



Universidad
Nacional
de Córdoba



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

Índice	2
1. Introducción	5
1.1. Oportunidad	5
1.2. Motivación	6
1.3. Objetivo	6
1.4. Alcance	7
1.5. Modelo de Desarrollo	8
1.6. Requerimientos Generales	8
1.6.1. Requerimientos funcionales	8
1.6.2. Requerimientos no funcionales	8
2. Base Teórica	10
2.1. Procesos e hilos	10
2.1.1. Estructura de los procesos	10
2.1.2. Estructura de los hilos	12
2.1.3. Estados de los procesos e hilos	13
2.1.4. Prioridad de los hilos	14
2.2. Planificación	14
2.2.1. <i>Multilevel Feedback Queue Scheduling</i>	15
2.2.2. Colas de ejecución de hilos y Cambios de Contexto	16
2.2.3. Planificador 4BSD para sistemas multiprocesador	17
2.3. Conceptos del proyecto integrador previo	18
2.3.1. Elección del planificador	18
2.3.2. Análisis de modificaciones realizadas	19
2.3.3. Modelado del hilo	19
2.3.4. Modelado del planificador	20
2.3.5. Jerarquía de transiciones	22
2.3.6. Marcado inicial	23
2.3.7. Métodos de interes para nuestra tesis	23

3. Desarrollo	24
3.1. Introducción	24
3.2. Metodologías de trabajo	24
3.2.1. Estrategia de ramificación	25
3.2.2. Conventional Commits	25
3.2.3. Repositorio de Documentación del Proyecto	26
3.3. Módulo de actualizaciones	27
3.3.1. Actualización al último release de la <i>v11</i>	27
3.3.2. Actualización a la versión 12	28
3.3.3. Actualización a versión 13	28
3.3.4. Resultados	28
3.3.5. Próximos pasos	29
3.4. Módulo de encendido/apagado de los procesadores	29
3.4.1. Objetivos	29
3.4.2. Primera Iteración: Implementación del módulo de encendido/apagado mediante cambios en la Red	30
3.4.3. Segunda Iteración: Soporte para el idlethread en la Red	32
3.4.4. Resultados	34
3.4.5. Próximos pasos	34
3.5. Módulo de monopolización de hilos por parte de los procesadores	35
3.5.1. Objetivos	35
3.5.2. Diseño de la implementación	35
3.5.3. Implementación	36
3.5.4. Resultados	38
3.5.5. Próximos pasos	38
3.6. Tareas Extra	38
3.6.1. Solución del problema de afinidad (procesadores sobrecargados)	39
3.6.2. Problema del uso de la placa de red en el sistema operativo	39
4. Análisis de resultados integrales	41
4.1. Resultados de la Implementación del Módulo de Encendido/Apagado de Procesadores	41
4.1.1. Estado Inactivo del Sistema	41

4.1.2. Comportamiento con el Módulo Inhabilitado	42
4.1.3. Comportamiento con el Módulo Habilitado	43
4.2. Resultados del módulo de Monopolización de Núcleos	44
4.2.1. Estado Normal del Sistema	44
4.2.2. Comportamiento con el Módulo Habilitado	44
4.3. Resultados de las actualizaciones en la versión del S.O.	45
4.4. Resultados del tareas extra	45
5. Conclusión y trabajos futuros	46
5.1. Conclusiones	46
5.2. Trabajos Futuros	46
5.2.1. Optimización del timeslice para procesadores apagados	46
5.2.2. Implementación de las políticas de afinidad mediante la Red	47
5.2.3. Solución definitiva al fallo de página con la placa de red encendida	47
5.2.4. Integración de los módulos a triggers automáticos del Sistema Operativo	47
Índice de figuras	48
Índice de cuadros	48
Referencias	49
A. Apéndice 1: Archivos de diferencias	50
A.1. <i>sched_switch</i> (v12 a v13): Firma de la función y variables	50
A.2. <i>sched_switch</i> (v12 a v13): Cambio de posición del bloque que bloquea el hilo previo al <i>resource_expulse_thread</i>	50
A.3. <i>sched_switch</i> (v12 a v13): Selección y ejecución de un nuevo thread	51

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 únicamente 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 última 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[2], 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.

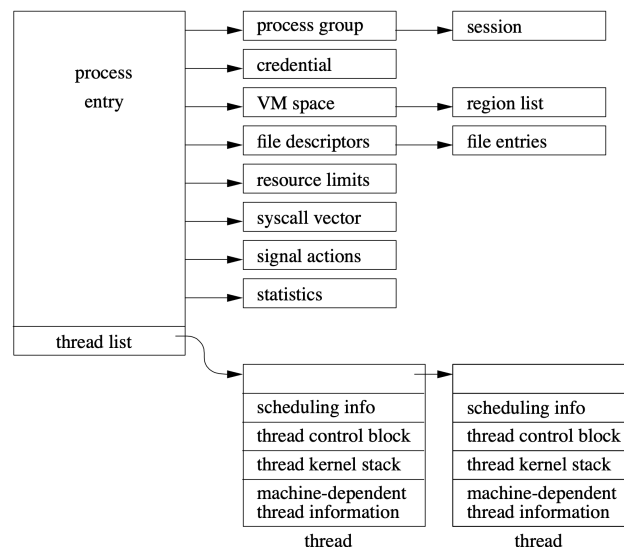


Figura 1: Estructura simplificada de un proceso.

Cada proceso cuenta con los punteros *p_pptr*, *p_children* y *p_sibling*, utilizados para establecer la relación 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 jerarquía de procesos.

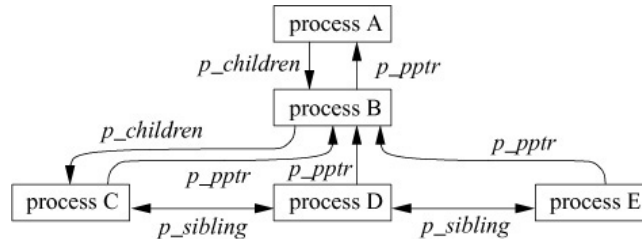


Figura 2: Jerarquía de grupo de procesos.

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 <i>RUNNABLE</i> , <i>SLEEPING</i> o <i>STOPPED</i>
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 *IDLE* 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

La planificación en sistemas operativos, implica decidir el orden y la duración de la ejecución de procesos e hilos del sistema. En este sentido, es el componente clave para optimizar el uso de recursos y el tiempo de ejecución de los programas.

FreeBSD cuenta con el planificador 4BSD, un planificador de tiempo compartido (*timeshare scheduler*) que asigna dinámicamente la prioridad de los procesos. El cálculo se realiza en función de varios parámetros de comportamiento previo de dicho proceso, como la cantidad de tiempo de CPU utilizado, la cantidad de recursos de memoria que posee o requiere para su ejecución, entre otros.

La política de planificación asigna inicialmente una prioridad de ejecución alta a cada hilo y permite

que ese hilo se ejecute durante un intervalo de tiempo fijo, o *time slice*. A los hilos que se ejecutan durante la duración de su *time slice* se les decrementa la prioridad; mientras que los hilos que ceden la CPU (generalmente debido a operaciones de entrada/salida) pueden mantener su prioridad. Por otro lado, los hilos inactivos incrementan su prioridad.

Esta forma de planificación beneficia especialmente a los programas interactivos. Los procesos que demandan una gran cantidad de tiempo de CPU ven reducida rápidamente su prioridad, mientras que los procesos interactivos que permanecen mayormente inactivos conservan una prioridad elevada. Esto permite que, cuando estén listos para ejecutarse, puedan desplazar a los procesos de baja prioridad que llevan ejecutándose durante un largo período de tiempo. Por ejemplo, supongamos que un editor de texto realiza una búsqueda dentro de un documento, lo que conlleva un aumento temporal en el uso de recursos computacionales. Durante este período, la prioridad del editor puede verse momentáneamente reducida para dar prioridad a otros procesos más urgentes. Sin embargo, una vez completada la búsqueda y sin actividad adicional del usuario, el editor recuperará su prioridad en la cola de planificación del sistema operativo.

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 otra cola de hilos de mínima prioridad que se ejecutan únicamente cuando ningún hilo en las colas de mayor prioridad está en un estado de posible ejecución.

2.2.1. *Multilevel Feedback Queue Scheduling*

El planificador 4BSD, utiliza un algoritmo de planificación basado en colas de retroalimentación multinivel *Multilevel Feedback Queue*. Todos los hilos en estado *RUNNABLE* reciben una prioridad que determina en qué cola de ejecución se colocan. Al seleccionar un nuevo hilo para ejecutar, el sistema escanea las colas de ejecución de mayor a menor prioridad y elige aquel que se encuentra en la primera cola no vacía. Si varios hilos residen en una cola, el sistema los ejecuta en orden circular (*Round Robin*); es decir, en el orden en que se encuentran en la cola, con igual cantidad de tiempo permitido.

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*. Cuando el hilo agota el *time slice*, se coloca al final de la cola de la que provino y se selecciona el hilo que se encuentra al comienzo de la misma para ejecutarse.

Cuanto más corto sea el *time slice*, mejor será la respuesta interactiva. Sin embargo, cuanto más largos sean los cuántums de tiempo, mayor será el rendimiento del sistema, ya que habrá menos sobrecarga debida a los cambios de contexto y se limpiarán menos a menudo las cachés del procesador. El *time slice* utilizado por 4BSD es de 0.1 segundo. Este valor se encontró empíricamente como el cuántum más largo que podía usarse sin pérdida de la respuesta deseada para trabajos interactivos como los editores.

2.2.2. Colas de ejecución de hilos y Cambios de Contexto

El *kernel* cuenta con un conjunto único de colas de ejecución para manejar todas las clases de hilos mencionadas anteriormente en la sección "Prioridad de los hilos". La cantidad de colas utilizadas para mantener la colección de hilos *RUNNABLE* impacta en el costo de su gestión. Mantener una sola cola ordenada simplifica la operación de búsqueda y selección del próximo hilo listo, pero otras operaciones se vuelven más costosas. Por otro lado, si se utilizara una cola por cada nivel de prioridad, la selección del próximo hilo listo sería demasiado costosa, dado que habría que buscar entre 256 de ellas. FreeBSD emplea 64 colas de ejecución para los hilos listos, de modo que para ubicar un hilo en una cola, se divide la prioridad del mismo por 4, y el resultado indica la cola donde debería ser ubicado. Con el fin de optimizar el tiempo, los hilos en cada cola no son reordenados según sus prioridades.

Las colas de ejecución contienen todos los hilos listos en la memoria principal, excepto el hilo que se está ejecutando actualmente. La Figura 3 muestra cómo se organiza cada cola como una lista doblemente enlazada de estructuras de hilo. La cabecera de cada cola de ejecución se mantiene en un arreglo. Asociado con este arreglo existe un vector de bits que se utiliza para identificar las colas de ejecución no vacías.

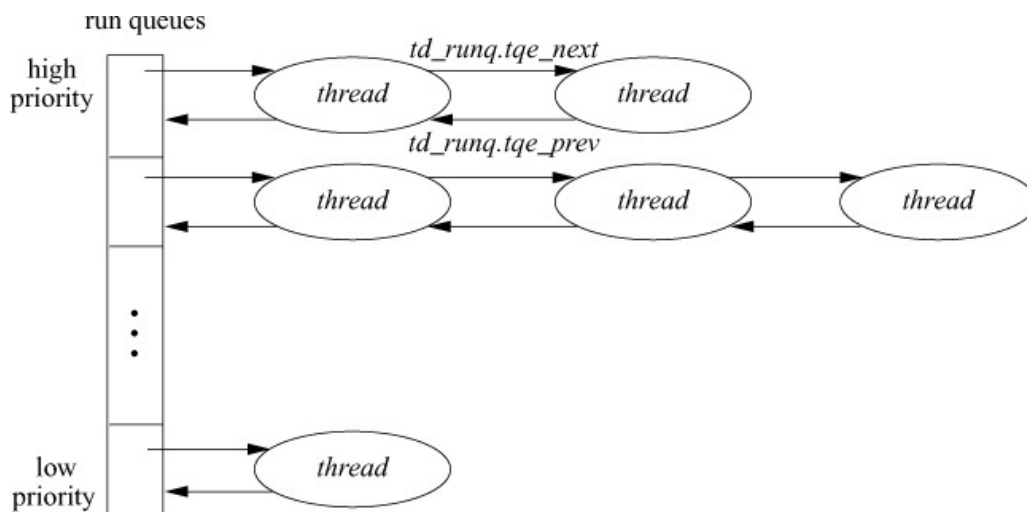


Figura 3: Estructura de colas para hilos *RUNNABLE*

El código de cambio de contexto se divide en dos partes: una parte independiente de la máquina, que reside en *mi_switch()*, y otra parte dependiente de la máquina, que reside en *cpu_switch()*.

Los **cambios de contexto voluntarios**, es decir, cuando un hilo decide ceder la CPU de forma deliberada, ocurren cuando dicho hilo llama a la rutina *sleep()*, colocándolo en una cola de espera e invocando *mi_switch()* para programar el siguiente hilo a ejecutar.

La rutina *mi_switch()* no puede ser llamada desde el código que se ejecuta a nivel de interrupción, ya que debe ser invocada dentro del contexto del hilo en ejecución. Por ello, para **cambios de contexto involuntarios**, se activa la bandera TDF_NEEDRESCHED del hilo en ejecución y luego se envía

una trampa de sistema asíncrona (AST). Una AST es una trampa que se le entrega al hilo que se estaba ejecutando previo a una interrupción, para que en lugar de reanudar su ejecución (al volver de la interrupción), se llame a la rutina *mi_switch()*.

2.2.3. Planificador 4BSD para sistemas multiprocesador

El planificador 4BSD en FreeBSD está diseñado para gestionar la carga de trabajo en sistemas multiprocesador mediante un enfoque que equilibra la carga de los procesadores. Para lograr esto, el planificador emplea tanto las colas de ejecución específicas de cada CPU como una cola de ejecución global. Tanto la cola global como las específicas de cada CPU están estructuradas como colas de múltiples niveles con realimentación (Multilevel Feedback Queue), lo que permite priorizar y gestionar los hilos de manera eficiente según su comportamiento y necesidades de tiempo de CPU.

La cola de ejecución global contiene hilos que pueden ser ejecutados por cualquier CPU del sistema. Sirve como un *pool* central de hilos listos para ejecutarse, permitiendo una distribución flexible de la carga de trabajo. Cuando una CPU queda inactiva, revisa tanto su propia cola de ejecución como la cola de ejecución global, seleccionando el hilo con mayor prioridad para ejecutarlo. En los casos en donde la cola de ejecución propia está vacía, la CPU tomará un hilo de la cola global, garantizando que todas las CPUs tengan trabajo disponible si hay suficientes hilos listos para ejecutarse.

Por otro lado, cada CPU en el sistema tiene su propia cola de ejecución. Estas colas contienen hilos que están designados para ejecutarse en CPUs específicas. Para optimizar el rendimiento, el planificador 4BSD considera varias restricciones y preferencias de afinidad cuando decide dónde colocar un hilo cuando termina su *time slice* o debe abandonar por otras razones.

Algunos hilos pueden estar vinculados a un conjunto específico de CPUs debido a consideraciones de rendimiento. El planificador 4BSD respeta esta afinidad al determinar en qué CPU encolar el hilo mediante un mecanismo de iteración sobre las CPUs asignadas al hilo, excluyendo aquellas para las que no existe afinidad. Luego, selecciona la CPU cuya cola tenga la menor cantidad de hilos listos para ejecutarse.

La función de *bounding*, por otro lado, implica restringir un hilo a un procesador específico. Por ejemplo, al ejecutar el método *kern_reboot* en FreeBSD, encargado de preparar el sistema para un reinicio, apagado o parada, es necesario vincular primero dicho hilo al procesador 0. Esto garantiza que el código se ejecute en la CPU 0, una práctica crucial en sistemas que solo permiten la ejecución de instrucciones en dicha CPU.

Por último, los hilos pueden ser *pinned* o anclados a una CPU particular. Cuando un hilo está anclado, el planificador se asegura de encolarlo siempre en el último procesador en el que se ejecutó. En el contexto del kernel de FreeBSD, esto se realiza para mantener la coherencia de caché o para garantizar la eficiencia de la ejecución de tareas críticas evitando la migración del hilo entre CPUs durante su ejecución. El método que ancla un hilo, es el que finalmente libera al hilo de esta restricción.

2.3. Conceptos del proyecto integrador previo

En el contexto del proyecto integrador previo, se llevó a cabo un análisis detallado del comportamiento del *scheduler* de un sistema operativo, con un enfoque específico en el proceso de asignación de tiempo de procesador. El objetivo fundamental fue identificar oportunidades de mejora en su funcionamiento, migrando de una visión estadística a un sistema más complejo implementado mediante Redes de Petri, en donde las decisiones de encolado de los hilos en una CPU se derivan de la información representada en el modelo, incluidos los estados globales y los de cada hilo.

En las secciones siguientes, se presentan los puntos clave abordados en este estudio, incluyendo la naturaleza de las modificaciones realizadas. Además, se resaltan los puntos de interés surgidos durante este análisis, los cuales orientarán el desarrollo de los distintos módulos destinados a ampliar las funcionalidades de planificación como la gestión del encendido y apagado de procesadores, así como la monopolización de los mismos por parte de los hilos.

2.3.1. Elección del planificador

La planificación a corto plazo en FreeBSD ha experimentado una evolución constante desde sus primeras versiones basadas en el sistema operativo 4BSD[1]. A lo largo de su historia, FreeBSD ha confiado en dos planificadores para gestionar la asignación de recursos del sistema, siendo el planificador 4BSD uno de los más notables. Este planificador, inicialmente diseñado para sistemas monoprocesador, ha sido adaptado y optimizado para entornos multiprocesador, convirtiéndose en un componente fundamental de la arquitectura del sistema operativo.

Aunque el planificador 4BSD es más simple en comparación con los más modernos, ha demostrado ser rápido y eficiente para satisfacer las necesidades de muchos usuarios. Sin embargo, con el objetivo de mejorar aún más el rendimiento y la capacidad de respuesta del sistema, se introdujo el planificador ULE en la versión 5.0 de FreeBSD. El ULE ofrece características avanzadas que lo hacen más adecuado para entornos con cargas de trabajo más complejas y exigentes.

A pesar de que el planificador ULE se ha convertido en el predeterminado en las versiones más recientes de FreeBSD (a partir de la versión 8.0), el planificador 4BSD sigue siendo una opción estable y confiable. La comunidad de FreeBSD continúa manteniendo y brindando soporte para el planificador 4BSD, asegurando su disponibilidad para aquellos que prefieren su simplicidad y eficiencia. Además, no hay planes inmediatos por parte del equipo de desarrollo de FreeBSD para dejar de darle soporte.

Nuestro proyecto se basa en el trabajo previo que se centró en el planificador 4BSD. Para obtener una comprensión más detallada y contextualizada de esta elección, se puede consultar las secciones de *Alcance* (1.4) y *Diferencias entre el planificador ULE y el 4BSD* (2.3.3) del proyecto integrador previo[2]. Aunque nuestro enfoque principal no radica en realizar cambios inmediatos en el sistema operativo ni en contribuir directamente a la comunidad de FreeBSD, continuamos con la fase de investigación e implementación centrada en este tipo de planificadores.

2.3.2. Análisis de modificaciones realizadas

Como se explicó previamente, la planificación en sistemas operativos consiste en decidir el orden y la duración de la ejecución de los distintos hilos del sistema. Tradicionalmente, los sistemas operativos han basado estas decisiones en estadísticas del comportamiento previo, utilizando datos históricos para optimizar el rendimiento.

Un aspecto importante de esta planificación ocurre cuando un hilo debe ceder la CPU a otro. En el contexto del planificador 4BSD, esto implica retirar un hilo de la cola de ejecución (RUNNABLE) y ponerlo en ejecución, mientras que el hilo que estaba en ejecución vuelve a encolarse. Aquí es donde se enfocó el trabajo anterior: en modelar el sistema de encolado y desencolado de hilos utilizando redes de Petri. Esta técnica permite representar de manera precisa y formal los estados y transiciones de los hilos, mejorando así la eficiencia y previsibilidad del planificador.

Las modificaciones realizadas en el código del planificador 4BSD se centraron en adaptar las funciones principales para que se ajusten al uso de redes de Petri. Estos cambios se llevaron a cabo en el archivo *sched_4bsd.c*, que contiene las funciones principales del planificador 4BSD.

Para comprender la naturaleza de los cambios realizados en el trabajo previo, detallaremos los resultados finales en términos de modelado de sistemas mediante redes de Petri.

2.3.3. Modelado del hilo

Uno de los modelos representados mediante redes de Petri es el de los estados de un hilo, como se muestra en la Figura 4.

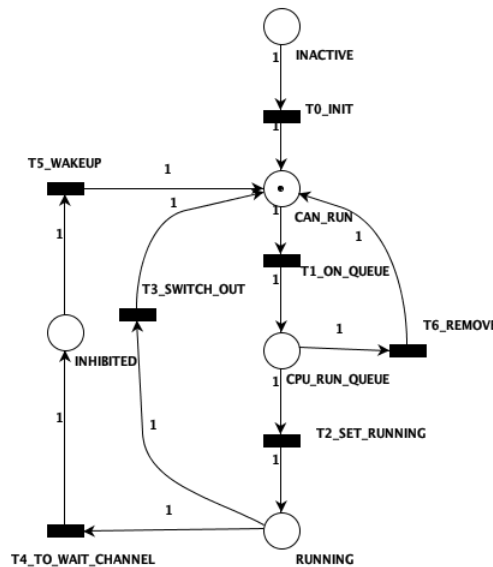


Figura 4: Modelo del hilo.

Las plazas de la Red de Petri la Figura 4 se corresponden a los estados posibles del hilo, siendo posible encontrarse en un solo estado a la vez. Las transiciones, por otro lado, representan los eventos que pueden llevar al hilo de un estado a otro. A continuación, se detallan las transiciones que se modelaron en la Red de Petri para el hilo:

- **T0_INIT:** Transición del estado INACTIVE a CAN_RUN, que ocurre cuando el hilo se agrega al planificador. Esta tarea no corresponde al scheduler, por lo que inicialmente un hilo en el planificador se inicializa en el estado CAN_RUN. Aunque esta transición no se dispara en la práctica, se incluyó en el modelo por razones representativas.
- **T1_ON_QUEUE:** El hilo se coloca en una cola local de una CPU específica o en la cola global, según la disponibilidad.
- **T2_SET_RUNNING:** El hilo se retira de la cola y comienza a ejecutar las instrucciones del programa asignado, ocupando así el procesador.
- **T3_SWITCH_OUT:** El planificador interrumpe el hilo y lo devuelve a su estado de CAN_RUN, seleccionando otro hilo de mayor prioridad para ejecutar y realizando un cambio de contexto.
- **T4_TO_WAIT_CHANNEL:** El hilo se bloquea debido a algún evento, semáforo o espera, y se coloca en una cola de espera hasta que se desbloquee.
- **T5_WAKEUP:** El hilo se desbloquea y puede volver a encolarse. Este evento ocurre fuera del planificador, y el hilo espera cambiar de estado según corresponda.
- **T6_REMOVE:** Se ejecuta cuando un hilo debe eliminarse de la cola en la que se encuentra.

2.3.4. Modelado del planificador

Por otro lado, se realizó el modelado de la red de Petri del planificador, o red de recursos, que se muestra en la Figura 5. En esta figura se detallan las plaza y transiciones de una sola CPU, siendo extensible a tantas CPUs como el sistema multiprocesador posea.

Las plazas del planificador, podemos separarlas en las siguientes categorías:

- **Plazas de la actividad del procesador:** Entre ellas encontramos las plazas PLACE_CPU, PLACE_TOEXEC y PLACE_EXECUTING que representan el estado de ejecución de un hilo sobre el procesador en cuestión.
- **Plazas de colas del procesador:** PLACE_QUEUE y PLACE_CANTQ, que representan las colas del procesador en cuestión, con un sistema diseñado para soportar la penalización de procesadores.
- **Plazas globales o de sistema:** PLACE_GLOBAL_QUEUE, PLACE_SMP_READY y PLACE_SMP_NOT_READY que representan las colas globales y el estado del sistema SMP en un determinado momento.

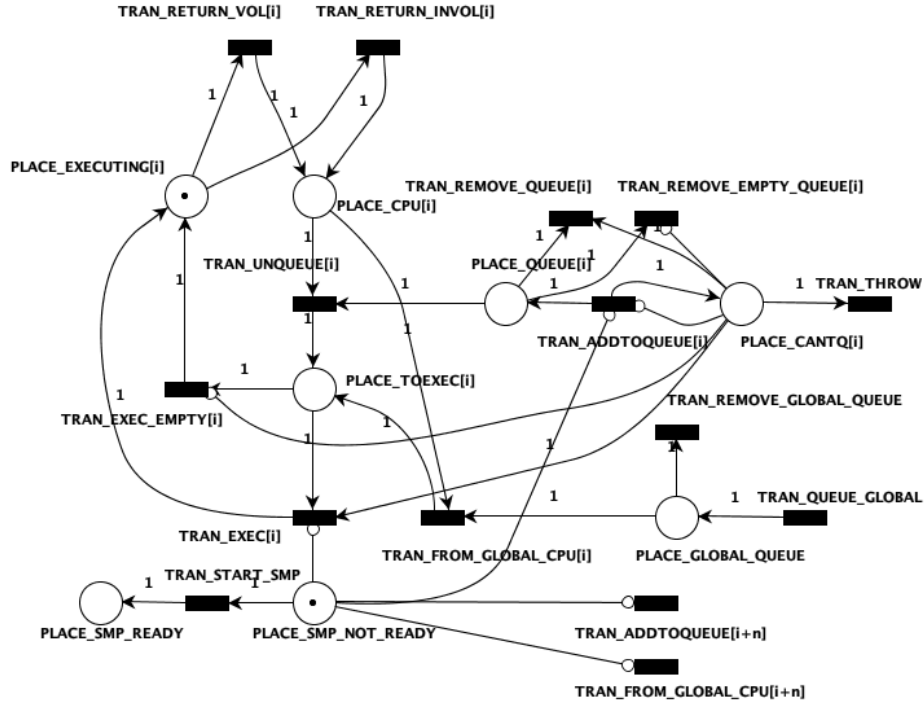


Figura 5: Modelo del planificador.

En cuanto a las transiciones de la red, podemos hacer la misma categorización:

■ **Transiciones de la actividad del procesador:**

- **TRAN_UNQUEUE:** Esta transición elimina al hilo próximo a ejecutarse de la cola del CPU, dejándolo listo para su ejecución.
- **TRAN_EXEC:** El hilo pasa a ejecutarse, ocupando el recurso del procesador. Esta transición elimina el token de la plaza de habilitación, permitiendo que un nuevo hilo pueda ser encolado. Cabe destacar que esta transición está inhibida en modo monoprocesador ya que en este modo los hilos que se planifican, son buscados en la cola global.
- **TRAN_EXEC_EMPTY:** Similar a TRAN_EXEC, esta transición permite la ejecución de hilos provenientes de la cola global.
- **TRAN_RETURN_VOL:** Representa la liberación del recurso del procesador para permitir la ejecución de otro hilo de la cola. Se dispara cuando un hilo no puede continuar su ejecución debido a la espera de un evento o recurso.
- **TRAN_RETURN_INVOL:** Al igual que TRAN_RETURN_VOL, esta transición se dispara cuando un hilo libera el procesador, pero en este caso, la interrupción se produce porque el hilo ha consumido su tiempo asignado de CPU o ha finalizado su tarea.

■ **Transiciones de colas del procesador:**

- **TRAN_ADDTOQUEUE:** Un hilo se agrega a la cola del CPU correspondiente. Esta transición está inhibida cuando el sistema está en modo monoprocesador (al inicio del sistema operativo) y cuando ya existe un hilo en la cola. Esta última decisión recae en un sistema de colas y de penalización de procesadores implementada en el trabajo integrador previo.
- **TRAN_REMOVE_QUEUE:** Expulsa un hilo de la cola y resta un token de habilitación de la CPU, desinhibiendo el encolado de otro hilo del sistema de penalización.
- **TRAN_REMOVE_EMPTY_QUEUE:** Funciona igual que TRAN_REMOVE_QUEUE, pero se ejecuta cuando la plaza de habilitación no tiene ningún token.
- **TRAN_THROW:** Esta transición se dispara automáticamente cuando todas las plazas de habilitación de las CPU tienen al menos un token. Su objetivo es habilitar las colas con la menor cantidad de hilos que estaban inhibidas, una vez que todas las colas se han nivelado.

■ **Transiciones globales o de sistema:**

- **TRAN_FROM_GLOBAL_CPU:** Representa el desencolado de un hilo desde la cola global.
- **TRAN_REMOVE_GLOBAL_QUEUE:** Expulsa un hilo de la cola global sin afectar la habilitación de la CPU.
- **TRAN_START_SMP:** Se dispara cuando el sistema cambia de monoprocesador a multiprocesador.
- **TRAN_QUEUE_GLOBAL:** Agrega un hilo a la cola global.

2.3.5. 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.

2.3.6. 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`).

2.3.7. Métodos de interes para nuestra tesis

Las principales modificaciones realizadas en el planificador se encuentran en las siguientes funciones, donde se implementaron las transiciones correspondientes para asociar y ajustar el funcionamiento del planificador original, al modelo de la red de Petri:

- Funciones de inicialización del planificador: Se añadieron funciones para inicializar la red de Petri, incluyendo el marcado inicial y la configuración de las transiciones y plazas en el modelo, representadas mediante variables.
- Encolado de hilos: En la función que gestiona el encolado de hilos, se ajustó la lógica para alinearse con la red de Petri, permitiendo que un hilo sea encolado en la cola global o en la de un procesador específico.
- Desencolado de hilos: En la función encargada de desencolar hilos, se integró el disparo de la transición correspondiente, permitiendo desencolar un hilo tanto de la cola global como de la cola de un procesador.
- Selección del hilo a ejecutar: En la función que elige el próximo hilo a ejecutar, se gestionan los hilos de la cola global, las colas de los procesadores y el hilo `IDLETHREAD`, ejecutado en casos de ausencia de otros hilos listos.
- Selección del Procesador: La función que selecciona el procesador adecuado para ejecutar un hilo fue completamente rediseñada para basarse en el estado de la red de Petri.
- Expulsión de Hilos: Función donde se gestiona la expulsión de hilos del planificador.
- Cambio de Contexto: Se añadieron las transiciones necesarias para tener en cuenta el cambio de contexto entre hilos debido a la preempción.

3. Desarrollo

3.1. Introducción

En este capítulo, detallaremos el desarrollo de las mejoras implementadas en el planificador a corto plazo basado en Redes de Petri para el sistema operativo FreeBSD. El trabajo realizado se fundamenta en la colaboración y continuidad de proyectos previos desarrollados por estudiantes de nuestra institución.

La comprensión y depuración del código existente, así como la adaptación de los archivos del código fuente de FreeBSD relacionados con el planificador 4BSD, fueron tareas esenciales en nuestro proceso de desarrollo. Este esfuerzo intenta reflejar buenas prácticas en ingeniería de software, siendo ésta, una prioridad durante las primeras etapas de desarrollo. Nuestro enfoque integró principios de trabajo en equipo y metodologías ágiles, permitiendo una iteración continua y mejorada sobre el código. A lo largo del desarrollo, se implementaron prácticas de gestión de versiones y estrategias de ramificación para garantizar la calidad y estabilidad del código en cada una de sus etapas. En las siguientes secciones, se detallarán las metodologías utilizadas y los módulos específicos desarrollados, subrayando la importancia de cada componente en el contexto del planificador 4BSD.

3.2. Metodologías de trabajo

Para abordar el desarrollo de este proyecto de manera efectiva, establecimos una serie de metodologías de trabajo que nos permitieron mantener una organización rigurosa y una documentación detallada en cada etapa del proceso. Nuestro objetivo era no solo avanzar de manera sistemática, sino también crear un recurso valioso para futuros trabajos en este ámbito.

La primera etapa consistió en una profunda revisión del informe del trabajo integrador previo, así como del código base de los archivos de interés en el planificador 4BSD. Este proceso se realizó de forma iterativa, asegurándonos de comprender y extender la documentación de ambos componentes antes de proceder. Ampliar la documentación en esta etapa de análisis inicial, fue crucial para establecer una base sólida sobre la cual construir el resto del proyecto.

Una vez completada la fase de comprensión y documentación, planificamos nuestras metodologías de trabajo, enfocándonos en estrategias que facilitaran la colaboración y la gestión eficiente del proyecto. Implementamos una estrategia de ramas estandarizada en el repositorio fork de FreeBSD, permitiendo un control preciso de las versiones y facilitando la integración continua. Además, creamos un repositorio externo dedicado a almacenar toda la documentación y los recursos necesarios para el desarrollo, garantizando un acceso fácil y organizado a la información relevante.

Estas metodologías no solo nos permitieron trabajar de manera ordenada y eficiente, sino que también sentaron las bases para una colaboración efectiva y una fácil transición del conocimiento. En las siguientes secciones, se detallarán las estrategias específicas utilizadas.

3.2.1. Estrategia de ramificación

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

Además, esta estrategia facilita el mantenimiento de un historial completo y bien organizado de las modificaciones realizadas al código, lo cual es esencial para el seguimiento y la continuidad del proyecto. En nuestro caso, definimos el prefijo de branches utilizando los apellidos de los integrantes del grupo de trabajo, como por ejemplo, *DrudiGoldmanPI/*. Este enfoque nos ayuda a identificar y localizar fácilmente todos los cambios realizados en este proyecto integrador.

Para definir los nombres de las ramas, establecimos tres casos principales por los cuales se crearían nuevas ramas:

- **Testing:** Ramas creadas para realizar pruebas y verificar que ciertas partes del código fuente funcionen según lo esperado. Este tipo de ramas se utilizan principalmente para la depuración mediante el uso de logs.
- **Update:** Ramas dedicadas a la actualización del código base con las versiones más recientes de FreeBSD. Esto es especialmente relevante en la fase inicial del proyecto, cuando se deben integrar los cambios previos realizados en el proyecto integrador anterior.
- **Feature:** Ramas utilizadas para el desarrollo de nuevas funcionalidades o la adición de código no existente previamente. Estas ramas permiten la implementación y prueba de nuevas características sin afectar la estabilidad de la rama principal.

Con estos tres casos principales (o causas), se definió la estrategia de nombrado de ramas de la siguiente forma:

`DrudiGoldmanPI/<CAUSA>_<DESCRIPCIÓN>-<VERSION_FREEBSD>`

Donde `<DESCRIPCIÓN>` es una breve explicación del contenido de los cambios y `<VERSION_FREEBSD>`, es la versión de FreeBSD que se utilizó para realizar los cambios.

Por ejemplo, *DrudiGoldmanPI/feature_cpuMonopolized-13.1.0* es una rama para la implementación del módulo de monopolización que se detallará posteriormente en esta sección de desarrollo; y *DrudiGoldmanPI/update_petriNetScheduler-12.3.0* es una rama para la actualización del planificador 4BSD mediante Redes de Petri a la versión 12.3.0 del sistema operativo.

3.2.2. Conventional Commits

En nuestro proyecto, utilizamos la convención de *Conventional Commits* para mantener un historial de cambios claro y consistente. Esta práctica facilita la lectura y comprensión de los cambios realizados, tanto para el equipo de desarrollo como para futuros colaboradores.

Los mensajes de *commit* siguen un formato estandarizado que describe de manera concisa el tipo de cambio, la versión del sistema operativo afectada y una breve descripción de la modificación. A continuación, se presentan algunos ejemplos extraídos de nuestro proyecto:

- `feat(13.1.0): Move thread lock before resource_expulse_thread()`
- `fix(13.1.0): Fix sched_pickcpu for cases when resource_choose_cpu returns -1 (NOCPU)`
- `refactor(13.1.0): Change kernel config file identifier`
- `feat(12.3.0): Short term scheduler modeling with PN - Add PI code to release/12.3.0`

Los tipos de cambios más comunes que utilizamos incluyen:

- **feat:** Para la adición de nuevas funcionalidades.
- **fix:** Para la corrección de errores.
- **refactor:** Para modificaciones que mejoran el código sin alterar su funcionalidad.

Para la integración de cambios, hemos adoptado la estrategia de *squash merge*. Esta técnica permite consolidar todos los cambios de una rama en un único *commit* bien detallado, lo cual simplifica el historial del proyecto y facilita el seguimiento de avances importantes. Cada *squash merge* incluye un resumen comprensivo de todos los *sub-commits* que forman parte del cambio, garantizando así una documentación clara y precisa del progreso del proyecto.

3.2.3. Repositorio de Documentación del Proyecto

El repositorio de la documentación del proyecto está organizado de manera estructurada y detallada, facilitando el acceso y la gestión de la información relevante para el desarrollo del mismo. A continuación, se presenta una descripción breve de su estructura principal:

- **Archivos del Proyecto**
 - **Archivos de Migraciones:** Contiene subdirectorios correspondientes a diferentes versiones de migración del sistema, incluyendo archivos de configuración y código fuente relevante.
 - **Logs:** Registro de eventos y mensajes generados durante pruebas y actualizaciones del sistema.
 - **Módulos:** Módulos de kernel adicionales para poner en funcionamiento los nuevos desarrollos.
 - **Otros:** Programas auxiliares para seguir el comportamiento de las transiciones de la Red de Petri.
 - **Programas de Test:** Programas utilizados para pruebas de rendimiento y funcionalidad.
- **Documentación del Proyecto**

- **Uso del Repositorio:** Guías sobre cómo manejar las ramas y versiones del repositorio.
- **Iteraciones:** Detalles de las distintas iteraciones del proyecto.

Esta organización meticulosa asegura que todos los aspectos del proyecto, desde el código fuente hasta la documentación y los informes, estén bien gestionados y accesibles para los miembros del equipo.

Para más detalles, puede acceder al repositorio de documentación del proyecto en: <https://github.com/drudilea/free-bsd-scheduler>.

3.3. Módulo de actualizaciones

El desarrollo del proyecto se organizó en módulos, cada uno con objetivos específicos y alcanzables para asegurar avances concretos en cada etapa.

En la fase inicial, nos enfocamos en una tarea esencial en cualquier proyecto de software: mantener la compatibilidad con las diferentes versiones del sistema operativo. De esta forma, obtuvimos beneficios significativos en términos de estabilidad y compatibilidad, resultando en un sistema más confiable. Además, estando alineados con la última versión del mismo, nos acerca a la posibilidad de contribuir al proyecto de una manera directa.

El trabajo comenzó con la versión 11 de FreeBSD, basado en el trabajo integrador previo realizado por Nicolás Papp y Tomás Turina, y se extendió hasta la última versión estable, la 13.1.

Para gestionar las actualizaciones de manera ordenada y efectiva, así como también, comprender mejor los cambios entre versiones; el módulo se dividió en tres etapas progresivas: actualización a la última *release* de la versión 11, y actualizaciones a las versiones 12 y 13 de FreeBSD.

3.3.1. Actualización al último release de la *v11*

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 release de ésta versión, 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 el valor de la bandera *td_owepreempt* en 1, indicando la necesidad de desalojar al que se encuentra en ejecución, para así tomar el control.

Si bien el método de *preemption* no se encuentra modelado en nuestra Red de Petri de Recursos, este cambio impactó en el código del planificador de nuestro trabajo ya que implicó la remoción de algunos disparos de transiciones que se provocaban en el cambio de contexto inmediato. Con la nueva

estrategia, fue necesario ajustar el código de nuestra Red de Petri para reflejar el cambio en la política de planificación.

3.3.2. Actualización a la versión 12

Al comparar las diferencias entre las versiones 11.4.0 y 12.3.0 (último *release*) de FreeBSD, no se encontraron cambios significativos en los archivos relevantes para nuestro proyecto del planificador. Debido a esto, no fue necesario realizar ajustes específicos para esta versión.

3.3.3. Actualización a versión 13

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

El primero de ellos (ver apéndice A.1) ya se encuentra en la firma de la función y sus variables: previamente, *newtd* era un parámetro de *sched_switch()* que se utilizaba para indicar explícitamente el hilo que se iba a ejecutar a continuación. Sin embargo, en la versión 13, el parámetro *newtd* se eliminó y con ello se simplificó la lógica de selección del hilo (ver apéndice A.3), teniendo que ajustar algunos disparos de la Rdp que previamente eran necesarios para ajustar los recursos a estos desencholados en el planificador.

En la versión 13 de FreeBSD, se realizó una reorganización significativa de los locks en la función *sched_switch()* (ver apéndice A.2). Estos cambios introdujeron comportamientos inesperados en la red, como el disparo de transiciones no sensibilizadas que desajustaban el estado de la red y causaban fallos en el sistema. Después de depurar el código, identificamos que el problema principal era el manejo incorrecto de la transición RETURN_VOL. Específicamente, el método *resource_expulse_thread()* estaba disparando esta transición sin bloquear previamente el *td_lock*. Sin este bloqueo, no era posible cambiar el estado del hilo, lo que impedía que regresara al estado de *CAN_RUN* o listo para ejecutarse. Como resultado, ningún hilo podía ser ejecutado por los procesadores más de una vez, lo que bloqueaba el sistema. Tras realizar los ajustes necesarios, logramos resolver este problema y garantizar el correcto funcionamiento de la red.

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 esto 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).

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.

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 resumen, 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()`, removiéndolo 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 sobrescribe 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 módulo 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.

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.

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; éste 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.

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.

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.

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 6 proporciona una representación visual de este estado.

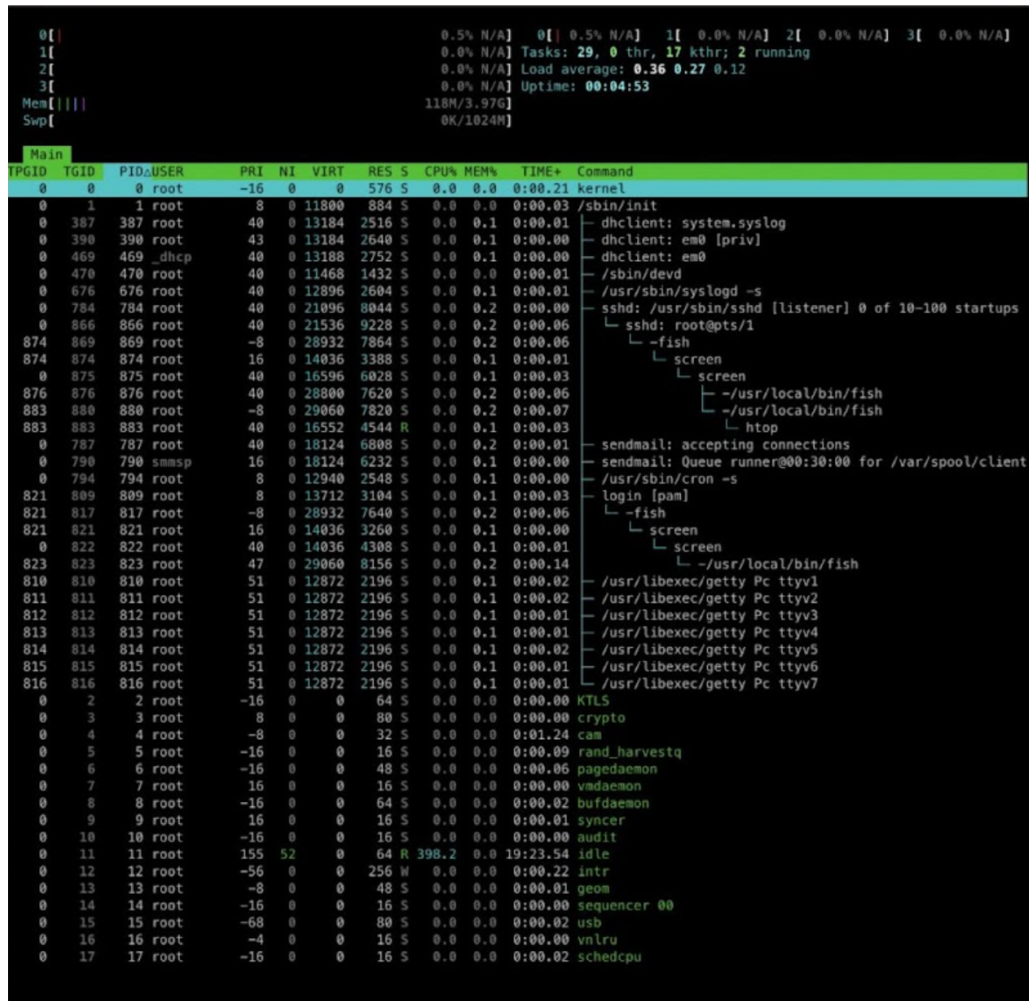


Figura 6: Estado inactivo del sistema.

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 7 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.

```

0[|||||] [100.0%N/A] 0[100.0%N/A] 1[100.0%N/A] 2[100.0%N/A] 3[100.0%N/A]
1[|||||] [100.0%N/A] Tasks: 30, 0 thr, 17 kthr; 4 running
2[|||||] [100.0%N/A] Load average: 2.98 0.97 0.36
3[|||||] [100.0%N/A] Uptime: 00:04:04
Mem[|||||] 114M/3.97G
Swp[|||||] 0K/1024M

Main
TPGID TID PWD USER PRI NI VIRT RES S CPU% MEM% TIME+ Command
0 0 0 root -16 0 0 576 S 0.0 0.0 0:00.15 kernel
0 1 1 root 8 0 11800 1108 S 0.0 0.0 0:00.01 /sbin/init
0 379 379 root 40 0 13184 2512 S 0.0 0.1 0:00.00 dhclient: system.syslog
0 382 382 root 4 0 13184 2640 S 0.0 0.1 0:00.00 dhclient: em0 [priv]
0 444 444 root 40 0 13188 2752 S 0.0 0.1 0:00.00 dhclient: em0
0 445 445 root 40 0 11468 1432 S 0.0 0.0 0:00.00 /sbin/devd
0 651 651 root 40 0 12896 2604 S 0.0 0.1 0:00.01 /usr/sbin/syslogd -s
0 759 759 root 40 0 21096 8044 S 0.0 0.2 0:00.00 sshd: /usr/sbin/sshd [listener] 0 of 10-100 startups
0 792 792 root 40 0 21536 9224 S 0.0 0.2 0:00.07 sshd: root@pts/0
800 795 795 root -8 0 29188 7740 S 0.0 0.2 0:00.10 -fish
800 800 800 root 16 0 14036 3260 S 0.0 0.1 0:00.01 screen
800 801 801 root 40 0 16596 6016 S 0.0 0.1 0:00.05 screen
802 802 802 root -8 0 33156 8208 S 0.0 0.2 0:00.15 -/usr/local/bin/fish
829 829 829 root 8 0 12728 2000 S 0.0 0.0 0:00.00 -/usr/local/bin/fish
829 830 830 root 65 0 12728 1996 R 97.7 0.0 1:21.38 ./parallel_calc_fork
829 831 831 root 65 0 12728 1996 R 97.7 0.0 1:21.38 ./parallel_calc_fork
829 832 832 root 65 0 12728 1996 R 97.7 0.0 1:21.39 ./parallel_calc_fork
829 833 833 root 65 0 12728 1996 R 97.7 0.0 1:21.39 ./parallel_calc_fork
809 806 806 root -8 0 29060 7836 S 0.0 0.2 0:00.06 -/usr/local/bin/fish
809 809 809 root 40 0 16552 4548 R 0.0 0.1 0:00.07 htop
0 762 762 root 40 0 18124 6808 S 0.0 0.2 0:00.01 sendmail: accepting connections
0 765 765 root 16 0 18124 6264 S 0.0 0.2 0:00.00 sendmail: Queue runner@00:30:00 for /var/spool/client
0 769 769 root 8 0 12940 2548 S 0.0 0.1 0:00.00 /usr/sbin/cron -s
784 784 784 root 40 0 12872 2196 S 0.0 0.1 0:00.01 /usr/libexec/getty Pc ttyv0
785 785 785 root 40 0 12872 2196 S 0.0 0.1 0:00.00 /usr/libexec/getty Pc ttyv1
786 786 786 root 40 0 12872 2196 S 0.0 0.1 0:00.01 /usr/libexec/getty Pc ttyv2
787 787 787 root 40 0 12872 2196 S 0.0 0.1 0:00.01 /usr/libexec/getty Pc ttyv3
788 788 788 root 40 0 12872 2196 S 0.0 0.1 0:00.01 /usr/libexec/getty Pc ttyv4
789 789 789 root 40 0 12872 2196 S 0.0 0.1 0:00.00 /usr/libexec/getty Pc ttyv5
790 790 790 root 40 0 12872 2196 S 0.0 0.1 0:00.00 /usr/libexec/getty Pc ttyv6
791 791 791 root 40 0 12872 2196 S 0.0 0.1 0:00.00 /usr/libexec/getty Pc ttyv7
0 2 2 root -16 0 0 64 S 0.0 0.0 0:00.00 KTLS
0 3 3 root 8 0 0 80 S 0.0 0.0 0:00.00 crypto
0 4 4 root -8 0 0 32 S 0.0 0.0 0:01.26 cam
0 5 5 root -16 0 0 16 S 0.0 0.0 0:00.07 rand_harvestq
0 6 6 root -16 0 0 48 S 0.0 0.0 0:00.04 pagedaemon
0 7 7 root 16 0 0 16 S 0.0 0.0 0:00.00 vmdaemon
0 8 8 root -16 0 0 64 S 0.0 0.0 0:00.01 bufdaemon
0 9 9 root 16 0 0 16 S 0.0 0.0 0:00.01 syncer
0 10 10 root -16 0 0 16 S 0.0 0.0 0:00.00 audit
0 11 11 root 155 52 0 64 R 5.5 0.0 10:41.36 idle
0 12 12 root -56 0 0 256 M 0.0 0.0 0:00.26 intr
0 13 13 root -8 0 0 48 S 0.0 0.0 0:00.02 geom
0 14 14 root -16 0 0 16 S 0.0 0.0 0:00.00 sequencer 00
0 15 15 root -68 0 0 80 S 0.0 0.0 0:00.01 usb
0 16 16 root -4 0 0 16 S 0.0 0.0 0:00.00 vnlru
0 17 17 root -16 0 0 16 S 0.0 0.0 0:00.01 schedcpu

```

Figura 7: 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.

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

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 ejecutarán hilos de clase IDLE en caso de inactividad del sistema.

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.

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.

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.

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.

4.4. Resultados del tareas extra

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. Conclusiones

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 visión 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

1.	Estructura simplificada de un proceso.	11
2.	Jerarquía de grupo de procesos.	12
3.	Estructura de colas para hilos <i>RUNNABLE</i>	16
4.	Modelo del hilo.	19
5.	Modelo del planificador.	21
6.	Estado inactivo del sistema.	42
7.	Estado de los núcleos con el modulo encendido/apagado inhabilitado.	43

Índice de cuadros

1.	Descripción de los estados del proceso en FreeBSD	13
2.	Clases de hilos por rango de prioridad.	14
3.	Matriz base previa a la nueva funcionalidad.	31
4.	Matriz de incidencia base con modulo de encendido/apagado de procesadores.	31
5.	Matriz de incidencia previa a la nueva funcionalidad.	31
6.	Matriz de incidencia con las modificaciones pertinentes.	32
7.	Matriz base con las modificaciones pertinentes.	33
8.	Matriz de incidencia con las modificaciones pertinentes.	33

Referencias

- [1] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Professional, 1996.
- [2] Tomás Turina y Nicolás Papp. Modelado del planificador a corto plazo con redes de petri. 2019.

A. Apéndice 1: Archivos de diferencias

A.1. *sched_switch* (v12 a v13): Firma de la función y variables

```
- void sched_switch(struct thread *td, struct thread *newtd, int flags)
+ void sched_switch(struct thread *td, int flags)
{
+   struct thread *newtd;
    struct mtx *tmtx;
    struct td_sched *ts;
    struct proc *p;
    int preempted;

-   tmtx = NULL;
+   tmtx = &sched_lock;
    ts = td_get_sched(td);
    p = td->td_proc;

    THREAD_LOCK_ASSERT(td, MA_OWNED);
...
}
```

A.2. *sched_switch* (v12 a v13): Cambio de posición del bloque que bloquea el hilo previo al *resource_expulse_thread*

```
+   if (td->td_lock != &sched_lock) {
+       mtx_lock_spin(&sched_lock);
+       tmtx = thread_lock_block(td);
+       mtx_unlock_spin(tmtx);
+   }

if ((td->td_flags & TDF_NOLOAD) == 0)
    sched_load_rem();

td->td_lastcpu = td->td_oncpu;
preempted = (td->td_flags & TDF_SLICEEND) == 0 && (flags & SW_PREEMPT) != 0;
td->td_flags &= ~(TDF_NEEDRESCHED | TDF_SLICEEND);
td->td_owepreempt = 0;
td->td_oncpu = NOCPU;

resource_expulse_thread(td, flags);

-   if (td->td_lock != &sched_lock) {
-       mtx_lock_spin(&sched_lock);
-       tmtx = thread_lock_block(td);
-       mtx_unlock_spin(tmtx);
-   }
...
```

A.3. *sched_switch* (v12 a v13): Selección y ejecución de un nuevo thread

```
+ newtd = choosethread();
- if (newtd) {
-     /*
-      * The thread we are about to run needs to be counted
-      * as if it had been added to the run queue and selected.
-      * It came from:
-      * * A preemption
-      * * An upcall
-      * * A followon
-      */
-     KASSERT((newtd->td_inhibitors == 0),
-             ("trying to run inhibited thread"));
-     newtd->td_flags |= TDF_DIDRUN;
-     TD_SET_RUNNING(newtd);
-     if ((newtd->td_flags & TDF_NOLOAD) == 0)
-         sched_load_add();
-     if (ts->ts_runq != &runq){
-         resource_fire_net(newtd, TRAN_UNQUEUE + (PCPU_GET(cpuid)*CPU_BASE_TRANSITI
-     }
-     else{
-         resource_fire_net(newtd, TRAN_FROM_GLOBAL_CPU + (PCPU_GET(cpuid)*CPU_BASE_
-     }
- } else {
-     newtd = choosethread();
-     MPASS(newtd->td_lock == &sched_lock);
- }

resource_execute_thread(newtd, PCPU_GET(cpuid));
```