestra en la Figura 4.1. Su diseño se basa en el problema de productores y consumidores, en donde se utiliza algún mecanismo entre los participantes (utilizado internamente por el socket) para hacer fluir los datos desde los productores hasta los consumidores. En el caso del socket, los productores son los threads del núcleo que procesan los paquetes a través de las capas del modelo OSI y colocan la información

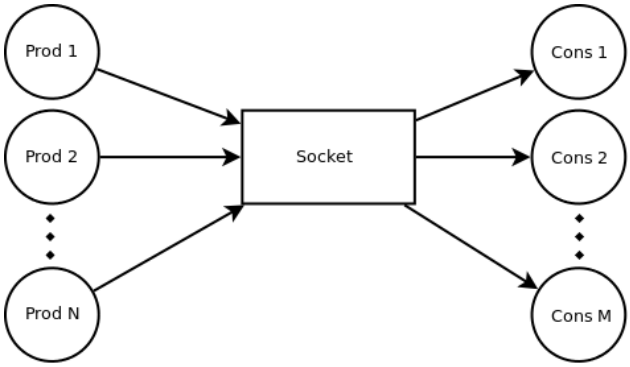

Figura 4.1: Arquitectura del modelo implementado

en la cola de recepción del socket. De la misma manera, los consumidores representan a los threads por el lado de la aplicación que acceden a datos disponibles en el socket mediante una llamada a primitivas de sistema como `read`, `recvmsg`, entre otras.

En el caso inicial, se modela el funcionamiento actual del socket de Linux, por lo que las estructuras utilizadas por el socket para el intercambio de información entre productores y consumidores tienen que ser replicadas, como se visualiza en la Figura 4.2.

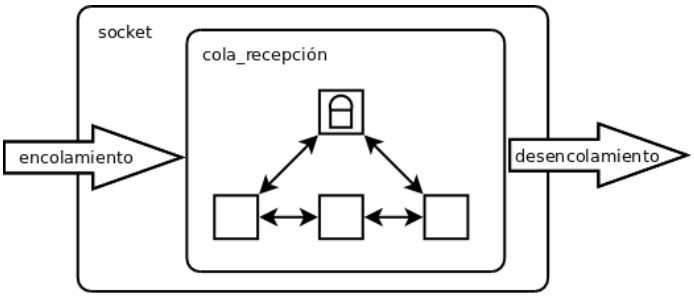

Figura 4.2: Modelamiento del socket en user space

Esta simplificación posee las siguientes características:

- *Uso de pthreads para emular hilos de ejecución,* los cuales son usados para simular procedimientos tanto del Kernel como de una aplicación que intenta leer datos desde el socket. Se eligieron estos threads debido a que programarlos no representa mayor dificultad, y éstos son ejecutados paralelamente en cada CPU disponible, siendo tratados como procesos independientes por el scheduler de Linux.
- *Ausencia de lock global del socket.* Si bien con esto se simplifica la implementación actual del socket en el Kernel, anteriormente se explicó que este lock debía ser levantado en conjunto con la característica de Memory Accounting de los sockets. Este modelo se deshace de ambas características.
- *Cola de paquetes FIFO, implementada con una lista doble enlazada con cabecera,* lo cual logra reproducir la estructura de datos existente en el núcleo para almacenar la información que llega al socket de manera asíncrona con la aplicación.
- *Sincronización de la cola de paquetes mediante un spinlock,* protegiendo la cola de accesos simultáneos y evitando que la estructura de datos quede en un estado inconsistente

debido a las modificaciones concurrentes que se realizan sobre ella.

Este último punto representa un problema desde el punto de vista programático: ninguna librería estándar provee un spinlock como mecanismo de sincronización de código, debido a que estas primitivas garantizan la exclusión mutua mediante *busy-waiting*, es decir, la ejecución continua de instrucciones de máquina que intentan acceder a la zona crítica, por lo que el rendimiento de una aplicación que usa spinlocks puede ser más bajo que el deseado o esperado.

4.1.2. Sincronización de estructuras

En el caso del socket, el Sistema Operativo utiliza spinlocks para garantizar el acceso exclusivo a la cola de paquetes, debido a que estas primitivas de sincronización fueron diseñadas para proteger pequeñas porciones de código, dado el bajo rendimiento que podrían presentar. En este escenario, el spinlock sólo resguarda las operaciones con punteros al insertar y eliminar desde una lista doble enlazada, proceso que puede ser llevado a cabo en tres asignaciones.

Sin embargo, para poder utilizar estas primitivas en una aplicación en el espacio de usuario de Linux, éstas debieron ser implementadas completamente en lenguaje ensamblador, compiladas y enlazadas en el binario final.

Según la literatura [19], los spinlocks son las primitivas de sincronización de más bajo nivel, las cuales se apoyan en instrucciones de hardware *CAS*(*Compare-And-Swap*) para intentar cambiar el valor de una variable atómicamente. Cuando se ha producido dicho cambio, se dice que el spinlock ha sido adquirido, y se garantiza la exclusión mutua. En caso de que un hilo de ejecución no logre cambiar el valor, no ha logrado obtener el lock correspondiente, por lo que lo sigue intentando reiteradamente hasta conseguirlo.

El hecho de ejecutar continuamente una instrucción del tipo que se describe antes disipa mucha energía, por lo que se introducen algunas optimizaciones antes de intentar nuevamente obtener el spinlock. Estas optimizaciones consisten en la ejecución de una instrucción de máquina que relaja la CPU, antes de volver a intentar tomar el lock nuevamente, con lo que se logra disminuir la cantidad de energía consumida por un spinlock.

De manera esquemática, un spinlock se visualiza en el trozo de código mostrado en la Figura 4.3.

Si bien este código representa una forma simple de visualizar su funcionamiento, el spinlock usado en este modelo debió ser programado en código ensamblador, por lo que la implementación usada difiere de lo antes expuesto. Implementaciones concretas y una explicación a fondo de su funcionamiento se puede encontrar en [19].

Sin perjuicio de lo anterior, adicionalmente se probará el rendimiento de los modelos con otro mecanismo de sincronización: los *mutexes*, los cuales se caracterizan por suspender la ejecución del thread si éste se encuentra bloqueado por otro hilo de ejecución.

```
int lock = 0;
/* swap intercambia atómicamente el valor de una variable */
/* con una constante, retornando el valor antiguo existente */
/* en la memoria */
while(swap(1, lock)){
    /* Valor antiguo de lock = 1 => lock ocupado */
    /* Relajar CPU */
    pause();
}
/* Valor antiguo de lock = 0 => este hilo de ejecución
 * logró tomar el spinlock */
```

Figura 4.3: Código esquemático de un spinlock

La principal diferencia entre los spinlocks y la primitiva antes mencionada, es que estas últimas reducen el consumo de CPU en escenarios de alta contención, debido a que previenen que la tarea cope la CPU mientras espera a que se libere el mutex.

4.1.3. Emulación de las pruebas

Una vez implementado el modelo antes descrito, se debió reproducir las pruebas de stress sobre el esquema emulado, para lo cual se establecen los siguientes parámetros de ejecución:

- *Cantidad de productores de información,* que emularán la última tarea que realizan los threads de Kernel al procesar un paquete recibido en el socket: insertar esta información en la cola de recepción.
- *Cantidad de consumidores de información,* los cuales simulan las llamadas al sistema que intentan obtener información desde el socket, extrayendo un paquete por cada invocación desde la cola de recepción. Cualquier thread que intente leer datos desde una cola que no posee paquetes, suspenderá su ejecución hasta que los datos sean provistos por algún thread de Kernel, para lo cual se utilizaron *file descriptors de eventos*, que proveen funcionalidad de semáforos en una de sus configuraciones, como se documenta en [11].
- *Cantidad de lecturas realizadas por cada consumidor,* con el fin de controlar el flujo de paquetes simulado que ingresa a los sockets y que son extraídos de la estructura por los consumidores. Si bien en el núcleo mismo existen, los descartes de paquetes no serán parte de la simulación implementada.

Estos grados de libertad permiten establecer configuraciones que se acercan a las condiciones reales sobre las cuales trabaja el Kernel: flujo de paquetes desde la red, cantidad de threads leyendo datos desde el socket y cantidad de threads de Kernel que procesan paquetes de manera paralela.

4.2. Esquema de encolamiento propuesto

La naturaleza multithread del problema presentado sugiere que la inclusión de una sola cola de paquetes no es la solución correcta, debido a la sincronización requerida por los hilos de ejecución para mantener la estructura de datos compartida siempre en estado consistente. Es por esto que se propone el uso de múltiples colas de paquetes, cada una de las cuales debe ser protegida por un spinlock para garantizar su consistencia.

En este caso, la Figura 4.2 se verá un poco modificada, como se muestra en la Figura 4.4.

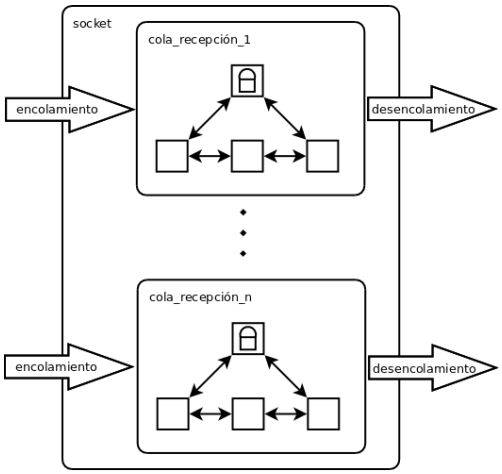

Figura 4.4: Modelo propuesto para el socket

Esta modificación permite tener tantas colas de recepción de paquetes como threads lectores hayan en la aplicación, evitando que el spinlock que se utiliza para mantener consistente la lista doble enlazada contenga threads de aplicación entre sí. La idea principal es que dicha primitiva de sincronización se utilice sólo para proteger la estructura mientras el Kernel o el thread de usuario introduce modificaciones en ella, sin que eso afecte a otros posibles hilos de ejecución del usuario o threads que procesan paquetes. Sin embargo, este esquema introduce el problema de cómo distribuir los paquetes en las colas desde el lado de los threads de Kernel, y cómo los lectores obtienen los paquetes desde las distintas colas utilizadas.

En escenarios similares a los servidores DNS, las consultas están muy probablemente contenidas completamente dentro de un único paquete UDP (según [13]), por lo que cada paquete proveniente de Internet es equivalente a cualquier otro en el sentido que contiene una consulta que debe ser resuelta por el servidor. Además, el tiempo de respuesta de una única consulta no influye significativamente en el desempeño general del servicio DNS, llegando incluso a ser resistente a la perdida de paquetes. Es así como cualquier mecanismo de distribución de los paquetes entre las colas de recepción garantizará que toda consulta será atendida por algún thread de atención del servidor, sin necesidad de preservar un orden FIFO en los paquetes.

Sin ir más lejos, en la solución antes descrita se implementa un esquema de distribución *round robin* circular entre las colas, con lo que se garantiza que todas las consultas son

equitativamente distribuidas entre todas las colas de recepción. Asimismo, se utiliza el mismo esquema de lectura desde las estructuras antes mencionadas, en donde cada thread lector recorre de manera circular todas las colas para extraer paquetes desde ellas. Esto es posible debido a que todos los paquetes de la prueba están uniformemente distribuidos entre todas las listas de consultas.

En resumen, el esquema antes descrito presenta las siguientes mejoras teóricas:

- *Evita gran parte de las contenciones del spinlock,* ya que utiliza instancias distintas de esta primitiva para sincronizar pares de threads productor-consumidor.
- *Permite extracciones paralelas,* debido a que utiliza múltiples colas de recepción, una por cada thread que lee. La cantidad de colas a crear inicialmente puede ser un parámetro pasado al crear el socket. Sin embargo, esta decisión se escapa del alcance de este documento.
- *Cada thread lee o escribe circularmente sobre todas las listas,* evitando una lógica mayor para determinar desde cual cola extraer o depositar los datos.

4.3. Metodología de las pruebas

Con el fin de mantener la consistencia entre los resultados presentados en capítulos anteriores, las pruebas que se describen en esta sección serán ejecutadas en la máquina antes descrita en el Capítulo 2:

- Sistema Operativo Fedora Core 17.
- Kernel Linux 3.5.2.
- Procesador Intel Core i5–2400 @ 3.10 GHz (4 núcleos).
- 16 GB memoria RAM DDR3 @ 1333 MHz.

Por otro lado, las pruebas de stress sobre ambos modelos de encolamiento fueron aplicadas a distintas configuraciones, con el fin de estudiar a cabalidad ambas situaciones. Dichas configuraciones se basan en:

- *Número de productores,* que emulan los threads de Kernel que procesan paquetes desde Internet y los dejan disponibles en las colas de recepción de un socket. Este parámetro se fijó en cada conjunto de pruebas.
- *Número de lectores,* representando, como antes se dijo, a las lecturas del socket. Este número se varía manteniendo constante la métrica anterior.
- *Número de colas,* las cuales corresponden a la cantidad de lectores existentes en las prueba, con la cual cada lector tendrá una cola exclusiva. Sin embargo, y dado que los paquetes serán distribuidos uniformemente entre estas colas, los threads no leerán desde su correspondiente cola, sino que lo harán en esquema *round robin*, como antes se dijo.
- *Número de paquetes,* el cual se mantuvo fijo en 1,000,000 por cada lector, midiendo el tiempo que transcurre entre el momento en que son lanzados todos los threads requeri-

dos hasta que éstos finalizan completamente. Dichas mediciones de tiempo son tomadas mediante la primitiva `gettimeofday` de Linux, retornando mediciones en μs.

Cada conjunto de pruebas consistirá en una cantidad fija de threads productores, los cuales variarán desde uno hasta cuatro, debido a que la máquina en la cual fueron ejecutados los test cuenta con cuatro procesadores. Además, cada conjunto se caracteriza por hacer variar el número de lectores concurrentes para la cantidad fija de productores presentes. Esta cantidad varía desde uno hasta diez lectores simultáneos, modificando además, la cantidad de colas que se utilizan en cada prueba. Estas configuraciones se aplicarán tanto para las versiones que utilizan spinlock, como para las que utilizan mutex como primitivas de sincronización.

De manera adicional a lo anterior, se aplicará una metodología de escalamiento simétrico de threads. A lo largo de este conjunto de pruebas, la cantidad de threads de Kernel coincidirá con la cantidad de threads lectores, variando estos valores desde uno hasta diez. Este esquema se aplicará sólo a las soluciones con spinlock, debido a que es ésta la primitiva de sincronización que se utiliza en el Kernel de Linux.

Por otra parte, y debido a que en muchos escenarios el scheduler del Sistema Operativo deberá programar la ejecución de cada thread, cada prueba será ejecutada cinco veces con el fin de corregir errores en las mediciones que podrían ser provocados por el scheduler de Linux. Este último hecho se debe a que hay más hilos de ejecución que procesadores disponibles.

Finalmente, y para mejorar la visualización de los datos al momento de presentarlos, cualquier escenario cuya ejecución tome sobre 25 segundos no será considerada en el análisis.

4.4. Resultados obtenidos

La ejecución de las pruebas siguiendo la metodología antes descrita presenta los siguientes resultados (todos los tiempos mostrados en las tablas 4.1 a 4.13 están medidos en μs):

4.4.1. Escalamiento de lectores: Esquema original

1 thread Kernel

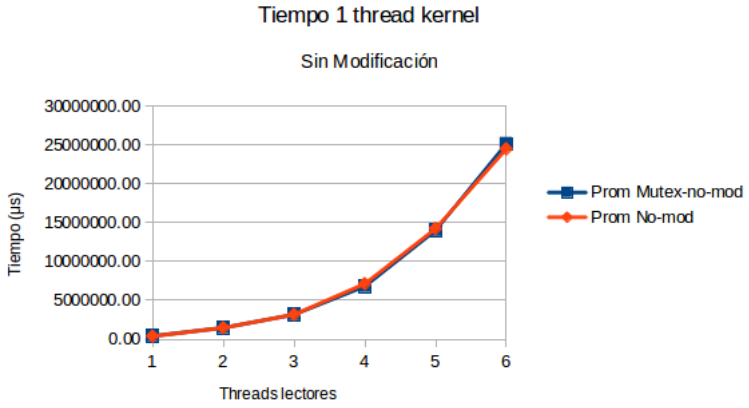

Figura 4.5: Rendimiento esquema original, 1 thread Kernel

Threads-lectores	Mutex	Spinlock
1	408170.00	377796.00
2	1463287.40	1459001.60
3	3159926.20	3161144.20
4	6789361.00	7117657.40
5	13990727.80	14208547.80
6	25133471.00	24472926.60

Tabla 4.1: Resultados esquema original, 1 thread Kernel

2 threads Kernel

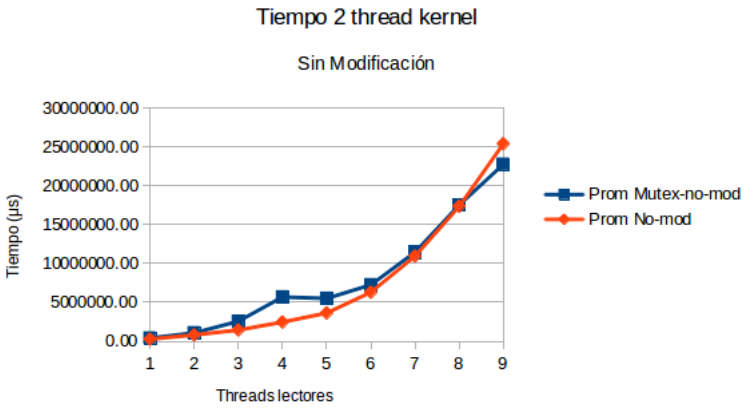

Figura 4.6: Rendimiento esquema original, 2 threads Kernel

Threads-lectores	Mutex	Spinlock
1	399751.40	264378.40
2	1069169.40	804175.40
3	2564234.00	1439088.00
4	5687968.20	2459780.20
5	5503660.40	3627482.60
6	7246283.00	6295573.80
7	11465158.00	10941532.00
8	17524919.40	17346057.00
9	22764535.60	25403367.00

Tabla 4.2: Resultados esquema original, 2 threads Kernel

3 threads Kernel

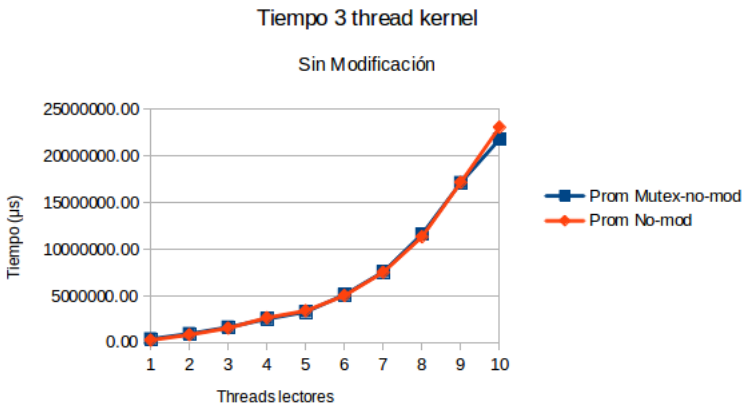

Figura 4.7: Rendimiento esquema original, 3 threads Kernel

Threads-lectores	Mutex	Spinlock
1	387761.20	271349.40
2	947534.60	847930.00
3	1643575.60	1559440.60
4	2550185.00	2669116.80
5	3295275.20	3417716.80
6	5136681.20	5044798.20
7	7594446.80	7542271.20
8	11617914.00	11339818.00
9	17144331.40	17178026.80
10	21810116.00	23093468.40

Tabla 4.3: Resultados esquema original, 3 threads Kernel

4 threads Kernel

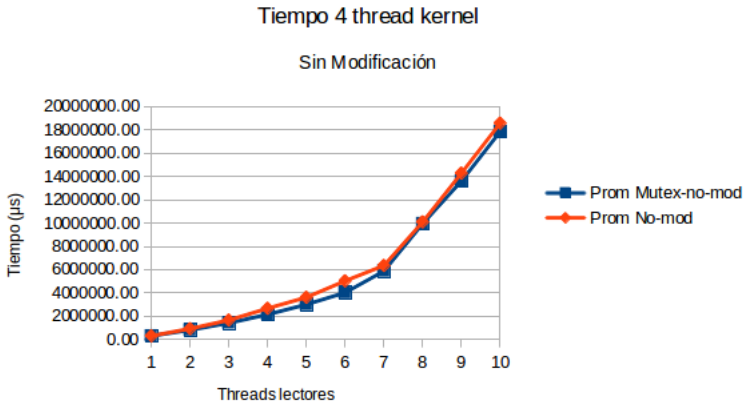

Figura 4.8: Rendimiento esquema original, 4 threads Kernel

Threads-lectores	Mutex	Spinlock
1	314640.40	317462.20
2	824224.80	934673.20
3	1420975.40	1669701.40
4	2153774.40	2678547.60
5	3016027.40	3626982.40
6	4033095.40	5036375.40
7	5881773.00	6355217.60
8	9945356.00	10097719.40
9	13604075.40	14282045.60
10	17850837.60	18574633.20

Tabla 4.4: Resultados esquema original, 4 threads Kernel

4.4.2. Escalamiento de lectores: Esquema modificado

1 thread Kernel

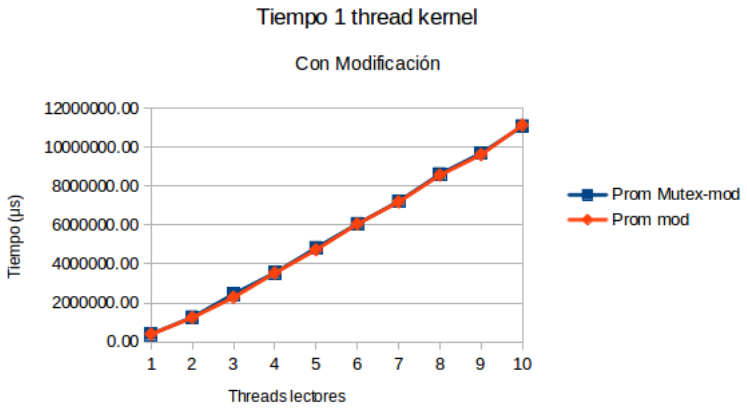

Figura 4.9: Rendimiento múltiples colas, 1 thread Kernel

Threads-lectores	Mutex	Spinlock
1	401877.00	393634.60
2	1263345.20	1244060.60
3	2457644.80	2286679.80
4	3547033.80	3534966.20
5	4827060.20	4725674.20
6	6061446.80	6053370.80
7	7214528.60	7184949.60
8	8614769.20	8558629.00
9	9690610.80	9612233.80
10	11078497.40	11122940.20

Tabla 4.5: Resultados múltiples colas, 1 thread Kernel

2 threads Kernel

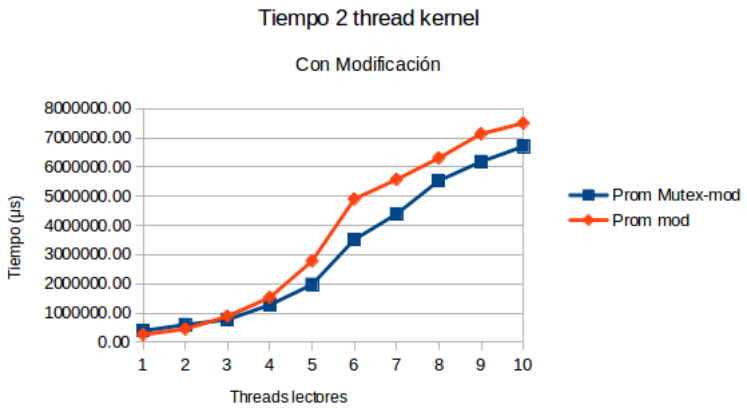

Figura 4.10: Rendimiento múltiples colas, 2 threads Kernel

Threads-lectores	Mutex	Spinlock
1	391412.80	261736.80
2	594117.00	451552.80
3	774811.20	888337.80
4	1277557.40	1532658.80
5	1974466.00	2779138.40
6	3514376.80	4902408.40
7	4390104.80	5570957.20
8	5529239.40	6305759.80
9	6182666.40	7135695.80
10	6702608.20	7499049.00

Tabla 4.6: Resultados múltiples colas, 2 threads Kernel

3 threads Kernel

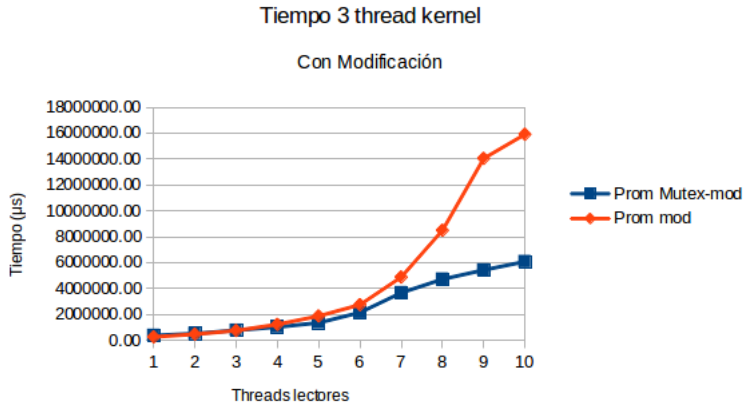

Figura 4.11: Rendimiento múltiples colas, 3 threads Kernel

Threads-lectores	Mutex	Spinlock
1	379323.00	268369.00
2	523877.00	476329.40
3	779433.20	757530.00
4	1031655.60	1233255.80
5	1356543.20	1878643.80
6	2148864.80	2751877.20
7	3674993.20	4883395.20
8	4719246.80	8495378.00
9	5430243.60	14042304.80
10	6077541.00	15903316.20

Tabla 4.7: Resultados múltiples colas, 3 threads Kernel

4 threads Kernel

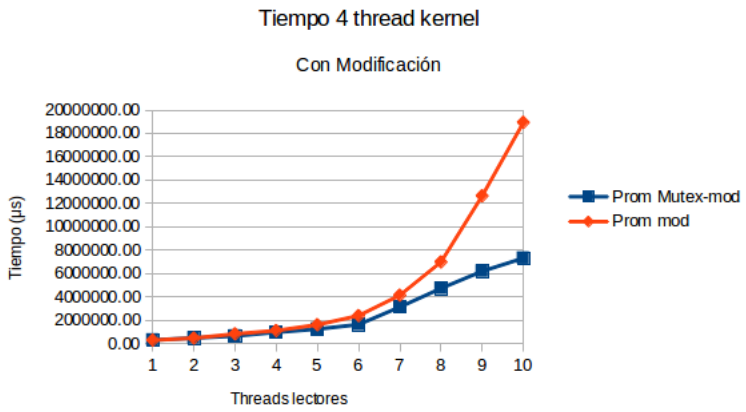

Figura 4.12: Rendimiento múltiples colas, 4 threads Kernel

Threads-lectores	Mutex	Spinlock
1	308836.80	298532.40
2	484477.20	474145.20
3	652344.60	830370.80
4	974134.00	1118317.80
5	1246573.00	1616666.20
6	1636235.80	2396740.20
7	3138906.40	4145349.20
8	4734949.60	6993168.00
9	6201419.40	12657827.60
10	7323411.20	18939318.00

Tabla 4.8: Resultados múltiples colas, 4 threads Kernel

4.4.3. Escalamiento simétrico

Para esta prueba se calculó el throughput alcanzado por ambas soluciones, recordando que cada thread lector realiza 1,000,000 accesos al modelo de socket. Esta última métrica está dimensionada en paquetes por segundo leídos:

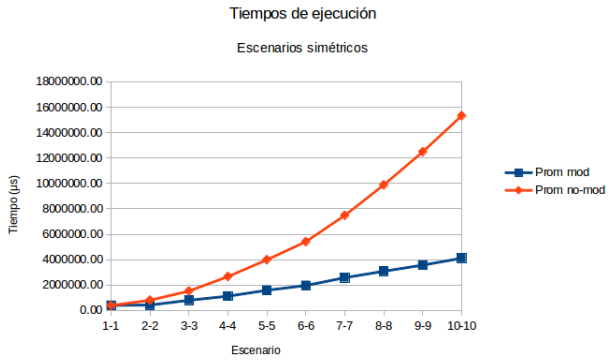

Figura 4.13: Rendimiento de escalamiento simétrico, modelos con spinlock

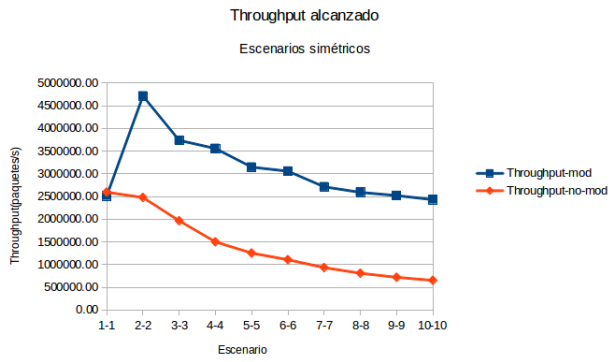

Figura 4.14: Throughput de escalamiento simétrico, modelos con spinlock

No. Kernel/Lectores	Modificado	Sin modificar	Throughput-mod	Throughput-no-mod
1-1	397626.40	385171.00	2514923.56	2596249.46
2-2	424138.00	806477.80	4715446.39	2479919.47
3-3	802529.00	1525675.80	3738182.67	1966341.74
4-4	1123915.20	2664974.40	3558987.37	1500952.50
5-5	1588013.60	3986408.60	3148587.64	1254261.79
6-6	1961381.20	5412884.60	3059068.78	1108466.27
7-7	2578014.60	7487434.00	2715267.79	934899.73
8-8	3085837.60	9881425.00	2592488.99	809599.83
9-9	3568460.80	12488385.80	2522095.80	720669.60
10-10	4112944.80	15331499.40	2431347.97	652251.93

Tabla 4.9: Resultados de escalamiento simétrico, modelos con spinlock

4.4.4. Resumen esquema original versus múltiples colas

En las siguientes páginas se muestra de manera conjunta los resultados del modelo sin modificar versus el modelo modificado, ambos utilizando spinlocks como mecanismo de sincronización. Adicionalmente se muestra el throughput de cada prueba, calculado de la misma forma anterior y dimensionado en paquetes por segundo leídos.

1 thread Kernel

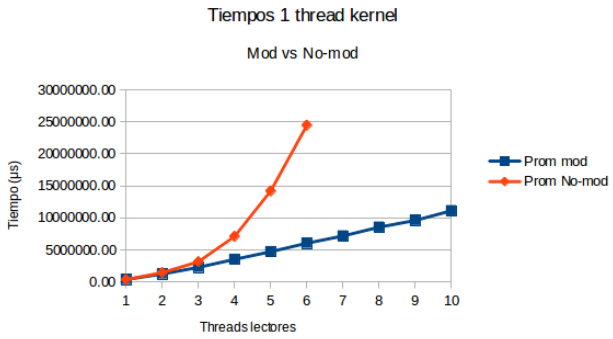

Figura 4.15: Resumen modificación versus original, 1 thread Kernel

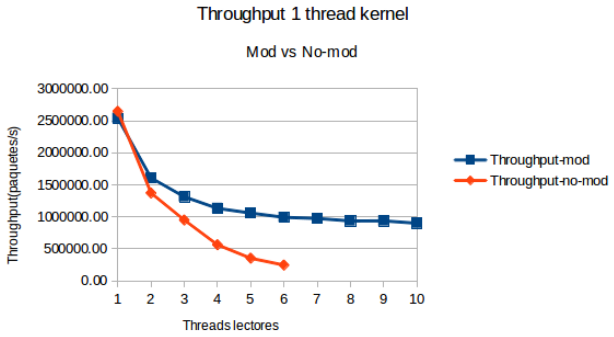

Figura 4.16: Throughput modificación versus original, 1 thread Kernel

Threads-lectores	Modificado	Sin modificar	Throughput-mod	Throughput-no-mod
1	393634.60	377796.00	2540427.09	2646931.15
2	1244060.60	1459001.60	1607638.73	1370800.42
3	2286679.80	3161144.20	1311945.82	949023.46
4	3534966.20	7117657.40	1131552.54	561982.65
5	4725674.20	14208547.80	1058050.09	351900.85
6	6053370.80	24472926.60	991183.29	245168.88
7	7184949.60	-	974258.75	-
8	8558629.00	-	934729.15	-
9	9612233.80	-	936306.81	-
10	11122940.20	-	899042.86	-

Tabla 4.10: Resultados resumen modificación versus original, 1 thread Kernel

2 threads Kernel

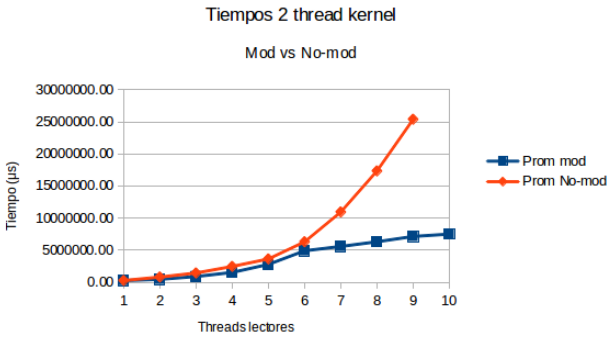

Figura 4.17: Resumen modificación versus original, 2 threads Kernel

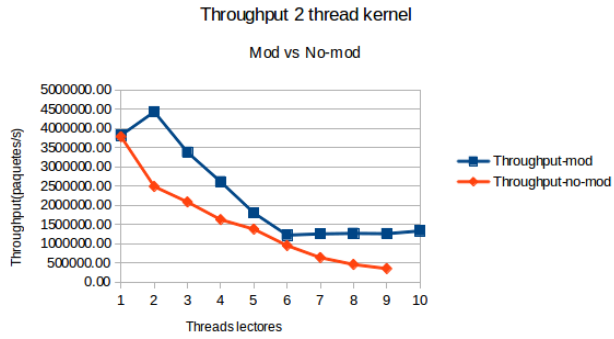

Figura 4.18: Throughput modificación versus original, 2 threads Kernel

Threads-lectores	Modificado	Sin modificar	Throughput-mod	Throughput-no-mod
1	261736.80	264378.40	3820632.02	3782457.27
2	451552.80	804175.40	4429160.89	2487019.62
3	888337.80	1439088.00	3377093.71	2084653.61
4	1532658.80	2459780.20	2609843.76	1626161.56
5	2779138.40	3627482.60	1799118.75	1378366.36
6	4902408.40	6295573.80	1223888.24	953050.54
7	5570957.20	10941532.00	1256516.56	639764.16
8	6305759.80	17346057.00	1268681.37	461199.91
9	7135695.80	25403367.00	1261264.53	354283.75
10	7499049.00	-	1333502.42	-

Tabla 4.11: Resultados resumen modificación versus original, 2 threads Kernel

3 threads Kernel

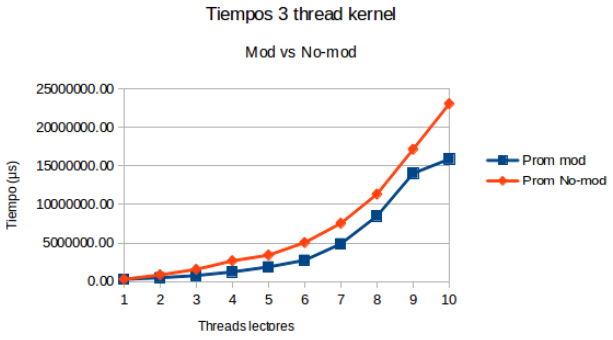

Figura 4.19: Resumen modificación versus original, 3 threads Kernel

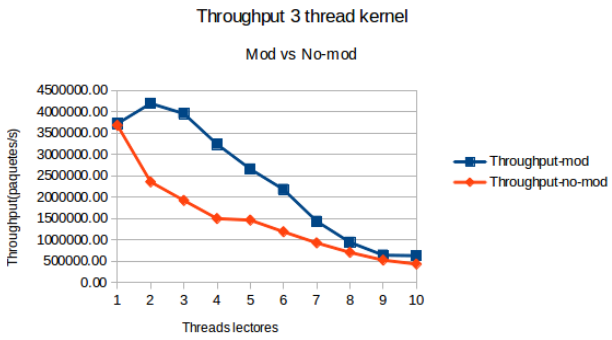

Figura 4.20: Throughput modificación versus original, 3 threads Kernel

Threads-lectores	Modificado	Sin modificar	Throughput-mod	Throughput-no-mod
1	268369.00	271349.40	3726212.79	3685285.47
2	476329.40	847930.00	4198775.05	2358685.27
3	757530.00	1559440.60	3960239.20	1923766.77
4	1233255.80	2669116.80	3243447.14	1498623.07
5	1878643.80	3417716.80	2661494.42	1462964.98
6	2751877.20	5044798.20	2180329.85	1189343.91
7	4883395.20	7542271.20	1433428.94	928102.40
8	8495378.00	11339818.00	941688.53	705478.69
9	14042304.80	17178026.80	640920.43	523925.13
10	15903316.20	23093468.40	628799.67	433022.87

Tabla 4.12: Resultados resumen modificación versus original, 3 threads Kernel

4 threads Kernel

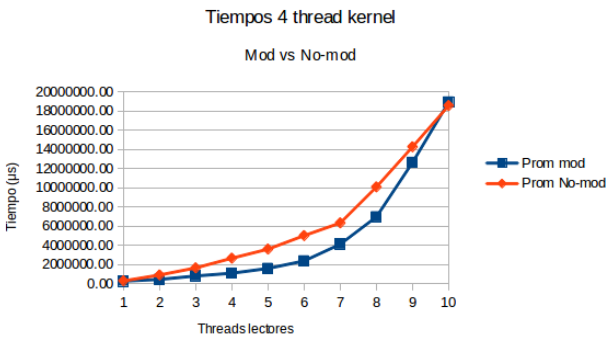

Figura 4.21: Resumen modificación versus original, 4 threads Kernel

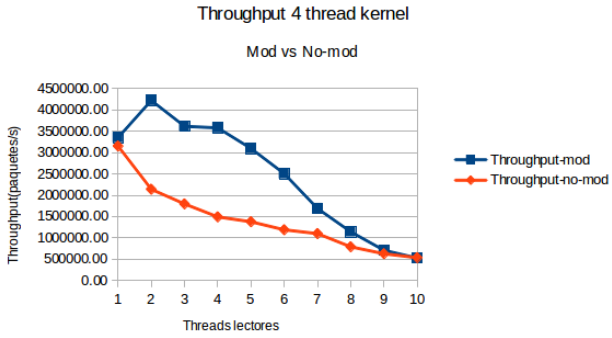

Figura 4.22: Throughput modificación versus original, 4 threads Kernel

Threads-lectores	Modificado	Sin modificar	Throughput-mod	Throughput-no-mod
1	298532.40	317462.20	3349720.16	3149981.32
2	474145.20	934673.20	4218117.15	2139785.33
3	830370.80	1669701.40	3612843.80	1796728.44
4	1118317.80	2678547.60	3576800.80	1493346.62
5	1616666.20	3626982.40	3092784.40	1378556.46
6	2396740.20	5036375.40	2503400.24	1191332.96
7	4145349.20	6355217.60	1688639.40	1101457.17
8	6993168.00	10097719.40	1143973.66	792258.10
9	12657827.60	14282045.60	711022.48	630161.83
10	18939318.00	18574633.20	528002.12	538368.64

Tabla 4.13: Resultados resumen modificación versus original, 4 threads Kernel

4.5. Discusión

Los resultados antes descritos permiten observar que la introducción del mutex en el esquema original de encolamiento de paquetes no logra mejoras significativas en el rendimiento en ningún escenario; el comportamiento exponencial del rendimiento se mantiene. Al margen de esto, el mutex logró mostrar leves mejorías en escenarios en donde existe alta contención de threads. Esto se puede visualizar al observar la Figura 4.8 y la Tabla 4.4, escenarios en los cuales el mutex siempre mantuvo rendimientos inferiores al spinlock.

Por otro lado, a partir de los resultados de la modificación introducida con las múltiples colas podemos observar diferentes situaciones. En el caso de 1 thread de Kernel (Figura 4.9 y Tabla 4.5), no existe diferencia sustancial entre la solución con spinlock y con mutex, presentando rendimiento lineal en ambos casos. Estos resultados son completamente explicables debido a que comienzan a presentar insuficiencia de datos en las colas dada la existencia de un único productor de estos datos. Es por esto que la velocidad de transferencia estará dado,

principalmente, por la velocidad a la cual el productor es capaz de colocar datos disponibles en las colas de los consumidores, lo cual explica el rendimiento lineal de este escenario. Sin embargo, el análisis del modelo normal contra la modificación nos ofrece mejores observaciones. Estos casos pueden ser examinados tanto desde el punto de vista del tiempo como desde el throughput promedio alcanzado.

Al analizar el comportamiento del tiempo se puede observar que el modelo presenta rendimientos exponenciales al variar el número de threads lectores en todas las situaciones. De manera similar se comporta la solución propuesta, en donde siempre se presenta un rendimiento exponencial, para luego linealizar el crecimiento en un determinado punto. Este comportamiento parece depender directamente de la cantidad de threads de Kernel simulados en la solución, en donde una mayor cantidad de productores se refleja en una mayor sección de comportamiento exponencial. A pesar de esto, la solución siempre presenta mejores tiempos que el modelo original. Al margen de lo anterior, es interesante observar que la solución utilizando mutexes como primitiva de sincronización presenta un mejor comportamiento asintótico que la versión utilizando spinlocks, logrando incluso mejores tiempos. Sin embargo, mejores análisis se extraen desde la comparativa de los valores de throughput.

En el caso del modelo original podemos observar que en todos los casos el throughput decae de manera exponencial al aumentar la cantidad de threads lectores y, por consiguiente, la cantidad de colas de recepción de paquetes. En este ámbito, la solución propuesta presenta ganancia en todos los casos, presentando incluso mejores valores al tener un único thread de Kernel en conjunción con múltiples hilos lectores cada uno con su respectiva cola. Asimismo, en los escenarios en donde existen múltiples threads productores, la solución tuvo resultados similares al modelo original cuando existía un único thread lector y cola de recepción, dado a que ambos esquemas degeneran en lo mismo en este caso. Sin embargo, al agregar threads lectores se nota la diferencia de las implementaciones.

Al contar con solo dos threads lectores, en todos las situaciones con distinto número de threads productores la solución logró aumentar el throughput, en contraste al modelo original que sigue presentando el decrecimiento exponencial. Adicionalmente, el modelo propuesto presenta mejores throughput que la situación inicial de sí mismo (configuración de un thread Kernel, un lector y un cola de recepción), mientras la cantidad de lectores y de colas no sobrepase la cantidad de threads de Kernel.

Finalmente, al observar el escalamiento simétrico de threads mostrado en las Figuras 4.13 y 4.14 se muestra claramente que el escalamiento simétrico de threads presenta un mejor crecimiento del tiempo de ejecución para el procesamiento de la misma cantidad de paquetes, lo cual se traduce en que la solución siempre presenta mejores valores de throughput, con el mismo comportamiento observado en el análisis anterior. En este resumen, se visualiza el punto óptimo en donde se maximiza esta métrica: dos threads Kernel en conjunción con dos threads lectores.

4.6. Conclusiones

Los resultados muestran que la introducción de múltiples colas de recepción de paquetes logra ganancias en el tiempo de procesamiento de paquetes y, lo que es más importante, importantes ganancias en la cantidad de paquetes que los threads lectores son capaces de leer. Sin embargo, la solución propuesta es muy dependiente del número de colas y cantidad de threads lectores, por lo que es necesario ajustar estos parámetros hasta la configuración óptima, la cual depende directamente de la cantidad de procesadores disponibles en la máquina que ejecuta la solución. Sin ir más lejos, un número de colas igual a la mitad de procesadores disponibles puede ofrecer una buena ganancia de tiempo, dado que permite ejecuciones tanto de threads de Kernel como de threads de la aplicación de manera simultánea.

Paralelamente se logró demostrar que la introducción de múltiples colas de lectura aumenta la cantidad de paquetes extraídos desde el socket, incluso en escenarios en donde se tiene un solo thread de Kernel recibiéndolos y procesándolos. Este hecho se evidencia, incluso, de manera más clara al aumentar la cantidad de threads de Kernel, en donde se logró aumentar el throughput del modelo del socket introduciendo múltiples colas y permitiendo ejecuciones paralelas de extracción.

Esto nos permite concluir que el esquema de encolamiento existente en el Kernel no minimiza las contenciones producidas, por lo que una implementación que sí lo haga es necesaria, llegando a aumentar el rendimiento de los sockets en cuatro o más veces, dependiendo de la cantidad de threads ejecutando paralelamente y de la cantidad de CPUs disponibles.

Conclusión

Los procesadores modernos tienen la característica de tener múltiples núcleos imbuidos en un mismo chip, lo que permite la ejecución de manera paralela de varios threads simultáneos. De este hecho intentan tomar ventaja cualquier tipo de aplicaciones ya que el paralelismo permite reducir drásticamente el tiempo de ejecución del mismo proceso. Sin embargo, al tratarse de aplicaciones que leen información desde Internet, no se logra ganancia alguna de rendimiento.

Empresas como *Facebook* o *Toshiba* han documentado esta observación [20, 18], sin investigar más profundamente que situar el origen de esto en el Kernel de Linux. El trabajo realizado caracterizó el problema, detectándolo al intentar paralelizar las lecturas a un mismo socket de Linux. Se mostró, asimismo, que dichos accesos son ejecutados de manera secuencial, sin lograr ningún tipo de ganancia al introducir threads simultáneos, hecho que se visualizó como tiempos de ejecución crecientes y lineales al aumentar los hilos de ejecución. Dicho problema fue replicado en las versiones más recientes del Kernel obteniendo los mismos resultados antes expuestos, logrando determinar que el hecho aún se presenta, con lo que se comenzó la tarea de determinar su origen y hallar una explicación a este fenómeno.

El primer punto de falla puesto a prueba fue la librería *libc*, analizando específicamente su primitiva `read`, la cual mostró no ser la raíz de la falla detectada, logrando accesos paralelos al obtener datos desde otros orígenes distintos a un socket, lo cual limitaba el problema al núcleo del Sistema. Sin embargo, el mecanismo utilizado por Linux para atender las llamadas a sistema (como `read`) podría haber estado serializando los accesos, lo que obligó a estudiar su comportamiento.

Este estudio se realizó mediante la implementación de una primitiva de sistema creada con fines de pruebas, midiendo cómo se comportaba Linux ante llamadas concurrentes a ésta. Este stress sobre el código implementado mostró que dichas llamadas sí pueden ser paralelizadas, con lo cual se pudo establecer en porciones de código pertenecientes al núcleo la causa de la serialización.

Esto último sumado al hecho de presentar rendimientos perfectamente lineales desde el punto de vista de la aplicación, llevó a pensar que la raíz del problema se ubicaba en alguna estructura de datos compartida utilizada por el socket y la aplicación, desde donde la información es insertada desde el núcleo y extraída por la aplicación. En específico, se sospechó del mecanismo de sincronización utilizado para mantener consistente esta estructura, lo que obligó a estudiar el comportamiento de éstas en el núcleo.

El desafío antes expuesto requirió la modificación del núcleo, con el fin de que éste fuese capaz de recolectar información acerca de las primitivas de sincronización utilizadas en las estructuras compartidas, monitoreando el proceso de recepción de un paquete desde que éste es recibido en la tarjeta de red hasta que es extraído por una aplicación. Específicamente, fue registrado el comportamiento del spinlock utilizado para proteger el socket completo (introducido con el *Memory Accounting*) en conjunto con las instancias de estas primitivas usadas para mantener consistentes las listas de paquetes del socket. En particular sólo fue registrado el comportamiento del spinlock de la lista de recepción de paquetes.

Los resultados demostraron que los mecanismos de sincronización utilizados en el socket corresponden a una de las primitivas más costosas del núcleo, sin que su eliminación subsane el problema real. Esta modificación logra disminuir levemente el tiempo de procesamiento de un paquete, pero no elimina la secuencialidad de los accesos. Por este hecho, se propuso la creación de un esquema de encolamiento de paquetes que sea capaz de soportar los accesos concurrentes, tanto por el lado del núcleo como de la aplicación. Esta solución consistía en la introducción de múltiples colas de recepción, evitando gran parte de la sincronización requerida para mantener coherente la lista de recepción de paquetes.

Sin embargo, cualquier solución que se proponga para este problema requiere la introducción de código en el núcleo de Linux, caracterizado por ser *monolítico*, lo que conlleva a compilar completamente el sistema al introducir cualquier cambio en éste. De modo de simplificar esta restricción, se elaboró un modelo del socket en un programa C, replicando tanto las estructuras de datos como las primitivas de sincronización que son utilizadas en el Kernel. Esta decisión se tomó con el fin de poder introducir cambios fácilmente, evitando los grandes tiempos de compilación antes mencionados.

Sobre este modelo fueron replicadas las mismas pruebas de stress aplicadas anteriormente de manera de validar la simplificación, logrando replicar el comportamiento encontrado en las pruebas reales. Asimismo, la validación del modelo permite determinar de manera muy cercana cómo se comportaría la solución propuesta una vez que ésta sea implementada realmente en el núcleo.

Los resultados de las ejecuciones mostraron que no sólo se logró una ganancia de tiempo al implementar múltiples colas de recepción, sino que también se pudo aumentar el throughput de paquetes del socket, llegando en algunos escenarios a duplicar esta métrica. Esto mismo permitió establecer el punto óptimo para el cual la solución propuesta maximiza su rendimiento: copar completamente todos los procesadores de la máquina con threads, tanto productores como de kernel; en otras palabras, creando tantos threads lectores y colas de recepción como la mitad de procesadores de la máquina.

Este simple hecho permite que se prevenga la mayor cantidad de contenciones que el modelo original no previene, razón que se estableció como el problema de fondo que evitaba un mejor rendimiento de las estructuras de datos del socket. Sin embargo, la solución propuesta requiere de mayores caracterizaciones que no fueron abordadas en este trabajo, por lo que a modo de continuidad de la investigación, queda como trabajo futuro:

- Finalizar la caracterización de las múltiples colas en el modelo.
- Proponer otros mecanismos de encolamiento de paquetes e implementarlos en el modelo,

utilizando otras estructuras de datos.

- Determinar su comportamiento mediante la ejecución de las pruebas.
- Analizar el comportamiento, tanto de la solución de este documento como la de alguna otra estructura propuesta, en máquinas con mayor cantidad de procesadores.
- Implementar el encolamiento definitivo en el Kernel de Linux.
- Probar la nueva implementación en un servidor real. Un DNS representaría un muy buen escenario de pruebas.

Con la finalización de este proyecto, se lograrán mejoras sustanciales en el índice de costo-efectividad presentados en servidores DNS actuales, los cuales deben recurrir a configuraciones artesanales que logran subsanar el problema, como lo es la configuración de máquinas virtuales en una misma máquina física.

Asimismo, se persigue uno de las principales premisas de Linux: explotar de la mejor manera posible la máquina sobre la que corre el sistema operativo. Esta investigación apuntó a probar que aún hay trabajo que hacer el núcleo, demostrando que se pueden implementar mejores formas de administrar los paquetes que entran a un socket en máquinas multi-procesadores.

Bibliografía

[1] CHRISTIAN BENVENUTI. *Understanding Linux Network Internals*. O'Reilly Media Inc., Sebastopol, CA, USA, 2006.

[2] DIEGO CALLEJA. Linux 2 6 25. <http://kernelnewbies.org/Linux_2_6_25>, Mayo 2008.

[3] ERIC DUMAZET. netdev - [rfc net-next-2.6] udp: Dont use lock_sock()/release_sock() in rx path. <http://lists.openwall.net/netdev/2009/10/13/153>, Octubre 20009.

[4] KLAUS WEHRLE et al. *Linux Networking Architecture*. Prentice Hall, Upper Saddle River, NJ, USA, 2004.

[5] T. BERNERS-LEE et al. Rfc1738 - a gopher url format. <http://tools.ietf.org/html/rfc1738>, Diciembre 1994.

[6] T. BERNERS-LEE et al. Rfc1945 - hypertext transfer protocol – http/1.0. <http://tools.ietf.org/html/rfc1945>, Mayo 1996.

[7] WENJI WU et al. The performance analysis of linux networking – packet receiving. *Computer Communications 30 (2007) 1044–1057*, 2006.

[8] THE LINUX FOUNDATION. kernel_flow. <http://www.linuxfoundation.org/collaborate/workgroups/networking/kernelflow>, Noviembre 2009.

[9] INFORMATION SCIENCES INSTITUTE. Rfc791 - internet protocol. <http://tools.ietf.org/html/rfc791>, Septiembre 1981.

[10] INFORMATION SCIENCES INSTITUTE. Rfc793 - transmission control protocol. <http://tools.ietf.org/html/rfc793>, Septiembre 1981.

[11] MICHAEL KERRISK. eventfd(2) - linux manual page. <http://man7.org/linux/man-pages/man2/eventfd.2.html>, Agosto 2010.

[12] P. MOCKAPETRIS. Rfc1034 - domain names - concepts and facilities. <http://tools.ietf.org/html/rfc1034>, Noviembre 1987.

[13] P. MOCKAPETRIS. Rfc1035 - domain names - implementation and specification. <http://tools.ietf.org/html/rfc1035>, Noviembre 1987.

[14] INGO MOLINAR. Runtime locking correctness validator. <https://www.kernel.org/doc/Documentation/lockdep-design.txt>, Julio 2006.

[15] JOSE MIGUEL PIQUER. Jsockets: Clientes/servidores de red. <https://wiki.dcc.uchile.cl/cc3301/#clase_21jsocketsclientes_servidores_de_red>, Abril 2009.

[16] J. POSTEL. Rfc793 - user datagram protocol. <http://tools.ietf.org/html/rfc768>, Agosto 1980.

[17] STEVEN ROSTEDT. ftrace - function tracer. <https://www.kernel.org/doc/Documentation/trace/ftrace.txt>, Octubre 2008.

[18] PAUL SAAB. Scaling memcached at facebook. <https://www.facebook.com/note.php?note_id=39391378919>, Diciembre 2008.

[19] DANIEL P. BOVET y MARCO CESATI. *Understanding the Linux Kernel*. O'Reilly Media Inc., Sebastopol, CA, USA, 3rd edition, 2006.

[20] TATUYA JINMEI y PAUL VIXIE. Implementation and evaluation of moderate parallelism in the bind9 dns server. USENIX 2006 Annual Technical Conference Refereed Paper, <http://static.usenix.org/events/usenix06/tech/full_papers/jinmei/jinmei_html/>, Abril 2006.

[21] PETER ZIJLSTRA. Lock statistics. <https://www.kernel.org/doc/Documentation/lockdep-design.txt>, Octubre 2007.