
Universidad Nacional de Córdoba

FCEFyN

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS y NATURALES

# Modelado y optimización de un Planificador del Sistema Operativo mediante Redes de Petri

## Integración de Funcionalidades de Monopolización de Núcleos y Control de Encendido/Apagado de Procesadores

**Autores:** Leandro Agustin Drudi, Nicolás Goldman

10 de diciembre de 2022

**Email:** drudilea@gmail.com, nicolasgoldman07@gmail.com

**Teléfonos:** (351)5307196, (3547)560104

**Legajos:** 40.245.918, 40.416.606

**Director:** Maximiliano Eschoyez

**Co-Director:** Nicolás Papp

## Resumen

El sistema operativo se encarga de gestionar los recursos de hardware de un dispositivo electrónico y de brindar servicios a los programas de aplicación. La gestión de los procesos e hilos y los recursos de un sistema operativo son fundamentales para su correcto funcionamiento y desempeño. En este contexto, la planificación es una tarea crítica para asegurar el uso eficiente de los recursos del sistema.

En este trabajo de tesis, se parte del modelado y la optimización del planificador a corto plazo mediante Redes de Petri realizado en un proyecto integrador previo por dos alumnos de la FCEFyN. Las Redes de Petri son una herramienta matemática formal de modelado que permite representar y analizar sistemas de eventos discretos, como es el caso de la planificación de procesos. El uso de ellas, nos permitirá comprobar con mayor facilidad la ausencia de problemas de concurrencia muy comunes en la planificación, así como también aportarán a la hora de captar los estados y eventos del planificador.

En el marco de esta investigación, comenzamos con la adaptación del proyecto integrador previo a la última versión del sistema operativo. La actualización del proyecto no solo nos brinda acceso a nuevas funcionalidades avanzadas, sino que también nos permite aprovechar las mejoras en términos de seguridad implementadas en dicha versión. Además, las versiones anteriores eventualmente dejan de tener soporte por parte de la comunidad, dificultando la colaboración y el intercambio de conocimientos.

Planteadas las actualizaciones correspondientes, se abordan dos aspectos cruciales en la optimización de la gestión de recursos en sistemas operativos. En primer lugar, se examina el "Encendido y Apagado de Procesadores" como un primer paso en la implementación de soluciones de ahorro de energía a nivel de software, antes de considerar enfoques de hardware. La finalidad es permitir la gestión activa de la energía al apagar procesadores no utilizados, lo que conlleva a una reducción significativa en el consumo energético, contribuyendo a una mayor eficiencia operativa del sistema. En segundo lugar, se introduce el concepto de "Monopolización de Núcleos" como una estrategia para priorizar procesos críticos por sobre otros, minimizando así interrupciones no deseadas. La capacidad de asignar núcleos específicos a tareas es fundamental para garantizar un rendimiento consistente y predecible en diversos entornos.

# Índice

# 1. Introducción

La planificación a corto plazo de un sistema operativo es la parte del sistema que se encarga de tomar decisiones sobre qué tareas se deben ejecutar y en qué orden. El objetivo principal de la misma es hacer un uso eficiente de los recursos de CPU disponibles y minimizar el tiempo de respuesta de los procesos.

El sistema operativo elegido para realizar este proyecto y las modificaciones pertinentes, es FreeBSD. Éste es un sistema operativo libre y de código abierto, basado en Unix, que se utiliza principalmente en servidores y estaciones de trabajo. Es conocido por su escalabilidad y robustez, así como por su capacidad de adaptarse y personalizarse según las necesidades del usuario.

El planificador de bajo nivel se ejecuta cada vez que un hilo se bloquea o interrumpe, y se debe seleccionar un nuevo hilo para ejecutar. Para ser eficiente, al ejecutarse miles de veces por segundo, debe tomar decisiones rápidamente con la menor cantidad de información posible. Para simplificar su tarea, el kernel mantiene un conjunto de colas de ejecución para cada CPU. Cuando una tarea se bloquea en una CPU, la responsabilidad del planificador de bajo nivel es encontrar y elegir el hilo con la máxima prioridad en la cola de ejecución asignada a esa CPU.

Este trabajo se construye sobre los cimientos establecidos en un proyecto integrador previo, donde se abordó la mejora del *scheduler* en el sistema operativo FreeBSD. En dicha investigación, se identificó la naturaleza estadística del proceso de asignación de tiempo de procesador y se propuso un modelo basado en redes de Petri para introducir determinismo y reducir la incertidumbre en la toma de decisiones. En este proyecto, buscamos sumarnos a esta trayectoria, aprovechando los conocimientos adquiridos para abordar nuevas dimensiones en la optimización de la gestión de recursos en sistemas operativos.

## 1.1. Oportunidad

El proyecto integrador previo estableció una base sólida para la comprensión integral del funcionamiento de los sistemas operativos y demostró la relevancia de la planificación en el rendimiento del sistema. Este logro requirió un compromiso considerable y una laboriosa inversión de tiempo en la construcción de modelos, así como en el análisis de los resultados obtenidos.

No obstante, dada la continua evolución tecnológica y la evolución de las necesidades de los usuarios, se hace imperativo mantener una constante actualización y refinamiento de dicha implementación.

La presente oportunidad de trabajo se centra en la incorporación de nuevas funcionalidades y mejoras al planificador a corto plazo del sistema operativo. El objetivo principal radica en alcanzar niveles superiores de eficiencia y desempeño en las operaciones que se ejecutan en el sistema. Para ello, aprovecharemos el conocimiento y las ventajas que ofrecen las Redes de Petri, herramientas que nos permitirán introducir innovaciones de manera efectiva, al tiempo que conservamos las características fundamentales del sistema base.

En este contexto, identificamos dos oportunidades específicas. La primera consiste en eliminar la participación de los CPUs en determinadas tareas, priorizando así el consumo de energía sobre el rendimiento. Para abordar esta oportunidad, se planteó desarrollado un módulo que permita bloquear el *encolado* de un procesador, simulando así su “apagado” y reduciendo el consumo energético global

del sistema. La segunda oportunidad se centra en optimizar el rendimiento en tareas de tiempo real, evitando interrupciones al saltarse el sistema de colas. Para esta finalidad, planteamos la creación de otro módulo que permita a un hilo apropiarse de un CPU, evitando que otros hilos hagan uso del mismo mientras se ejecuta la tarea prioritaria. Si bien estos cambios aún requerirán de desarrollos adicionales para implementarse concretamente en los procesos y situaciones específicas, representan un primer paso hacia la mejora de la eficiencia energética y el rendimiento del sistema operativo, al proporcionar herramientas para el control más preciso de los recursos de hardware y la gestión de tareas críticas.

## 1.2. Motivación

La investigación previa en el área de la planificación a corto plazo de sistemas operativos, permitió encontrar una solución efectiva para reducir el indeterminismo en este tipo de sistemas. En concreto, la implementación del planificador a corto plazo de FreeBSD utilizando Redes de Petri, ha demostrado muchas ventajas mediante esta técnica de modelado.

La administración eficiente de la energía es un tema de interés en la actualidad, debido al aumento de la demanda de energía y al impacto ambiental. En este sentido, el uso de técnicas de ahorro de energía en los sistemas operativos es esencial para reducir los costos y minimizar dicho impacto.

Por otro lado, la gestión eficiente de las tareas en tiempo real es esencial en una amplia gama de aplicaciones. La capacidad de priorizar y ejecutar estas tareas de manera oportuna y predecible es crucial para garantizar un funcionamiento confiable del sistema.

En este trabajo integrador, proponemos avanzar con la implementación de funcionalidades de control de recursos y priorización de tareas en el sistema operativo, aprovechando el potencial de las Redes de Petri como herramienta de modelado y análisis en el ámbito de los sistemas operativos, sentando las bases para futuras investigaciones en este campo y contribuyendo así al avance de la informática en este aspecto.

Otra oportunidad significativa que surge para el desarrollo de este proyecto integrador es la actualización del código del planificador a la última versión del sistema operativo. La versión desde la que partimos como base es FreeBSD 11, y buscamos la migración hacia la 13.1. Esta actualización nos brinda la posibilidad de capitalizar las mejoras de rendimiento, seguridad y estabilidad introducidas en las últimas versiones. Por otro lado, los avances realizados en ambos trabajos estarían más cerca de representar una contribución al código base del sistema, y al mismo tiempo nos permitiría trabajar a la par de los contribuyentes de FreeBSD.

## 1.3. Objetivo

Objetivos principales:

- Actualizar el modelado e implementación del planificador del sistema operativo FreeBSD mediante Redes de Petri. Este se realizó para la versión 11 del mismo, mientras que FreeBSD se encuentra cursando la versión 13. Como comentamos en la sección de motivación, hacerlo compatible con

las últimas versiones nos permite aprovechar las nuevas funcionalidades y mejoras de seguridad, evitar que se vuelva obsoleto con el paso del tiempo y acercarnos a la comunidad.

- Desarrollar una funcionalidad que permita encender y apagar procesadores según las necesidades del sistema en diferentes momentos. Al permitir que el sistema gestione activamente el estado de los procesadores, restará unicamente implementar estrategias que utilicen este módulo para optimizar el consumo de energía en función de la carga de trabajo.
- Desarrollar un mecanismo que le brinde a cualquier hilo la posibilidad de ejecutarse en un procesador, evitando que otros hilos se encolen en éste. Mediante esta funcionalidad, se prioriza la ejecución del hilo correspondiente y se reduce el tiempo de espera y las demoras, logrando que el hilo alcance su objetivo antes. Como consecuencia se evitan pérdidas de rendimiento causadas por cambios de contexto.

Objetivos secundarios:

- Analizar y aprender exhaustivamente acerca del código fuente del sistema operativo FreeBSD.
- Profundizar los conocimientos en la depuración del kernel y las diferentes herramientas de debugging.
- Mejorar la documentación del proyecto, estrategia de ramas y commits en el repositorio de desarrollo priorizando las buenas prácticas de programación. Esto permitirá dejar mejores bases para quienes decidan continuar con la investigación a futuro.
- Automatizar y documentar los procesos repetitivos que se llevan a cabo en las diferentes etapas del proyecto, como por ejemplo la instalación de máquinas virtuales, paquetes que nos ayudarán a la hora del desarrollo, configuraciones de red, instalación y compilación de kernel, entre otras.
- Compartir e interactuar con la comunidad de FreeBSD a través de foros y listas de difusión.

## 1.4. Alcance

La fase inicial del proyecto se enfocará en la actualización de la implementación existente a la ultima versión del sistema operativo FreeBSD. Se considerará completada, una vez que el código del proyecto integrador previo se encuentre actualizado y se compruebe el funcionamiento adecuado.

Luego se procederá a implementar las dos nuevas funcionalidades mencionadas anteriormente. Esta etapa presenta un mayor nivel de complejidad, ya que estas funcionalidades pueden ser implementadas de diversas formas. Por esta razón, se ha decidido delimitar el alcance de la implementación en dos módulos independientes. La activación y desactivación de estos módulos se realizará mediante la carga o descarga de un módulo de kernel que incluirá parámetros relevantes en cada caso.

En el caso de la funcionalidad de encendido y apagado de procesadores, la activación permitirá elegir los procesadores que se desean desactivar del proceso de selección de colas. Por otro lado, en el caso de la funcionalidad de monopolización de CPUs, la activación del módulo permitirá seleccionar el hilo que se apropiará de la CPU correspondiente, también elegido mediante parámetros.

Adicionalmente, se implementarán pruebas para validar la funcionalidad precisa de estas incorporaciones. El alcance general de este trabajo es establecer un sólido fundamento para la adición de módulos independientes, capaces de integrarse a la Red de Petri para alcanzar diversos propósitos. En este contexto, se anticipa que los resultados obtenidos en este proyecto puedan servir como un valioso punto de partida para futuras investigaciones en el campo de la planificación a corto plazo de sistemas operativos.

## 1.5. Modelo de Desarrollo

Nuestro enfoque para este proyecto se centró en tres objetivos principales, como se mencionó anteriormente: actualizar el planificador a la última versión del sistema operativo y desarrollar los módulos de encendido/apagado de procesadores, y de monopolización de CPUs. Estos objetivos nos guiaron en la definición de metas específicas y alcances concretos, lo que nos permitió avanzar de manera ágil y precisa en el desarrollo. Aunque en este informe presentaremos estos objetivos como tres módulos separados, es importante destacar que evolucionaron de forma incremental a lo largo del proceso de desarrollo.

## 1.6. Requerimientos Generales

### 1.6.1. Requerimientos funcionales

- Adaptar la implementación actual del planificador a corto plazo de FreeBSD para garantizar su compatibilidad con las últimas versiones del sistema operativo.
- Diseñar y ejecutar pruebas para validar la correcta integración del modelo a las nuevas versiones de FreeBSD y para evaluar el funcionamiento óptimo de los nuevos desarrollos.
- Desarrollar e integrar dos nuevos módulos con funciones de gestión de procesadores y de monopolización de CPUs.
- Ejecutar pruebas para asegurar la correcta implementación e integración de las nuevas características desarrolladas en el planificador existente de modo que no afecten negativamente su rendimiento.

### 1.6.2. Requerimientos no funcionales

- Abordar las actualizaciones y los nuevos módulos del planificador a corto plazo de manera modular, organizada y documentada para facilitar su mantenimiento y actualización en el futuro.
- Detectar posibles integraciones de los nuevos módulos en el sistema operativo para proyectos futuros.
- Asegurar que la implementación del planificador a corto plazo de FreeBSD mediante Redes de Petri sea segura y no comprometa la seguridad del sistema operativo.

- Garantizar que las nuevas funcionalidades implementadas en el planificador a corto plazo de FreeBSD sean fáciles de usar y comprender para los usuarios finales. Para ello, se deberá crear documentación clara y concisa que explique la configuración, el uso y las ventajas de las nuevas funcionalidades.

# 2. Base Teórica

La planificación es un componente esencial de cualquier sistema operativo, encargado de asignar tiempo de procesamiento a los diferentes hilos y procesos en ejecución. En este capítulo, nos sumergiremos en la comprensión de la planificación a corto plazo, con especial atención en el contexto del sistema operativo FreeBSD y su planificador 4BSD. Como ya se detalló previamente en el capítulo 1, este trabajo se basa y extiende directamente desde investigaciones previas[1], donde se introdujo el concepto de Redes de Petri, para el modelado de procesos.

Exploraremos los conceptos fundamentales de procesos e hilos, detallando su estructura, estados y dinámicas de ejecución. Además, nos adentraremos en el funcionamiento interno del planificador, desglosando las operaciones esenciales como el cambio de contexto, el encolado de procesos, la selección de procesador y la remoción de hilos de la cola. Estos aspectos son esenciales para comprender cómo el planificador 4BSD toma decisiones en tiempo real sobre la ejecución de procesos y la administración de recursos.

Al profundizar en esta base teórica, no solo sentaremos las bases para entender el funcionamiento del planificador de FreeBSD, sino que también estableceremos una sólida conexión con los avances y las decisiones tomadas en trabajos anteriores. Esto nos permitirá visualizar el progreso continuo y las oportunidades de mejora que surgieron a partir de esos primeros cambios implementados en el sistema operativo.

## 2.1. Procesos e hilos

En sistemas operativos, los procesos son entidades aisladas que representan la ejecución de una tarea o aplicación en particular. Cada proceso cuenta con su propio espacio de direcciones, que es un área reservada de memoria virtual donde se aloja el código del programa, las variables y los recursos necesarios para su ejecución. Además, disponen de acceso a los recursos del kernel a través de llamadas a sistemas.

Dentro de cada proceso, puede haber uno o varios hilos de ejecución. Estos hilos son unidades de ejecución independientes que comparten los recursos del proceso padre. Cada hilo se asocia con un procesador virtual, que tiene su propio contexto y un stack de ejecución alojado en el espacio de direcciones del proceso.

El kernel del sistema operativo crea la ilusión de ejecución concurrente de múltiples procesos, distribuyendo los recursos del sistema entre los procesadores listos para ejecutar tareas.

### 2.1.1. Estructura de los procesos

Cada proceso en el sistema recibe un identificador único llamado *identificador de proceso* (PID). Los PID son el mecanismo común utilizado por las aplicaciones y el kernel para hacer referencia a los procesos. Existen dos identificadores que son de especial importancia para cada proceso: el PID del proceso en sí, y el PID del proceso padre.

La estructura simplificada de un proceso se puede observar en la Figura 1. Esta estructura tiene

como objetivo facilitar la gestión de múltiples hilos que comparten un espacio de direcciones y otros recursos dentro del proceso. Algunas de las categorías que componen esta estructura son las siguientes:

- Identificación del grupo de procesos: el grupo de procesos y la sesión a la que pertenece el mismo son elementos importantes para la gestión y el control de los procesos en el sistema.
- Credenciales de usuario: los identificadores de usuario y grupo determinan los permisos de acceso a los recursos del sistema.
- Gestión de memoria: esta parte de la estructura es crucial para asignar y gestionar el espacio de direcciones virtuales del proceso, así como para administrar otros aspectos relacionados con la memoria.
- Descriptores de archivos: una matriz de punteros que indica los archivos abiertos por el proceso e información relevante de los mismos.
- Vector de llamadas al sistema: estructura de datos que mapea las llamadas al sistema con las funciones correspondientes en el kernel del sistema operativo.
- Contabilidad de recursos: estructura que describe la utilización de los recursos del sistema por parte del proceso.
- Estadísticas: estadísticas del proceso sobre su ejecución, temporizadores y profiling.
- Acciones de señal: la acción a tomar cuando se envía una señal a un proceso.
- Estructura de hilo: el contenido de la estructura de hilos del proceso.

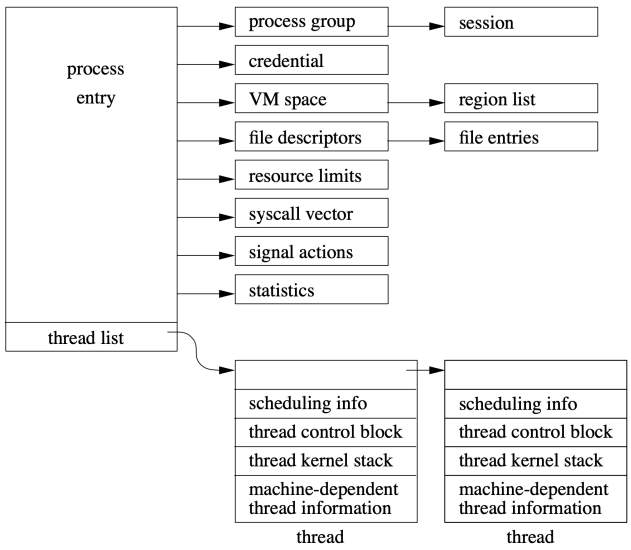


Figura 1: Estructura simplificada de un proceso.

Cada proceso cuenta con los punteros *p_pptr*, *p_children* y *p_sibling*, utilizados para establecer la relacion entre procesos. Cuando se crea un proceso hijo, se agrega a la lista *p_children* de su padre. El proceso hijo también mantiene un enlace a su padre mediante su puntero *p_pptr*. Si un proceso tiene más de un hijo activo al mismo tiempo, los hijos están asociados entre sí a través de las entradas de la lista *p_sibling*. La Figura 2 muestra un ejemplo de jerarquia de procesos.

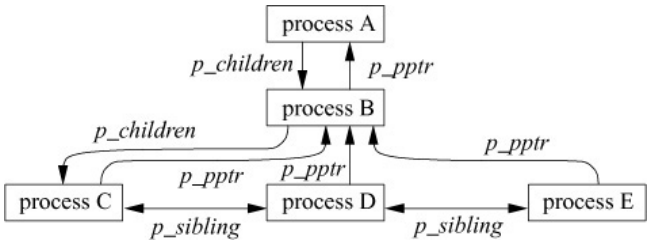


Figura 2: Jerarquía de grupo de procesos.

### 2.1.2. Estructura de los hilos

Un hilo, en sistemas operativos modernos, constituye una unidad fundamental de ejecución dentro de un proceso. Se trata de una entidad independiente que representa una secuencia de instrucciones ejecutables dentro del contexto de un proceso. Cada hilo posee su propio contador de programa, registros de CPU y pila de ejecución, lo que le permite ejecutar código de manera concurrente dentro del mismo proceso. Aunque los hilos comparten recursos como el espacio de direcciones y otros recursos del proceso principal, también pueden comunicarse y cooperar entre sí para llevar a cabo tareas específicas de manera más eficiente.

En el caso de FreeBSD, el sistema adopta el modelo 1:1, donde cada hilo de usuario se corresponde con un hilo a nivel de kernel para mejorar la eficiencia de las aplicaciones.

La estructura de un hilo, que se muestra en la Figura 1, contiene la información necesaria para ejecutarse en el kernel del sistema operativo:

- Información para la planificación: se refiere a la prioridad del hilo en modo kernel y en modo usuario, la cantidad de tiempo que ha pasado suspendido y el uso reciente de la CPU. Además, se indica el estado de ejecución del hilo, banderas de estado adicionales; y si el hilo se encuentra suspendido, información sobre el canal y evento por el cual espera.
- TSB (thread state block): estado de ejecución del hilo en modo usuario y modo kernel. La estructura incluye registros de propósito general, punteros de pila, contador de programa, registros de gestión de memoria, entre otros.
- Pila del kernel: pila para usar al ejecutar en el kernel. Las pilas del kernel deben mantenerse pequeñas para evitar desperdiciar memoria física.
- Estado de la máquina (*machine-dependent state*): se refiere a la información del hilo en relación a detalles que son específicos de la arquitectura de la CPU (registros de estado de punto flotante, información de interrupciones, información de registros de segmento de memoria, etc.).

### 2.1.3. Estados de los procesos e hilos

La estructura de un proceso en FreeBSD incluye un campo que indica su estado actual. Los estados de un proceso son fundamentales para entender su comportamiento en el sistema y están estrechamente relacionados con el funcionamiento del planificador 4BSD. Cuando se crea un proceso utilizando la llamada al sistema *fork*, inicialmente se marca como nuevo (NEW). Este estado indica que el proceso está en su fase de creación y aún no ha recibido suficientes recursos para comenzar la ejecución.

Una vez que se asignan los recursos necesarios, el estado del proceso se cambia a NORMAL. En este estado, los hilos del proceso pueden encontrarse en diferentes subestados: ejecutable (RUNNABLE) cuando están listos para ejecutarse o actualmente en ejecución, durmiendo (SLEEPING) cuando están esperando un evento, o detenidos (STOPPED) cuando han sido pausados por una señal o por el proceso padre. El estado NORMAL persiste hasta que el proceso completa su tarea.

Cuando un proceso ha finalizado su ejecución, entra en el estado ZOMBIE. En este estado, el proceso ha liberado sus recursos pero aún no ha notificado formalmente su terminación al proceso padre. Es responsabilidad del sistema operativo limpiar los procesos en estado ZOMBIE y comunicar al proceso padre que el hijo ha finalizado.

La relación entre estos estados y el planificador 4BSD es crucial para comprender cómo se gestionan los recursos del sistema y cómo se asigna el tiempo de CPU a los procesos en FreeBSD.

En la tabla 1, se proporciona una descripción concisa de cada estado del proceso y su significado en el contexto del sistema operativo.

| **Estado** | **Características** |
|---|---|
| NEW | En fase de creación, aún sin recursos asignados |
| NORMAL | Ejecución activa, sus hilos alternaran entre *RUNNABLE*, *SLEEPING* o *STOPPED* |
| ZOMBIE | En fase de finalización |

Tabla 1: Descripción de los estados del proceso en FreeBSD

En FreeBSD, el sistema organiza las estructuras de procesos en dos listas principales: *zombproc* y *allproc*. Los procesos en estado ZOMBIE se encuentran en la lista *zombproc*, mientras que los procesos activos están en la lista *allproc*. Esta distinción permite optimizar las operaciones del sistema, como la llamada al sistema *wait* que busca procesos terminados, así como las operaciones del planificador que identifican procesos listos para ejecutarse.

Los hilos de un proceso, por su lado, excluyendo los que están en ejecución, se distribuyen en tres colas principales: *run_queue*, *sleep_queue* y *turnstile_queue*. Los hilos listos para ejecutarse se ubican en la *run_queue*, mientras que aquellos que están bloqueados esperando eventos se encuentran en la *sleep_queue* o en la *turnstile_queue*. Es importante destacar que las colas *run_queue* están organizadas según la prioridad de planificación de hilos establecida por el planificador 4BSD. La diferencia entre la *turnstile_queue* y la *sleep_queue*, radica en que esta última se utiliza para hilos bloqueados con locks de tipo *sleepable*, mientras que la *turnstile_queue* alberga hilos bloqueados con locks de tipo *non-sleepable*.

### 2.1.4. Prioridad de los hilos

Las prioridades de los hilos son un componente importante para la planificación. Estas prioridades, que van desde 0 hasta 255 (donde 0 denota la prioridad más alta), determinarán el orden en que se ejecutarán los hilos. En la Tabla 2, se describen los diferentes rangos de prioridades.

Las prioridades en el rango de 0 a 47 son asignadas de forma predeterminada por el sistema y se destinan a las tareas de interrupción.

Las prioridades de los hilos en tiempo real se encuentran en el intervalo de 48 a 79 y deben ser configuradas previamente por las aplicaciones mediante la llamada al sistema *rtprio*. A continuación, se encuentran los hilos con prioridades en el rango de 80 a 119, conocidos como hilos del kernel superior (top-half kernel threads). Estos hilos se encargan de gestionar operaciones críticas del kernel que afectan a todo el sistema.

Los hilos con prioridades entre 120 y 223 pertenecen a la clase de hilos de tiempo compartido. Están destinados a ejecutar tareas de usuario convencionales y sus prioridades son ajustadas de manera automática por el kernel en función del uso de la CPU.

Cuando no hay tareas activas que requieran el uso de la CPU, los hilos de la clase ÏDLE"pueden ejecutarse. Estos hilos tienen la finalidad de mantener el sistema en un estado inactivo, consumiendo recursos mínimos y estando listos para responder a tareas prioritarias.

| **Rango** | **Clase** | **Tipo de hilo** |
|---|---|---|
| 0 - 47 | ITHD | Bottom-half kernel (interrupt) |
| 48 -79 | REALTIME | Real-time user |
| 80 - 119 | KERN | Top-half kernel |
| 120 - 223 | TIMESHARE | Time-sharing user |
| 224 - 255 | IDLE | Idle user |

Tabla 2: Clases de hilos por rango de prioridad.

## 2.2. Planificación

Planificar es decidir cómo, cuándo y por cuánto tiempo vamos a correr los hilos que se encuentran en nuestro sistema; tanto los hilos propios del sistema operativo, como de aplicaciones que se encuentran en ejecución.

En este sentido, la planificación es fundamental para equilibrar la utilización de los recursos con el tiempo necesario para completar los programas. FreeBSD implementa por defecto, un planificador *compartido por tiempo*, el cual calcula la prioridad de los procesos de manera periódica. Este cálculo se realiza en base a datos previos, como la cantidad de tiempo utilizado de CPU o la cantidad de recursos de memoria que el proceso mantiene o requiere para su ejecución. No obstante, algunas tareas

requieren un control más preciso sobre el proceso, como el caso de la planificación de tiempo real. FreeBSD también implementa esta funcionalidad mediante una cola separada para los hilos en cuestión, los cuales no se ven interrumpidos por otros hilos a menos que tengan igual o mayor prioridad.

Además, el *kernel* de FreeBSD cuenta con una cola de hilos de mínima prioridad que se ejecutan únicamente cuando ningún otro hilo en las colas de mayor prioridad está en un estado de posible ejecución.

En cuanto al método de planificación por tiempo, FreeBSD favorece a los programas interactivos. Asigna una prioridad alta a cada hilo y permite que se ejecute por un periodo fijo de tiempo, conocido como *time slice*. A medida que el hilo se ejecuta, su prioridad disminuye, mientras que aquellos suspendidos por E/S mantienen su prioridad. Por su parte, los hilos que se mantienen inactivos mejoran su prioridad en la cola.

## 2.3. Conceptos del proyecto integrador previo

### 2.3.1. Introducción

El proyecto integrador del cual partimos, consiste en modelar el planificador 4BSD con Redes de Petri. Los hilos de ejecución de un proceso como el planificador del sistema operativo pueden considerarse como sistemas que pueden modelarse utilizando esta herramienta.

Las decisiones de encolado de los hilos en una CPU se toman a partir de la información representada en el modelo; así como también los estados globales y los de cada hilo.

TODO: Agregar un poco mas para mejorar

### 2.3.2. Elección del planificador

La planificación a corto plazo en FreeBSD ha experimentado una notable evolución a lo largo de los años. Desde sus inicios con el planificador 4BSD hasta el actual planificador ULE, se han implementado cambios significativos que han contribuido a mejorar la eficiencia y el rendimiento del sistema.

Aunque el planificador ULE es el predeterminado en las versiones actuales de FreeBSD, este trabajo se basa en el trabajo previo realizado en el marco del proyecto integrador, que se centró en el planificador 4BSD. Para entender con mayor detalle y contexto dicha elección, visitar la sección 2.3.3. del proyecto integrador previo[1].

En este momento, nuestro enfoque principal no radica en realizar un cambio inmediato en el sistema operativo ni en contribuir directamente a la comunidad de FreeBSD. En su lugar, estamos continuando con la fase de investigación e implementación centrada en este tipo de planificadores.

Al mismo tiempo, el planificador 4BSD ha sido mantenido por la comunidad de FreeBSD durante décadas y no hay planes inmediatos para dejar de darle soporte. Esto significa que todavía sigue siendo una opción estable y confiable para el desarrollo de nuestro proyecto.

— PLANIFICADOR

### 2.3.3. Funcionamiento del planificador 4BSD

El planificador 4BSD, inicialmente diseñado para sistemas monoprocesador, ha evolucionado para adaptarse eficazmente a sistemas multiprocesador. Comprender su funcionamiento es esencial para apreciar cómo este componente crítico del sistema FreeBSD gestiona la asignación de recursos y el tiempo de ejecución de los hilos.

En esencia, el planificador 4BSD organiza los hilos en múltiples colas según su prioridad. Cada hilo tiene una prioridad asignada y reside en una cola específica de acuerdo con esta prioridad.

Un aspecto crucial del planificador 4BSD es la gestión de tiempos de ejecución equitativos. A cada proceso se le asigna un pequeño período de tiempo, conocido como "timeslice". Una vez que un proceso agota su timeslice, se suspende temporalmente para permitir que otros procesos tengan la oportunidad de ejecutarse. Este proceso cíclico de asignación de tiempos de ejecución, basado en el algoritmo Round Robin", asegura que ningún proceso monopolice los recursos del sistema durante largos períodos, contribuyendo así a un rendimiento estable y justo del sistema, garantizando la equidad en el uso de los recursos del sistema.

En lo que respecta a las prioridades, el planificador 4BSD las ajusta periódicamente en función del tiempo de CPU consumido por cada hilo. Este proceso implica elevar la prioridad de los hilos que han utilizado menos tiempo de CPU y reducir la prioridad de aquellos que han consumido más recursos de procesador. De esta manera, el planificador se encarga de equilibrar la carga de trabajo de manera equitativa entre todos los procesos en ejecución en el sistema.

#### Funciones basicas del planificador

intro en donde se explica que todas las funciones de aca abajo basicamente fueron las que tuvieron que cambiar para la planificacion con RdP

Encolado (add) El sistema usa 64 colas, seleccionando una cola para un determinado hilo y dividiendo la prioridad del hilo por 4. Para ahorrar tiempo, los hilos en cada cola no se vuelven a dividir en prioridades.

Estas colas pueden ser de tipo run queue, turnstile queue o sleep queue. Los hilos en estado RUNNABLE se ubican en las colas de tipo run queue; mientras que los que están bloqueados o esperando un evento, son posicionados en los otros dos tipos de colas.

Si un hilo agota su tiempo (o intervalo de tiempo) permitido, se coloca al final de la cola de la que procede, y el próximo hilo (ahora al principio de la cola) se selecciona para ejecutarse.

Si un hilo se bloquea, no se vuelve a colocar en la cola de ejecución (run queue). En su lugar, se coloca en una turnstile queue o en una sleep queue.

Las operaciones relacionadas al encolado se realizan dentro de la función sched_add(). Ésta función recibe como parámetros el hilo a encolar y flags con información acerca del mismo.

Su primera tarea es corroborar que el hilo se encuentre en un estado permitido para ser encolado, es decir, en estado CAN_RUN o RUNNING. Al pasar esta verificación, se adquiere el lock del planificador y se lo pasa al estado RUNQ para luego elegir en cuál cola de CPU va a ser asignado. Esto es posible a

través de la función sched_pickcpu(), la cual elige un CPU de la siguiente forma:

1. Consulta si el hilo ya se había ejecutado en un procesador anterior y si es posible volverlo a encolar en el mismo. En caso de que sea verdadero, almacena este valor en una variable. En caso contrario, dentro de esa variable guarda el valor NOCPU (igual a -1).

2. Luego itera a través de cada CPU disponible en el sistema. En caso de que el CPU que se está iterando no permita el encolado, continúa con el siguiente.

3. Para el caso en que esté permitido el encolado para el CPU que se está iterando, existen dos opciones basadas en el valor de la variable inicializada en el primer punto.

   *a*) En caso de que la variable haya sido inicializada con el valor NOCPU, se la sobreescribe con el CPU de la iteración en la que se encuentra el programa en ese momento.

   *b*) En caso de que la variable haya sido inicializada con el valor del último CPU en el que había sido ejecutado el hilo, se consulta si la cola del CPU que se está iterando, tiene menor cantidad de procesos que la del CPU elegido en el punto 1. Caso afirmativo, se reemplaza el valor de la variable por el CPU que se está iterando, ya que esto significa que éste procesador contiene menos hilos en su cola de ejecución. Caso contrario, el valor de la variable no se sobreescribe.

4. Al finalizar la iteración de cada CPU, se retorna el valor de la variable con el CPU más apropiado para el hilo.

Una vez elegido el mejor CPU para el hilo, se procede a realizar los cambios de contexto necesarios para agregarlo a la cola del procesador correspondiente.

Si el procesador al que se agrega el hilo es diferente al que está ejecutando la función en ese instante, éste último envía una señal IPI (inter-processor interrupt) para comunicar que existe este nuevo hilo en su cola.

En caso de que el hilo se agregara a la cola del CPU actual, se llama a las funciones correspondientes para saber si este hilo tiene mayor prioridad que el que está en ejecución actualmente y debe reemplazarlo, esto se conoce como preemption.

Cambios de contexto (switch y throw) Al hablar de cambios de contexto de los hilos, se hace referencia a dos funciones en particular.

La función sched_switch() se encarga de expulsar al hilo que recibe por parámetro (hilo actual en ejecución), o en caso de que el hilo continúe en estado RUNNING por alguna razón, se vuelve a poner en cola a través de la función sched_add().

Una vez finalizada la expulsión se solicita un nuevo hilo para ejecutar. Esto se hace a través de la función choosethread() y sched_choose().

Una vez obtenido el nuevo hilo, se verifica que éste no sea el mismo que fue puesto en cola anteriormente, si esto sucediera, sólo continúa la ejecución. Pero en caso de que sean dos hilos diferentes, se procede a realizar el cambio de contexto haciendo uso de la función cpu_switch().

La función cpu_switch() guarda el contexto del hilo anterior y restaura el contexto del nuevo hilo. Así como también, se asegura de que el estado del nuevo hilo se marque como TD_RUNNING (en ejecución).

Otra de las funciones que se encargan del cambio de contexto es sched_throw(), la cual se encarga de cambiar un hilo por otro en el mismo CPU que se está ejecutando. Recibe un hilo como parámetro (el cual puede ser nulo), y lo expulsa del planificador. Luego a través de la misma función utilizada anteriormente, choosethread(), obtiene un nuevo hilo y continúa con su ejecución.

Elección de hilos (choose) La elección del próximo hilo a ejecutar se hace a través de la función sched_choose() previamente invocada por la función choosethread() nombrada anteriormente.

El funcionamiento consiste en elegir al hilo con mayor prioridad dentro de alguna de las colas no vacías del CPU o de la cola global. Cada hilo habilitado para ser ejecutado, es decir en estado RUNNABLE, tiene una prioridad asignada.

La función comienza obteniendo el primer hilo de la cola global y el primero de la cola del CPU; guardando ambos en dos variables diferentes. Luego compara las prioridades de ambos y se queda con el de mayor prioridad.

Una vez elegido el hilo, se procede a removerlo de la cola correspondiente, ya sea la global o la del CPU, y se retorna este hilo.

Para el caso en que ambos hilos sean nulos, es decir, no hay hilos para ejecutar, se simula una ejecución con el idlethread y se retorna el mismo.

Remoción de hilos de la cola (rem) A diferencia de la función anterior, en la que al elegir un hilo para correr, éste era removido de la cola en la que estaba ubicado, la función sched_rem() se utiliza para quitar un hilo (especificado por parámetro) de una cola por dos razones principales.

Una de estas es el ajuste de prioridad que consiste en otorgarle una nueva prioridad al hilo por lo que es necesario removerlo de la cola en la que se encuentra y luego volver a encolarlo a través de la función sched_add() en la posición correspondiente.

El otro motivo por el cual se querría hacer uso de esta función es en caso de que un hilo tenga afinidad con algún CPU, por lo que el procedimiento será igual al mencionado en el párrafo anterior, con el objetivo de que se encuentre en la cola correcta.

— RED DE PETRI

### 2.3.4. Modelado del hilo

Uno de los modelos representados a través de la Red de Petri, es el de los estados de un hilo. En la figura es posible observar esta representación.

Agregar red de petri del hilo

- **T0_INIT**: El paso del estado INACTIVE a CAN_RUN. Esto sucede cuando el hilo se agrega al planificador. Esto sucede generalmente en el momento de creación de un proceso o cuando el mismo realiza un fork. Esta tarea no corresponde al scheduler, por lo que inicialmente un hilo en

el planificador se encuentra inicializado en el estado CAN_RUN. Esta transición nunca se dispara, solo se la incorpora al modelo de modo representativo.

- **T1_ON_QUEUE**: El hilo se pone en una cola local de una determinada CPU o en la cola global dependiendo de la disponibilidad. Esta cola organiza los hilos de acuerdo a sus prioridades de ejecución.
- **T2_SET_RUNNING**: El hilo se quita de la cola y pasa a ejecutar las instrucciones del programa que tiene asignadas. En este instante el procesador se encuentra ocupado por dicho hilo.
- **T3_SWITCH_OUT**: El scheduler interrumpe el hilo y lo vuelve a colocar en una cola. El planificador toma otro hilo de la cola (el de mayor prioridad) y realiza un cambio de contexto.
- **T4_TO_WAIT_CHANNEL**: Algún evento, semáforo o espera bloquea al hilo. Se agrega en una sleep queue o turnstile, en la cual el hilo queda a la espera de un evento que le quitará el bloqueo.
- **T5_WAKEUP**: Se desbloquea el hilo y puede volver a encolarse nuevamente. El evento que lo desbloquea se genera fuera del scheduler. El hilo queda a la espera para poder cambiar de estado cuando corresponda.
- **T6_REMOVE**: Se ejecutará cada vez que un hilo deba ser expulsado de la cola en que se encuentra actualmente.

### 2.3.5. Modelado del planificador

Para modelar el planificador se hace un modelo específico para un procesador y se extiende para los demás. En este proyecto se consideran cuatro procesadores.

Agregar red de petri de 1 procesador

Transiciones

- TRAN_ADDTOQUEUE: Un hilo es agregado a la cola del CPU correspondiente. Se encuentra inhibida cuando todo el sistema se encuentra en modo monoprocesador (al inicio del sistema operativo); y cuando ya existe un hilo en la cola.
- TRAN_UNQUEUE: Se quita al hilo próximo a ejecutar de la cola del CPU. En este punto el hilo se encuentra listo para ser ejecutado.
- TRAN_EXEC: El hilo pasa a ejecución por lo que el recurso del procesador se encuentra ocupado. Esta transición elimina el token de la plaza de habilitación, permitiendo así que un nuevo hilo pueda ser encolado. A su vez, ésta transición se encuentra inhibida en el modo monoprocesador.
- TRAN_EXEC_EMPTY: Esta transición se comporta de igual manera que la anterior, pero no depende de la plaza de habilitación. Su utilidad surge para los hilos provenientes de la cola global.

- TRAN_RETURN_VOL: Representa un retorno del recurso procesador para que pueda ejecutar otro hilo de su cola. Más precisamente, se dispara cuando la interrupción de la ejecución se debe a que el hilo no puede continuar porque espera por un evento o un recurso.
- TRAN_RETURN_INVOL: El funcionamiento es igual a la transición anterior, pero en este caso se dispara cuando la interrupción se produce porque el hilo consumió su tiempo asignado de CPU o bien finalizó su tarea.
- TRAN_FROM_GLOBAL_CPU: Representa el desencolado de un hilo desde la cola global.
- TRAN_REMOVE_QUEUE: Expulsa un hilo de la cola y también resta un token de habilitación de la CPU, es decir, se premia a la misma para que pueda encolar.
- TRAN_REMOVE_EMPTY_QUEUE: Su funcionamiento es igual al anterior pero se ejecuta en los casos en que la plaza de habilitación no posea ningún token.
- TRAN_REMOVE_GLOBAL_QUEUE: Expulsa un hilo de la cola global. No premia al CPU.
- TRAN_START_SMP: Se dispara cuando el sistema pasa de monoprocesador a multiprocesador.
- TRAN_THROW: Se ejecutará automáticamente cada vez que todas las plazas de habilitación de las CPU tengan al menos un token. El objetivo de esta transición consiste en habilitar las colas con la menor cantidad de hilos que estaban inhibidas, una vez que todas se han emparejado.
- TRAN_QUEUE_GLOBAL: Agrega un hilo a la cola global.

### 2.3.6. Jerarquía de transiciones

Para llevar a cabo la conexión entre las redes de los hilos y la red de recursos de las CPU se utiliza el concepto de redes jerárquicas. Es decir que al dispararse cierta transición en la red de recursos, también debe dispararse su transición correspondiente en la red del hilo.

Las jerarquías están definidas de la siguiente forma:

- Transiciones TRAN_ADDTOQUEUE y TRAN_QUEUE_GLOBAL de la red de recursos son jerárquicas a la transición T1_ON_QUEUE del hilo.
- Transiciones TRAN_EXEC y TRAN_EXEC_EMPTY de la red de recursos son jerárquicas a la transición T2_SET_RUNNING del hilo.
- Transiciones TRAN_REMOVE_QUEUE, TRAN_REMOVE_EMPTY_QUEUE y TRAN_ REMOVE_GLOBAL_QUEUE de la red de recursos son jerárquicas a la transición T6_ REMOVE del hilo.
- Transición TRAN_RETURN_INVOL de la red de recursos es jerárquica a la transición T3_ SWITCH_OUT del hilo.
- Transición TRAN_RETURN_VOL de la red de recursos es jerárquica a la transición T4_TO _WAIT_CHANNEL del hilo.

### 2.3.7. Marcado inicial

La red de recursos se inicializará siempre con un token en la plaza que indica que el sistema se encuentra funcionando en modo monoprocesador. Además, se inicializan las plazas que representan a las CPU con un token, excepto la de la CPU0 ya que la misma, inicialmente se encuentra ejecutando el hilo inicial del sistema, por lo que esta última debe inicializarse con un token en la plaza de ejecución (PLACE_EXECUTING_0).

# 3. Desarrollo

## 3.1. Introducción

En el marco de este trabajo de investigación, implementamos mejoras en el planificador a corto plazo basado en Redes de Petri para el sistema operativo FreeBSD. Este proyecto supuso un gran desafío, ya que implicó comprender y depurar el código previo y los archivos del código fuente de FreeBSD relacionados con el funcionamiento del scheduler 4BSD, que había sido desarrollado por estudiantes de nuestra misma institución en un proyecto integrador previo. Fue un proceso arduo de lectura y relectura constante, con el objetivo de adquirir un panorama general y detallado sobre el funcionamiento del scheduler.

## 3.2. Metodologías de trabajo

Luego de superar la fase inicial, dedicamos un tiempo a planificar la forma en que íbamos a abordar el desarrollo del proyecto. Creíamos crucial establecer una estructura organizada para avanzar de manera sistemática y dejar registros detallados de nuestras actividades en cada etapa del proyecto, lo cual podría ser útil para trabajos futuros.

Durante esta fase, nos enfocamos en documentar los procesos de instalación y depuración, establecer una estrategia de ramas estandarizada en el repositorio fork de FreeBSD y definir un repositorio externo donde alojar todos los recursos que podrían ser útiles para cualquier persona que desee involucrarse en la implementación del proyecto. Estas tareas fueron de gran importancia para asegurarnos de que estábamos trabajando de manera ordenada y eficiente, y para crear un recurso valioso para la comunidad.

### 3.2.1. Estrategia de ramificación

Definir una estrategia de ramificación o branch strategy en los proyectos es importante porque ayuda a mantener el control sobre el flujo de trabajo, a mantener la organización del proyecto y a asegurar que las modificaciones se integren sin problemas en la rama principal del código. Una estrategia de ramificación bien definida establece un conjunto de reglas claras y consistentes para crear y fusionar ramas de código.

Al mismo tiempo, ayuda a mantener un historial completo y bien organizado de las modificaciones realizadas al código, lo que puede ser útil para fines de seguimiento y continuidad del proyecto. Por esta razón, definimos el prefijo de branches conformado por los apellidos de los integrantes del grupo de trabajo, en nuestro caso, por ejemplo, DrudiGoldmanPI/. Esto también nos ayuda a encontrar con mayor facilidad todos los cambios realizados en este proyecto integrador.

Mostrar las branch strategies

### 3.2.2. Conventional commits

Los Conventional Commits son una herramienta esencial para mantener una estructura y formato consistente en los mensajes de confirmación del código. En el contexto de un trabajo de tesis, su importancia radica en que el trabajo probablemente seguirá avanzando con el tiempo por otros alumnos o personas que deseen seguir el mismo camino.

El uso de Conventional Commits ayuda a mantener una estructura bien organizada de los pequeños pasos que se fueron dando en cada una de las ramas, lo que permite a cualquier persona que consulte el repositorio tener una comprensión clara y rápida del progreso del proyecto y de los cambios que se realizaron en cada paso.

### 3.2.3. Pruebas del kernel

La depuración del kernel es una tarea crucial en el desarrollo de sistemas operativos, ya que permite encontrar y corregir errores que de otra manera podrían pasar desapercibidos. En nuestro trabajo, la herramienta protagonista en este ámbito, fue KGDB.

kgdb permite la depuración de archivos de núcleo del kernel. Cuando se produce un kernel panic o una falla del sistema, podemos utilizar esta herramienta para analizar el dump core y encontrar la causa subyacente del problema. Dentro de este modo, podemos examinar el estado del sistema en el momento de la falla y determinar qué parte del código del kernel causó la falla.

## 3.3. Módulo de actualizaciones

Se inició el proceso de desarrollo del proyecto en sí mismo, que se dividió en módulos. En cada módulo se establecieron objetivos cortos y alcanzables, buscando lograr avances concretos en cada etapa.

En la etapa inicial de nuestro trabajo, nos centramos en una tarea crucial dentro de cualquier proyecto de software: mantenerlo actualizado en los diferentes lanzamientos del Sistema Operativo. Esta práctica nos brinda una serie de beneficios significativos en términos de rendimiento, estabilidad y compatibilidad, lo que a su vez resulta en un sistema operativo más eficiente y confiable en general. Además, al llevar a cabo este proceso de actualización, abrimos la posibilidad de contribuir al proyecto de FreeBSD en colaboración con futuros trabajos realizados por estudiantes de nuestra misma universidad. Comenzamos nuestro trabajo en la versión 11 (realizado por Nicolas Papp y Tomas Turina) y continuamos hasta alcanzar la última versión estable, la 13.1, que representa el estado actual del sistema operativo.

Para comprender mejor los cambios significativos entre las versiones, hemos dividido el módulo en tres partes progresivas: actualización a la versión máxima del 11, actualización del 11 al 12 y actualización del 12 al 13.

### 3.3.1. Actualización a la versión máxima del release 11

En esta primera iteración se buscó llevar los cambios del proyecto integrador previo realizados en la versión 11.0.0 del sistema operativo FreeBSD a la última versión de éste release, es decir, al 11.4.0.

Durante dicha actualización, nos encontramos con un cambio significativo que tuvo un impacto relevante en nuestra tesis: la modificación en la función maybe_preempt. En la versión anterior, esta función se encargaba de determinar si un nuevo hilo debía tomar el control del procesador inmediatamente, reemplazando al hilo actual en ejecución. En caso afirmativo, se realizaba el cambio de contexto y comenzaba su ejecución, sustituyendo al hilo que se encontraba en el procesador.

En la versión más reciente, se introdujo una nueva estrategia, posponer el cambio de contexto para un momento más adecuado mediante un sistema de banderas. En lugar de realizar el cambio de contexto de manera inmediata, se implementó un mecanismo donde el hilo recién encolado establece la bandera td_owepreempt en 1, indicando que se necesita desalojar al que se encuentra en ejecución, en favor de dicho hilo.

Este cambio en la función maybe_preempt proporciona una mayor flexibilidad y control sobre el reemplazo inmediato de hilos en el planificador de FreeBSD. En el contexto de nuestra tesis, este ajuste fue relevante, ya que tuvimos que tener en cuenta esta nueva lógica al analizar y evaluar el comportamiento del planificador.

### 3.3.2. Actualización a versión 12

Tras analizar las diferencias entre la versión 11.4.0 (último release de la 11) y la versión 12.3.0 (último release de la 12), se determinó que los cambios eran mínimos sobre los archivos de relevancia para nuestro proyecto del planificador mediante el uso de Redes de Petri; por esta razón, no se llevó a cabo la actualización. En su lugar, nos enfocamos en las actualizaciones posteriores que presentaban cambios más significativos en el planificador, permitiéndonos un análisis más profundo y enriquecedor de nuestra investigación.

### 3.3.3. Actualización a versión 13

La versión 13 trae con ella varios cambios significativos en el código del scheduler. Los cambios principales, y que mayor tiempo de análisis e investigación tomaron, fueron los cambios dentro de la función sched_switch(), la función encargada de realizar el cambio de contexto entre hilos en el planificador de tipo 4BSD.

El primero de ellos (appendix diff) ya se encuentra los parámetro de dicha función: newtd se utilizaba para indicar explícitamente el hilo que se iba a ejecutar a continuación. Sin embargo, en la versión nueva, el parámetro newtd se eliminó. La eliminación de éste parámetro y la inclusión de la lógica de selección del hilo dentro de sched_switch simplifica el código y puede permitir una mayor flexibilidad en la selección del hilo y adaptarse mejor a posibles cambios o mejoras en la política de planificación en futuras versiones de FreeBSD.

El segundo cambio (appendix diff) fue el que más problemas trajo a la hora de la actualización con los cambios de la Red de Petri al release 13 y se trató principalmente sobre la reorganización de los locks tomados por los hilos, con la finalidad de mejorar la eficiencia y evitar interrupciones innecesarias en el camino crítico de ejecución.

### 3.3.4. Resultados

Durante la actualización a las diferentes versiones del sistema operativo FreeBSD, se logró garantizar el funcionamiento correcto de todas las versiones, lo cual fue un logro significativo. Esto permitió no solo mantener el proyecto al día, sino también adentrarnos en el código del sistema operativo y comprender su funcionamiento de manera más integral.

Se tuvo la oportunidad de leer y analizar el código fuente en profundidad. Esto nos permitió adquirir un conocimiento detallado del proyecto y nos familiarizó con su estructura y características particulares. Esta inmersión en el código nos brindó una base sólida para abordar tareas posteriores de manera más eficiente y efectiva.

Durante el proceso de actualización, también se identificaron y documentaron algunos problemas menores que se trasladaban de versiones anteriores del proyecto. Estos problemas fueron registrados tanto en un repositorio de documentación específico como en este documento, con el propósito de mantener un registro detallado y accesible para futuros trabajos.

### 3.3.5. Próximos pasos

Como se desarrolló previamente, es de suma importancia mantener el proyecto continuamente actualizado con las últimas versiones del sistema operativo.

Para lograr este objetivo, es fundamental estar al tanto de las actualizaciones necesarias y evitar quedarse rezagado con respecto al release más reciente.

La tarea de mantener el software actualizado en proyectos de código abierto como FreeBSD, nos permite trabajar con un sistema operativo más eficiente, seguro y compatible con las tecnologías más recientes. Y no menor, es necesario destacar que facilita la integración con la comunidad de desarrolladores y usuarios.

Al mantener el proyecto actualizado, se está en sintonía con las últimas características, mejoras y correcciones de errores introducidas por la comunidad, lo cual brinda la oportunidad de contribuir al desarrollo colaborativo y recibir retroalimentación valiosa.

Sugerimos que los trabajos futuros en el ámbito del proyecto consideren la importancia que tiene este apartado y se comprometan a dedicar el tiempo y los recursos necesarios para llevarlo a cabo de manera regular.

## 3.4. Módulo de encendido/apagado de los procesadores

Partiendo de un sistema operativo funcional y actualizado, el siguiente enfoque fue implementar una nueva característica que se encuentra dentro del área de ahorro energético y control de los núcleos del procesador.

Más precisamente lo que se busca, es poder controlar el estado de cada núcleo del procesador en el que se está ejecutando el sistema operativo, es decir, tener el control para elegir cuál o cuáles núcleos permitirán encolar y ejecutar hilos de procesos en diferentes momentos temporales y a su vez, que ésto

pueda cambiar dinámicamente de acuerdo a las necesidades del sistema o a definiciones especificadas por el desarrollador.

### 3.4.1. Objetivos

Como objetivo general se propone crear una funcionalidad propia del scheduler para permitir habilitar o inhabilitar los diferentes núcleos del procesador, con el objetivo de trasfondo de ahorrar energía en contextos que así lo requieran.

A partir del objetivo mencionado anteriormente, podemos destacar diferentes situaciones inherentes al mismo, para dejar en evidencia las ventajas de éste módulo.

Es necesario partir de la suposición de que ésta funcionalidad será utilizada en diversos casos. Por ejemplo, cuando el procesador tenga pocas tareas o que las mismas no precisen de una gran capacidad de procesamiento, por lo que podrían ser resueltas con menos núcleos que el total disponible en el sistema.

Otro escenario podría basarse simplemente en la decisión de restringir la cantidad de núcleos disponibles de un sistema, por lo cual se elegiría inhabilitar los demás y de ésta manera, obtener un ahorro en el consumo energético.

La versatilidad de éste módulo, permitiría realizar estas decisiones mientras el sistema operativo se encuentra en funcionamiento, es decir que no sería necesario reiniciarlo para aplicar estos cambios.

### 3.4.2. Primera Iteración: Implementación del módulo de encendido/apagado mediante cambios en la Red

Para comenzar con la implementación del módulo se debatió el enfoque que le daríamos al desarrollo, es decir, si el cambio que debíamos generar iba a ser en las funciones del scheduler directamente o en la Red de Petri. Básicamente definir quién tomaría las decisiones.

Luego de observar con detenimiento el código, optamos por la modificación en la Red de Petri, ya que sería un cambio más favorable a futuro por los beneficios que se obtienen de un sistema de modelado como éste.

El primer acercamiento que se planteó fue el de crear algún indicador que permita conocer el estado de cada procesador, es decir, que tenga la información acerca de su estado, ya sea encendido o apagado.

Para representar esta idea dentro de la Red de Petri, se creó una plaza a la que se le dió el nombre de PLACE_SUSPENDED. En el caso en que la misma se encuentre sin ningún token, se puede deducir que el procesador puede ser utilizado normalmente o que está disponible; caso contrario, el procesador no debería ejecutar ningún hilo.

La plaza de estado del procesador estará acompañada de dos transiciones encargadas de agregar o quitar el token correspondiente al estado del CPU en cuestión. La cantidad de tokens máxima que puede contener esta plaza, es de uno, ya que su funcionamiento refleja un comportamiento booleano. Para asegurar que esta lógica se cumpla, se agregó un arco inhibidor a la transición que aporta tokens a la plaza, es decir, una vez que agrega un token, ya no se disparará nuevamente hasta que este token

sea eliminado por la transición correspondiente.

Esta plaza es la encargada de inhabilitar las transiciones de encolado correspondientes, la propia del CPU y la global. De esta forma cuando el CPU se encuentra suspendido, internamente se evita que algún hilo pueda ingresar a la cola y luego ejecutarse.

Un detalle que se debe tener en cuenta es que éste desarrollo, si bien evita que el procesador continúe encolando hilos, no elimina aquellos que ya se encontraban dentro de la cola asociada al CPU. En otras palabras, los hilos ya encolados, serán ejecutados en el momento correspondiente pero no podrán volver a la cola de este CPU inhabilitado.

Ésta aclaración se puede observar en la Red de Petri directamente, ya que la nueva plaza de suspensión sólo inhibe la transición de desencolado de la cola global y la transición encargada de agregar hilos a la cola del procesador, pero no interviene en las transiciones relacionadas a la ejecución de hilos, ni en la de desencolado propia del procesador (TRAN_UNQUEUE).

IMAGEN DE LA RED DE UN SOLO PROCESADOR CON EL SUSPEND PROC PERO SIN EL IDLE

Como primer paso para la implementación a nivel código de esta funcionalidad, fue la modificación de las matrices (matriz base e incidencia) para dar soporte al cambio realizado en la red.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | -1 | 0 | 0 | 0 | 0 | -1 | 0 |
| 1 | -1 | 0 | 0 | 0 | 0 | 0 | -1 | -1 |
| 0 | -1 | 0 | 0 | 1 | 1 | -1 | 0 | 0 |
| 0 | 1 | -1 | -1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | -1 | -1 | 0 | 0 | 0 |

Tabla 3: Matriz base previa a la nueva funcionalidad.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | -1 | 0 | 0 | 0 | 0 | -1 | 0 | 0 | 0 |
| 1 | -1 | 0 | 0 | 0 | 0 | 0 | -1 | -1 | 0 | 0 |
| 0 | -1 | 0 | 0 | 1 | 1 | -1 | 0 | 0 | 0 | 0 |
| 0 | 1 | -1 | -1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | -1 | -1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 |

Tabla 4: Matriz de incidencia base con modulo de encendido/apagado de procesadores.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Tabla 5: Matriz de incidencia previa a la nueva funcionalidad.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

Tabla 6: Matriz de incidencia con las modificaciones pertinentes.

Como complemento al cambio en las matrices, se agregó una función con la posibilidad de ser invocada desde un módulo de kernel, encargada de alternar el estado de un CPU elegido por el usuario que cargue el módulo en cuestión. Es decir, dentro de este módulo se elige el procesador a suspender o encender, luego se compila y carga el mismo.

Una vez implementada esta funcionalidad a nivel código, descubrimos que no funcionaba como se esperaba ya que recibimos alertas de transiciones no sensibilizadas con intentos de disparo.

Para resolverlo comenzamos imprimiendo en la terminal, la información relacionada a los disparos y de esta manera descubrimos que el problema a solucionar estaba relacionado con el funcionamiento del scheduler para el caso en que no existan hilos a ejecutar en la cola propia ni en la global.

En el próximo apartado se explica cómo solucionamos esta problemática.

### 3.4.3. Segunda Iteración: Soporte para el idlethread en la Red

Partiendo de las impresiones en la terminal realizadas en el paso previo, se pudo identificar que se generaba un error en los casos en que un CPU no tenía hilos para ejecutar.

Para resolver esta problemática, es necesario entender que en casos como éste, el scheduler 4BSD, encola y ejecuta el idlethread. Este funcionamiento ya existía previamente al inicio del proyecto integrador; es propio del sistema operativo.

El idlethread representa un hilo dummy, es decir un hilo que no genera una carga para el procesador que lo ejecute. Es posible encolarlo, suspenderlo y ejecutarlo como cualquier otro hilo.

Cada núcleo del CPU tiene asociado su idlethread, el cual ejecuta continuamente un bucle simple que normalmente implica detener la CPU o ejecutar una instrucción de bajo consumo. De este modo, se asegura de que la CPU permanece ocupada incluso cuando no hay otras tareas o hilos listos para ejecutarse.

En lugar de permitir que un núcleo de CPU permanezca libre, FreeBSD lo mantiene ocupado con el idlethread. Este enfoque tiene beneficios que van desde un menor consumo de energía hasta mejoras en la capacidad de respuesta y en el rendimiento general del sistema.

Cuando no hay hilos o procesos ejecutables listos para ejecutarse en una CPU, el planificador selecciona el idlethread como el siguiente hilo a ejecutar. Este hilo es el que tiene la prioridad más baja entre todos, por lo que sólo se programa cuando no hay otras tareas de mayor prioridad para ejecutar, y cualquier otro hilo lo reemplazará en caso de necesitar tiempo de procesador.

Partiendo de esta base pudimos concluir que nuestra Red de Petri planteada como modelo del planificador, no estaba representando correctamente lo que sucedía internamente. Por esto se decidió hacer ajustes en la red de recursos para contemplar este caso.

El problema surgía al intentar disparar la transición TRAN_UNQUEUE del procesador suspendido ya que no existían hilos en la cola asociada al mismo. Pero el motivo por el cual se intentaba este disparo, es justamente por la funcionalidad ya integrada del idlethread, es decir, que el planificador desencola este hilo para mantener al núcleo del procesador activo. Por esto es que se observaba una inconsistencia con el modelo.

Para resolver esta problemática decidimos agregar una nueva transición (TRAN_EXEC_IDLE) que permite ejecutar el idlethread cuando no exista ningún hilo en la cola del procesador, incluyendo de esta forma, el caso en que el CPU se encuentre suspendido. Una vez aplicados estos cambios, el sistema representó fielmente el proceso interno llevado a cabo por el planificador.

IMAGEN DE LA RED DE UN SOLO PROCESADOR CON EL SUSPEND PROC CON EL IDLE

Para desarrollar esta actualización dentro del código se decidió realizar modificaciones de la Red de Petri de recursos, en las matrices base y de inhibición.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | -1 | 0 | 0 | 0 | 0 | 0 | -1 | 0 | 0 | 0 |
| 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | -1 | 0 | 0 |
| 0 | -1 | 0 | 0 | -1 | 1 | 1 | -1 | 0 | 0 | 0 | 0 |
| 0 | 1 | -1 | -1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | -1 | -1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 |

Tabla 7: Matriz base con las modificaciones pertinentes.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

Tabla 8: Matriz de incidencia con las modificaciones pertinentes.

Además de modificar el archivo base de la Red de Petri, se agregaron cambios dentro de la función sched_choose (encargada de indicar cuál es el mejor hilo a ejecutar) del planificador para que en los casos de CPU suspendido, el hilo elegido para pasar a ejecución sea el idlethread hasta que el procesador se activado nuevamente.

Para lograr esto, se consulta directamente a través de una función, si la plaza de suspensión del procesador que se encuentra ejecutando el código, contiene algún token. En caso de que esto sea verdadero, estaríamos ante la presencia de un procesador suspendido. Caso contrario el funcionamiento es el habitual.

### 3.4.4. Resultados

Una vez finalizado el desarrollo de esta nueva funcionalidad pudimos comenzar a realizar las pruebas pertinentes.

En resúmen, la funcionalidad propuesta para el planificador, que permite habilitar o inhabilitar núcleos del procesador, ha sido resuelta permitiendo así un control sobre los mismos que previamente no existía.

A su vez, se logró la versatilidad propuesta para este módulo, ya que es posible poner en funcionamiento esta mejora en tiempo real sin necesidad de reiniciar el sistema operativo.

Los detalles de los resultados serán explicados con mayor profundidad en el capítulo de “Análisis de resultados”.

Un detalle a tener en cuenta es la imposibilidad de suspender el procesador cero. Esto es debido a que el sistema operativo utiliza este núcleo por defecto para tareas esenciales y al apagarlo, el sistema comienza a ralentizarse hasta que el kernel entra en pánico, resultando en un reinicio del mismo.

Para evitar que esta situación pueda darse, se realizaron los cambios necesarios en el código para prohibir la elección del núcleo en cuestión.

### 3.4.5. Próximos pasos

La implementación de este módulo no tiene mejoras o impacto inmediato en el sistema; debido a que es un módulo independiente, activado mediante una señal enviada en un momento elegido por el desarrollador.

Observando a futuro, creemos que el próximo enfoque debería apuntar a una integración de esta funcionalidad con elementos propios del sistema operativo, tales como hilos, procesos o programas que controlen los mismos.

De esta forma se cambiaría el estado de los procesadores en tiempo real y de acuerdo a necesidades reales del sistema en conjunto; no solo a través de un módulo de kernel.

## 3.5. Módulo de monopolización de hilos por parte de los procesadores

...https://github.com/drudilea/freebsd-src/compare/DrudiGoldmanPI/feature_cpuOnOffModule-13.1.0\protect\protect\futurelet\ellipsis@token\let\ellipsis@one=,\def{.:;!?}.\kern\fontdimen3\font.\kern\fontdimen3\font.\spacefactor\@m{}\relaxdrudilea:freebsd-src:DrudiGoldmanPI/testing_cpuMonopolized-13.1.0_titens

Una vez finalizado el módulo de encendido/apagado del que se comentó en la sección previa, se plantearon diferentes formas posibles para la continuación del trabajo integrador. Una de ellas, fue la monopolización de hilos por parte de los procesadores. En este contexto, se ha introducido la capacidad de anclar hilos a una CPU específica, lo que implica que dicha CPU solo puede encolar y ejecutar ese hilo en particular.

### 3.5.1. Objetivos

El objetivo de este módulo es proporcionar al planificador 4BSD la capacidad de anclar hilos a procesadores específicos en cualquier momento durante la ejecución del sistema operativo. Esto permitirá definir políticas de asignación de recursos más precisas y adaptables a las necesidades del sistema y garantizar una distribución óptima de la carga de trabajo en el hardware disponible.

Es importante destacar que el módulo de monopolización de hilos se limitará a la funcionalidad de anclado en sí mismo, pero a partir de ello, se abre un abanico de posibilidades de mejora en el sistema operativo, tales como mayor flexibilidad al sistema para adaptarse a cambios en la carga de trabajo o a requisitos específicos de algunas tareas, asignación de hilos críticos o de alta prioridad a núcleos de procesadores específicos mejorando su tiempo de respuesta y reduciendo la posibilidad de conflictos de acceso a recursos.

Con la implementación exitosa de este módulo, se espera que el planificador 4BSD de FreeBSD obtenga una funcionalidad avanzada y altamente configurable, brindando una solución potente y flexible para la asignación y gestión de hilos en entornos de computación de alto rendimiento.

### 3.5.2. Diseño de la implementación

La implementación del módulo de monopolización de hilos por parte de los procesadores en el planificador del sistema operativo FreeBSD requirió una etapa amplia de planificación y diseño. Durante esta fase, se exploraron diferentes soluciones con el objetivo de lograr la funcionalidad deseada.

A grandes rasgos, se consideraron dos enfoques principales para la implementación. El primero, implicaba realizar modificaciones tanto en la Red de Petri de recursos como en el código del planificador. Esta opción requería adaptar la estructura de la Red para incorporar el concepto de monopolización y realizar los cambios correspondientes en el código para garantizar la correcta ejecución de los hilos monopolizados.

El segundo enfoque consistía en ajustar únicamente el código del planificador sin modificar la Red. En este caso, se buscaba encontrar una manera de introducir la funcionalidad de monopolización dentro del código existente. Esto implicaba analizar y modificar las secciones pertinentes del código del planificador para asegurar que los hilos pudieran ser asignados y ejecutados de manera exclusiva en una CPU específica.

La elección entre estos dos enfoques se basó en consideraciones técnicas y de viabilidad. Se evaluaron factores como la complejidad de las modificaciones requeridas y la compatibilidad con la Red existente. Tras un análisis y evaluación, se optó por la segunda opción.

### 3.5.3. Implementación

Para comprender el desarrollo del módulo, es fundamental entender cómo el planificador 4BSD gestiona los cambios de contexto. Cuando un hilo en ejecución en el procesador agota su tiempo de ejecución o queda bloqueado por una operación de E/S, se genera una interrupción que indica la necesidad de cambiar el contexto y seleccionar un nuevo hilo para la ejecución. En este punto, se hace un llamado a la función mi_switch( ), encargada de llevar a cabo el cambio de contexto en general.

mi_switch( ) llama a sched_switch( ), método encargado de salvar el estado del hilo a expulsar en la estructura de control (en caso de que se opte por el cambio), y limpiar las banderas y contadores relacionados con la planificación en dicho hilo para que pueda ser correctamente reprogramado en el futuro.

sched_switch( ) hace un llamado a la función sched_add( ) para agregar el hilo nuevamente a la cola de algún procesador. Lo realiza teniendo en cuenta su afinidad, sensibilidad en las transiciones de la red y las políticas específicas del planificador. De esta forma, se garantiza que el hilo se gestionará adecuadamente para su ejecución futura.

Por último, el scheduler busca el nuevo hilo a ejecutar mediante sched_choose( ), removiendolo de la run queue mediante el disparo de la transición correspondiente. Cabe destacar que en esta sección, donde elegimos el nuevo hilo, comparamos si es el mismo que queríamos sacar, y en base a esta

comparación, realizamos o no el cambio de contexto mediante cpu_switch( ).

Como se detalló en el apartado de objetivos, en este módulo se busca brindarle la posibilidad al scheduler de anclar hilos determinados a los diferentes CPUs manejados por el sistema operativo, haciendo que dichos CPUs ejecuten ese y solo ese hilo en todo momento, hasta que se desee lo contrario. Para lograrlo, se realizaron cambios dentro de la función sched_add( ) nombrada previamente.

El módulo consiste básicamente en un arreglo (pinned_threads_per_cpu) de tantos elementos como procesadores haya en el sistema, en donde la posición de cada elemento se corresponde con el ID del procesador. El valor de cada uno de los elementos del arreglo corresponde al ID del hilo que está tomando control de ese núcleo. Se utiliza el valor -1 en los casos en los que el procesador se encuentra libre para cualquier hilo.

En el siguiente ejemplo, se detalla un caso en el que el hilo con ID 100101 tomó control sobre el CPU1 y donde el resto de los procesadores se encuentran funcionando normalmente.

int pinned_threads_per_cpu[CPU_NUMBER] = { -1, 100101, -1, -1 };

Al momento de encolar el hilo, se busca cuál de los procesadores disponibles sería la mejor opción para continuar la ejecución del mismo. Ésta decisión se toma dentro del método resource_choose_cpu desarrollado en el trabajo integrador previo y extendido actualmente, para hacer uso de este nuevo arreglo de hilos asociados a procesadores. Recibe como parámetro el hilo que se encuentra a encolar, y cuenta con tres condicionales que determinarán dicho procesador:

- Como primera condición, si el hilo se encuentra dentro del arreglo de pinned_threads_per_cpu ya estamos en condiciones de elegir dicho procesador como el indicado para el encolado.
- Si el hilo no se encuentra dentro del arreglo, se intenta asignar a la cola del último procesador en el que se ejecutó. Esto es posible solo si la transición TRAN_ADDTOQUEUE de dicho procesador se encuentra sensibilizada y el procesador no está monopolizado por otro hilo.
- Por último, si no se cumplen ninguna de las dos condiciones previas, se recorren los diferentes núcleos del procesador y se retorna el primero que cumpla las condiciones necesarias para el encolado.

Como parte del desarrollo del módulo, también se implementaron algunos métodos complementarios encargados de manejar el estado del arreglo pinned_threads_per_cpu.

- toggle_pin_thread_to_cpu: Método encargado de conmutar el estado de monopolización de un procesador. Recibe el ID de un hilo y de un procesador como parámetros y realiza diferentes operaciones de acuerdo al estado del arreglo:
  - Si el procesador se encuentra libre, se agrega el ID del hilo a la posición correspondiente en el arreglo.
  - Si el procesador ya estaba monopolizado por otro hilo, se sobreescribe con el nuevo ID.
  - En caso de que el hilo ya se encuentre monopolizando al procesador, lo libera escribiendo el valor -1 en la posición correspondiente.

- cpu_available_for_thread: Método utilizado por resource_choose_cpu para saber si un hilo puede utilizar un procesador, o si este se encuentra monopolizado por otro. Recibe el ID de un hilo y de un procesador como parámetros y retorna 1 en caso de que el procesador se encuentre habilitado para encolar dicho hilo; retorna 0 en caso de que dicho procesador se encuentre tomado por otro hilo.
- get_monopolized_cpu_by_thread_id: Método utilizado por resource_choose_cpu para obtener el ID del procesador al que se encuentra asociado el hilo enviado por parámetro. Retorna -1 en caso de que no esté anclado a ningún CPU.

### 3.5.4. Resultados

Los resultados del módulo fueron exitosos, mediante el código desarrollado, se logró la monopolización de procesadores por parte de los hilos.

Para visualizar este comportamiento se llevaron a cabo pruebas utilizando una herramienta de monitoreo y un programa de estrés, similar al módulo de encendido y apagado. En el programa de estrés, se generan cuatro subprocesos que se ejecutan en todos los núcleos del sistema durante un período de tiempo significativo.

Una vez iniciado el programa, seleccionamos uno de los hilos correspondientes a estos subprocesos y lo vinculamos explícitamente a uno de los procesadores disponibles (entre CPU1 y CPU3). Esto nos permitió observar cómo el hilo seleccionado permanece constantemente en el procesador al que se ha anclado, y cómo este último no ejecuta ningún otro hilo que no esté vinculado.

Como parte del resultado, también aclaramos que no se encuentra contemplado el caso del CPU0 en este módulo, ya que al igual que en el modulo de encendido/apagado, nos trajo problemas a la hora de la monopolización, debido a que es el encargado de manejar algunas tareas de administración del sistema.

Los detalles de los resultados serán explicados con mayor profundidad en el capítulo de “Análisis de resultados”.

### 3.5.5. Próximos pasos

Al igual que en el módulo previo, la implementación de este módulo no tiene mejoras o impacto inmediato en el sistema; esto es debido a que son módulos completamente independientes que son activados por señales o por alguna otra parte del kernel.

Puede ser especialmente útil en escenarios en los que se necesite garantizar la ejecución de tareas críticas en tiempo real o cuando ciertos hilos requieran una capacidad de procesamiento dedicada y preferente.

Además de la priorización manual de hilos, esta modificación puede tener otras funcionalidades relacionadas. Por ejemplo, podría permitir la asignación de hilos a procesadores específicos según criterios como la afinidad de memoria o la afinidad de caché, optimizando así el rendimiento del sistema. También podría ser utilizado en entornos de computación distribuida, donde se necesite asignar tareas a CPUs específicas para aprovechar recursos especializados.

## 3.6. Tareas Extra

Durante el proceso de desarrollo de los módulos, se fueron trabajando además algunas otras problemáticas encontradas.

### 3.6.1. Solución del problema de afinidad (procesadores sobrecargados)

La primera de ellas, y la más crucial, fue un problema encontrado con la afinidad de los procesos.

El síntoma ocurría durante nuestras pruebas de estrés; nos dimos cuenta de que los procesadores nunca alcanzaban el 100 % de su capacidad. Descubrimos que esto se debía a un problema de asignación de CPU que generaba una sobrecarga (overhead). El sistema operativo realizaba demasiados cambios de contexto entre los hilos, debido a un cambio en el código en relación con la afinidad, la limitación y la fijación a núcleos específicos.

Estos cambios provocaban que el planificador eligiera los procesadores de forma aleatoria para encolar los hilos, sin considerar las indicaciones que podrían mejorar el rendimiento en la asignación de recursos y los cambios de contexto.

IMAGEN DEL HTOP

A continuación, se dejan algunas pruebas de estrés realizadas previa y posteriormente a la implementación de la mejora. Si se observa detalladamente el programa htop en cada uno de los casos, se puede ver que en el primero, el promedio de uso de los procesadores es de un 78,62 % y el cálculo de todos los números primos hasta 100.000 tomó once segundos realizarlo (ocho veces, una por cada subproceso).

En la segunda imagen, se muestran los resultados obtenidos al hacer los cambios correspondientes para mantener las funcionalidades de afinidad, limitación (bound) y la fijación a núcleos específicos (pin) en el planificador.

Los cambios fueron positivos, logrando un uso pleno de los procesadores, que se mantuvieron al 100 % durante todas las pruebas de estrés realizadas y redujeron el tiempo de cálculo en un 63,63 %, completando la tarea en cuatro segundos.

SEGUNDA IMAGEN DEL HTOP

### 3.6.2. Problema del uso de la placa de red en el sistema operativo

Durante el desarrollo del proyecto integrador nos encontramos con otro problema recurrente relacionado a la placa de red del sistema operativo en sus diferentes versiones.

Precisamente este problema surgió al utilizar SSH para facilitar la conexión con la máquina virtual. De todas formas, se observó que el problema no se producía si se iniciaba la máquina virtual con la placa de red desactivada evitando así el uso de la funcionalidad SSH.

El problema en sí, consiste en un kernel panic relacionado a un error por page fault. Esto quiere decir que el sistema ingresa en un estado de pánico en el cual no se permite seguir interactuando con el mismo; se imprime una mínima información relacionada al error en la consola y luego se reinicia. Durante el booteo del sistema, luego de este error, se crea un archivo con la información del sistema previo al kernel panic; ésto se conoce como crash dump file.

Luego de un análisis exhaustivo sobre el problema, decidimos hacer uso del foro oficial de FreeBSD para recibir ayuda de la comunidad. Una vez planteada la situación que enfrentábamos, obtuvimos diferentes puntos de vista y posibles soluciones al problema, que desafortunadamente no nos llevó a la resolución del problema.

Agregar link al hilo del blog en el anexo

Igualmente, gracias a esta comunidad, pudimos aprender en profundidad acerca de los diferentes debuggers para el kernel del sistema operativo, a interpretar correctamente la información contenida en estos crash dumps files generados, y encontramos en FreeBSD una comunidad muy interesada con el proyecto del planificador mediante Redes de Petri y dispuesta a ayudar a solucionar problemáticas en este tipo de casos.

IMAGEN DE KGBD

Creemos que este es un punto a investigar y mejorar a futuro, ya que, si bien no es bloqueante, contribuye a una mejor experiencia al desarrollar y trabajar con el proyecto.

# 4. Análisis de resultados integrales

En este capítulo, se presentan los resultados integrales de nuestra investigación, que se centra en dos aspectos cruciales para la optimización de la gestión de recursos en sistemas operativos. En primer lugar, introduciremos los resultados del módulo de Encendido y Apagado de Procesadores. En segundo lugar, se presentarán los del módulo de Monopolización de Núcleos.

Ambos mecanismos desarrollados proporcionan al sistema operativo y a la red las herramientas necesarias para mejorar la gestión de sus recursos. Es fundamental tener presente que estos mecanismos representan un paso esencial hacia la consecución de objetivos más amplios, específicamente relacionados con mejoras en la eficiencia energética y el rendimiento general del sistema. La exposición de resultados tiene como objetivo destacar la funcionalidad adecuada de la implementación de los módulos previamente desarrollados en este informe, acercándonos a los objetivos más amplios mencionados.

Como se detalló en la sección de desarrollo de este informe, previo a la implementación de los módulos mencionados llevamos a cabo una actualización del código del kernel, para incorporar al último lanzamiento estable de FreeBSD, las modificaciones realizadas en el proyecto integrador previo. Decidimos incluir los resultados de esta actualización al final de este capítulo, dado que consideramos que los módulos son más propensos a un análisis detallado y a comprender sus resultados. En contraste, las actualizaciones simplemente requieren que el sistema continúe funcionando correctamente.

ESCRIBIR ALGO SOBRE LAS HERRAMIENTAS QUE SE IMPLEMENTARAN A LA HORA DE LOS ANÁLISIS.

## 4.1. Resultados de la Implementación del Módulo de Encendido/Apagado de Procesadores

La implementación del módulo de encendido/apagado de procesadores en FreeBSD, utilizando el enfoque de Redes de Petri, se realizó como punto de inicio del camino hacia la mejora de eficiencia y gestión de recursos del sistema operativo. En esta sección, presentaremos los resultados obtenidos de nuestras pruebas y analizaremos cómo este módulo afecta el rendimiento del sistema.

### 4.1.1. Estado Inactivo del Sistema

En situaciones de inactividad del sistema operativo, caracterizadas por la ausencia de tareas pendientes, cada procesador opera en un modo especial ejecutando hilos de la clase IDLE. Este escenario está diseñado para que el sistema mantenga un estado pasivo, preparado para abordar de manera eficiente cualquier tarea que pueda emerger.

Creemos importante resaltar que, independientemente de si nuestro módulo de encendido/apagado está activo para alguno de los procesadores en este contexto, hemos considerado y abordado esta situación en nuestra implementación para garantizar la estabilidad del sistema. La Figura 3 proporciona una representación visual de este estado.

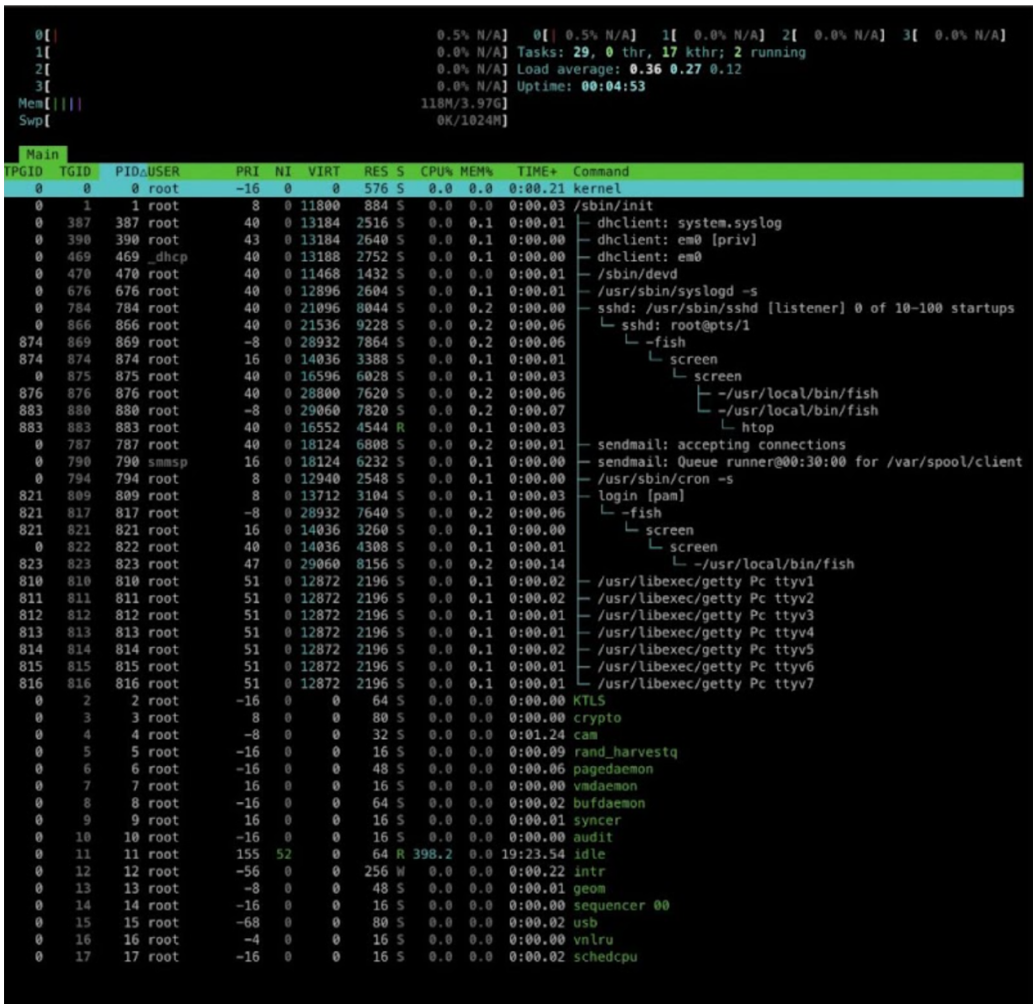


Figura 3: Estado inactivo del sistema.

### 4.1.2. Comportamiento con el Módulo Inhabilitado

Cuando el módulo de encendido/apagado de procesadores no está habilitado para ningún núcleo, el sistema operativo sigue su funcionamiento convencional bajo la lógica del planificador. En este estado, el *scheduler* opera mediante el uso de la Red de Petri con las plazas y transiciones de cada uno de los procesadores del sistema, sin hacer uso de la adición de plazas y transiciones que corresponden al módulo, introducidas en el desarrollo del mismo.

Para confirmar dicho comportamiento, llevamos a cabo una prueba de estrés sobre los procesadores, ejecutando un programa que genera múltiples procesos que realizan cálculos matemáticos complejos. Durante la ejecución del programa, supervisamos el sistema y observamos que todos los procesadores operaban al máximo de su capacidad, funcionando al 100 % para completar la tarea. La Figura 4 proporciona una representación visual de este estado.

Este escenario de funcionamiento con el módulo deshabilitado proporciona un punto de referencia inicial para comprender el impacto de nuestra implementación en la gestión de recursos y el rendimiento general del sistema operativo.

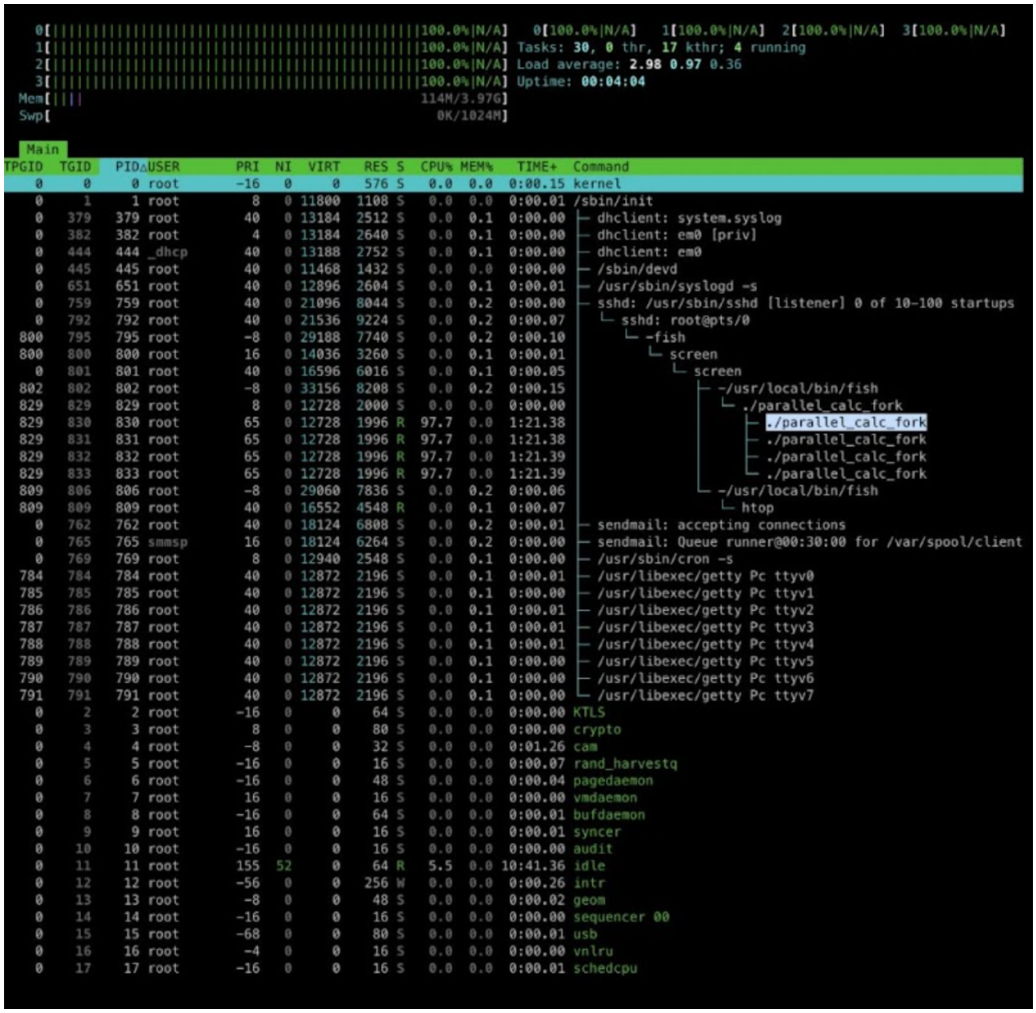


Figura 4: Estado de los núcleos con el modulo encendido/apagado inhabilitado.

### 4.1.3. Comportamiento con el Módulo Habilitado

Basándonos en las observaciones anteriores, llevamos a cabo la misma prueba de estrés, pero activando previamente el módulo de encendido/apagado para inhabilitar el Procesador 2.

Este enfoque nos permitió no solo corroborar la funcionalidad adecuada de la implementación, sino también analizar la respuesta del sistema ante un cambio tan significativo como el bloqueo del encolado en uno de sus núcleos.

En relación a los resultados obtenidos durante esta evaluación, observamos que, al suspender uno de

los procesadores del sistema, este continuó funcionando de manera estable. Se puede apreciar que la carga del procesador suspendido se mantuvo en cero, mientras que los demás procesadores continuaron ejecutando los cálculos. Esto demuestra de manera concluyente que el sistema fue capaz de adaptarse sin dificultades a la reducción de recursos de procesamiento.

IMAGEN CON PROCESADOR SUSPENDIDO

En relación al tiempo necesario para completar el cálculo del programa, notamos que dicho tiempo aumentó proporcionalmente a la disminución en el número de procesadores activos, destacando la correlación entre la disponibilidad de recursos de procesamiento y el tiempo total requerido para finalizar el programa. En la FIGURA TANTO podemos comprobar el funcionamiento del sistema con la inhabilitación simultánea de dos procesadores.

IMAGEN CON DOS PROCESADORES SUSPENDIDOS

TABLA CON LOS TIEMPOS DE EJECUCION CON TODOS LOS PROC, CON 3 Y CON 2 + explicacion + diciendo que el objetivo no es tener apagos los procesadores en tiempos de estres, sino que se muestra solo para ejemplificar, y que la parte del kernel que en un futuro haga uso del modulo para reducir los gastos energeticos, tiene que entender cuando conviene apagar. (Medio como conclusion del modulo)

branch con el codigo

## 4.2. Resultados del módulo de Monopolización de Núcleos

La implementación del módulo de Monopolización de Núcleos en FreeBSD constituyó el próximo paso en el desarrollo de este proyecto integrador. A continuación, se expondrán en detalle los resultados alcanzados al concluir dicho módulo.

### 4.2.1. Estado Normal del Sistema

En este apartado se analizará el comportamiento general del sistema con el módulo de monopolización deshabilitado. Es relevante señalar que bajo esta configuración, los procesadores operan de forma predeterminada, utilizando la Red de Petri como planificador para mantener el equilibrio de la carga. De esta forma, los núcleos que en algún momento carezcan de tareas tomarán hilos de la cola global o ejecutaran hilos de clase IDLE en caso de inactividad del sistema.

ENCONTRAR UNA FORMA DE MOSTRAR COMO LOS HILOS SE PLANIFICAN SIN TENER EN CUENTA LA MONOPOLIZACION, CAMBIANDO SU EJECUCION ENTRE LOS DISTINTOS NUCLEOS

### 4.2.2. Comportamiento con el Módulo Habilitado

Habiendo expuesto el comportamiento del sistema en su condición normal, se sienta una base para la posterior comparación con el desempeño del módulo.

Siguiendo un procedimiento similar al empleado con el módulo de encendido y apagado, procedimos a ejecutar el programa de estrés, orientado al cálculo de números primos. Durante su ejecución, la herramienta de monitoreo (htop) permite la observación de los distintos subprocesos vinculados a este programa y la información específica de cada uno, incluyendo el identificador de cada hilo, el cual adquiere relevancia antes de activar el módulo.

En este momento, nos enfocamos en los procesadores que ejecutan cada subproceso y cómo pueden variar según las decisiones del planificador.

2 imágenes con los subprocesos, que se pueda ver una comparación de cómo cambia el CPU para alguno de los subprocesos, es decir, subproceso 1 primer imagen corre en el cpu 1, segunda imagen ese mismo subproceso corre en cpu 3

Con el programa en ejecución y el monitoreo activado, el siguiente paso implica la configuración del módulo; en otras palabras, es el momento de actualizar las variables dentro del código del módulo. La primera variable a considerar es el identificador del subproceso que deseamos anclar a un núcleo específico, obtenido previamente del programa de monitoreo. Una vez que tenemos el identificador, procedemos a seleccionar en qué núcleo asociaremos este hilo, compilando y cargando el módulo de kernel adecuado.

Los resultados obtenidos concuerdan con la información proporcionada por la herramienta de monitoreo. En este caso, hemos seleccionado el hilo con el identificador 878172 y el CPU 2. Como era de esperar, tras la ejecución del módulo, este subproceso estuvo vinculado al CPU durante toda su ejecución. Además, ningún otro hilo o proceso se ejecutó en ningún momento en este núcleo, evidenciando así su exclusividad y el correcto funcionamiento del desarrollo.

Imagen con el hilo asociado al núcleo, quizás también podemos tener un link a video para mostrar hasta que termina de ejecutarse, y con la desactivación del módulo para ver como el cpu vuelve a estar disponible para los otros procesos

Un caso relevante es el estado del sistema una vez que el programa de estrés ha concluido pero el módulo permanece activado. En estas condiciones, se puede observar cómo el CPU al que se había anclado el hilo permanece "reservado", sin ejecutar ningún hilo que no posea el ID especificado, mientras que los procesos restantes solo se ejecutan en los núcleos liberados. Con esto en mente, se puede identificar una similitud con el funcionamiento del módulo de encendido/apagado al desactivar un núcleo.

Al desactivar el módulo, el sistema continúa operando según lo esperado, con todos sus núcleos disponibles, tal como lo haría en su estado normal.

branch con el codigo

## 4.3. Resultados de las actualizaciones en la versión del S.O.

En esta sección, se expondrán los resultados derivados de las actualizaciones implementadas en el marco del proyecto integrador previo. Es relevante subrayar que estas actualizaciones se llevaron a cabo con el propósito de sincronizar el proyecto con la versión más reciente y estable del sistema operativo (versión 13.1). Dado que la implementación original demostró su eficacia en la versión 11, el énfasis

principal en esta sección se dirigirá a garantizar la continuidad de dicho rendimiento.

Esta fue la primera tarea abordada en este proyecto, y la elección de priorizarla desde el inicio se fundamentó en la necesidad de mantenerse alineado con la comunidad y acceder a posibles recursos de apoyo cuando fuese necesario. Asimismo, esta estrategia facilitó el desarrollo sobre una base de código más actualizada, evitando conflictos significativos que pudieran surgir en fases finales del desarrollo.

A pesar de la necesidad de revisar numerosos conflictos generados entre el código del planificador en la versión 11 y la versión 13 de FreeBSD, los resultados finales obtenidos de esta etapa confirman que la actualización se realizó con éxito. El sistema operativo, con el planificador de redes de Petri, mantuvo la estabilidad y el funcionamiento adecuado en las pruebas realizadas.

El código correspondiente a esta etapa se encuentra en una rama específica del repositorio del proyecto. Debido a la estructura de ramas y modalidad de trabajo que se planteó para la realización de este trabajo integrador, también podemos encontrar una rama por cada actualización que se fue realizando progresivamente desde la 11 hasta la 13.1.

MOSTRAR ALGUNAS IMAGENES DEL PROYECTO CORRIENDO CON LA VERSION 13.1 Y UTILIZANDO LA RED

## 4.4. Resultados del tareas extra OPCIONAL - VER SI LO AGREGAMOS

# 5. Conclusión y trabajos futuros

En este capítulo, se presentarán las conclusiones derivadas del desarrollo de las actualizaciones implementadas en el proyecto, el Módulo de encendido/apagado de procesadores, y el Módulo de monopolización de núcleos en el sistema operativo FreeBSD.

Además, se discutirán las implicancias y aplicaciones prácticas de los resultados obtenidos, destacando los logros alcanzados y las perspectivas para futuras investigaciones y mejoras en la gestión de recursos.

## 5.1. Conclusiónes

La actualización exitosa del código del planificador a la última versión del sistema operativo FreeBSD 13.1 fue un paso crucial en la evolución de nuestro trabajo. Este proceso no solo demostró la continuidad del funcionamiento del planificador mediante redes de Petri, sino que también validó la vision del proyecto en curso y de futuros proyectos que sigan esta línea de desarrollo.

La modularidad se reveló como una ventaja esencial en nuestra implementación. La facilidad con la que pudimos agregar módulos independientes que interactúan con la red general del sistema abrió nuevas posibilidades para trabajar con procesos e hilos. Esta flexibilidad no solo facilita la comprensión de los puntos críticos del sistema operativo, como los tiempos de respuesta y el manejo eficiente de procesadores, sino que también abre puertas a futuras mejoras. La implementación exitosa del Módulo de encendido/apagado de procesadores, así como del Módulo de monopolización de núcleos, han demostrado la viabilidad del enfoque.

Por otro lado, la colaboración con la comunidad de desarrolladores de FreeBSD fue esencial. En momentos críticos, la ayuda de personas experimentadas que comprenden a un nivel profundo un código complejo resultó fundamental. Esto resalta la importancia de mantenerse actualizado con la comunidad para superar desafíos y mejorar el código de manera efectiva.

## 5.2. Trabajos Futuros

Mirando hacia el futuro, nuestra implementación proporciona una base sólida para investigaciones adicionales y mejoras continuas. Identificamos áreas potenciales que podrían beneficiarse de un estudio más profundo.

El éxito de los módulos desarrollados abre la puerta a diversas oportunidades para futuras investigaciones y mejoras en la gestión de recursos en sistemas operativos. A continuación, intentaremos desarrollar algunas de las áreas en donde detectamos potenciales mejoras a lo implementado hasta el momento.

### 5.2.1. Optimización del timeslice para procesadores apagados

Una mejora considerable para tener en cuenta en futuras iteraciones consiste en la implementación de una gestión dinámica de las ventanas de tiempo asignadas a los hilos.

En el caso del módulo de encendido/apagado, esta mejora evitaría la necesidad de realizar operaciones repetidas de encolado y desencolado del idlethread. Este enfoque contribuiría significativamente a la eficiencia del sistema, reduciendo la carga asociada con estas operaciones y dirigiendo el módulo hacia una mejora sustancial en la gestión de recursos y la eficacia en el manejo de la energía del sistema operativo.

En cuanto al módulo de monopolización de núcleos, la aplicación de esta estrategia permitiría una asignación de tiempo prolongada. Esto posibilitaría que los procesos anclados se ejecuten de manera continua en los procesadores, acelerando así la finalización de los mismos, contribuyendo a un rendimiento general mejorado del sistema operativo.

### 5.2.2. Implementación de las políticas de afinidad mediante la Red

En la actualidad, FreeBSD dispone de métodos nativos para gestionar la afinidad entre hilos y procesadores. La migración de esta implementación hacia el enfoque basado en Redes de Petri fue postergada para priorizar la introducción de los primeros módulos en la red.

Sin embargo, reconocemos la relevancia de realizar los ajustes correspondientes con el fin de incorporar estas políticas en la red de recursos. De esta manera, el sistema operativo estaría capacitado para tomar todas las decisiones a través de la red, evitando que algunas de ellas queden vinculadas al código existente que utiliza flags de afinidad para la toma de decisiones; contribuyendo a una gestión más cohesiva y centralizada de las políticas de afinidad en el sistema operativo.

### 5.2.3. Solución definitiva al fallo de página con la placa de red encendida

Como se comentó en la sección 3.3.4.2. del presente informe, actualmente tenemos problemas en el uso regular del sistema operativo cuando la placa de red se encuentra encendida. Si bien es una falla que se puede detectar también en la versión del proyecto integrador previo, consideramos que es un punto importante a tener en cuenta para mejorar la experiencia de trabajo sobre este proyecto.

### 5.2.4. Integración de los módulos a triggers automáticos del Sistema Operativo

Este punto de mejora futura, y probablemente la razón del presente desarrollo, se relaciona con la integración de los módulos en el sistema operativo en sí. Hasta el momento, la activación o desactivación de estos módulos se realiza de forma manual. Esta decisión fue tomada desde el inicio del proyecto, convencidos de que la importancia de este desarrollo recaía en comenzar el camino a la mejora de gestión y energía del sistema operativo mediante la implementación de los respectivos módulos.

No obstante, la visión es que la toma de decisiones sobre la activación o desactivación de estos módulos recaiga en el propio sistema operativo mediante la implementación de señales o mensajes que permitan disparar las transiciones correspondientes en la red, habilitando así los módulos. La intención es delegar al sistema operativo la toma de estas decisiones de manera automatizada, aprovechando momentos de inactividad (para el módulo de encendido/apagado de procesadores) y durante la ejecución de tareas críticas (para el módulo de monopolización de núcleos).

# Índice de figuras



# Índice de cuadros

# 6. Bibliografía

## Referencias

[1] Tomás Turina, Nicolás Papp. (2019). *Modelado del planificador a corto plazo con redes de Petri.*

[2] Marshall Kirk McKusick, George V. Neville-Neil, Robert N.M. Watson. (2014). *The Design and Implementation of the FreeBSD Operating System (2nd Edition).*