Universidad Nacional de Córdoba

Facultad de Ciencias Exactas Físicas y Naturales

Modelado del planificador a corto plazo con redes de Petri

Autores: Tomás Turina, Nicolás Papp

31 de enero de 2019

Email: turinatomas@gmail.com, nicolaspapp@gmail.com

Teléfonos: (03573)-15410308, (03547)-15575806

Legajos: 37.524.275, 37.439.047

Director: Maximiliano Eschoyez
Co-Director: Orlando Micolini

Resumen

El sistema operativo se basa en un conjunto de programas que controlan los procesos básicos de una computadora o sistema embebido. Su principal función es permitir a los programas de usuario acceder a los recursos del *hardware* utilizado. Dentro de sus tareas más importantes se encuentra la planificación, que consiste en el reparto de tiempo de procesador a los diferentes hilos y procesos que se encuentran en ejecución. Esta tarea la realiza el *scheduler* o planificador. Existen dos tipos de planificación: estática y dinámica. La planificación estática, utilizada mayormente en sistemas de tiempo real, mantiene prioridades fijas para los hilos, lo cual da previsibilidad pero a costa de flexibilidad. La planificación dinámica, utilizada generalmente en sistemas operativos de propósito general, se basa en estadísticas *post mortem* de tiempos de ejecución para cambiar prioridades.

En este trabajo se presenta una propuesta de planificación dinámica que permita una fácil reasignación de prioridades. Si el planificador pudiera conocer el orden de ejecución de los procesos, se podría disminuir la incertidumbre y agregar determinismo. De esta manera, con conocimiento *a priori* de cada proceso, se mejoraría la información disponible en tiempo de ejecución para su planificación. Lo anterior resultaría de especial importancia para decidir cuántos recursos se necesitan mantener disponibles para la cantidad de procesos en ejecución en un determinado momento.

El planificador que proponemos está modelado con una red de Petri dado que es una herramienta matemática formal. Esta herramienta será de fundamental utilidad a la hora de captar los estados y eventos del planificador.

Índice general

Capítulo 1

Introducción

El sistema operativo se encarga de gestionar los recursos de *hardware* de un dispositivo electrónico y de brindar servicios a los programas de aplicación. Una de sus principales tareas consiste en la ejecución de procesos del sistema y del usuario. Un proceso se puede definir como un programa en ejecución, el cual puede estar compuesto por varios hilos de ejecución.

La planificación de procesos que realiza el sistema operativo involucra varios niveles, categorizados según su plazo de ocurrencia. La planificación a largo plazo decide cuándo un nuevo proceso debe ingresar a la cola de ejecutables del sistema. A mediano plazo administra los procesos que están esperando un evento para continuar con su ejecución. Finalmente, a corto plazo se encarga de determinar el orden de los próximos hilos a ejecutarse y cuándo interrumpir a los que se están ejecutando. El foco de estudio de este trabajo será la planificación a corto plazo.

1.1. Oportunidad

Los sistemas embebidos disponen cada día de nuevas características y mayor capacidad. A su vez, soportan sistemas operativos más potentes. Sin embargo, varios de los sistemas operativos adaptados a sistemas embebidos se diseñaron para computadoras de propósito general, limitando posibilidades para reducción de energía mediante apagado de procesadores y de aprovechar hardware específico del sistema.

Actualmente, los métodos estadísticos son la forma utilizada para tomar las decisiones de planificación en los sistemas operativos de propósito general. Si se contara con información sobre la ejecución en los próximos instantes, el planificador podría cruzarla con el estado actual del sistema para tomar decisiones de forma determinista. Una forma de lograrlo requeriría conocer con anterioridad los recursos que van a necesitar los procesos y así poder gestionarlos a conveniencia. De esta manera, se lograría adaptar el planificador a las necesidades del sistema embebido que gestione.

Utilizar los conocimientos adquiridos sobre redes de Petri nos convertir una

implementación en código fuente a un modelo formal, que mantiene su funcionalidad existente y nos permite agregar funcionalidad propia fácilmente. También, permite cambiar de una lógica definida por un código fuente a una lógica basada en operaciones matriciales, que en un futuro podría ser resuelto con un hardware específico para dicho fin.

1.2. Motivación

El planificador del sistema operativo ha sido siempre la parte encargada de asignar tiempo de procesador a los hilos de diferentes procesos. Los planificadores actuales toman decisiones en base al comportamiento observado del hilo. Nuestro objetivo es buscar un reparto más equitativo del tiempo de CPU mediante la detección de eventos deterministas dentro de los procesos. Como métrica se propone medir los tiempos de finalizaciòn para un grupo de tareas finitas utilizadas como patrón, que pueden ser de uso exhaustivo del CPU o interactivas. También estamos interesados en la detección de estos eventos para determinar cuántas CPU mantener activas al mismo tiempo, y en qué momentos conviene activarlas o desactivarlas.

Se va a tomar como herramienta para desarrollar esta solución las redes de Petri. Esta herramienta tiene ventajas en comparación con una máquina de estados a la hora de determinar el número de estados de cada componente.

El Laboratorio de Arquitectura de Computadoras (LAC) ha realizado varios trabajos que involucran el uso de las redes de Petri para el modelado de sistemas concurrentes. En base a estos trabajos, pensamos que tanto los hilos de ejecución de un proceso como el planificador del sistema operativo pueden considerarse como sistemas que pueden modelarse utilizando esta herramienta. Lograr un modelo correcto permitirá reducir el indeterminismo introducido por la estadística de los modelos actuales. Por lo tanto, nuestra principal motivación surge de disponer de las herramientas necesarias para modelar un nuevo planificador que nos permita dar un gran paso sobre la oportunidad presentada. Además, esta experiencia nos permitirá ampliar nuestros conocimientos sobre *scheduling* en sistemas operativos, particularmente en FreeBSD, sobre el cual desarrollaremos nuestro trabajo.

1.3. Objetivo

Objetivo principal:

- Proponer un modelo del planificador del sistema operativo que mantenga sus hilos en cola durante el menor tiempo posible.

- Proponer un modelo de *scheduler* que permita disminuir la cantidad de decisiones estadísticas a cambio de agregar una mayor información determinista sobre el sistema.

- Proponer un modelo capaz de ser adaptado para aplicaciones específicas.

Objetivos Secundarios:

- Arribar a la implementación de un *scheduler* seguro y con mayor determinismo.
- Planificar todos los hilos y procesos del sistema operativo a través de una red de Petri.
- Estudiar la estructura del sistema operativo FreeBSD y los mecanismos utilizados para el manejo de procesos e hilos.
- Investigar sobre prácticas de desarrollo y depuración del *kernel* del sistema operativo.
- Virtualizar el sistema operativo de desarrollo y depuración y configurar ambiente de desarrollo y depuración.
- Modelar los estados de los hilos.
- Modelar los estados de las CPU y un sistema de planificación de hilos.
- Implementar un modelo de red de Petri para conocer en todo momento el estado de los hilos y las CPU.

1.4. Alcance

El proyecto consiste en tomar como base el planificador 4BSD para hacer una propuesta de planificador modelado con redes de Petri. Posteriormente se lo implementará para probar si el modelo es viable. No se realizarán pruebas exhaustivas, simplemente es un primer prototipo por lo que sólo se probará su funcionamiento.

La elección del planificador 4BSD en vez del planificador ULE se fundamenta en su mayor sencillez y robustez de desarrollo a lo largo del tiempo. Las ventajas del planificador ULE toman relevancia en sistemas con gran cantidad de procesadores (ej: servidores o clusters) y dichos sistemas escapan el alcance de este desarrollo. Se puede encontrar mayor información acerca de las diferencias de estos dos planificadores en la Sección 2.3.3 de este trabajo.

1.5. Modelo de desarrollo

La metodología de trabajo escogida es el modelo iterativo. En cada iteración se plantea un modelo inicial y se lo implementa, para después analizarlo y ver qué requerimiento se debe incluir para el modelo de la siguiente iteración.

1.6. Requerimientos Generales

En esta sección se aclaran los lineamientos de requerimientos generales para el trabajo. Se identifican tanto requerimientos funcionales como no funcionales.

1.6.1. Requerimientos funcionales

Los requerimientos funcionales son:

- Las decisiones de encolado de los hilos en una CPU se toman mediante la red de Petri.
- Los estados globales y los de cada hilo deben estar correctamente representados en la red de Petri.

1.6.2. Requerimientos no funcionales

Los requerimientos no funcionales son:

- El sistema debe ser seguro.
- El sistema no debe permitir interbloqueo.

Capítulo 2

Base Teórica

Dado que el foco principal de estudio será la planificación a corto plazo del sistema operativo, se desarrollarán los contenidos teóricos necesarios para entender el lugar que ocupa un planificador en el núcleo del sistema operativo. Posteriormente, se detallarán conceptos de procesos y sincronización entre procesos, los cuales resultan muy importantes para comprender los objetivos de este trabajo. Estos conceptos abarcados serán abordados en relación al sistema operativo FreeBSD. Este sistema, como se detallará más adelante, esta formado por dos *scheduler*: 4BSD y ULE. Se hará un análisis de sus fortalezas y debilidades y se explicará porqué se ha elegido el 4BSD.

2.1. Introducción

Los niveles de **planificación** se clasifican según su frecuencia de ocurrencia. En los sistemas operativos de propósito general existen 3 niveles de planificación:

- Planificación a corto plazo: se encarga de elegir cuál es el siguiente hilo a ejecutar por el microprocesador.
- Planificación a mediano plazo: se encarga de mover procesos entre memoria principal y disco.
- Planificación a largo plazo: se encarga de inicializar nuevos procesos en el sistema y de finalizarlos.

El *scheduler* de un sistema operativo es el encargado de la planificación a corto plazo. Se encarga de planificar y distribuir el tiempo disponible de los microprocesadores entre todos los procesos que están en estado de ejecución en un determinado momento.

En FreeBSD los procesos tienen tres estados:

- NEW: cuando el proceso se encuentra en etapa de creación.
- NORMAL: una vez inicializado, el proceso va a contar con uno o varios hilos disponibles para ejecutarse.
- ZOMBIE: cuando el proceso está en estado de finalización.

El *scheduler* se encarga de planificar aquellos hilos cuyos correspondientes procesos tienen estado *NORMAL*.

Por otra parte, en FreeBSD los hilos que forman un proceso tienen los siguientes estados:

- INACTIVE: aquellos que están inactivos, en proceso de creación.
- INHIBITED: aquellos que se encuentran inhibidos de su ejecución, ya sea en espera de un recurso del sistema o de un determinado evento.
- CAN_RUN: aquellos que se encuentran correctamente inicializados y pueden agregarse a una cola de ejecución.
- RUNQ: aquellos que se encuentran en una cola de ejecución esperando ser ejecutados.
- RUNNING: cuando se encuentran en ejecución.

Las principales tareas que debe llevar a cabo el planificador son:

1. Ser capaz de agregar aquellos hilos que se encuentran en estado ejecutable a la cola de ejecución. Pasaje de estado: CAN_RUN ⇒ RUNQ.
2. Disponer del recurso CPU y asignarla entre los hilos disponibles en la cola de ejecución de la mejor forma posible. Pasaje de estado: RUNQ ⇒ RUNNING.
3. Extraer de ejecución aquellos hilos que no pueden continuar. Pasaje de estado: RUNNING ⇒ INHIBITED/CAN_RUN.

2.2. Procesos e hilos

Los procesos consisten en un espacio de direcciones que contienen un mapeo sobre el código del programa y variables globales. También tiene una serie de recursos del *kernel* que puede nombrar u operar utilizando llamadas a sistemas.

Cada proceso posee uno o más hilos de ejecución. Cada hilo representa un procesador virtual con un contexto asociado y su propio *stack* mapeado en el espacio de direcciones. También, cada hilo tiene un hilo de *kernel* asociado que representa el hilo de usuario cuando está ejecutando en el *kernel* como resultado de una llamada de sistema, fallo de página o envío de señal.

El *kernel* soporta la ilusión de ejecución concurrente de múltiples procesos repartiendo los recursos del sistema sobre los procesadores que están listos para ejecutar.

El hilo de un proceso opera en modo de usuario o en modo *kernel*. En el espacio de usuario, el hilo ejecuta código de aplicación en un modo sin privilegio. Cuando un hilo pide servicio de parte del sistema operativo, se realiza un cambio de privilegio mediante un mecanismo de protección.

Los recursos que necesita un hilo se dividen en dos partes:

1. Los necesarios para ejecutar en modo usuario (definidos por la arquitectura de la CPU).

2. Los necesarios para ejecutar en modo *kernel* como ser los registros por *hardware*, registros, contador de programa y el puntero al *stack*.

2.2.1. Estados de los procesos

A cada proceso se le asigna un identificador único llamado Identificador de Proceso (PID). El *kernel* los utiliza para referenciar al proceso. Hay dos PID de importancia para un proceso:

1. El PID del proceso en sí.

2. El PID del proceso padre.

El hilo es la unidad de ejecución de un proceso. Los hilos compartiendo un espacio de direcciones se planifican independientemente y pueden realizar llamadas a sistema simultáneamente. La estructura del hilo sólo contiene la información mínima necesaria para correr en el *kernel*.

Para mejor eficiencia en las aplicaciones, FreeBSD usa un modelo de 1:1 en donde cada hilo de usuario se corresponde con un hilo a nivel de *kernel*.

Como muchos sistemas operativos, FreeBSD utiliza la API de *threading* de *POSIX* conocida como `pthreads`. Este modelo incluye una serie de primitivas como ser creación, planificación, coordinación, envío de señales y destrucción de los hilos de un proceso. Provee de semáforos para asegurar el acceso a los recursos.

En su forma más liviana, los hilos de FreeBSD comparten los recursos incluyendo el PID. Cuando se necesita mayor computación paralela, se crea un nuevo hilo con el llamado a la función `pthread_create`. Como todos los hilos comparten la estructura del proceso los mismos tienen un solo PID, por lo que se forman como una sola entrada en la lista de los procesos.

Hay formas de crear un nuevo proceso que sólo comparta un grupo selecto de recursos del padre. Para esto se utiliza la llamada `rfork`. Esta llamada asocia un PID con cada hilo creado y lo gestiona de la misma forma que a los otros procesos del sistema.

2.2.2. Estructura de los procesos e hilos

Un proceso posee las siguientes categorías de información:

- Identificador de grupo de procesos: el grupo de procesos y sesión a la cual pertenece.

- Credenciales de usuario: credenciales de usuarios y grupos para dicho proceso.

- Administración de memoria: describe la asignación del espacio de memoria virtual usada por el proceso.

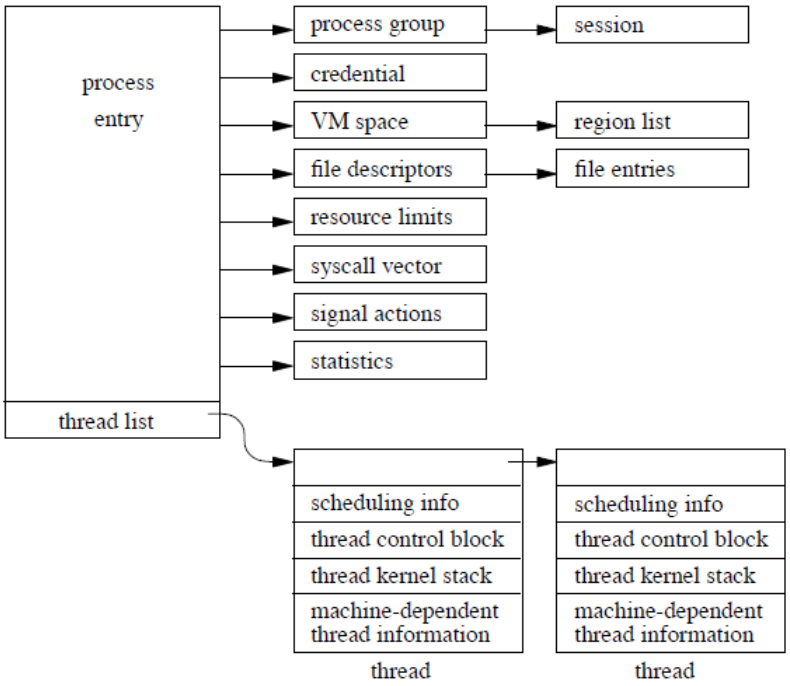

Figura 2.1: Estructura de procesos en FreeBSD, [1, Capítulo 4].

- Descriptores de archivo: un arreglo de punteros a entradas de archivos indexados por los descriptores de archivos abiertos por el programa.
- Vector de llamadas a sistemas: un mapeo de número de llamada a acciones.
- Recursos: la utilización de los recursos del sistema.
- Estadísticas: incluye información de ejecución, tiempos y *profiling*.
- Señales: la acción a tomar cuando se le envía una señal a un proceso.
- Estructura de hilo: el contenido de la estructura de hilos del proceso.

El elemento de estado en la estructura de un proceso mantiene el valor actual del estado del proceso. Los posibles estados se muestran en la figura 2.2.

Cuando un proceso se crea, se pone en el estado *NEW*. Cuando se le asignan los recursos que necesita para ejecución su estado cambia a *NORMAL*. Una vez en estado *NORMAL*, sus hilos pueden estar en uno de tres estados: *RUNNABLE*, *SLEEPING* o *STOPPED*. El estado *RUNNABLE* implica que está listo para ejecutarse, *SLEEPING* quiere decir esperando por un evento, y *STOPPED* cuando el proceso padre le envía una señal para que se detenga. Al finalizar un proceso se lo marca como *ZOMBIE* hasta que haya liberado sus recursos y haya comunicado su estado de terminación a su proceso padre.

State	Description
NEW	undergoing process creation
NORMAL	thread(s) will be RUNNABLE, SLEEPING, or STOPPED
ZOMBIE	undergoing process termination

Figura 2.2: Estados de los procesos, [1, Capitulo 4].

Los hilos no bloquedos y en estado *NORMAL* esperan en una de estas tres colas: `run_queue, sleep_queue o turnstile_queue`. Los hilos en ejecución se encuentran en la `run_queue`, mientras que los bloqueados esperando por algún evento pueden estar localizados en la `sleep_queue, turnstile_queue` o en ninguna de las anteriores. Tanto la `sleep_queue` como la `turnstile_queue` son tablas de datos de tipo *hash* cuyo índice es un identificador de evento.

También se utilizan punteros `p_ptr` y `p_children` para ver la jerarquía de procesos, como puede verse en la figura 2.3.

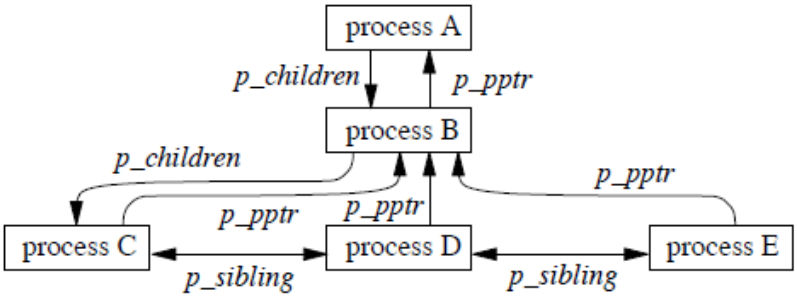

Figura 2.3: Orden jerárquico entre procesos padres e hijos [1, Capitulo 4].

El tiempo en el planificador se concede de acuerdo a la clase y prioridad del proceso planificado.

En la figura 2.3 se observan las clases y prioridades que pueden tener los hilos. Las prioridades de los hilos en la clase *realtime* se establecen por las aplicaciones usando la llamada de sistema `rtprio` y el *kernel* no puede reajustar el valor seteado. La clase *ithd* se establece cuando se configura el dispositivo y nunca se cambia. Las prioridades de los hilos corriendo en el modo compartido por tiempo *(timeshare)* se ajustan por el *kernel* basado en los recursos utilizados de la CPU.

La estructura de los hilos incluye:

- Planificación: compuesta por:

    1. Prioridad del hilo en modo *kernel*

Range	Class	Thread type
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

Figura 2.4: Rango y clases de los hilos [1, Capitulo 4].

2. Prioridad del hilo en modo usuario
3. Utilización reciente de la CPU
4. Tiempo utilizado en la última suspensión

- TSB: estado de ejecución en modo usuario y modo *kernel*.
- Kernel Stack: el *stack* de ejecución para el *kernel*.
- Machine State: la información de hilo dependiente del *hardware*.

Cada hilo potencialmente ejecutable debe tener su *stack* residente en memoria porque una de las tareas del *stack* es manejar los fallos de página. Si no estuviera residente, habría una falla de página cuando el hilo se empiece a ejecutar y no habría *stack* del *kernel* disponible para dar servicio a ese fallo de página.

Los que implementan código deben ser cuidadosos al escribir código que se ejecuta en el *kernel* para evitar usar variables locales muy grandes y subrutinas con mucho anidamiento, para evitar un sobrepasamiento en el *stack* de ejecución.

2.3. Planificación

Las políticas de planificación típicamente se usan para intentar balancear la utilización de recursos comparándolo con el tiempo que tarda en completarse el programa. El planificador por defecto de FreeBSD es el compartido por tiempo, en donde la prioridad de un proceso se recalcula periódicamente basándose en los parámetros previos, como la cantidad de tiempo usado de CPU o la cantidad de recursos de memoria que mantiene o requiere para ejecutarse. Algunas tareas requieren control más preciso sobre el proceso (*real-time scheduling*). El sistema operativo FreeBSD implementa planificación de tiempo real utilizando una cola separada de la utilizada por procesos regulares. Los hilos en esa cola no se degradan en prioridad ni se interrumpen por otros hilos, salvo que tengan igual o mayor prioridad.

El *kernel* FreeBSD también tiene una cola con hilos de mínima prioridad, que se ejecutan solamente cuando ningún otro hilo en las colas de mayor prioridad está en un estado de posible ejecución.

El *scheduler* por tiempo de FreeBSD usa un método que favorece a los programas interactivos. La ejecución asigna una prioridad alta a cada hilo y permite que el hilo se ejecute por un periodo fijo de tiempo (conocido como *time slice*). A medida que se ejecutan disminuye su prioridad, mientras que a los suspendidos (por I/O) mantienen su prioridad. Los hilos que se mantienen inactivos mejoran su prioridad.

Hay tareas que se hacen en pasos muy pequeños, ningún paso individual corre el tiempo suficiente como para que se degrade su prioridad. Para solucionar este problema, la prioridad del proceso hijo se propaga al padre. Cuando se crea un proceso hijo comienza con la prioridad del padre.

Otro objetivo del *scheduler* es la de minimizar el *thrashing*, un fenómeno que ocurre cuando cuando la memoria principal está tan llena que se pasa mayor tiempo planificando y en fallos de páginas que ejecutando aplicaciones. El *thrashing* se reduce marcando el último proceso en ejecución como que no se le permite correr.

2.3.1. Planificación de hilos

El planificador de FreeBSD tiene una serie de *APIs (Application Programming Interface)* que permiten soportar diferentes tipos de planificadores. El *kernel* FreeBSD tiene dos planificadores disponibles:

- El planificador ULE, que se encuentra en `/sys/kern/sched_ule.c`, es el utilizado por defecto.
- El planificador tradicional de 4BSD en `/sys/kern/sched_4bsd.c`. Este planificador se sigue manteniendo como alternativa al ULE.

Ya que un sistema ocupado realiza millones de decisiones de planificación por segundo, el mecanismo con el cual se toman las decisiones de planificación es crítica para la performance del sistema.

FreeBSD requiere seleccionar el planificador en el momento en que se construye el *kernel*. Todas las llamadas en el código del planificador se resuelven en tiempo de compilación en lugar de realizar una llamada indirecta cada vez que se toma una decisión de planificación.

2.3.2. Planificador de bajo nivel

El planificador se divide en dos partes: un planificador de bajo nivel que se ejecuta constantemente y uno más complejo de alto nivel que se ejecuta unas cuantas veces por segundo.

Para mayor eficiencia, se necesita tomar una decisión rápidamente con la menor cantidad de información. Para simplificar esto, el *kernel* mantiene varias colas para cada CPU, organizadas de alta a baja prioridad. La responsabilidad del planificador de bajo nivel es seleccionar el hilo de más alta prioridad de la cola de la CPU. El planificador de alto nivel se encarga de elegir las prioridades

de los hilos y decidir en que cola de CPU ponerlo.

El planificador de alto nivel asigna una prioridad a todos los hilos ejecutables que determina en qué cola de ejecución se encolarán.

Si se acumulan múltiples hilos en una cola de tarea, el sistema se convierte en un *round robin*, lo que quiere decir que se ejecutan en el orden que se encuentran en la cola. Si el hilo se bloquea, se pone en una `turnstile_queue`. Si utiliza su ventana de tiempo, se reubica al final de la cola de la que provino a la espera de una nueva ventana de ejecución.

2.3.3. Diferencias entre el planificador ULE y el 4BSD

Como se mencionó anteriormente, FreeBSD dispone de dos tipos de *scheduler*. Ambos cumplen la misma funcionalidad pero el ULE incorpora nuevas características sobre todo destinadas a plataformas SMP/SMT con cargas pesadas.

Se realizarán comparaciones del uso del *scheduler*, la afinidad de CPU, el reajuste de prioridades y el tratamiento de procesos interactivos.

Uso del scheduler:

- El 4BSD fue diseñado originalmente para sistemas monoprocesador o de trabajos con carga de procesador. Posteriormente, se amplió para sistemas multiprocesador.
- El ULE fue diseñado a partir de las nuevas necesidades que 4BSD no satisface para plataformas SMP/SMT.

Afinidad de CPU:

- Si bien ambos sistemas soportan afinidad de CPU, el planificador ULE crea una topología de CPU que describe la relación entre las CPU del mismo sistema, lo que permite identificar la CPU con menor costo de migración disponible.

Prioridades:

- En 4BSD las prioridades se actualizan periódicamente en base al tiempo de procesador que cada hilo ha consumido, aumentando la prioridad en aquellos que consumieron menos tiempo y penalizando a los de mayor utilización.
- En ULE no hay un ajuste periódico de prioridad de los hilos. La inanición se previene manteniendo dos colas, la de corrientes y la de siguientes. Cada hilo entrante es asignado a la cola de siguientes. Se ejecuta la cola de corrientes en orden de prioridad hasta vaciarla, en ese momento se invierten las colas, la de siguientes pasa a ser la de corrientes y viceversa. Esto hace que cualquier hilo se ejecute al menos una vez cada dos inversiones de las colas sin importar su prioridad.

Procesos interactivos:

- Tanto en 4BSD como en ULE se incrementa la prioridad de aquellos procesos que son considerados como interactivos. En 4BSD, para determinar si un proceso es interactivo se miden los tiempos de espera tanto voluntarios como involuntarios, permitiéndole a una tarea no interactiva convertise en interactiva porque el sistema operativo no la dejó correr mucho tiempo debido a la carga del sistema.
- En ULE se miden solamente los tiempos de espera haciéndolo más preciso.

2.3.4. Colas de hilos ejecutables y cambios de contexto

Como observamos en la figura 2.3 las prioridades se encuentran distribuidas entre 0 y 255. Los hilos con prioridades en el rango de 120 a 223 recalculan su prioridad en forma constante (tiempo compartido). Los hilos de tiempo real y los *idle threads* establecidos por las aplicaciones tienen los rangos de 48 a 79 y 224 a 255, respectivamente. Los rangos de 0 a 47, son inicializados por el sistema y permanecen constantes.

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

La cabeza de cada cola está contenida en un arreglo. Asociado con cada arreglo hay un vector de bits `rq_status` que se usa para identificar las colas que no están vacías. Hay dos rutinas, `runq_add` y `runq_remove`, utilizadas para poner o sacar un hilo de una cola. El corazón del algoritmo de planificación es `runq_choose`. Esta rutina es responsable de seleccionar un nuevo hilo para correr, siguiendo los siguientes pasos:

1. Se asegura que se tome el cerrojo asociado a la cola de ejecución.
2. Encuentra la primera fila que no esté vacía identificando el primer bit que no sea cero en el arreglo `rq_status`. Si `rq_status` es 0, no hay ningún hilo para correr y selecciona un hilo que hace un bucle de espera.
3. Saca el primer hilo de la cola no vacía.
4. Si se vacía la cola al remover el hilo, pone el bit a 0 de `rq_status`.
5. Devuelve el hilo seleccionado.

La parte del código independiente de la arquitectura reside en `mi_switch`, mientras que la parte dependiente reside en `cpu_switch`. Esta separación es importante porque es una característica constructiva de FreeBSD. Esto se distingue para poder ampliar los horizontes de compatibilidad multiplataforma.

El mecanismo de cambio de contexto involuntario funciona poniendo en 1 la bandera `TDF_NEEDRESCHED` del hilo y publicando un *Asynchronous System Trap (AST)*. Una AST consiste en una trampa en el hilo que retorna el control al sistema.

2.3.5. Planificador de hilos de tiempo compartido

Una de las tareas del planificador es asegurar que la CPU esté siempre ocupada. El procesador que está corriendo el hilo lo remueve si completa una tarea, termina su turno o bien se bloquea esperando recursos. Cuando se migra un hilo se pierde la *cache* y ésta se carga en la nueva CPU a la que ha sido migrado el hilo.

El término afinidad del procesador describe un planificador que sólo migra hilos cuando es necesario. Esto sucede exclusivamente para darle a un procesador disponible algo que hacer.

Los procesadores con muchas CPU necesitan balancear las cargas de cada *core*. Con este fin, se define una jerarquía de CPU. En ella se ubican primero las CPU más rápidas de migrar, que son las que se encuentran en la misma pastilla. Se prosigue a cubrir el resto hasta incluir todos los *core* en la jerarquía. Cuando se decide a qué procesador migrar el hilo, se fija dentro de los *cores* disponibles cuál *core* está más alto en la jerarquía ya que va a tener el menor costo.

El algoritmo de planificación es quien necesita estar consciente de las múltiples CPU, razón por la cual se comienzó con el desarrollo del planificador ULE, cuyo propósito de desarrollo fue el de compensar los problemas del planificador BSD tradicional para planificar en multiprocesadores.

Las razones para el nuevo planificador son las siguientes:

- Brindar afinidad en sistemas con multi-procesador.
- Distribuir las cargas de las CPU en sistemas con multiprocesador.
- Dar mejor soporte a procesadores con múltiples núcleos en un chip.
- Mejorar el planificador para que no sea dependiente del número de hilos en el sistema.

El planificador ULE está compuesto por dos algoritmos o partes ortogonales entre sí; los que pueden manejar la afinidad y distribución de hilos sobre CPU y los que son responsables del orden y duración de los hilos. ULE evalúa el comportamiento de los hilos mediante el análisis de los eventos para diferenciar a los procesos interactivos de los que son por lote.

Existe un algoritmo que evalúa la interactividad de los hilos mediante una puntuación, el cual es un número entre 0 y 100. Se busca diferenciar aquellos hilos que no están corriendo debido a que tienen una prioridad inferior a aquellos que están periódicamente esperando por entrada de usuario.

Una dificultad existe en aplicaciones que consumen una porción significativa de recursos por breves periodos de tiempo mientras se deben mantener interactivas. Para lidiar con estas situaciones, se lleva un histórico de varios segundos del comportamiento de cada hilo y que va decayendo gradualmente. Un hilo se considera interactivo si el cociente entre el tiempo detenido voluntariamente comparado contra el tiempo de ejecución está por debajo de cierto factor. Ese

factor está definido por el código de ULE y no es configurable.

Para hilos cuyo tiempo en estado detenido excede el tiempo de ejecución:

$$\text{puntuacion interactiva} = \frac{\text{factor escalamiento}}{\frac{\text{detenido voluntariamente}}{\text{ejecutando}}}$$

Para el caso contrario:

$$\text{puntuacion interactiva} = \frac{\text{factor escalamiento}}{\frac{\text{ejecutando}}{\text{detenido}}} + \text{factor escalamiento}$$

El factor de escalamiento es el mayor puntaje de interactividad dividido por dos. La rutina llamada `sched_interac_update` se utiliza en varios puntos de la existencia de un hilo, por ejemplo en la llamada a la rutina `wakeup` para actualizar los tiempos de ejecución y de detención.

Cada hilo guarda la utilización de la CPU como la cantidad de *ticks*, de típicamente 1 milisegundo, durante el cual se ejecutó una ventana de tiempo definido por el primer y último *tick*. El planificador mantiene alrededor de 10 segundos de históricos.

El planificador implementa *round-robin* para la asignación de ventanas de tiempo. Una ventana de tiempo es un intervalo fijo en el que se ejecuta a un hilo antes de que el planificador elija otro hilo de igual prioridad para correr. El intervalo de tiempo previene la inanición entre hilos de igual prioridad.

El planificador también debe trabajar para evitar la inanición de los trabajos con carga de CPU en baja prioridad. El planificador ULE crea tres arreglos de colas para cada CPU en el sistema. Tener una cola por cada CPU hace que sea posible de implementar afinidad de procesador en un sistema multiprocesador. Uno de esos arreglos de colas es la `idle_queue` donde guarda todos los hilos con esa prioridad. El arreglo se ordena de la prioridad más alta a la más baja. El segundo arreglo contiene a la cola de hilos de tiempo real, que está organizada de la misma forma. El tercer arreglo de colas es la cola de tiempo compartido. En vez de ordenarse por prioridad, se manejan como una cola por calendario. Un puntero referencia la entrada, y avanza una vez por cada *tick* de sistema, aunque puede no avanzar hasta que la cola seleccionada esté vacía.

Muchos de los parámetros de los algoritmos pueden explorarse en tiempo real mediante `sysctl kern.schedtree`. El resto son constantes de tiempo de compilación y están documentados en la parte superior del archivo fuente del planificador (ubicada en `/sys/kern/sched_ule.c`). Los hilos se eligen para ejecutar en orden de prioridad, desde la cola de tiempo real hasta que ésta esté vacía, punto en el cuál los hilos de la cola de tiempo compartido empezarán a ejecutarse.

2.3.6. Planificación de multiprocesadores

El planificador balancea afinidades para crear la ilusión de un planificador global, que tiene colas locales para cada procesador, y para tener un buen rendimiento. Las decisiones de balanceo se toman en base a la topología de CPU aportada por código dependiente de la arquitectura que describe la relación entre las CPU del mismo sistema.

La topología del sistema identifica cuáles CPU son simétricas, cuáles comparten alguna *cache*, etcétera. Luego, la función recursiva llamada `cpu_search` analiza la topología. Cuando un hilo pasa a la cola de ejecutables como resultado de despertarse, liberarse, crearse o cualquier otro tipo de evento, la función `sched_pickcpu` decide en donde se ejecutará. El planificador ULE determina cuál es la mejor CPU basado en el siguiente criterio:

1. Si el hilo tiene afinidad a una sola CPU o está ligado por corto plazo eligen a la única CPU permitida.
2. Si el hilo es de interrupciones que se planifican por manejos de interrupción a nivel de *hardware* se ejecutan en el CPU donde se originó la interrupción siempre y cuando su prioridad sea la suficiente para correr inmediatamente.
3. Si el hilo tiene afinidad a más de una CPU se recorre de forma inversa el árbol desde la última CPU en el que ha sido planificado hasta la CPU donde se encuentra una afinidad válida que pueda correr el hilo inmediatamente.
4. Si el hilo no tiene afinidad pero tiene alta prioridad, el sistema completo busca la CPU con la menor carga que esté corriendo un hilo de menor prioridad al que se quiere planificar.
5. Si el hilo no tiene afinidad y su prioridad es baja se busca en el sistema completo por la CPU con menor carga.

Posteriormente se compara la CPU actual con el resultado de estas búsquedas para ver si sería una decisión preferible ejecutarlo en una CPU diferente a la actual. Cuando un procesador no tiene nada en ejecución se pone en 1 un bit en una máscara compartida por todos los procesadores. La CPU disponible llama a `tdq_idled` para buscar en otras CPU trabajos que puedan migrarse para mantenerse ocupada.

También se puede mover el trabajo a una CPU disponible. Cuando una CPU activa está por agregar trabajo a su propia cola, primero se fija si tiene un exceso de trabajo y si no hay alguna otra CPU en espera en el sistema. Si se encuentra una, entonces el hilo migra a esta última usando una interrupción entre procesos (*Interprocessor Interrupt (IPI)*).

2.4. Cambios de contexto

El *kernel* cambia de contexto permanentemente para compartir la CPU de una manera efectiva. Cuando un hilo en ejecución se bloquea, el *kernel* encuentra otro hilo para correr y realiza un cambio de contexto. El sistema también

puede interrumpir al hilo en ejecución para correr un hilo desencadenado por un evento asincrónico, como una interrupción del dispositivo.

Cambiar entre hilos ocurre sincrónicamente respecto al hilo en ejecución, mientras que atender interrupciones ocurre de forma asincrónica. Los cambios de contexto entre procesos, por otro lado, se miden en voluntarios o involuntarios.

Un cambio de contexto involuntario se da cuando el hilo se ejecuta durante el tiempo asignado de ejecución o bien el sistema identifica un hilo con mayor prioridad para correr.

Un cambio de contexto voluntario se da con una llamada a la subrutina `sleep`, mientras que un cambio involuntario ocurre por mecanismos encontrados en las rutinas `mi_switch` y `setrunnable`.

2.4.1. Cambios de contexto a bajo nivel

La localización del contexto de un proceso dentro de su estructura interna le permite al *kernel* realizar cambios de contexto simplemente cambiando la noción de la estructura del hilo en ejecución (o del proceso) y restaurando el contexto descripto en la TSB[1] del hilo.

2.4.2. Cambio de contexto voluntario

El cambio de contexto voluntario se produce cuando un hilo debe esperar la disponibilidad de un recurso o que ocurra un evento. En FreeBSD, los cambios de contextos voluntarios inician a través de pedidos para obtener semáforos tomados por otro hilo o por llamadas a la subrutina `sleep`. Cuando un hilo ya no necesita la CPU, se suspende esperando en un canal de espera (*wait channel*) y obtiene una prioridad para cuando despierte. Cuando un hilo se bloquea por un semáforo, el canal de espera usualmente es la dirección del semáforo. Cuando el bloqueo es por la espera de un evento o recurso, el canal de espera usualmente es la dirección o alguna estructura de datos que identifica ese evento o recurso.

Cuando un proceso padre hace un *wait* para esperar la terminación de sus hijos, debe esperar que uno de sus hijos termine. Como no sabe cuál de los hijos va a terminar primero y como sólo puede esperar en un canal de espera, se duerme con la dirección de su propia estructura del proceso. Cuando el hijo termina despierta la estructura de proceso de su padre en vez de la de sí mismo. Una vez corriendo, debe escanear la lista de hijos para determinar cuál fue el que terminó.

Un hilo puede bloquearse por un tiempo corto, medio o largo dependiendo de la espera. Si es por tiempo corto se asocia a un pedido de semáforo, y se maneja con una estructura de datos llamada `turnstile`. Esta estructura se encarga de seguir al propietario de cada semáforo y la lista de hilos esperando para adquirirlo.

[1] *Thread State Block - Bloque que contiene toda la información de un hilo*

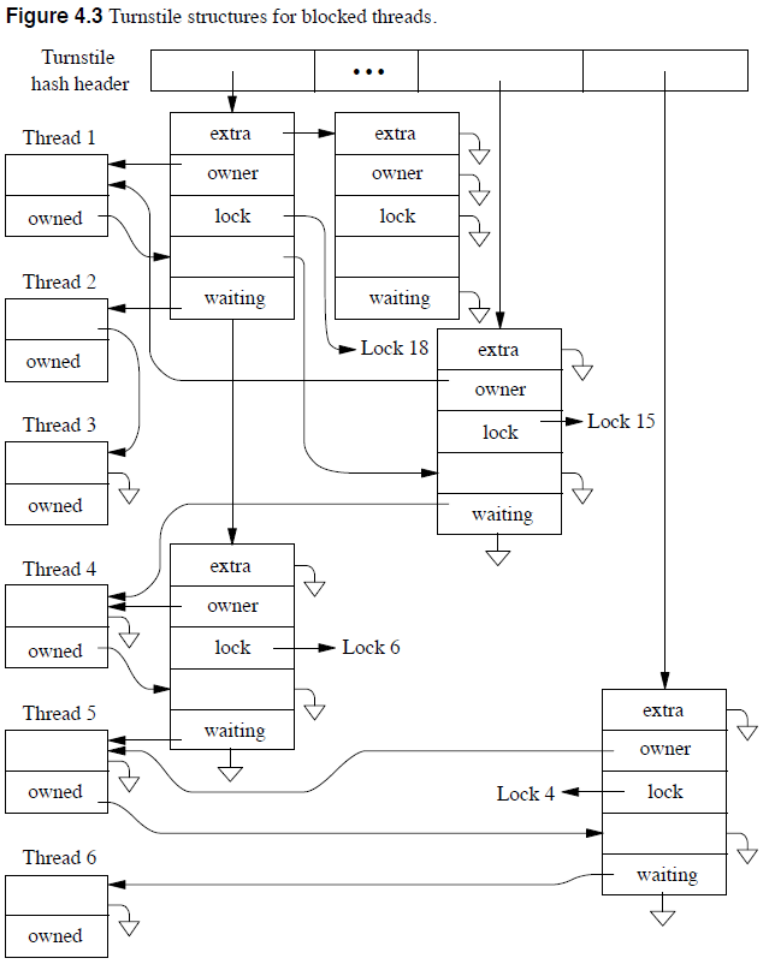

Figura 2.5: Estructura de *turnstiles* para un bloque de hilos, [1, Capitulo 4].

El encabezado es una tabla *hash*[2] para encontrar los semáforos con hilos en espera. Si se encuentra un `turnstile` provee un puntero al hilo que tiene el semáforo y lista los hilos en espera para acceder. Esta estructura se utiliza para identificar los hilos a despertar al soltarse un semáforo. Una estructura `turnstile` es necesaria cada vez que un hilo se bloquea en un semáforo. Ya que los bloqueos en semáforos son comunes, sería muy lento adquirir y liberar memoria de ésta estructura cada vez que se ejecuta un bloqueo, así que cada hilo asigna un `turnstile` al crearse. Ya que el hilo sólo puede estar bloqueado en un semáforo sólo se necesita uno por hilo. Si el hilo es el primero en bloquearse, se utiliza el `turnstile` de ese hilo. Si no es el primero, entonces se utiliza la estructura del hilo anterior. Los `turnstile` adicionales inutilizados se mantienen en una lista.

Esta estructura a su vez permite evitar casos de inversión de prioridad. Una inversión de prioridad ocurre cuando se intenta adquirir un semáforo adquirido por un hilo con menor prioridad que la del hilo que intenta acceder. Se resuelve mediante la propagación de prioridades. El hilo con mayor prioridad se ejecuta y si se bloque al adquirir un semáforo ocupado por un hilo de menor prioridad, se propaga la prioridad del hilo de mayor prioridad a la de menor. Una vez terminada su adquisición, el hilo con la prioridad incrementada temporalmente suelta el semáforo y en ese momento decrementa nuevamente su prioridad permitiendo que el hilo de mayor prioridad tome ese semáforo.

Para los procesos de mediano y largo plazo, se usa una estructura de datos llamada `sleepqueue` en la que no se necesita saber quién retiene el semáforo, ya que no necesitan propagar prioridades. A diferencia de los procesos bloqueados a corto tiempo, los semáforos de mediano y largo plazo tienen un tiempo límite para que el hilo despierte y retornan un mensaje de error en caso de que no hayan podido tomar el semáforo. Los procesos bloqueados a largo plazo pueden ser interrumpibles, esto quiere decir que despiertan si se les envía una señal a ellos o el evento para el cuál se encuentran esperando sucede antes de tiempo.

Cuando los hilos corren concurrentemente, el proceso de *adaptive spinning*[3] asegura que no se bloqueen. Como se liberan de mayor a menor prioridad, el hilo de mayor prioridad será normalmente el que tome el semáforo. No habría necesidad de propagación de prioridad. Más bien, el semáforo se da desde los hilos de mayor prioridad hacia los de menor prioridad.

2.5. Sincronización

El *kernel* de FreeBSD soporta tanto arquitecturas de multiprocesamiento simétrico (SMP) como acceso no unificado a la memoria (NUMA). Una arquitectura SMP es aquella en la que todas las CPU están conectadas a una memoria principal mientras que la arquitectura NUMA es aquella en la cual las CPU están conectadas a una memoria no uniforme.

[2] hash: Función computable que convierte un conjunto de elementos en un rango de salida finito

[3] *adaptive spinning*: Ocurre cuando se espera por un semáforo sin bloquearse el hilo que lo espera

La forma más simple de sincronización se logra empleando las secciones críticas. Una sección crítica es una porción de código que se debe tratar como una porción atómica en ejecución para evitar los problemas generados por la concurrencia. Cuando un hilo está corriendo en una sección crítica no puede migrar a otra CPU ni interrumpirse por otro hilo. Una sección crítica permite la ejecución concurrente de las CPU, por lo que no puede ser protegida a nivel de estructura de datos de sistemas. Una sección crítica comienza con `critical_enter` y continua hasta recibir un llamado a la función `critical_exit`.

2.5.1. Sincronización por semáforos

Los semáforos son el método primario utilizado para la sincronización a corto plazo. Se deben tener en cuenta las siguientes consideraciones de diseño:

- Tomar y soltar un semáforo debe ser tan rápido como sea posible.
- Los semáforos deben tener la información y espacio de almacenamiento para soportar propagación de prioridad.
- Un hilo debe poder adquirir un semáforo recursivamente si el mismo está inicializado para soportar la recursión.

Los semáforos están construidos a partir de una instrucción en *hardware* que permite hacer un *compare-and-swap*. Esta instrucción se ejecuta con destino a la dirección de memoria del semáforo. Cuando el semáforo está libre, `MTX\_UNOWED` está guardado en esa dirección de memoria. Cuando el semáforo está tomado, se reemplaza ese valor por un puntero al hilo que tomó ese semáforo. Cuando se intenta adquirir un semáforo se compara esa dirección de memoria con `MTX\_UNOWED` y si es igual se realiza el intercambio por la dirección de memoria del hilo.

Hay dos tipos de semáforos, los bloqueantes y los no bloqueantes. Los semáforos que no duermen a los hilos de ejecución se llaman *spin mutexes*. Estos forman un ciclo esperando a que otro hilo de otra CPU suelte el semáforo. El *spin* puede resultar en un interbloqueo si un hilo bloquea al hilo que tiene el semáforo y luego intenta adquirir ese mismo semáforo. Para evitar que esto suceda se corre este tipo de semáforo dentro de una sección crítica con interrupciones deshabilitadas. Estos tipos de semáforos se utilizan cuando se pretende adquirirlos por un breve periodo de tiempo. Los hilos no pueden dormirse mientras retengan uno de estos semáforos.

Utilizar un semáforo de *spin* en lugares donde los recursos van a permanecer tomados durante grandes periodos de tiempo desperdicia recursos de una PC. A su vez, este tipo de semáforos no son apropiados para monoprocesadores, por lo que se convierten en semáforos convencionales en este caso.

2.5.2. Interfaz de semáforos

Primero se debe inicializar el semáforo mediante la función `mtx_init`. Luego se utiliza `mtx_lock` para adquirirlo. Cabe aclarar que el método `mtx_lock_spin`

es similar al anterior, excepto que lo hace utilizando el *spin* hasta que el semáforo se encuentre disponible.

Es posible para el hilo adquirir el semáforo de forma recursiva y sin efectos secundarios si durante la inicialización se lo configuró con el *flag* `MTX_RECURSE`.

- El método `mtx_trylock` intenta adquirir el semáforo y si no lo logra retorna 0.
- El método `mtx_unlock` desbloquea el semáforo.
- El método `mtx_unlock_spin` libera un semáforo del tipo *spin* que se adquirió usando `mtx_lock_spin`.
- El método `mtx_destroy` destruye el semáforo, liberando la memoria ocupada.

Se debe inicializar con un cerrojo todo semáforo que se quiera destruir posteriormente. No se le permite a un hilo tomar un semáforo destruido. En caso de que esto ocurra, se produce un *kernel panic*.

2.5.3. Sincronización y variables de condición

La sincronización entre procesos de un recurso generalmente se implementa con una estructura de tipo cerrojo *(lock)*. El *kernel* tiene manejadores de estas estructuras para gestionarlas. No todos los tipos de cerrojos soportan la misma cantidad de operaciones. Un cerrojo de escritura-lectura opera como un semáforo con la excepción que el cerrojo soporta tanto el acceso exclusivo como compartido.

Junto a los semáforos se usan variables de condición para esperar que estas condiciones ocurran. Los hilos esperan por una condición llamando `cv_wait`, `cv_wait_sig` (esperar hasta que sea interrumpido). Para desbloquear se llama `cv_signal` o `cv_broadcast`. La función `cv_wait_remove` extrae un hilo en espera en una cola de condición, si es que se encuentra en una.

El hilo debe tener tomado el semáforo antes de poder llamar a `cv_wait`, `cv_wait_sig`, `cw_timedwait` o `cv_timewait_sig`. También debe tenerlo tomado para llamar a `cv_signal` o `cv_broadcast`.

2.5.4. Prevención de deadlocks

Para evitar *deadlocks*, FreeBSD mantiene un orden parcial sobre todos los cerrojos. El orden se basa en dos reglas:

1. Un hilo no puede tomar más de un cerrojo de cada clase.
2. Un hilo solo puede tomar un cerrojo con número de clase mayor al número de clase máximo de los cerrojos que tiene adquiridos.

Estas reglas no son suficientes, por lo que se complementa con un módulo testigo que monitorea y refuerza el orden parcial de los cerrojos. Cada vez que

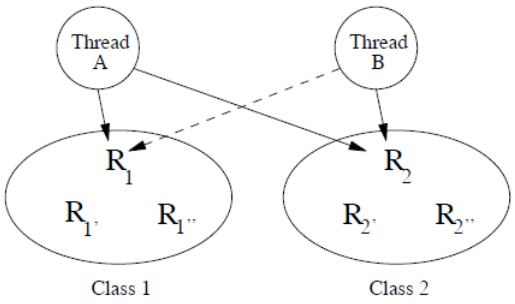

Figura 2.6: Reglas de prevención de *deadlocks*, obtenida de [1, Capitulo 4].

se adquiere un cerrojo, el módulo testigo usa estas dos listas para verificar que los cerrojos no se adquieran en el orden equivocado. Si se detecta una violación, se muestra un mensaje por consola detallando los cerrojos involucrados y su ubicación en el código. Este módulo testigo se puede configurar para entrar en pánico o mostrarlo en el *debugger* del *kernel* cuando se detecta una violación de orden o falla alguna otra verificación.

2.6. Elección del planificador a utilizar

En base a la información obtenida de los *scheduler* 4BSD y ULE, se opta por trabajar con el planificador 4BSD por las siguientes razones:

1. Al ser una primera incursión del grupo de trabajo del LAC en *scheduling*, se favorece a 4BSD al tener una menor curva de aprendizaje.

2. El objetivo del trabajo es presentar un modelo inicial que sustituya las funcionalidades básicas del *scheduler*, por lo que es conviente partir del más simple.

3. Las funcionalidades que aporta ULE por sobre 4BSD no resultan de interés en el objetivo a largo plazo ya que se pretende cambiar por completo la funcionalidad de esos módulos.

Capítulo 3

Desarrollo

El modelo propuesto para el desarrollo del trabajo es el modelo iterativo.

En cada iteración se presentará:

1. Planteo de los objetivos que se esperan alcanzar al finalizar la iteración.
2. Propuesta de un nuevo modelo de redes de Petri y/o modificaciones al anterior para cubrir los objetivos propuestos anteriormente.
3. Explicación detallada del funcionamiento del modelo propuesto.
4. Validación del modelo.
5. Implementación del modelo.
6. Análisis final en base a los objetivos planteados al inicio de la iteración y los resultados obtenidos a su finalización.
7. Pasos a seguir en la próxima iteración.

El entorno de trabajo necesario para la propuesta, el desarrollo, el análisis, la implementación y la validación de los modelos estará conformado por:

- Plataforma de simulación y análisis de redes de Petri PIPE. El correcto funcionamiento de todo modelo propuesto se validará en esta plataforma.
- Máquina virtual con sistema operativo FreeBSD. El *kernel* seleccionado al momento de su compilación y sobre el cual se hará foco en este trabajo es el 4BSD.
- Código fuente del *kernel* de FreeBSD. Se utilizará un editor de texto para incorporar las modificaciones propuestas.
- Servidor MobaXterm para conexión SSH a la máquina virtual. Los cambios en el código fuente y la compilación del *kernel* con las modificaciones se realizarán por esta vía.
- Marco teórico y bibliografía sobre *scheduling* en FreeBSD.

3.1. Primera iteración: Modelo inicial del hilo

3.1.1. Objetivos a alcanzar

En esta iteración se buscará proponer un modelo inicial de red de Petri para representar los diferentes estados en los que se puede encontrar un hilo.

3.1.2. Modelo propuesto

En base a los 5 estados en los que puede encontrarse un hilo se modeló la red de la figura 3.1.

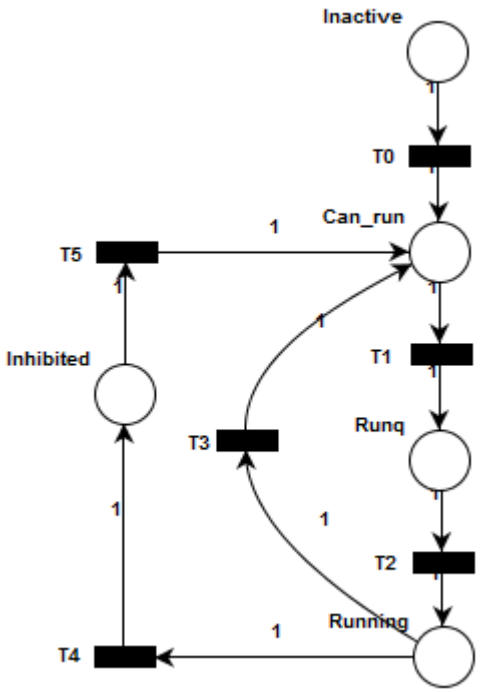

Figura 3.1: Modelo propuesto del hilo.

3.1.3. Análisis del modelo

Las plazas de la red De Petri propuesta coinciden con los estados de los hilos, que solamente puede estar en un estado a la vez.

Por su parte, las transiciones representan:

- T0: El paso del estado INACTIVE a CAN_RUN. Esto sucede cuando el hilo se agrega al *scheduler*. Esto sucede generalmente en momento de creación de un proceso o cuando el mismo realiza un *fork*. Esta tarea no corresponde al *scheduler*, por lo que inicialmente un hilo en el *scheduler* se encuentra inicializado en el estado CAN_RUN. Esta transición nunca se dispara, solo se la incorpora al modelo de modo representativo.

- T1: El hilo se pone en una cola local de una determinada CPU o en la cola global dependiendo de la disponibilidad. Esta cola organiza los hilos de acuerdo a sus prioridades de ejecución.
- T2: El hilo pasa de la cola ejecutando las instrucciones del programa que tiene asignadas. En este instante el procesador se encuentra ocupado por dicho hilo.
- T3: El *scheduler* interrumpe el hilo y lo vuelve a colocar en una cola. El planificador toma otro hilo de la cola (el de mayor prioridad) y realiza un cambio de contexto.
- T4: Algún evento, semáforo o espera bloquea al hilo. Se agrega en una *sleepq* o *turnstile*, en la cual el hilo queda a la espera de un evento que le quitará el bloqueo.
- T5: Se desbloquea el hilo y puede volver a encolarse nuevamente. El evento que lo desbloquea se genera fuera del *scheduler*. El hilo queda a la espera para poder cambiar de estado cuando corresponda.

La red representa los estados y transiciones de cada hilo, por lo cual asignamos una red por cada hilo al momento de crearlo.

3.1.4. Validación del modelo

En la figura 3.2 se pueden ver los resultados obtenidos utilizando la herramienta PIPE de análisis y simulación de Redes de Petri. Se evalúan tres aspectos fundamentales:

1. Que sea acotada
2. Que sea segura
3. Que no tenga interbloqueo

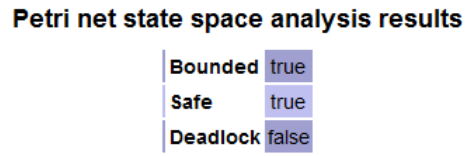

Figura 3.2: Análisis de la red propuesta.

El modelo propuesto es acotado, seguro y libre de interbloqueos.

3.1.5. Implementación del modelo

Para implementar el modelo en el código fuente, se procedió a:

1. Identificar la estructura *thread* dentro del código fuente. La misma se encuentra definida en el archivo de cabeceras `proc.h`.

2. Definir en `proc.h` los *macros* (directiva de pre compilación) que van a representar las plazas y transiciones de la red de Petri del *thread*.
3. Agregar a la estructura *thread* los siguientes campos:
   - `mark`: vector que tiene el marcado inicial, con tamaño PLACES_SIZE.
   - `sensitized_buffer`: vector que representa las transiciones sensibilizadas de su red asociada, con tamaño TRANSITIONS_SIZE.
4. Crear un archivo de cabeceras `sched_petri.h` que incluye a `proc.h` y que contiene la definición de las funciones necesarias para el funcionamiento de la red.
5. Crear un archivo fuente `sched_petri.c` que incluya a `sched_petri.h` para representar la red de Petri propuesta y su funcionamiento. Se declaró la matriz de incidencia (PLACES_SIZE * TRANSITIONS_SIZE) y el vector de marcado inicial (PLACES_SIZE), y a su vez se implementaron las funciones declaradas anteriormente:
   - `init_petri_thread`: esta función recibe un *thread* como parámetro y procede a inicializar su `mark` al marcado inicial.
   - `thread_get_sensitized`: recibe un *thread* como parámetro y analiza todas sus transiciones para actualizar su `sensitized_buffer` (función no utilizada).
   - `thread_petri_fire`: recibe un *thread* y una transición como parámetros y la dispara haciendo uso de la matriz de incidencia, actualizando finalmente su marcado.
   - `thread_search_and_fire`: recibe un *thread* como parámetro y calcula sus transiciones sensibilizadas para proceder a dispararlas (función no utilizada).
6. Identificar donde se inicializa y asigna memoria a la estructura *thread* dentro del código fuente. Esto se realiza en `kern_thread.c` en la función `thread_alloc`.
7. Llamar a `init_petri_net` en `thread_alloc` para inicializar y asignar memoria para la red.

3.1.6. Análisis de resultados

Al inicio de la iteración se planteó elaborar un modelo para representar los estados de un hilo. La validación del modelo y su posterior implementación nos lleva a concluir que el objetivo definido para esta iteración se cumplió. Al finalizar la iteración se encontraron una serie de instrucciones en `proc.h` con estados de hilos e instrucciones para pasar de un estado al otro. También se identificaron las partes del código fuente en donde ocurren dichas transiciones.

3.1.7. Próximos pasos

Con un modelo ya desarrollado para los hilos, el próximo paso consiste en proponer un modelo inicial para representar los estados y eventos de los recursos del sistema.

3.2. Segunda iteración: Red de recursos dinámica

3.2.1. Objetivos a alcanzar

En esta iteración se buscará proponer un modelo inicial de red de Petri para representar el reparto de las CPU para cada uno de los hilos.

3.2.2. Modelo propuesto

Para llevar a cabo la conexión entre las redes de los hilos y la red de reparto de las CPU se utilizará el concepto de redes jerárquicas. Existirá una red padre que tenga todos los recursos y que sensibilice o inhiba algunas transiciones de cada una de las redes locales (figura 3.3).

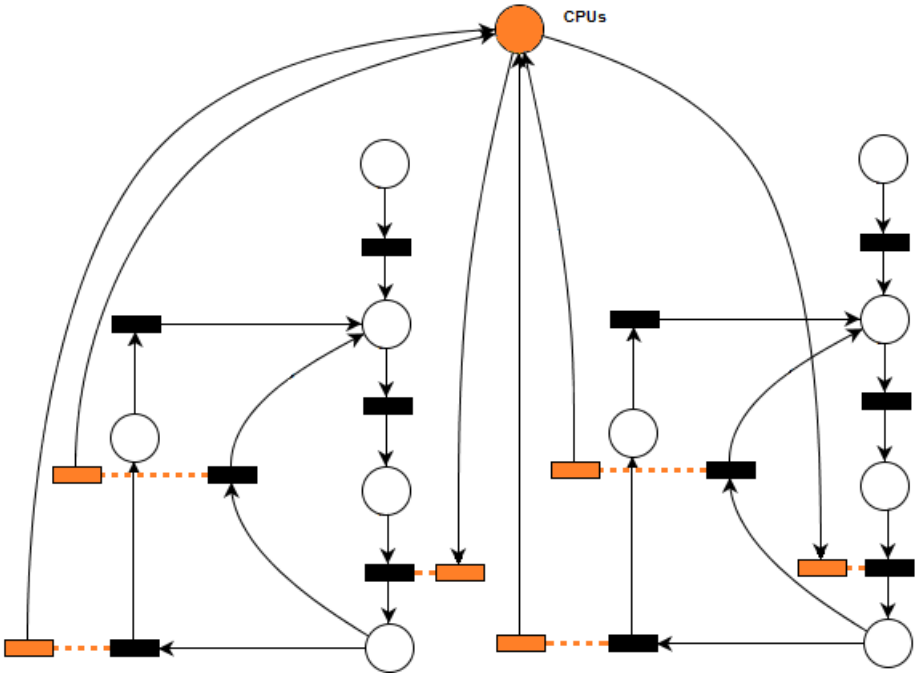

Figura 3.3: Modelo propuesto de la red.

3.2.3. Análisis del modelo

En un primer momento se pensó en llevar a cabo una red de recursos dinámica que agregara arcos representativos de las CPU a medida que se crean nuevos hilos.

3.2.4. Validación del modelo

En la figura 3.4 se pueden ver los resultados obtenidos utilizando la herramienta PIPE. Se evalúan tres aspectos fundamentales:

1. Que sea acotada
2. Que sea segura
3. Que no tenga interbloqueo

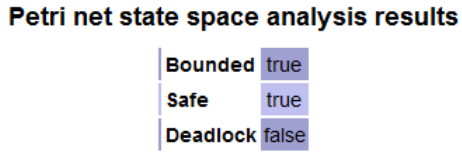

Figura 3.4: Análisis de la red propuesta.

El modelo propuesto es acotado, seguro y libre de interbloqueos.

3.2.5. Implementación del modelo

Al momento de implementar el modelo se determinó que no resulta muy eficiente trabajar con redes dinámicas por el hecho de tener que asignar posiciones de memoria dinámicamente para la misma cada vez que se crea un nuevo hilo, por lo tanto el mismo fue descartado.

3.2.6. Análisis de resultados

El objetivo inicial no pudo cumplirse, por lo que se debe pensar en otra alternativa para modelar el reparto de las CPU utilizando una red estática.

3.2.7. Próximos pasos

Descartada la red anterior, se determinó que la red a diseñar para representar el reparto de las CPU debe ser capaz de:

- Repartir la ejecución de hilos equitativamente entre las CPU disponibles. Para ello debe llevar un control de los hilos asignados a las diferentes colas de CPU.
- Favorecer con más asignaciones a aquellas CPU cuyos hilos tengan un tiempo de ejecución menor, es decir que aquellas CPU van a recibir mayor cantidad de hilos en sus colas que el resto.
- No permitir encolar más hilos a aquellas CPU que tengan más hilos pendientes de ejecución que los demás, es decir cuyas colas tengan más hilos que el resto.

3.3. Tercera iteración: Sistema de turnos de CPU

3.3.1. Objetivos a alcanzar

En esta iteración se buscará proponer una alternativa al modelo de red de Petri anterior con las especificaciones definidas, partiendo de un diseño simple sobre el cual se puedan ir incorporando nuevas funcionalidades.

3.3.2. Modelo propuesto

El modelo propuesto para esta iteración se presenta en la figura 3.5.

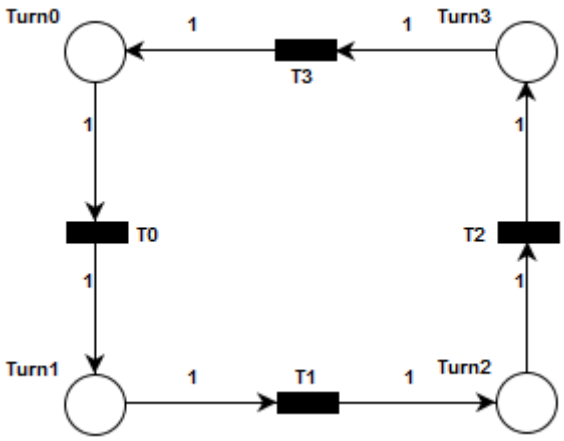

Figura 3.5: Modelo propuesto: turnos para 4 CPU.

3.3.3. Análisis del modelo

Para comenzar, se pensó la red como el ciclo de un *token* que circula por tantas plazas como CPU disponga el sistema. Esto nos da la posibilidad de ir tomando decisiones respecto a cada CPU y de poder movernos de una CPU a la siguiente en forma cíclica, pasando por todas las CPU antes de volver a la misma. Esto permite que la asignación de hilos dentro de cada CPU sea justa y equitativa.

El modelo anterior representa los 4 turnos que se necesitan para modelar una computadora dotada de 4 CPU.

3.3.4. Validación del modelo

En la figura 3.6 se pueden ver los resultados obtenidos utilizando la herramienta PIPE. Se evalúan los mismos aspectos fundamentales que fueron propuestos anteriormente.

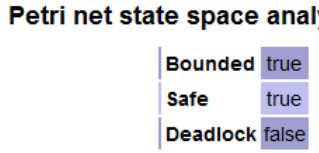

Figura 3.6: Análisis de la red propuesta.

El modelo propuesto es acotado, seguro y libre de interbloqueos.

3.3.5. Implementación del modelo

Para implementar el modelo en el código fuente, se procedió a:

1. Definir en `sched_petri.h` las instrucciones que van a representar las plazas y transiciones de la red de Petri de recursos propuesta.

2. Definir en `sched_petri.h` la cantidad de CPU con las que va a contar el sistema. A modo representativo, se trabajará siempre con 4 CPU.

3. Definir en `sched_petri.h` la estructura `petri_cpu_resource_net` con los siguientes campos:

   - `mark`: vector que contiene el marcado inicial de la red de recursos, con tamaño CPU_NUMBER_PLACES.
   - `sensitized_buffer`: vector que representa las transiciones sensibilizadas, con tamaño CPU_NUMBER_TRANSITIONS.
   - `cpu_owner_list`: vector con los identificadores de los *threads* que están utilizando cada CPU, con tamaño CPU_NUMBER (campo no utilizado).
   - `incidence_matrix`: matriz de tamaño CPU_NUMBER_PLACES * CPU_NUMBER_TRANSITIONS.
   - `inhibition_matrix`: matriz de tamaño CPU_NUMBER_PLACES * CPU_NUMBER_TRANSITIONS.
   - `is_automatic_transition`: vector con las transiciones automáticas de la red, de tamaño CPU_NUMBER_TRANSITIONS.

4. Agregar en `sched_petri.h` la definición de las funciones necesarias para el funcionamiento de la red.

5. Crear un archivo fuente `petri_global_net.c` que incluya `sched_petri.h` para representar la red de Petri de recursos propuesta y su funcionamiento. Se declaró la matriz de incidencia base y la matriz de inhibición base, y a su vez se implementaron las funciones declaradas anteriormente:

   - `init_resource_net`: función que inicializa `mark`, `incidence_matrix`, `ihibition_matrix` e `is_automatic_transition`.
   - `resource_get_sensitized`: analiza todas sus transiciones para actualizar su `sensitized_buffer` (función no utilizada).

- `resource_fire_net`: recibe un *thread* y una transición como parámetros, dispara la transición en la red global haciendo uso de la matriz de incidencia y dispara las transiciones automáticas que se habiliten.

6. En la función `sched_setup` del archivo `sched_4bsd.c` identificar donde se inicializa el *scheduler* para inicializar su red de recursos.

7. Llamar a `init_resource_net` en `sched_setup` para inicializar y asignar espacio de memoria para la red de recursos.

3.3.6. Análisis de resultados

El planteo de una red simple de recursos estática, definida al inicializar el *scheduler* del sistema con la cantidad de CPU que se deseen, permitió representar de manera simplificada el funcionamiento buscado para el reparto de las CPU. El objetivo definido para esta iteración se cumplió.

3.3.7. Próximos pasos

Sobre la red propuesta en esta última iteración se irán incorporando funcionalidades que brinda el *scheduler 4BSD*. El próximo paso nos llevará a incorporar la representación del encolado de los hilos en las diferentes colas de las CPU de forma equitativa.

3.4. Cuarta iteración: Encolado de hilos

3.4.1. Objetivos a alcanzar

En esta iteración se buscará modificar el modelo propuesto anteriormente para que el mismo contemple el encolado equitativo de hilos en las diferentes CPU que conforman el sistema.

3.4.2. Modelo propuesto

Partiendo del modelo anterior, se proponen la red con algunas modificaciones (figura 3.7).

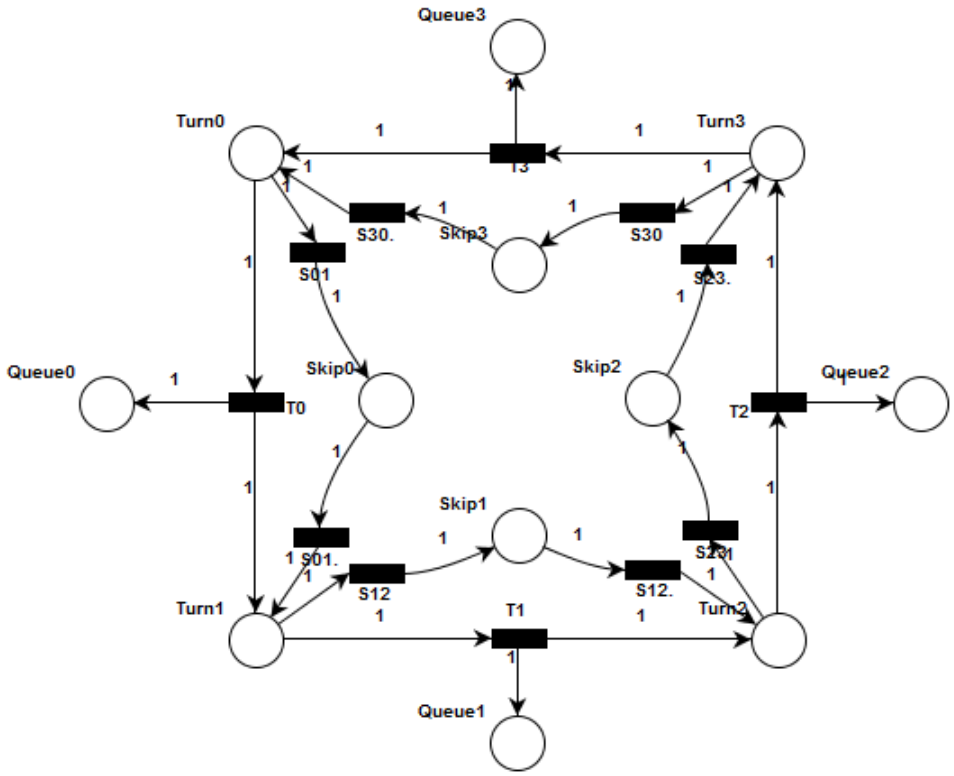

Figura 3.7: Modelo propuesto para el encolado de hilos.

3.4.3. Análisis del modelo

Resulta necesario representar la cantidad de elementos encolados por cada CPU. Cada cola va a ser representada por una plaza que, cuando el *token* circulante pase por la correspondiente CPU, aumentará en uno su cantidad de *tokens*. De esta forma se lograría una repartición equitativa de *tokens* (*threads*) entre las colas de las CPU.

Sin embargo, puede darse el caso de que la CPU no esté en condiciones de encolar debido a que la misma está desencolando más lento que el resto. Por este motivo se ve la necesidad de ofrecer caminos alternativos entre turnos que no encolen hilos en una CPU determinada.

Resumiendo, para pasar de turno pueden darse dos casos:

- Que la CPU esté en condiciones de encolar: se disparará una transición que pase el turno y agregue un *token* a la cola de esa CPU.
- Que la CPU no esté en condiciones de encolar: se disparará una transición que pasará el turno al siguiente sin agregar *tokens* en su cola.

3.4.4. Validación del modelo

En la figura 3.8 se pueden ver los resultados obtenidos utilizando la herramienta PIPE. Se evalúan los mismos aspectos fundamentales que fueron propuestos anteriormente.

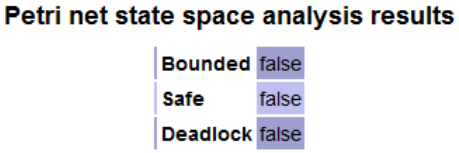

Figura 3.8: Análisis de la red propuesta.

El modelo propuesto es no acotado (debido a las plazas de encolado), inseguro y libre de interbloqueos. El modelo se convierte en acotado si se limitan las plazas de encolado a una cantidad máxima (lo cuál corresponde con el caso práctico).

3.4.5. Implementación del modelo

Para implementar el modelo en el código fuente, se procedió a:

1. Definir en `sched_petri.h` los macros de las nuevas plazas y transiciones incorporadas al modelo. Inicializarlas en `init_resource_net`.
2. Identificar donde se realiza el encolado de *threads* en el código fuente. Esto se realiza en `sched_4bsd.c` en la función `sched_add`.
3. Llamar a `resource_fire_net` en `sched_add` para contemplar en la red el encolado de los *threads* que ingresan al *scheduler* en la CPU que le corresponda.

3.4.6. Análisis de resultados

El objetivo propuesto de modelar el encolado de los *threads* pudo ser cumplido, pero sin embargo el modelo carece de un mecanismo que le permita decidir

qué camino tomar a la hora de pasar el turno. Esta información debe ser provista por la misma red, por lo que para lograr un modelo de encolado más determinista se deberá proponer alguna modificación.

3.4.7. Próximos pasos

Como próximo paso se deberá proponer un nuevo modelo que permita llevar el control de la cantidad de *tokens* asignados por cada cola para saber cuáles CPU son las que han recibido menor cantidad de hilos para ejecutar. Teniendo esta información, la decisión sobre qué camino tomar al pasar el turno va a ser provista por la misma red.

3.5. Quinta iteración: Control del encolado

3.5.1. Objetivos a alcanzar

Partiendo del modelo anterior se buscará introducir un sistema de autocontrol para las asignaciones de los hilos que permita determinar cuáles son las CPU que tienen la mayor cantidad de hilos en sus colas y de esta forma decidir si la CPU actual está en condiciones de encolar.

3.5.2. Modelo propuesto

Se propone la modificación presentada en la figura 3.9.

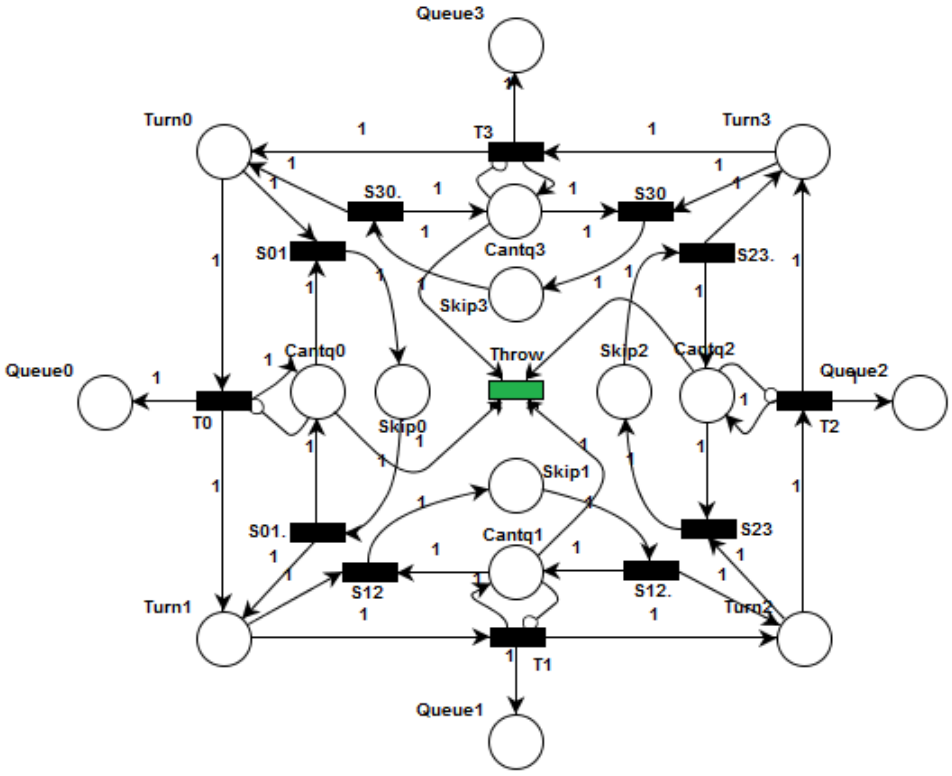

Figura 3.9: Modelo propuesto con el sistema de autocontrol de encolado.

3.5.3. Análisis del modelo

La solución propuesta consiste en incorporar una plaza por cada CPU que representará la posibilidad que tiene la misma de encolar o no. Esta plaza va a inhibir el camino que involucra el encolado de la CPU, es decir que funcionará como un cerrojo entre turnos que indique cuándo esté en condiciones de encolar de acuerdo a las especificaciones iniciales.

Por otra parte, se incorporó una transición de descarte global (es decir que habrá solo una sin importar la cantidad de CPU, representada de color verde) que se ejecutará automáticamente cada vez que todas las habilitaciones de las CPU tengan al menos un *token*. El objetivo de esta transición consiste en habilitar las colas con la menor cantidad de hilos que estaban inhibidas una vez que todas se han emparejado.

3.5.4. Validación del modelo

En la figura 3.10 se pueden ver los resultados obtenidos utilizando la herramienta PIPE.

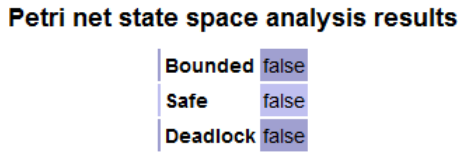

Figura 3.10: Análisis de la red propuesta.

El modelo propuesto es no acotado (debido a las plazas de encolado y las plazas de habilitación), es inseguro y libre de interbloqueos.

3.5.5. Implementación del modelo

Para implementar el modelo en el código fuente, se procedió a:

1. Definir en `sched_petri.h` los macros de las nuevas plazas y transiciones incorporadas al modelo. Inicializarlas en `init_resource_net`.

2. Definir como automática la transición de descarte al momento de inicializar el vector de transiciones automáticas en `sched_petri.h`. De esta forma cada vez que la misma quede sensibilizada va a ser disparada de inmediato, manteniendo siempre así en el modelo al menos una CPU que pueda encolar.

3.5.6. Análisis de resultados

El modelo propuesto hasta el momento tiene la información necesaria para determinar cuándo un hilo debe ser encolado en una determinada CPU, cumpliendo el objetivo inicial planteado. Sin embargo, al momento de analizarlo se descubrió que el mismo carece de información relacionada a la velocidad de ejecución de cada CPU. Puede darse el caso que en una CPU los hilos utilicen todo el tiempo asignado, mientras que en otra puede que existan muchos bloqueos y cambios de contexto. Esto puede generar desencolados más rápidos en algunas CPU, por lo que la velocidad de ejecución de cada CPU también debe ser tenida en cuenta para decidir sobre el encolado.

3.5.7. Próximos pasos

En primer lugar resulta necesario incorporar al modelo:

- La plaza que represente a la CPU.
- La transición de desencolado entre la CPU y su cola.
- La plaza donde irá a parar el hilo que adquiere la CPU y está en condiciones de ser ejecutado.
- La transición que inicia la ejecución del hilo.
- La transición de retorno de la CPU para cuando finaliza la ejecución.

Partiendo de las incorporaciones anteriores, se puede comenzar a pensar en un modelo que controle el encolado de cada CPU teniendo en cuenta también la velocidad con la que ejecuta cada uno.

Por otra parte, también se pueden empezar a definir las transiciones jerárquicas que van a existir entre la red de recursos y la red de cada *thread*.

3.6. Sexta iteración: Selección de hilo y ejecución

3.6.1. Objetivos a alcanzar

Como objetivo de esta iteración se buscará expandir el modelo de la red de recursos para representar tanto el encolado como el desencolado y la ejecución de los hilos, pudiendo de esta forma introducir en la decisión del encolado la velocidad a la que opera cada CPU, penalizando la posibilidad de encolar a aquellas CPU que estén despachando hilos a velocidades menores que el resto. Una vez que se tenga el modelo completo se definirán las conexiones necesarias entre la red de recursos y la red de los *threads*.

3.6.2. Modelo propuesto

Se presenta el modelo para el encolado y desencolado de hilos. Para mantener el gráfico entendible, de ahora en adelante se mostrará el modelo de una sola CPU teniendo en cuenta la simetría que mantiene el modelo a la cantidad de CPU disponibles. Las transiciones en naranja van a representar al resto de las CPU y se las incorporan al modelo para mostrar la penalización detallada (figura 3.11).

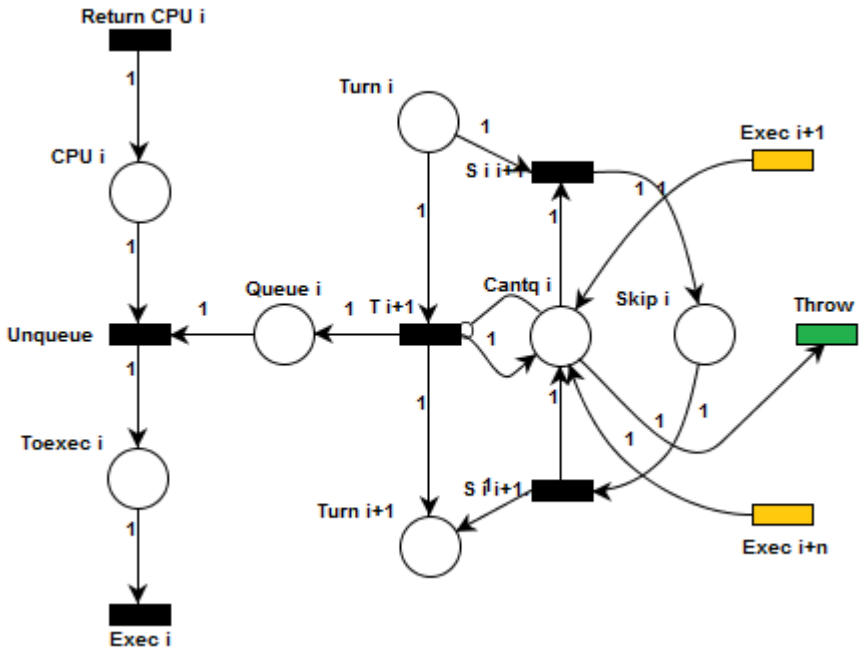

Figura 3.11: Modelo propuesto con encolado, desencolado y ejecución.

3.6.3. Análisis del modelo

Al momento de elaborar el modelo se analizaron dos alternativas para representar la influencia de la velocidad de ejecución en el encolado de la CPU:

1. Cada ejecución de un hilo ubicado en la cola de una determinada CPU va a premiar el encolado de la misma disminuyendo en uno la cantidad de *tokens* presentes en su plaza de habilitación para que quede habilitada a tomar hilos más rápido que el resto.

2. Cada ejecución de un hilo ubicado en la cola de una determinada CPU va a penalizar el encolado del resto aumentando en uno la cantidad de *tokens* en sus plazas de habilitación para que bloqueen sus encolados por trabajar a una menor velocidad.

Inicialmente se consideró que la primera solución podría resultar problemática para el caso que se generen grandes cantidades de hilos al mismo tiempo ya que se premiaría solo a las CPU que ejecuten rápido pero no se penalizaría a los que tienen velocidades de ejecución menor que el resto, lo que puede originar un cuello de botella. Por lo tanto, se optó por introducir al modelo la segunda alternativa ya que la misma contempla tanto a las CPU que ejecutan rápido como a las que ejecutan a menor velocidad.

A partir del último modelo detallado se puede llevar a cabo la conexión en términos de jerarquía que tendría esta red de recursos con cada una de las redes de los hilos existentes en el sistema. Para ello, se determinaron las siguientes transiciones jerárquicas:

- Transición CAN_RUN ⇒ RUNQ del hilo es la que corresponde a la transición de encolado de la red de recursos.
- Transición RUNQ ⇒ RUNNING del hilo es la que corresponde a la transición de desencolado de la red de recursos.
- Transición RUNNING ⇒ RUNQ del hilo es la que corresponde a la transición de retorno de CPU de la red de recursos.
- Transición RUNNING ⇒ INHIBITED del hilo es la que corresponde a la transición de retorno de la CPU de la red de recursos.

Como se puede observar, la mayoría de las transiciones de la red de Petri de un hilo pueden vincularse a la red de recursos. Sin embargo, es necesario realizar algunas aclaraciones:

1. Las últimas dos transiciones detalladas anteriormente generan un retorno de la CPU en la red de recursos. Es necesario categorizar el retorno en:

   - Voluntario: cuando la interrupción de la ejecución se debe a que el hilo no puede continuar porque espera por un evento o un recurso.
   - Involuntario: cuando la interrupción se produce porque el hilo consumió su tiempo asignado de CPU o bien finalizó su tarea.

   Para distinguir entre ambos retornos se incluirán dos transiciones diferentes para representarlas en el modelo.

2. La transición INACTIVE ⇒ CAN_RUN del hilo no se es tarea del *scheduler*, por lo que la misma no tiene jeárquica en la red de recursos.

3. La transición INHIBITED ⇒ CAN_RUN del hilo tampoco depende del *scheduler* y no tiene jerarquía con la red de recursos, sino que la misma depende de otras partes del sistema operativo que se encargan de generar los eventos o liberar los recursos que necesita el hilo. Por este motivo, esta transición solo va a monitorear cuando se produzca esto último para disparar en ese momento esta transición previo a disparar la de encolado.

3.6.4. Validación del modelo

Al analizar la red propuesta, completa con 4 CPU con la herramienta PIPE, se obtuvieron los resultados indicados en la figura 3.12.

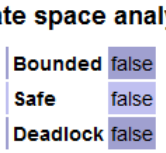

Figura 3.12: Análisis de la red propuesta.

El modelo propuesto es no acotado (debido a las plazas de encolado y las plazas de habilitación), es inseguro y libre de interbloqueos.

3.6.5. Implementación del modelo

Para implementar el modelo en el código fuente, se procedió a:

1. Definir en `sched_petri.h` los macros de las nuevas plazas y transiciones incorporadas al modelo. Inicializarlas en `init_resource_net`.

2. Agregar en `petri_global_net.c` dos vectores:

   - `Hierarchical_transitions`: vector con las transiciones jerárquicas de la red de recursos. Ordenadas de acuerdo al índice correspondiente con `hierarchical_corresponse`.
   - `Hierarchical_corresponse`: vector con las transiciones jerárquicas de la red del *thread*. Ordenadas de acuerdo al índice correspondiente con `hierarchical_transitions`.

3. Identificar donde se realiza el desencolado de *threads* en el código fuente. Esto se realiza en `sched_4bsd.c` en la función `sched_choose`.

4. Llamar a `resource_fire_net` en `sched_choose` para contemplar en la red el desencolado de los *threads* que van a pasar a ejecución en la CPU que le corresponda.

5. Identificar donde se manda a ejecutar a los *threads* en el código fuente. Esto se realiza en `sched_4bsd.c` en la función `sched_switch`.

6. Llamar a `resource_fire_net` en `sched_switch` para contemplar en la red los *threads* que pasan a ejecución en la CPU que le corresponda.

7. Identificar cuando finaliza la ejecución de los *threads* liberando la CPU en el código fuente. Esto se realiza en `sched_4bsd.c` en `sched_switch`.

8. Llamar a `resource_fire_net` en `sched_switch` para contemplar en la red los *threads* que finalizan su ejecución y retornan la CPU que corresponde.

9. Incorporar en `petri_global_net.c` en la función `resource_fire_net` el disparo de las transiciones jerárquicas de la red del *thread* para cuando una transición de la red de recursos que le corresponda se dispara.

3.6.6. Análisis de resultados

Como se planteó inicialmente, se incorporaron al modelo las plazas y transiciones necesarias para modelar el desencolado, la ejecución y la finalización de la ejecución de los hilos en las diferentes CPU. Además, pudo modelarse un sistema de transiciones jerárquicas que permite el disparo de las correspondientes del *thread* cada vez que se dispara una de la red de recursos.

Tal como se mencionó en el análisis del modelo, resulta necesario incorporar al modelo la nueva transición de retorno para distinguir entre cambios de contexto voluntario e involuntario. También resulta necesario utilizar algún mecanismo que permita identificar cuándo debe dispararse la transición del *thread* que lo saca de estado inhibido.

Continuando con el análisis, se llevó a cabo un análisis más minucioso del modelo planteando la siguiente cuestión:

- ¿Es necesario llevar un sistema de turnos por cada CPU?
- ¿No sería más eficiente que directamente se sensibilicen las transiciones de encolado cuyas colas estén habilitadas?

3.6.7. Próximos pasos

Se propone continuar llevando a cabo un análisis respecto a las cuestiones planteadas anteriormente.

3.7. Séptima iteración: Modelo sin turnos

3.7.1. Objetivos a alcanzar

En esta iteración se buscará proponer una simplificación al último modelo de la red de recursos teniendo en cuenta que el sistema de turnos propuesto desde un principio resulta innecesario e ineficiente.

3.7.2. Modelo propuesto

El modelo propuesto para esta iteración se presenta en la figura 3.13.

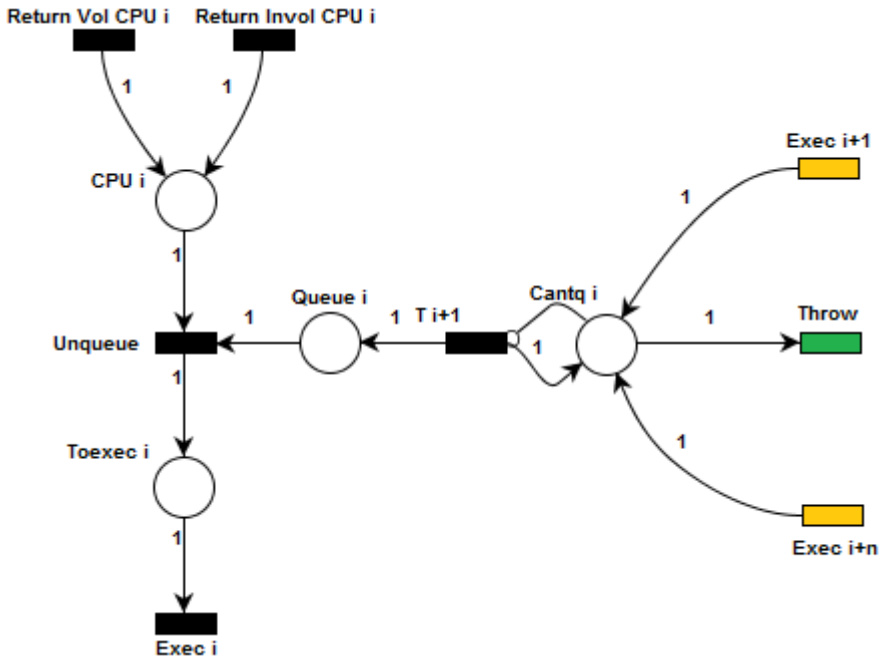

Figura 3.13: Modelo propuesto sin el sistema de turnos.

3.7.3. Análisis del modelo

Analizando la situación a nivel de *software*, al momento de tener que encolar un hilo en una cola de CPU el *scheduler* se encarga de llevar a cabo un monitoreo general de todas las transiciones de encolado de la red de recursos para determinar cuáles de estas se encuentran en el momento sensibilizadas y entre las mismas seleccionar una para disparar. Considerando este comportamiento, la simplificación propuesta en la iteración anterior se considera factible para el modelo.

El modelo presentado incluye los cambios propuestos:

- Notar que no se incluyen más ambos caminos, sino que todo pasa a ser una red más independiente para cada CPU.

- Se tuvo en cuenta ambos cambios de contexto (voluntario e involuntario) para este modelo.

3.7.4. Validación del modelo

Los resultados obtenidos al analizar la red propuesta ,utilizando 4 CPU, en la herramienta PIPE se encuentran en la figura 3.14.

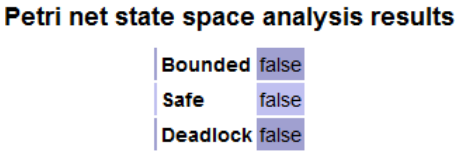

Figura 3.14: Análisis de la red propuesta.

El modelo propuesto es no acotado (debido a las plazas de encolado y las plazas de habilitación), es inseguro y libre de interbloqueos.

3.7.5. Implementación del modelo

Para implementar el modelo en el código fuente, se procedió a:

1. Definir en `sched_petri.h` los macros de las nuevas plazas y transiciones incorporadas al modelo. Inicializarlas en `init_resource_net`.

2. Dividir en `petri_global_net.c` las transiciones jerárquicas de cambio de contexto.

3. Agregar en `sched_petri.h` la definición de `resource_choose_cpu` e implementarla en `petri_global_net.c`:

   - `Resource_choose_cpu`: recibe un *thread* como parámetro, busca la transición de encolado de la CPU que esté disponible y la retorna.

4. Agregar en la función `sched_add` de `sched_4bsd.c` un llamado a la función `resource_choose_cpu` antes de realizar el encolado para tener la CPU correcta.

5. Añadir a la estructura *thread* en `proc.h` el campo `td_frominh` que va a indicar cuando un *thread* acaba de salir de estado inhibido. Esto va a ser cuando el mismo pase por un cambio de contexto voluntario.

6. Agregar en `sched_petri.h` la definición de `resource_expulse_thread` e implementarla en `petri_global_net.c`:

   - `Resource_expulse_thread`: recibe un *thread* como parámetro y las *flags* de cambio de contexto. Según el tipo de cambio de contexto actualiza el valor de `td_frominh` y dispara la transición de retorno de CPU correspondiente en la red de recursos y por ende su jerárquica.

7. Llamar a `resource_expulse_thread` en lugar de `resource_fire_net` en `sched_switch` para contemplar en la red los *threads* que finalizan su ejecución.

8. Disparar la transición del *thread* que lo saca del estado de inhibido en la función `sched_add` de `sched_4bsd.c` antes de encolarlo cuando el valor de `td_frominh` es de 1.

3.7.6. Análisis de resultados

Los objetivos planteados al inicio de la iteración pudieron ser cumplidos, por lo que se obtuvo una correcta simplificación en el modelo gracias a su comportamiento a nivel de *software*.

3.7.7. Próximos pasos

A partir de este punto se llevará a cabo un análisis más profundo del código del *scheduler* para incorporar todas las transiciones y plazas necesarias para representar el completo funcionamiento del mismo.

Como paso a seguir, se plantea analizar la afinidad que tienen los hilos respecto a las CPU donde corren. Un *scheduler* debe ser capaz de proveer un mecanismo de afinidad para los hilos, el cual a nivel de *hardware* es necesario ya que generalmente resulta más efectivo que el hilo vuelva a correr en la última CPU en la que se ejecutó.

3.8. Octava iteración: Afinidad de hilos

3.8.1. Objetivos a alcanzar

Partiendo del supuesto de que los hilos pueden tener cierta afinidad con alguna CPU o grupo de CPU, se buscará modelar el caso en el que ninguna CPU afín a un hilo que está por ser encolado se encuentre disponible.

3.8.2. Modelo propuesto

Para contemplar la afinidad de los hilos con las CPU, necesaria para tomar decisiones al momento del encolado, se debe incorporar al último modelo una nueva cola global para todas las CPU. El modelo propuesto se presenta en la figura 3.15.

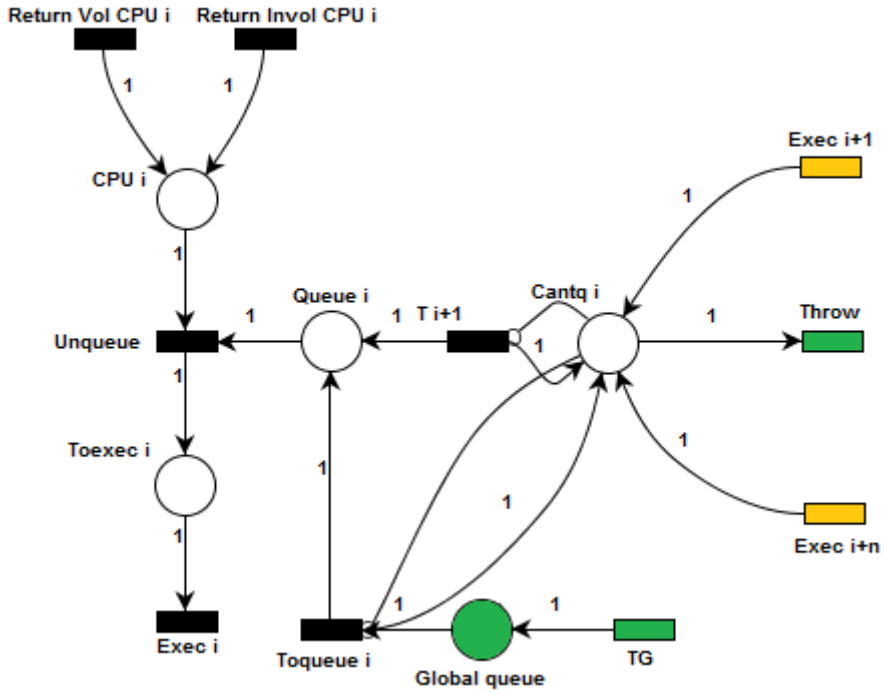

Figura 3.15: Modelo propuesto con la cola global.

3.8.3. Análisis del modelo

Si bien el modelo da una posibilidad de aplicar afinidad mediante *software*, el mismo no contempla el caso donde ninguna de las CPU marcadas con afinidad para un determinado hilo esté disponible. Para este fin se decide agregar, como está presente en el código, una cola global en la cual se encolarán los hilos cuyas CPU afines se encuentren inhabilitadas. Luego, mediante transiciones de traspaso, son reasignados a las colas que les correspondan.

En el modelo propuesto se representa la cola global del sistema, la transición de encolado global (ambas en verde para representar que son únicas) y la transición de traspaso de cola. Esta última transición va a ser una para cada CPU del sistema y, al igual que la transición de encolado normal, va a habilitarse cuando la plaza de habilitación lo permita para esa CPU y va a castigar el encolado de la misma cuando se ejecute.

3.8.4. Validación del modelo

Al analizar la red propuesta, completa con 4 CPU, utilizando la herramienta de análisis y simulación PIPE se obtienen los resultados presentados en la figura 3.16.

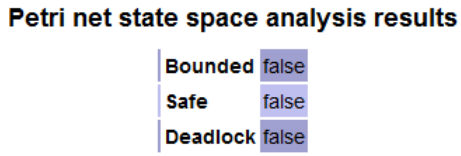

Figura 3.16: Análisis de la red propuesta.

El modelo propuesto es no acotado (debido a las plazas de encolado de cada CPU, encolado global y las plazas de habilitación), es inseguro y libre de interbloqueos.

3.8.5. Implementación del modelo

Para implementar el modelo en el código fuente, se procedió a:

1. Definir en `sched_petri.h` los macros de las nuevas plazas y transiciones incorporadas al modelo. Inicializarlas en `init_resource_net`.

2. Añadir en `petri_global_net.c` la transición jerárquica del *thread* a la transición de encolado global.

3. Analizar en la función `resource_choose_cpu` en `petri_global_net.c` las *flags* que indican las CPU afines a un *thread* y en base a esto tomar las decisión de retornar una cola de CPU o bien la cola global.

4. Tener en cuenta en la función `sched_add` de `sched_4bsd` la nueva transición de encolado global, disparándola cuando no se retorna ningúna CPU donde encolar.

3.8.6. Análisis de resultados

La incorporación de una nueva cola general para todas las CPU, donde van a parar los hilos que no tienen ninguna de sus CPU afines disponibles, permitió resolver el problema presentado inicialmente.

Por otro lado, analizando más detalladamente, se descubrió que al momento de inicializarse el sistema operativo las colas de las CPU se encuentran deshabilitadas hasta que todas las CPU son inicializadas, encontrándose sólo la CPU0 disponible desde el inicio. Esto podría presentar un problema para el encolado de hilos, pero se determinó que la cola global añadida al modelo podría también ser utilizada para encolar a todos los hilos hasta que las colas de las CPU se encuentren disponibles. De esta forma, va a ser necesario plantearse en las próximas iteraciones una forma de representar esta situación de inicio.

Continuando el análisis, también se presentó otra cuestión. A la hora de seleccionar el próximo hilo a ejecutar por una CPU, el *scheduler* pregunta tanto a la cola global como a la cola de la CPU cuál es el hilo con mayor prioridad para pasarlo directamente a ejecución. En base a este funcionamiento resulta innecesario realizar el cambio de cola propuesto anteriormente, en su lugar resulta más factible pasar el hilo presente en la cola global directamente a ejecución.

3.8.7. Próximos pasos

El análisis realizado anteriormente nos planteó 2 nuevos objetivos:

1. Representar el inicio del sistema con un comportamiento monoprocesador.
2. Replantear el cambio de cola propuesto, pasando directamente a ejecución a los hilos encolados en la cola global.

En la próxima iteración se optará por analizar la alternativa para el cambio de cola, y posteriormente en una futura iteración se abordará la cuestión de la inicialización de las CPU.

3.9. Novena iteración: Selección entre colas

3.9.1. Objetivos a alcanzar

En esta iteración se buscará una alternativa al cambio de cola propuesto anteriormente, para de esta forma adaptarse al comportamiento real del *scheduler*.

3.9.2. Modelo propuesto

Se propone la modificación ilustrada en la figura 3.17.

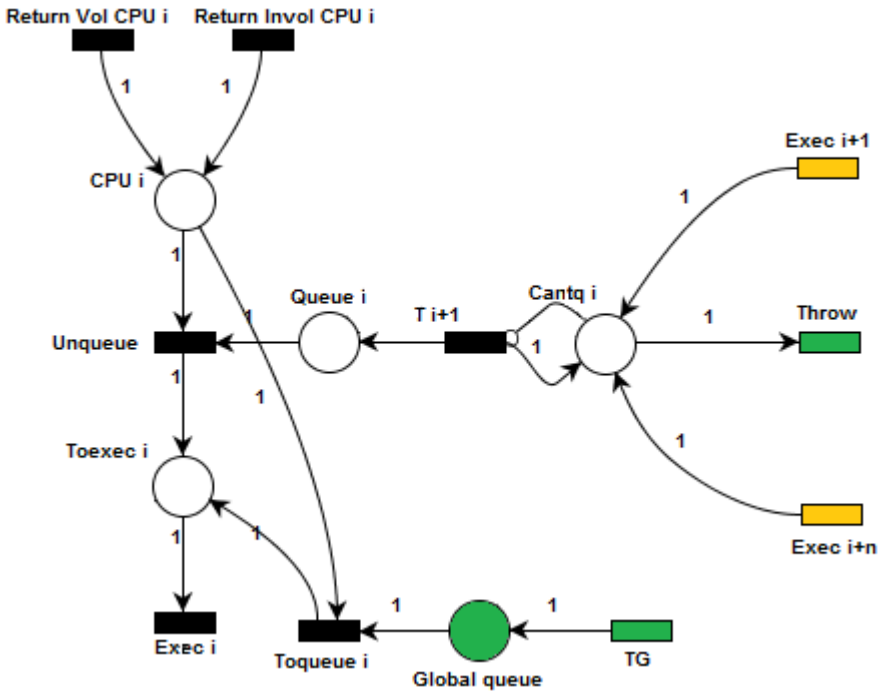

Figura 3.17: Modelo propuesto sin cambio de cola.

3.9.3. Análisis del modelo

Este cambio en el modelo puede realizarse eliminando los arcos entre ambas colas y creando nuevos arcos entre la CPU, la cola global y el estado de ejecución de cada CPU del sistema operativo. De esta manera, los hilos presentes en la cola global son tenidos en cuenta al momento en que una CPU elige el próximo hilo a ejecutar. Esta transición, que ahora también se corresponderá a una transición de desencolado, tendrá como jerárquica del hilo a RUNQ ⇒ RUNNING.

3.9.4. Validación del modelo

Al analizar la red propuesta en el software de análisis y simulación de Red de Petri PIPE, utilizando 4 CPU como modelo, se obtienen los resultados de la figura 3.18.

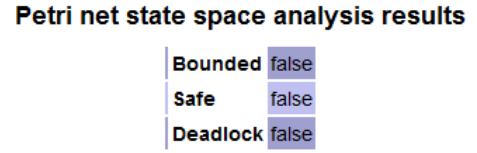

Figura 3.18: Análisis de la red propuesta.

El modelo propuesto es no acotado (debido a las plazas de encolado de cada CPU, encolado global y las plazas de habilitación), es inseguro y libre de interbloqueos.

3.9.5. Implementación del modelo

Para implementar el modelo en el código fuente, se procedió a:

1. Actualizar la función `init_resource_net` en `petri_global_net.c`.

2. Añadir en `petri_global_net.c` la transición jerárquica del *thread* a la transición de desencolado global.

3. Tener en cuenta en la función `sched_choose` de `sched_4bsd` la nueva transición de desencolado global, disparándola cuando el *thread* seleccionado se encuentra presente en la cola global.

3.9.6. Análisis de resultados

Tal como se planteó en la iteración anterior, resultaba necesario descartar el cambio de cola para los hilos presentes en la cola global y pasar a los mismos a ejecución cuándo la CPU los considere como el próximo en su lista de prioridad. El hecho de que un hilo se encuentre en la cola propia de la CPU o bien en la global no influye en la decisión sobre el próximo a ser ejecutado.

En esta iteración también se estudió con mayor profundidad el funcionamiento de los cambios de contexto en base al modelo obtenido. Cuando se produce un cambio de contexto, es decir que el hilo que está en ejecución libera la CPU y se lo cede al de mayor prioridad de la cola, se produce la siguiente secuencia:

1. El hilo en ejecución es expulsado, retornando la CPU y cambiando su estado RUNNING ⇒ CAN_RUN/INHIBITED.

2. El hilo saliente es agregado a una cola de ejecución en caso de que sea un cambio de contexto involuntario, cambiando de estado CAN_RUN ⇒ RUNQ.

3. Se elige el siguiente hilo a ejecutar y se lo manda a ejecución, asignándole la CPU y cambiando su estado RUNQ ⇒ RUNNING.

3.9.7. Próximos pasos

Resuelto el problema del cambio de cola, se considera necesario continuar el trabajo modelando el sistema monoprocesador de inicio del sistema.

Cuando inicializa el sistema operativo solamente la CPU0 se encuentra lista para ejecutar. Esta CPU es la que se encarga de inicializar el resto, por lo que inicialmente el sistema se encuentra en estado modo monoprocesador hasta que todas las CPU se encuentran en condiciones de ejecutar, momento donde el sistema pasa a modo multiprocesador (SMP). De esta forma, cuando el sistema se encuentra funcionando en monoprocesador se produce una acumulación de *tokens* en las plazas de habilitación del resto de las CPU, es decir que se está castigando por ejecutar "más lento" a las CPU que aún no se encuentran inicializadas.

Al problema de la imposibilidad del encolado de hilos en las colas de las CPU hasta que las mismas son inicializadas se suma el problema del castigo a las CPU no inicializadas. En la siguiente iteración se buscará solucionar estas cuestiones.

3.10. Décima iteración: Monoprocesador/ Multiprocesador

3.10.1. Objetivos a alcanzar

En esta iteración se buscará adaptar el modelo de tal forma que el mismo pueda representar tanto el comportamiento monoprocesador (NO SMP) como el comportamiento multiprocesador (SMP) del sistema operativo.

3.10.2. Modelo propuesto

Se propone un nuevo modelo de trabajo (figura 3.19).

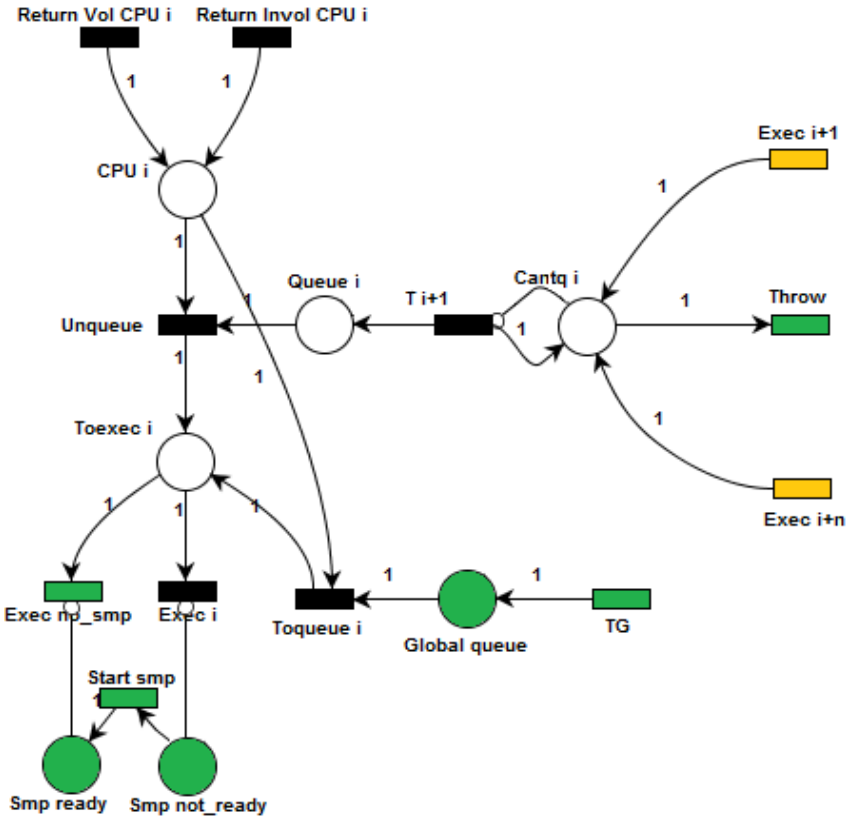

Figura 3.19: Modelo propuesto monoprocesador-multiprocesador.

3.10.3. Análisis del modelo

Como se planteó iteraciones atrás, el problema de la imposibilidad del encolado de hilos en las colas de las CPU fue resuelto utilizando la cola global para encolarlos. De esta manera, la CPU0 va a ejecutar los hilos presentes en esta cola hasta que las otras se habiliten.

Por otra parte, para solucionar la acumulación de *tokens* en las CPU que se encuentran inhabilitadas se propuso:

- Utilizar una plaza global para indicar que el sistema se encuentra en modo monoprocesador (modo NO SMP).
- Utilizar una plaza global para indicar que el sistema se encuentra en modo multiprocesador (modo SMP).
- Emplear una transición global entre ambas plazas, la cual se disparará cuando se inicialicen todas las CPU.
- Agregar una transición de ejecución global que va a ser utilizada únicamente por la CPU0 cuando el sistema se encuentre en modo monoprocesador y no va a castigar al resto de las CPU. Notar que esta transición no va a estar conectada al resto de las CPU del sistema operativo.

La transición global de ejecución es equivalente a la ya existente para cada CPU y tendrá como jerárquica la misma transición RUNQ ⇒ RUNNING del hilo.

3.10.4. Validación del modelo

Al analizar la red propuesta en la herramienta de análisis PIPE, utilizando un modelo de 4 CPU, se obtuvieron los resultados de la figura 3.20.

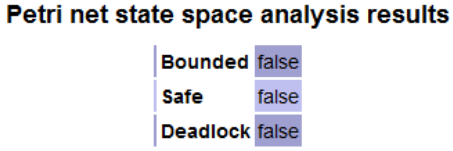

Figura 3.20: Análisis de la red propuesta.

El modelo propuesto es no acotado (debido a las plazas de encolado de cada CPU, encolado global y las plazas de habilitación), es inseguro y libre de interbloqueos.

3.10.5. Implementación del modelo

Para implementar el modelo en el código fuente, se procedió a:

1. Definir en `sched_petri.h` los macros de las nuevas plazas y transiciones incorporadas al modelo. Inicializarlas en `init_resource_net`.
2. Añadir en `petri_global_net.c` la transición jerárquica del *thread* a la transición de ejecución global.
3. Agregar en `petri_global_net.c` el campo `smp_set`, inicializado en 0, el cual va a permitir identificar el momento en que se inició el modo SMP.
4. Añadir en la función `resource_fire_net` en `petri_global_net.c` la comprobación del estado SMP del sistema representado por `smp_started`. Cuando `smp_started` se ponga en 1, se debe disparar la transición de traspaso a SMP en la red de recursos y poner en 1 a `smp_set`.

5. Agregar en `sched_petri.h` la definición de `resource_execute_thread` e implementarla en `petri_global_net.c`:
   - `Resource_execute_thread`: recibe un *thread* como parámetro y un número de CPU. Esta función ejecuta la transición de ejecución que corresponda, según el valor de `smp_set`.
6. Reemplazar en la función `sched_switch` en `sched_4bsd.c` el disparo de la transición de ejecución por un llamado a `resource_execute_thread`.
7. Modificar la función `resource_choose_cpu` en `petri_global_net.c` para que retorne siempre la transición de encolado global cuando el sistema se encuentre en NO SMP.

3.10.6. Análisis de resultados

La implementación y posterior simulación del nuevo modelo con el agregado de un sistema que permite el pasaje del modo monoprocesador a multiprocesador permitió resolver los problemas con los cuales se inició la actual iteración.

De esta manera, la cola global del sistema pasó a cumplir 2 funciones:

- Permitir el encolado de aquellos hilos cuyas CPU afines no se encuentran disponibles.
- Permitir encolar todos los hilos del sistema cuando el procesador se encuentra en modo monoprocesador y las colas de las CPU se encuentran inhabilitadas.

3.10.7. Próximos pasos

Una funcionalidad aún no modelada del *scheduler* es el cambio de colas que puede realizar para un hilo. Esto puede deberse a:

- Se asignó a cierto hilo afinidad con algúna CPU, por lo cual si el mismo se encuentra en la cola incorrecta debe ser expulsado para ser ubicado correctamente en la cola que le corresponda.
- El hilo sufrió un reajuste de prioridad y debe ser reubicado en una nueva cola.

Por otra parte, puede que el *scheduler* expulse un hilo que se encuentre en ejecución porque el mismo abandona el sistema operativo.

Se propone continuar en la siguiente iteración el modelado de estas últimas características.

3.11. Undécima iteración: Expulsión de hilos

3.11.1. Objetivos a alcanzar

Se buscará en esta iteración incorporar en el modelo los elementos necesarios que permitan representar la expulsión de un hilo de una determinada cola. También se buscará representar la expulsión de los hilos del sistema operativo cuando los mismos finalizan su ejecución.

3.11.2. Modelo propuesto

Esta última característica va a producir cambios en ambas redes.

Por lo tanto, se propone una nueva red para el hilo (figura 3.21).

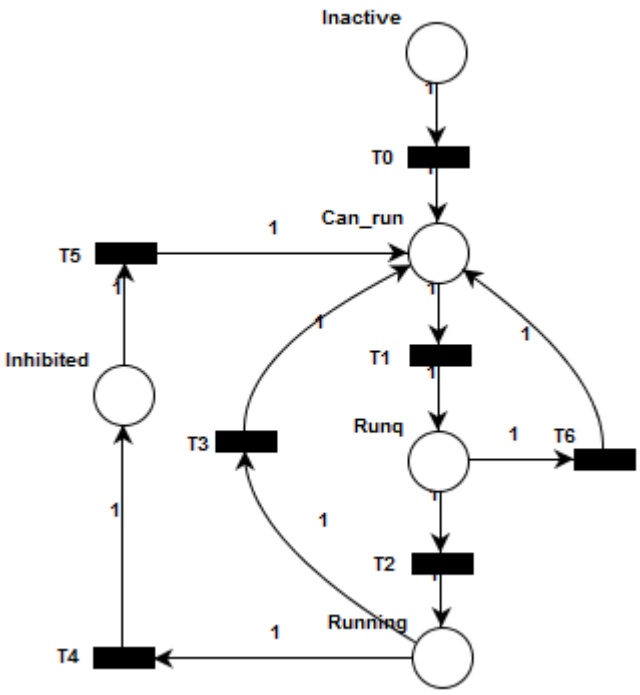

Figura 3.21: Modelo del hilo con cambio de cola.

Por su parte, la red de recursos también se verá influida para considerar este cambio (figura 3.22).

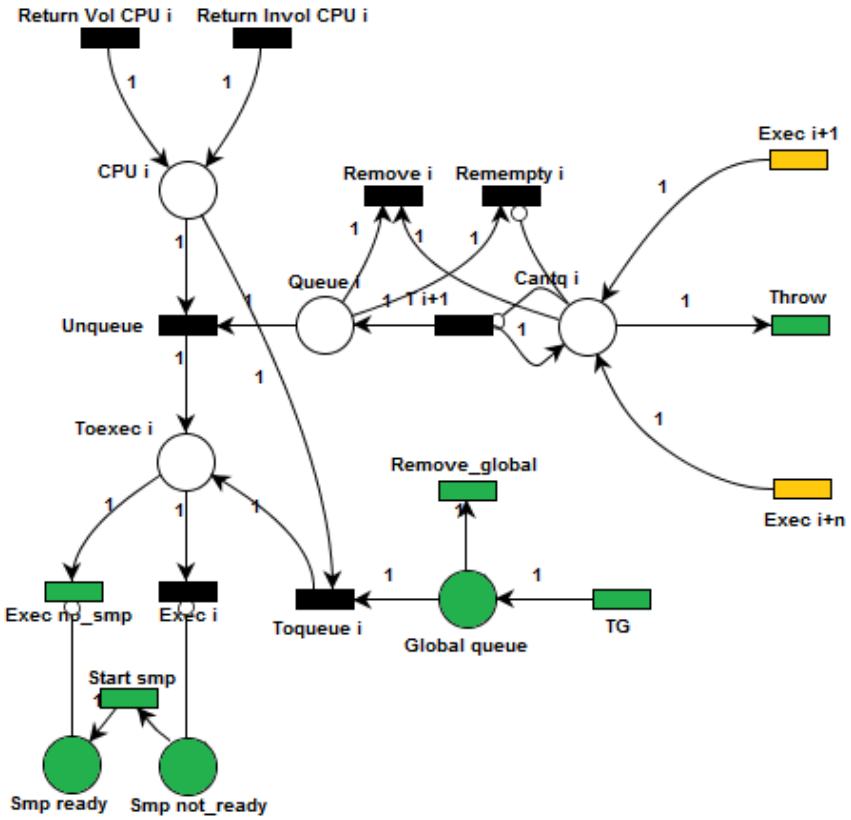

Figura 3.22: Modelo propuesto con expulsión de hilos.

3.11.3. Análisis del modelo

En el modelo del hilo se representa un nuevo cambio de estado para el hilo RUNQ ⇒ CAN_RUN. La transición T6 se ejecutará cada vez que un hilo deba ser expulsado de la cola en que se encuentra actualmente.

En cuanto al modelo de la red de recursos, para representar la expulsión de los hilos se van a incorporar dos transiciones de expulsión para cada CPU:

- La primera expulsará a un hilo de su cola cada vez que se ejecute y restará un *token* de habilitacón de la CPU, es decir que se premia a la CPU para que pueda encolar.
- La segunda expulsará a un hilo de su cola cada vez que se ejecute y la plaza de habilitación no tenga ningún *token*.

Por otra parte, se agregó también una transición global de expulsión para cuando el hilo expulsado se encuentre encolado en la cola global, la cual es única ya que no debe premiar el encolado de ningúna CPU.

Para realizar la nueva conexión entre ambas redes, se va a tener que tanto las transiciones de expulsión de cada CPU como la transición de expulsión global

van a tener como jerárquica la transición RUNQ ⇒ CAN_RUN del hilo.

Este último modelo contempla las siguientes funcionalidades del *scheduler*:

- Encolado de hilos, ya sea en cola global o de una CPU.
- Expulsión de hilos de una cola, para asignarlos a una correcta.
- Desencolado de hilos cuando se encuentra presente la CPU, ya sea desde la cola global o la de la CPU.
- Ejecución monoprocesador para la CPU0.
- Ejecución multiprocesador para todas las CPU.
- Retornos voluntarios e involuntarios de la CPU.
- Transiciones jerárquicas asignadas para la conexión con cada red de hilos.

3.11.4. Validación del modelo

Se analiza la red propuesta utilizando la herramienta de análisis y simulación PIPE. Para un modelo que utiliza 4 CPU los resultados son los presentados en la figura 3.23.

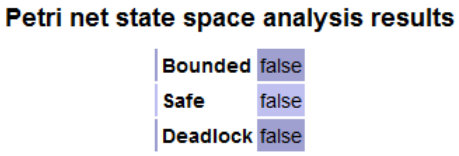

Figura 3.23: Análisis de la red propuesta.

El modelo propuesto es no acotado (debido a las plazas de encolado de cada CPU, encolado global y las plazas de habilitación), es inseguro y libre de interbloqueos.

3.11.5. Implementación del modelo

Para implementar el modelo en el código fuente, se procedió a:

1. Definir en `sched_petri.h` los macros de las nuevas plazas y transiciones incorporadas al modelo. Inicializarlas en `init_resource_net`.
2. Añadir en `petri_global_net.c` la transición jerárquica del *thread* a las transiciones de remoción de cada CPU y la global.
3. Agregar en `sched_petri.h` la definición de `resource_remove_thread` e implementarla en `petri_global_net.c`:
   - `Resource_remove_thread`: recibe un *thread* como parámetro y un número de CPU. Esta función ejecuta la transición de expulsión de la CPU que corresponda, según cuál sea la que se encuentre sensibilizada.

4. Identificar donde se expulsan los *threads* de su cola en el código fuente. Esto se realiza en `sched_4bsd.c` en la función `sched_rem`.

5. Llamar a `resource_fire_net` en `sched_rem` para expulsar a los *threads* que se encuentren actualmente en la cola global y deben ser reubicados, o bien llamar a `resource_remove_thread` para expulsar a los *threads* que se encuentren en una cola de CPU para reubicarlos.

6. Identificar donde son desechados los *threads* que finalizan su ejecución en el código fuente. Esto se realiza en `sched_4bsd.c` en la función `sched_throw`.

7. Llamar a `resource_expulse_thread` en `sched_throw` para expulsar a los *threads* que deben ser desechados. Posteriormente se debe seleccionar un nuevo *thread* de la cola y mandarlo a ejecución, para ello debe dispararse primero la transición de desencolado, al igual que se hace en `sched_choose`, y posteriormente llamar a `resource_execute_thread` con el *thread* elegido.

3.11.6. Análisis de resultados

Luego de probar este último modelo en el código y simularlo, el mismo resultó funcionar como se esperaba tanto para la red de los recursos como para la del *thread*, cumpliendo el objetivo propuesto para la iteración.

Sin embargo, se llevó a cabo un análisis más profundo y se pudieron resaltar algunas falencias en la red de recursos:

1. El método propuesto de castigar las CPU que ejecutan más lento resulta poco eficiente cuando hay pocos hilos ejecutándose en el sistema, ya que las colas están en su mayor tiempo vacías y no resulta necesario castigar a CPU inactivas.

2. Cuando un hilo pasa a ejecución se pierde control del estado en que se encuentra la red global. Además, para las transiciones de retorno de CPU no existe ningún mecanismo de control presente en la red global para controlar sus disparos.

Por otra parte, el modelo aún no cubre la funcionalidad del *scheduler* que permite a hilos que acaban de ser encolados pasar directamente a ejecución cuando su prioridad es mayor al que se encuentra actualmente ejecutando.

3.11.7. Próximos pasos

En el análisis de los resultados de la presente iteración se plantearon problemas tanto de falencias en la red de recursos como de funcionalidades que requieren ser incorporadas en el modelo de la misma.

Con el modelo de la red del *thread* completo y funcionando, se propone proceder en la siguiente iteración resolviendo los problemas identificados y que restan ver en la red de recursos.

3.12. Duodécima iteración: Hilos de alta prioridad

3.12.1. Objetivos a alcanzar

En esta iteración se buscará proponer una alternativa al modelo anterior para que el *scheduler* trabaje de manera eficiente en los momentos donde no está tan cargado. También se buscará incorporar un sistema de control interno en la red de recursos para poder saber cuándo una CPU se encuentra ejecutando un hilo.

Por otra parte, deberá implementarse una funcionalidad que contemple los hilos que son encolados y tienen mayor prioridad que el resto, ya que estos deben interrumpir al hilo actual y pasar directamente a ejecución.

3.12.2. Modelo propuesto

Partiendo en base a los objetivos planteados, se presenta el modelo propuesto en la figura 3.24.

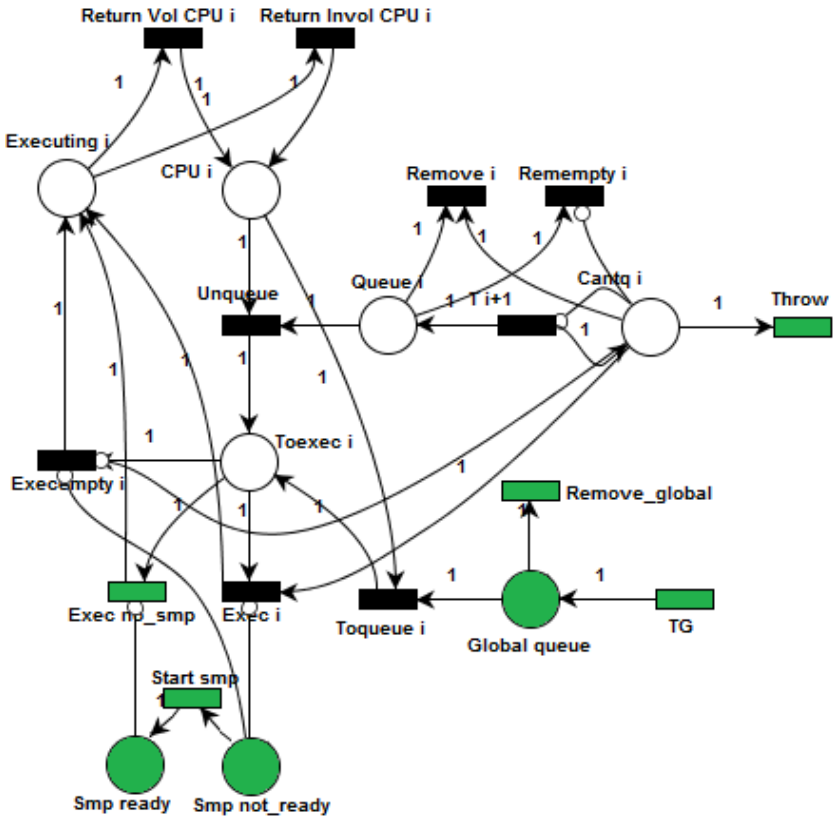

Figura 3.24: Modelo propuesto con dos transiciones de ejecución.

3.12.3. Análisis del modelo

Para disminuir el efecto de la ineficiencia del sistema para cuando se encuentra ejecutando pocos hilos, se pensó una alternativa teniendo en cuenta el modelo propuesto algunas iteraciones atrás de premiar a la propia CPU que ejecuta. Esta alternativa consiste en desglosar la transición de ejecución en dos transiciones (ambas jerárquicas a RUNQ ⇒ RUNNING), de manera que:

1. Una transición ejecutará el hilo y premiará a la CPU siempre y cuando la misma se encuentre inhabilitada a encolar (resta un *token* a la plaza de habilitación).

2. La otra transición solo se encargará de ejecutar el hilo sin premiar a la CPU siempre y cuando la misma se encuentre habilitada a encolar (inhibe con la plaza de habilitación).

En cuanto al sistema de control necesario para determinar cuándo una CPU se encuentra ejecutando, lo que se propone es incorporar al modelo una plaza por cada CPU que recibirá un *token* de control cuando un hilo pase a ejecución, mientras que dicho *token* abandonará la plaza al retornarse la CPU.

Respecto al pasaje directo a ejecución de los hilos encolados de alta prioridad no se contempla en el modelo, pero el mismo será tratado al momento de implementarlo en el código fuente.

3.12.4. Validación del modelo

Se analiza la red propuesta utilizando la herramienta PIPE, utilizando un modelo compuesto de 4 CPU. Los resultados se presentan en la figura 3.25.

Petri net state space analysis results

Bounded	false
Safe	false
Deadlock	false

Figura 3.25: Análisis de la red propuesta.

El modelo propuesto es no acotado (debido a las plazas de encolado de cada CPU, encolado global y las plazas de habilitación), es inseguro y libre de interbloqueos.

3.12.5. Implementación del modelo

Para implementar el modelo en el código fuente, se procedió a:

1. Definir en `sched_petri.h` los macros de las nuevas plazas y transiciones incorporadas al modelo. Inicializarlas en `init_resource_net`.

2. Añadir en `petri_global_net.c` la transición jerárquica del *thread* a ambas transiciones de ejecución.

3. Modificar la función `resource_execute_thread` en `petri_global_net.c` para que se realice la comprobación y se dispare la transición de ejecución que corresponda.

4. En la función `sched_add` de `sched_4bsd.c` realizar la comprobación de prioridad del hilo recientemente encolado haciendo uso para esto de la función `maybe_preempt`. Esta última función se encarga de:

   - Comparar prioridades del *thread* actual de ejecución y el *thread* que recibe como parámetro, es decir el que acaba de ser encolado.
   - En caso de tener mayor prioridad el *thread* que recibe, deberá pasárselo a `sched_switch` como *thread* que va a generar un cambio de contexto involuntario en el sistema. Caso contrario, la función finaliza.

5. Siguiendo el primer caso, en `sched_switch`, luego de expulsar y mandar a cola al *thread* que se encuentra en ejecución, se va a desencolar el *thread* que recibe (sin pasar por `sched_choose`) para luego mandarlo a ejecución.

3.12.6. Análisis de resultados

Incorporados los nuevos cambios, se pudo observar en la simulación un mejor comportamiento de la red de recursos.

Algunos posibles cambios que podrían realizarse para mejorar el control propio del modelo son:

- Inhibir el encolado de todas las CPU cuando se está en modo monoprocesador.
- Inhibir el desencolado de la cola global para todas las CPU excepto para la CPU0 cuando se está en estado monoprocesador).

Además, como simplificación del modelo se podría descartar la transición global de ejecución para el modo monoprocesador del sistema y reemplazar su funcionalidad con la transición de ejecución que no premia de la CPU0.

Por otra parte, al analizar el comportamiento de la CPU cuando no dispone de hilos en sus colas se descubrió que las mismas pasan a ejecutar hilos de baja prioridad que tienen asignados desde que son habilitadas. Esta funcionalidad también debe ser implementada.

3.12.7. Próximos pasos

Como paso a seguir en la próxima iteración se procederá a realizar los ajustes propuestos al modelo para que el mismo disponga de mayor autocontrol.

Finalmente, al momento de implementarlo se tendrán en cuenta también los hilos de baja prioridad que pasan a ocupar la CPU cuando la misma no dispone de hilos para ejecutar en ninguna de sus colas.

3.13. Décimo tercera iteración: Hilos de baja prioridad

3.13.1. Objetivos a alcanzar

En esta iteración se buscará aplicar los ajustes de control propuestos anteriormente. Además, se buscará implementar el funcionamiento de los hilos de baja prioridad que pasan a ocupar ls CPU cuando la misma no posee ningún hilo para ejecutar en su cola.

3.13.2. Modelo propuesto

Se propone el modelo completo de la red de recursos (figura 3.26).

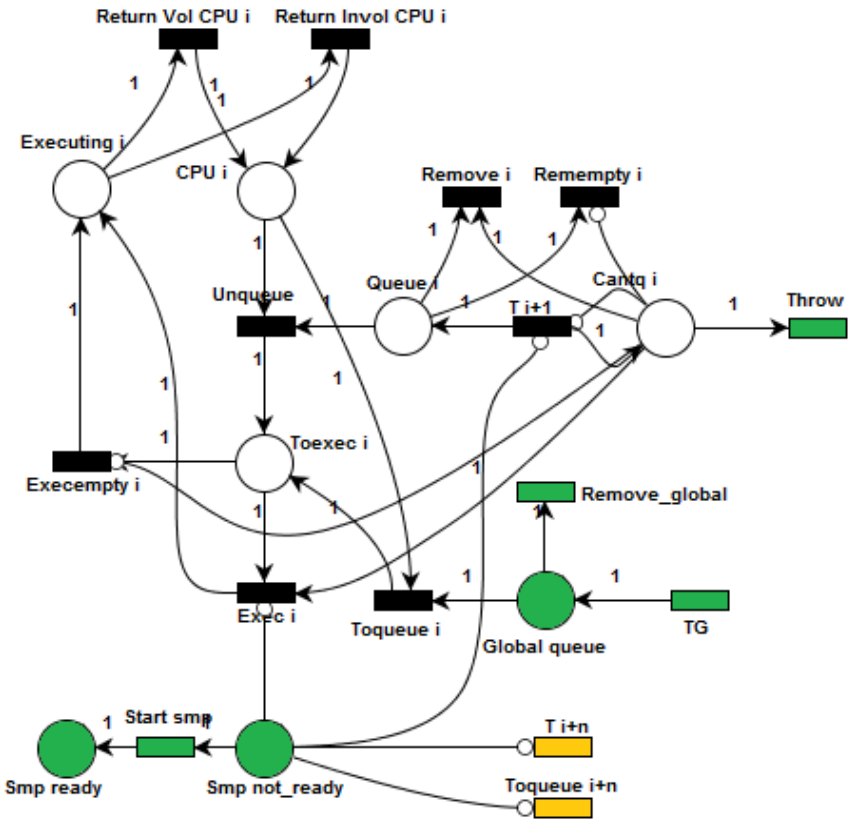

Figura 3.26: Modelo propuesto con los controles incorporados.

3.13.3. Análisis del modelo

Tal como se describió en la iteración anterior, se procedió a incorporar los arcos inhibidores necesarios para controlar el encolado de las distintas CPU. Se puede observar de color naranja las transiciones pertenecientes a otra CPU.

Por otra parte, se descartó la transición de ejecución para NO SMP y se reemplazó su funcionalidad por la transición de ejecución que no premia de la CPU0.

Habiendo arribado a este último modelo, se describirá el marcado inicial necesario para lograr el correcto funcionamiento en el *scheduler*, tanto para la red de hilos como para la red de recursos:

- Para la red del hilo, el marcado inicial del hilo debe iniciar con un *token* en su plaza CAN_RUN, ya que al momento de ingresar por primera vez al *scheduler* para ser encolado el mismo ya fue inicializado (INACTIVE ⇒ CAN_RUN ya ejecutado). Sin embargo existe un hilo particular, el hilo con id 100000, que se encarga de inicializar al resto, por lo que el mismo inicialmente va a encontrarse en estado RUNNING y se considera inicialmente ya corriendo en la CPU0.

- Para la red de recursos, la misma inicializará siempre con un *token* en la plaza que indica que el sistema se encuentra funcionando en modo monoprocesador. Además, se inicializarán las plazas que representan a las CPU con un *token*, excepto la de la CPU0 ya que la misma inicialmente se encuentra ejecutando al hilo inicial del sistema, por lo que para esta última deberá inicializarse con un *token* en la plaza que indica que se encuentra en ejecución.

En cuanto al funcionamiento de los hilos de baja prioridad, el mismo se verá contemplado en el análisis de implementación.

3.13.4. Validación del modelo

Se analiza la red anteriormente descripta, utilizando un modelo de 4 CPU, en el software de análisis PIPE. Los resultados se encuentran en la figura 3.27.

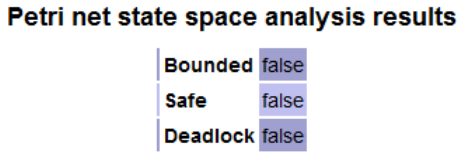

Figura 3.27: Análisis de la red propuesta.

El modelo propuesto es no acotado (debido a las plazas de encolado de cada CPU, encolado global y las plazas de habilitación), es inseguro y libre de interbloqueos.

Si se realiza el mismo análisis con las plazas mencionadas anteriormente limitando su capacidad a 1 se obtiene los resultados descriptos en la figura 3.28

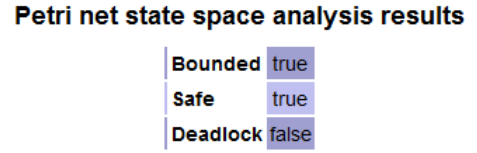

Figura 3.28: Análisis de la red propuesta limitando sus plazas.

El modelo pasa a ser acotado, seguro y libre de interbloqueos.

3.13.5. Implementación del modelo

Para implementar el modelo en el código fuente, se procedió a:

1. Eliminar en `sched_petri.h` el macro de la transición de ejecución para modo monoprocesador. Incorporar los cambios en `init_resource_net`.

2. Eliminar en `petri_global_net.c` la transición jerárquica del *thread* a la transición de ejecución descartada.

3. Inicializar correctamente el marcado de los *threads* en `init_petri_thread` de `sched_petri.c`. Notar que esta función, llamada al ser alocado en memoria un nuevo *thread*, nunca será llamada para el *thread0* inicial.

4. Inicializar correctamente el marcado de la red de recursos en la función `init_resource_net` de `petri_global_net.c`. Esta función se llamará al inicializar el *scheduler* en `sched_setup` de `sched_4bsdc`.

5. Inicializar en la función `sched_init` de `sched_4bsd.c` el marcado del *thread0*, función únicamente ejecutada por el mismo al inicializar el sistema.

6. Modificar la función `sched_choose` de `sched_4bsd.c` de tal forma que:

   - Si no encuentra ningún *thread* para ejecutar en ninguna de las colas, se ejecute la transición de encolado global del *idle_thread*. Como los *idle_threads* no se encuentran en las colas, para no perder el flujo de la red de los mismos se los encolará "de pasaje" en la cola global.
   - Inmediatamente después del desencolado, ejecutar la transición de desencolado global para finalizar el "pasaje" por la cola global.
   - El tratamiento de los *idle_threads* en `sched_switch` será igual al resto de los *threads* en cuanto a pasaje a ejecución y finalización de la misma. Solamente no se los tendrán en cuenta a la hora de encolarlos cuando los mismos son expulsados de la CPU.
   - Dado a que también pueden sufrir cambios de contexto voluntarios como involuntarios, para el caso que *td_frominh* sea 1 se deberá ejecutar la transición del *thread* que lo saca del estado de inhibido. Esto se realiza antes de encolar y desencolar el *idle_thread* en `sched_choose`.

3.13.6. Análisis de resultados

El resultado esperado en cuanto al funcionamiento de los hilos de baja prioridad fue correcto. Esta nueva incorporación permitió ejecutar un análisis completo de las redes en ejecución y los resultados en cuanto al seguimiento de sus marcados fueron los esperados para el modelado del sistema propuesto desde un inicio.

Por otra parte, el agregado de los sistemas de control permitió corroborar en ejecución que las transiciones que controlan son en todo momento correctamente disparadas, sin encontrar momentos donde un hilo las intente disparar sin encontrarse sensibilizadas.

3.13.7. Próximos pasos

El último modelo alcanzado, luego de realizar su implementación y ponerlo en ejecución, muestra un correcto funcionamiento total esperado para el *scheduler* del sistema operativo. Se alcanzó finalmente el objetivo propuesto inicialmente de modelar el correcto funcionamiento del sistema completo.

Como próximo y último paso se procederá a presentar el modelo completo y se elaborarán los diagramas de secuencia necesarios para explicar las distintas funcionalidades llevadas a cabo por el *scheduler* en conjunto con el modelo desarrollado.

3.14. Modelo completo

3.14.1. Modelo del hilo

En la figura 3.29 se presenta el modelo final de la red del hilo con el correspondiente marcado inicial.

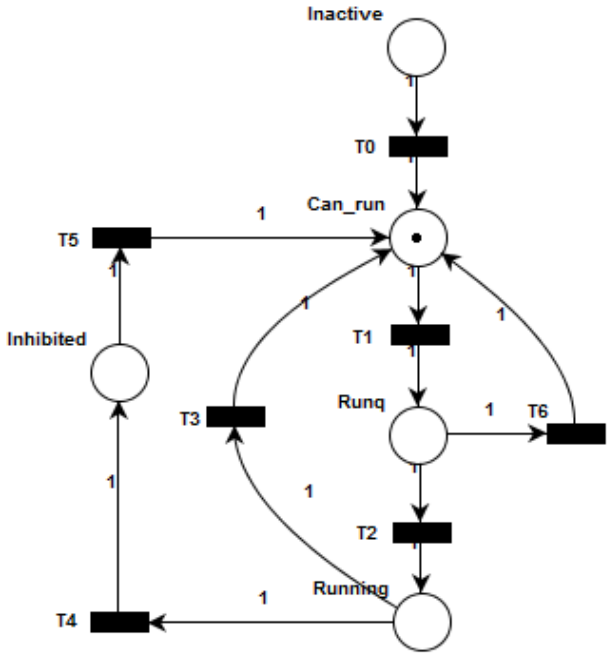

Figura 3.29: Modelo del hilo completo.

3.14.2. Modelo de la red de recursos

En la figura 3.30 se presenta el modelo final de la red de recursos con el correspondiente marcado inical para un sistema compuesto por 4 CPU. Se representan de color verde las transiciones y plazas globales de la red.

3.14.3. Jerarquía de transiciones

Se tiene la siguiente jerarquía de transiciones:

- Transiciones `ADDTOQUEUE` y `TG` de la red de recursos son jerárquicas a la transición CAN_RUN ⇒ RUNQ del hilo.
- Transiciones `EXEC` y `EXECEMPTY` de la red de recursos son jerárquicas a la transición RUNQ ⇒ RUNNING del hilo.
- Transiciones `REMOVE`, `REMEMPTY` y `REMOVEGLOBAL` de la red de recursos son jerárquicas a la transición RUNQ ⇒ CAN_RUN del hilo.

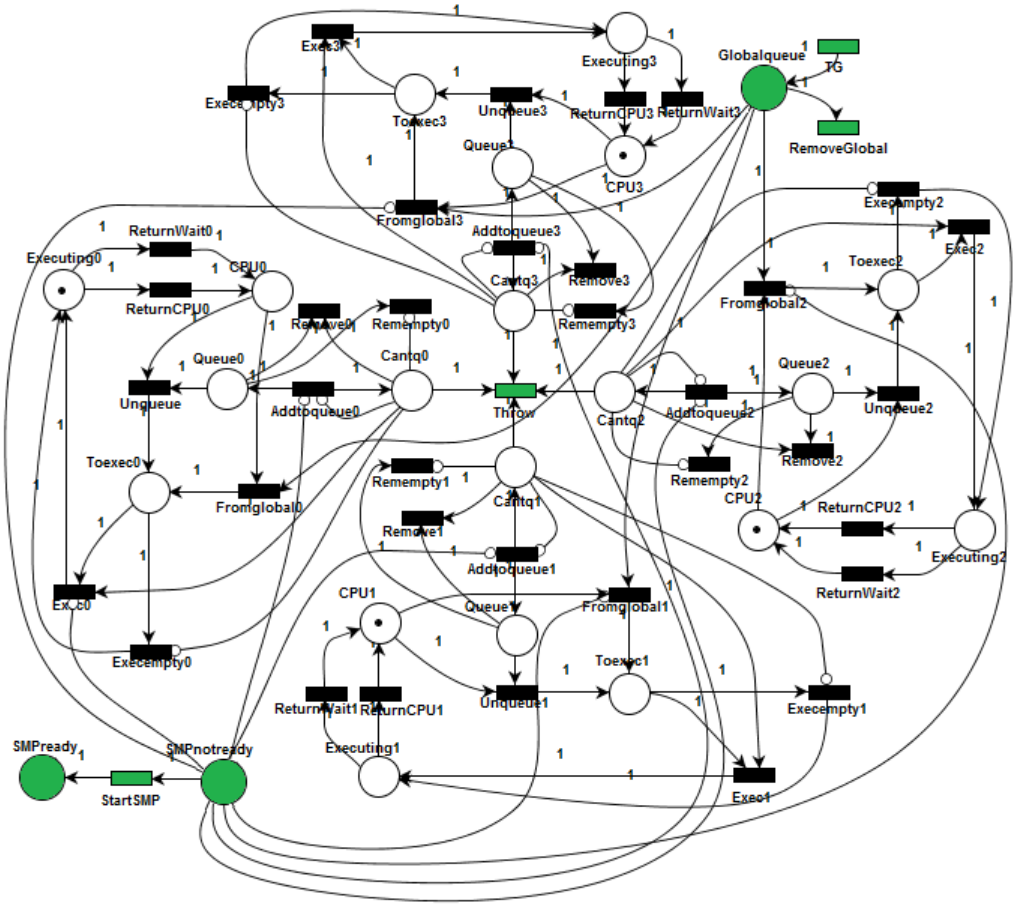

Figura 3.30: Red de recursos con 4 CPU.

- Transición `RETURNCPU` de la red de recursos es jerárquica a la transición RUNNING ⇒ CAN_RUN del hilo.
- Transición `RETURNWAIT` de la red de recursos es jerárquica a la transición RUNNING ⇒ INHIBITED del hilo.

3.14.4. Funciones principales

Las funciones principales del *scheduler* implementado son:

- `sched_add`: se encarga del encolado de los hilos. Hace uso de la función `resource_choose_cpu` para seleccionar la cola. Puede pasar al hilo directamente a ejecución si la función `maybe\_preempt` lo determina.
- `sched_choose`: se encarga de desencolar al hilo de mayor prioridad, fijándose tanto en la cola de la CPU actual como en la cola global.
- `sched_rem`: se encarga de remover al hilo de la cola, el cual debe ser reubicado. Hace uso de la función `resource_remove_thread`.
- `sched_switch`: se encarga de expulsar al hilo actual en ejecución haciendo uso de la función `resource_expulse_thread`, reubicarlo en una cola y seleccionar el próximo a ejecutar mediante `choose_thread` para mandarlo a ejecución haciendo uso de la función `resource_execute_thread`.
- `sched_throw`: igual a `sched_switch`, con la diferencia de que no reubica en una cola al hilo saliente ya que el mismo ha finalizado su ejecución.

3.14.5. Diagramas de secuencia

A continuación se presentan los diagramas de secuencia de las funciones principales:

Función sched_add

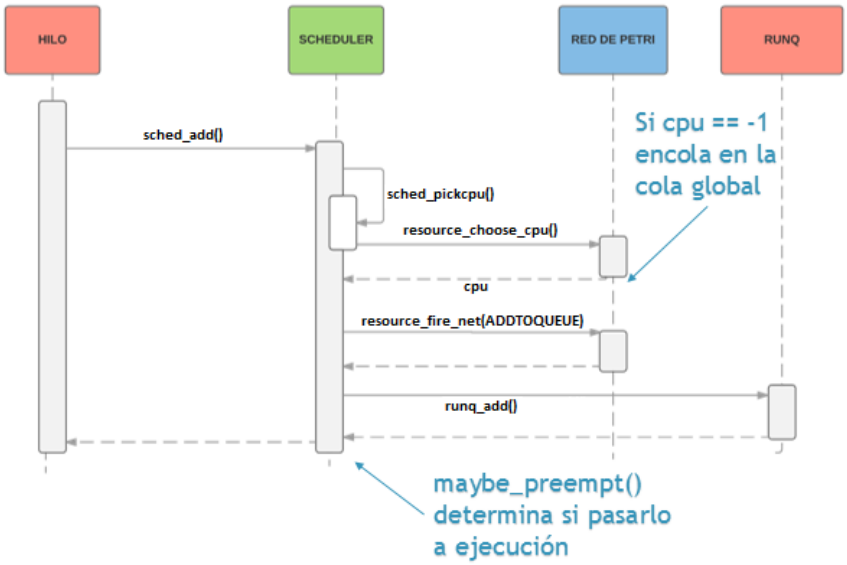

Figura 3.31: Diagrama de secuencia de `sched_add()`.

Función sched_choose

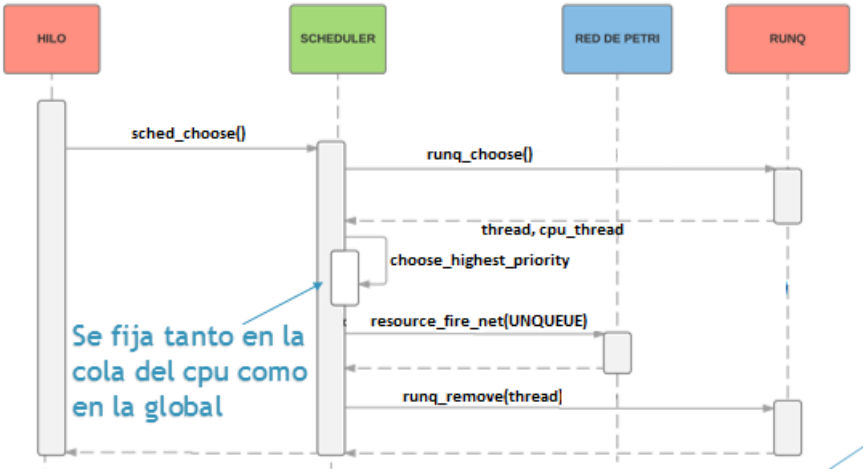

Figura 3.32: Diagrama de secuencia de `sched_choose()`.

Función sched_rem

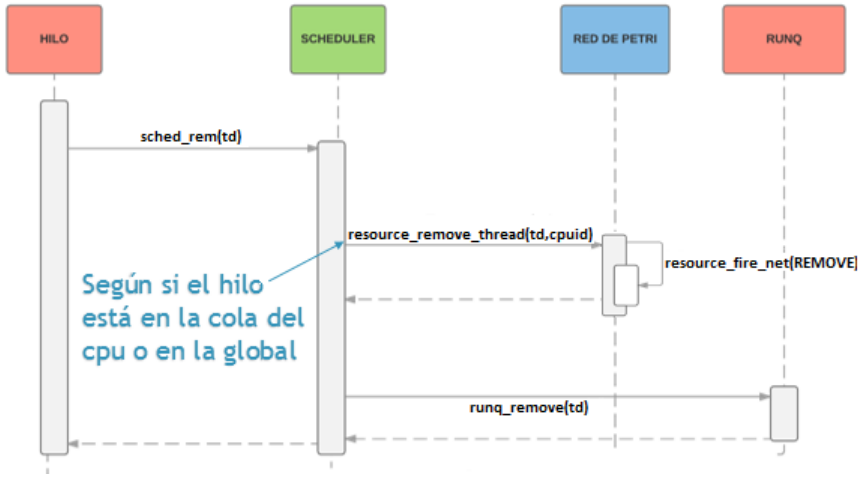

Figura 3.33: Diagrama de secuencia de `sched_rem()`.

Función sched_switch

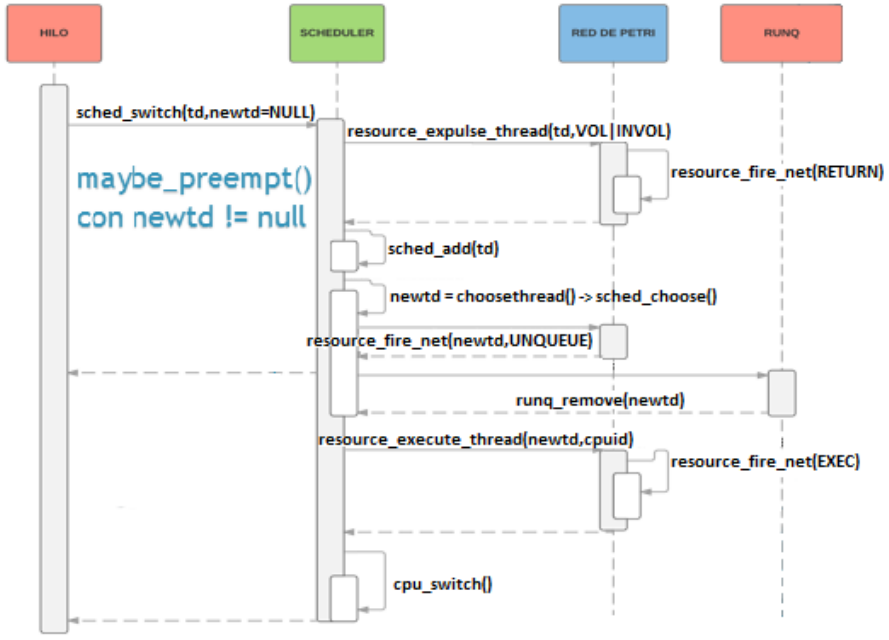

Figura 3.34: Diagrama de secuencia de `sched_switch()`.

Función sched_throw

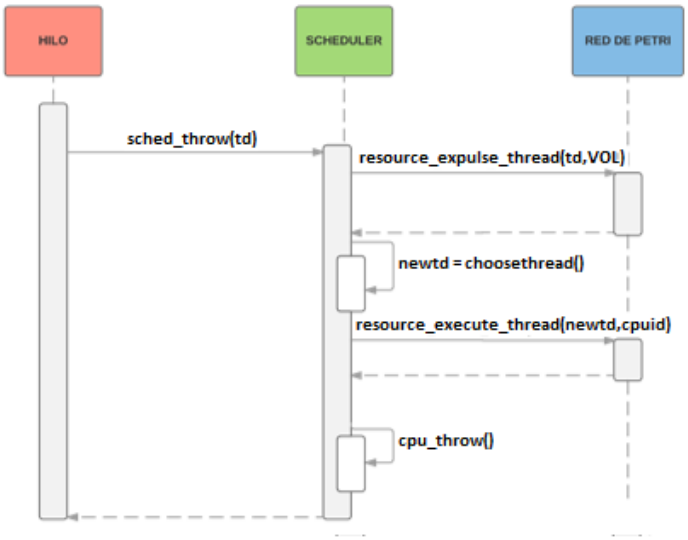

Figura 3.35: Diagrama de secuencia de `sched_throw()`.

Capítulo 4

Presentacion de resultados

4.1. Resultados obtenidos

En esta sección se llevarán a cabo algunas pruebas sobre el modelo de planificador desarrollado en funcionamiento. Las pruebas no son exhaustivas, sino que sólo se realizarán para demostrar que el modelo funciona. También se harán comparaciones de estos resultados con los del planificador 4BSD para observar si existen grandes diferencias entre ambos.

Si bien el objetivo del proyecto no propone mejorar el rendimiento en tiempo del *scheduler*, se propone realizar una serie de pruebas para poder observar la sobrecarga que el agregado de la red genera a la hora de realizar cambios de contexto de hilos. (Aclaración: Si bien las decisiones para elegir el próximo hilo son realizadas por la red, no se ha eliminado completamente la lógica del *scheduler*, anterior por lo cual hay lógica redundante en el mismo).

Se realizarán dos tipos de mediciones comparativas:

1. La primera prueba consiste en un programa en el cual un proceso padre crea un cierto número de procesos hijos y cada hijo realizará una cantidad fija de multiplicaciones de números enteros. Se mide el tiempo en el cual todos los hijos terminan de ejecutarse.

2. La segunda prueba consiste en una prueba idéntica a la anterior, con la diferencia de que los procesos en este caso intentarán realizar una cierta cantidad de impresiones en pantalla.

Ambas pruebas se realizarán en una máquina virtual de las siguientes características:

- Nucleos Asignados: 4 Núcleos.
- Ram Asignada: 2GB.
- Carga inicial del sistema operativo al iniciar el programa: Mínima.

Como puede observarse en la figura 4.1 la penalización obtenida a partir de las modificaciones realizadas son mínimas considerando la redundancia que

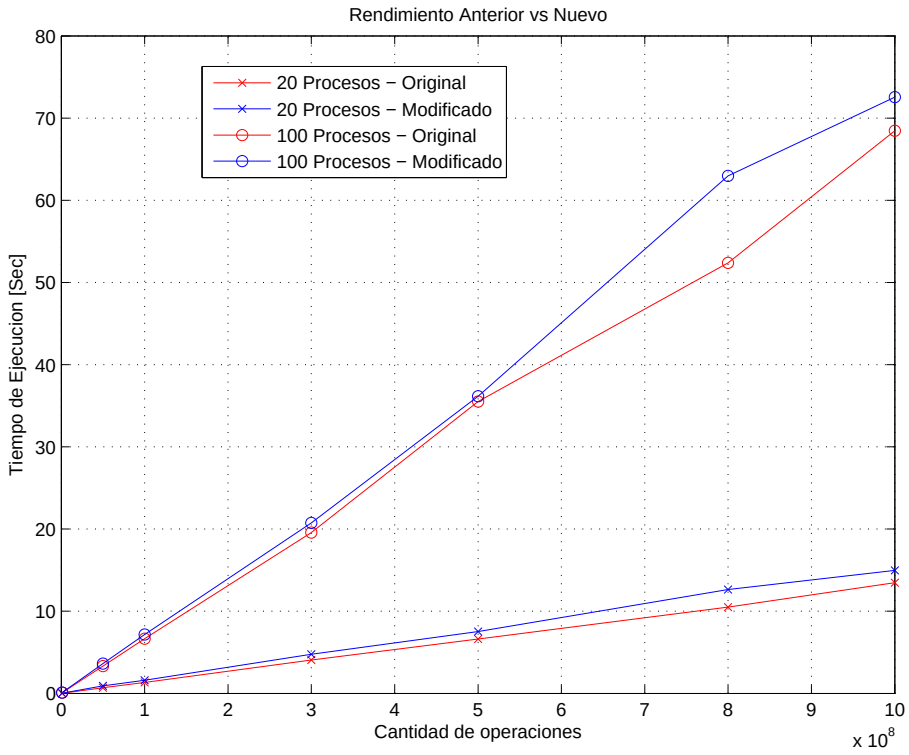

Figura 4.1: Resultados obtenidos en la primera prueba.

hay al no poder descartarse completamente el modelo anterior. Por el contrario, en la figura 4.2 pueden verse que los resultados son poco concluyentes ya que dependen mucho del estado del sistema en ese preciso momento de ejecución, pero a grandes rasgos puede verse que las modificaciones planteadas no lo afectan significativamente.

4.2. Overhead en memoria

Uno de los detalles a analizar a la hora de evaluar el desarrollo realizado es la cantidad de espacio de memoria adicional que requiere el agregado de una red de Petri al sistema operativo. Se agregan algunas referencias para mayor entendimiento del método de cálculo.

```
CPU_NUMBER = 4 (Cantidad de CPU)
CPU_BASE_PLACES = 5 (Cantidad de plazas bases por CPU)
CPU_BASE_TRANSITIONS = 9 (Cantidad de transiciones bases por CPU)
CPU_NUMBER_PLACES = (CPU_BASE_PLACES*CPU_NUMBER)+3
(Cantidad de plazas totales)
CPU_NUMBER_TRANSITION = (CPU_BASE_TRANSITIONS*CPU_NUMBER)+4
(Cantidad de transiciones totales)
```

El cálculo se realiza a continuación:

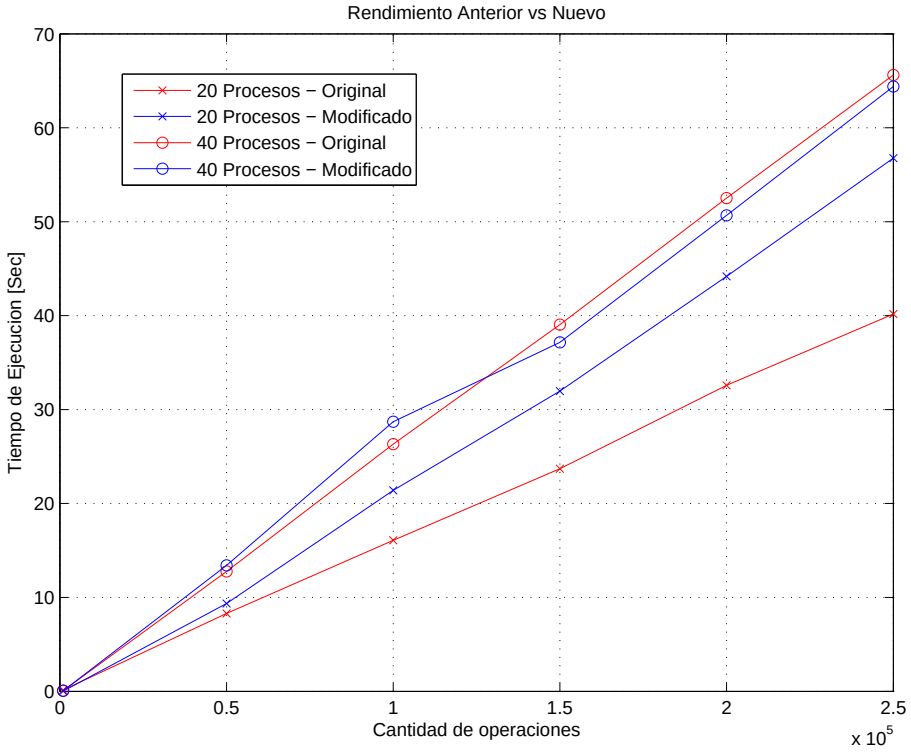

Figura 4.2: Resultados obtenidos en la segunda prueba

```
MARCADO = 4bytes * CPU_NUMBER_PLACES
BUFFER_SENSIBILIZADA = 4bytes * CPU_NUMBER_TRANSITION
ARREGLO_CPU_OWNER = 4bytes * CPU_NUMBER
MATRICES_INCIDENCIA_INHIBICION =
1byte* 2*(CPU_NUMBER_PLACES*CPU_NUMBER_TRANSITION)
AUTOMATICAS = 4bytes * CPU_NUMBER_TRANSITION
```

El resultado es la suma de los 5 componentes anteriormente mencionados.

Por ej, para 4 CPU se obtiene:

```
MARCADO = 92 bytes
BUFFER_SENSIBILIZADA = 160 bytes
ARREGLO_CPU_OWNER = 16 bytes
MATRICES_INCIDENCIA_INHIBICION = 1840 bytes
AUTOMATICAS = 160 bytes
```

Esto nos da un total de 2.27 KB para 4 CPU.

Capítulo 5

Conclusión y Trabajos Futuros

5.1. Conclusión

El *scheduler* es un componente fundamental dentro de cualquier sistema operativo. Su principal tarea es el reparto de tiempo de procesador a los diferentes hilos y procesos en ejecución. Esto permite dar una ilusión de ejecución simultánea y multitarea.

Nuestra propuesta se basó en tomar este componente del sistema operativo, analizar su comportamiento, el proceso por el cual asigna tiempo de ejecución y proponer un modelo que permita mejorar el existente. Se observó que el proceso de toma de decisiones de asignación de tiempo de procesador es totalmente estadístico, lo cual ignora información de ejecución que podría ser importante a la hora de elegir el próximo hilo a ejecutar. Si el planificador agregara información sobre el orden de ejecución de los procesos, se podría disminuir la incertidumbre y agregar determinismo.

Para realizar dicho modelo se tomó un planificador como modelo inicial. El sistema operativo elegido fue FreeBSD debido a las libertades de uso que ofrece la licencia que dispone y su creciente popularidad en el desarrollo de sistemas embebidos o en proyectos *open source*. FreeBSD posee dos *scheduler*: 4BSD y ULE. Tras un análisis exhaustivo se eligió 4BSD, tanto por su sencillez de funcionamiento como por su robustez y estabilidad.

El esfuerzo necesario para entender el funcionamiento del *scheduler* elegido fue difícil de estimar con anterioridad debido a que no se contaba con ninguna experiencia previa en este tipo de trabajo. Conocer su funcionamiento resultó sumamente necesario para emplearlo como punto de partida del modelo a desarrollar.

Para abordar el diseño del modelo, se utilizaron conocimientos adquiridos sobre redes de Petri, que ayudaron tanto para resolver los problemas de concurrencia como para validar formalmente la inexistencia de inanición e interblo-

queos.

Todas las iteraciones pueden ser consideradas como logros importantes desde una perspectiva general del trabajo. Cada una de ellas marcó un punto importante, ya sea porque permitió extender las funcionalidades del modelo, como por ejemplo la incorporación del funcionamiento monoprocesador y multiprocesador o el modelado de la afinidad de hilos con los distintos procesadores. También resaltaron un camino que parecía que era el correcto por el cual transitar, pero no resultaba así, como por ejemplo cuando no resultó conveniente la utilización de redes dinámicas para la asignación de procesadores.

El trabajo dedicado a cada una de las iteraciones y los logros alcanzados en cada una de ellas pueden ser resumidos desde una vista general como un solo logro superior al resto. La gran cantidad de tiempo de estudio dedicado a entender el funcionamiento de los planificadores actuales y la selección de un camino seguro que se fue ajustando sobre el transcurso del desarrollo de la idea permitieron llegar a cumplir los objetivos detallados para este trabajo. Así, se llegó a un resultado deseado superando la gran incertidumbre que había respecto a obtener resultados satisfactorios y abriendo una gran oportunidad para continuar investigando sobre este ámbito con altas probabilidades de éxito.

5.2. Trabajos Futuros

Para alcanzar la meta de un planificador a corto plazo que permita preveer la mejor forma de asignar todos los recursos del sistema operativo que gestione resta de mucho trabajo por realizar. Entre otros, se pueden enumerar los siguientes trabajos a futuro:

- Realizar pruebas más exhaustivas sobre el funcionamiento del modelo propuesto. Analizar su comportamiento en un sistema operativo con interfaz gráfica y gran carga de procesos.
- Modelar un reparto de CPU a nivel de procesos, asignando ciertos tipos de procesos a determinados CPU. Por ejemplo, asignar procesos que realizan operaciones complejas de cálculo a un CPU preparado para realizarlas, optimizando el funcionamiento general del sistema operativo.
- Agregar al modelo la posibilidad de pasar de un estado monoprocesador a multiprocesador y viceversa según la cantidad de procesos en estado de ejecución del sistema. Esto permitiría, por ejemplo, apagar algunos CPU que están siendo inutilizados.
- Reemplazar el sistema actual de cálculo de prioridades de los procesos, ya que en un sistema embebido todos los procesos se conocen de antemano. Se podría armar un plan de ejecución que no incluya cálculos estadísticos al momento de elegir el próximo hilo a ejecutar.
- Cambiar el enfoque del trabajo y analizar si el modelo puede utilizarse para mejorar la performance del sistema operativo. Analizar posibilidades de correr el planificador desarrollado en una FPGA, lo cuál podría aumentar drásticamente el desempeño del sistema.

- Modelar la versión ULE del sistema operativo FreeBSD, el cuál incorpora nuevas funcionalidades respecto a 4BSD.

Bibliografía

[1] *The design and implementation of the FreeBSD operating system*, Marshall Kirk MkKusick, George V. Neville-Neil, Robert N.M. Watson, 2nd Edition.

[2] *ULE: A Moder Scheduler for FreeBSD*, Jeff Roberson, The FreeBSD project.

[3] *Sistemas Operativos: Una visión aplicada* Jesús CARRETERO PÉREZ, Félix GARCÍA CARBALLEIRA, Pedro DE MIGUEL ANASAGASTI, Fernando PÉREZ COSTOYA Editorial Mc Graw Hill.

[4] *Profiling and Debugging the FreeBSD Kernel*, Ray Kinsella, May 2009.

Apéndice A

Anexo: Como preparar el ambiente de desarrollo

A.1. Como compilar un kernel FreeBSD en modo debug

Mientras kgdb es un *debugger offline* que provee una interfaz de usuario de alto nivel, hay cosas que no puede hacer. Lo principal es que no puede insertar *breakpoints* ni simples *steps* en el código del *kernel*. Por lo tanto, es necesario *debuggear* el *kernel* a bajo nivel utilizando una herramienta *online*. Para poder *debuggear* el *kernel* de FreeBSD se provee la herramienta de *debug* DDB. Esta herramienta es un *debugger* interactivo que permite al usuario ejecutar comandos específicos para inspeccionar detalles sobre el *kernel* en ejecución. Permite controlar la ejecución del mismo utilizando *breakpoints* y simples *steps*. Sin embargo, este *debugger* no puede acceder a los archivos source del *kernel* y solo tiene acceso a los símbolos globales y estáticos, no toda la información de *debug* como gdb abarca.

Para proceder con el *debug*, es necesario compilar el *kernel* FreeBSD y configurar las opciones de *debug* necesarias. Para esto se deben seguir los siguientes pasos:

1. Descargar el código fuente del *kernel* de FreeBSD en la carpeta */usr/freebsd* si es que no se encuentra disponible después de instalarlo. Se puede descargar tanto de GIT como de SVN, el versionado de la revisión utilizada es `FreeBSD: releng/11.0/README 300137 2016-05-18 10:43:13`.

2. Entrar en la carpeta del código fuente del *kernel* y copiar una configuración de *kernel* ya existente, posicionarse y ejecutar los siguientes comandos:
   ```
   # cd /usr/src/sys/amd64/conf
   # cp GENERIC KERNELCONF
   ```

3. Editar el archivo `KERNELCONF` recientemente creado para agregar las siguientes opciones de *debug*:

   ```
   options KDB
   ```

```
options DDB
options KDB_UNATTENDED
options GDB
```

4. Volver a la carpeta `/usr/src` y como root ejecutar el siguiente comando:
   `# make buildkernel KERNCONF=KERNELCONF`

5. Finalmente se debe instalar el nuevo *kernel* ejecutando:
   `# make installkernel KERNCONF=KERNELCONF`

Seguidos los pasos anteriores, el *kernel* ya se encuentra compilado con las opciones de *debug* necesarias. Luego de reiniciar la máquina con la nueva configuración, para ingresar a DDB desde la terminal y comenzar con el *debug* se ejecuta el siguiente comando:
`# sysctl debug.kdb.enter=1`

Los comandos DDB son similares a algunos de gdb. Por ejemplo, se utiliza *break function-name address* para establecer un breakpoint o "s" para saltar un simple step.

A.2. Debug del Kernel FreeBSD entre dos máquinas virtuales (debug remoto)

Para realizar un *debug* remoto del *kernel* FreeBSD utilizando kgdb se necesitan configurar dos máquinas virtuales, una como *deb* y la otra como *target*.

A continuación se presentan los pasos necesarios para configurar las VMs y conectarlas a través de puerto serie para el *debug* con kgdb:

1. Descargar el código fuente utilizado para compilar el *kernel* de la VM1 *"deb"* en la VM2 *"target"* mediante los siguientes comandos:
   `# cd /usr/src`
   `# fetch ftp://ftp.freebsd.org/pub/FreeBSD/releases/amd64/{RELEASE-VERSION}/src.txz` (la version utilizada fue *11.0-RELEASE*
   `# tar -C /usr/ -xzvf src.txz`

2. Dirigirse a la VM1 y configurar la siguiente información en puertos seriales tal como lo muestra la figura A.1.

   La opción de conectar a un pipe existente debe estar desmarcada ya que se busca crear uno nuevo.

3. Dirigirse a la VM2 y establecer la configuración de la misma manera sólo que se debe seleccionar la opción de conectarse a un pipe existente (figura A.2).

4. Para realizar una prueba del puerto serial entre ambas máquinas se deben prender las dos máquinas virtuales y en la VM1 escribir:

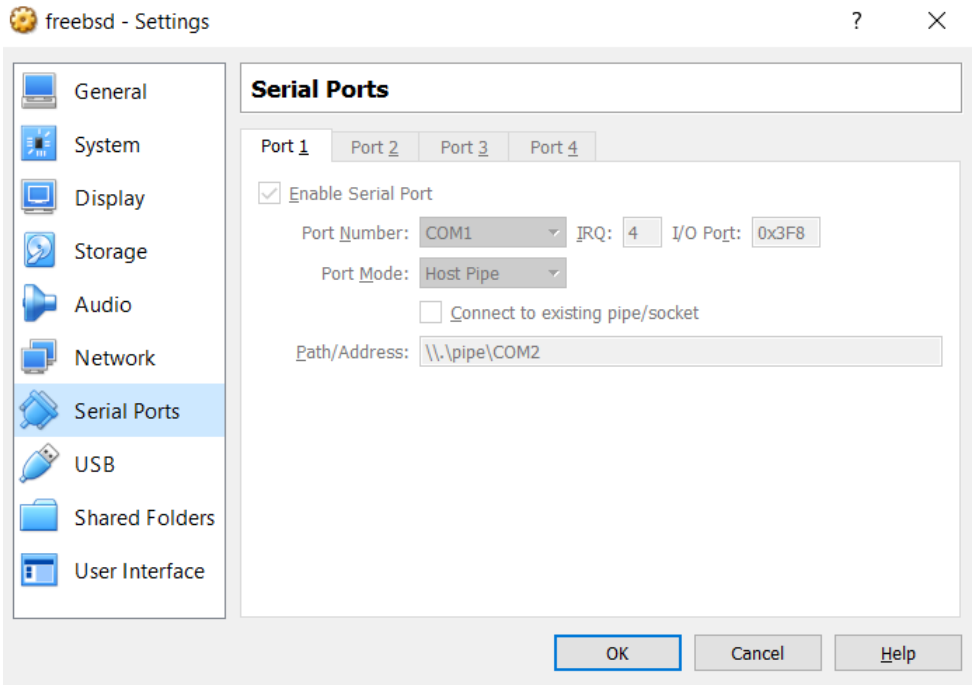

Figura A.1: Configuracion de los puertos serial de las máquinas virtuales para VM1.

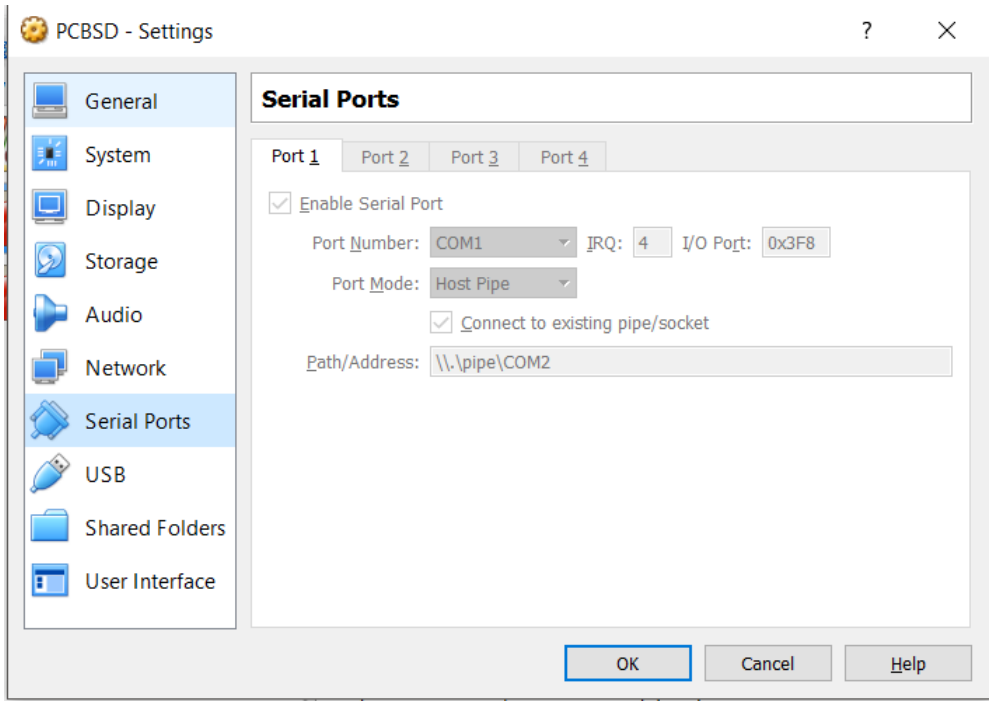

Figura A.2: Configuracion de los puertos serial de las máquinas virtuales para VM2.

```
# echo "Test String" >> /dev/cuau0
```

En la máquina VM2 se ejecuta el siguiente comando para leer el *Test String* recibido:

```
# cat /dev/cuau0
```

5. Para poder *debuggear* se deben seguir los siguientes pasos:
   - En la VM2 editar el archivo ubicado en `/boot/device.hints` para que se encuentre la siguiente línea.

     ```
     hint.uart.0.flags="0x80"
     ```

   - Copiar los archivos de `/usr/obj` desde la VM1 hacia la VM2 para poder *debuggear* el *kernel*.
   - Ejecutar en la VM2 los siguientes comandos:

     ```
     #  cd /usr/obj/usr/src/sys/KERNELCONF|
     # make gdbinit
     # kgdb -r /dev/cuau0 ./kernel.debug
     ```

   - Luego, en la VM1 ejecutar los comandos:

     ```
     # sysctl debug.kdb.enter=1
       db> gdb
     ```

   - Para *debuggear* en la VM2 utilizar:
     `n(next),s(step),bt(break),c(continue)`.

Para mayor información consultar en:

FreeBSD kernel debugging - http://chetanbl.blogspot.com.ar/2011/11/freebsd-kernel-module-debugging.html

Install FreeBSD kernel sources - http://unix.stackexchange.com/questions/204956/how-do-you-install-the-freebsd10-kernel-sources

A.3. Agregado de archivos al kernel

Se realizan agregando líneas al archivo `/usr/src/sys/conf/files` Por ejemplo:

```
kern/sched_4bsd.c optional sched_4bsd
kern/sched_ule.c optional sched_ule
kern/petri_global_net.c  standard
kern/sched_petri.c       standard
```

En este caso los archivos .c son agregados a la carpeta donde se encuentra el *kernel*: `/usr/src/sys/kern`, mientras que los .h son copiados en la carpeta `/usr/src/sys/sys`

A.4. Recompilado rápido del kernel

Existen una serie de banderas que pueden utilizarse para realizar un recompilado más rápido del *kernel*. Estas son las siguientes:

```
make NO_KERNELCLEAN=yes NO_KERNELDEPEND=yes
MODULES_WITH_WORLD=yes buildkernel KERNCONF=KERNELCONF
```

- **NO_KERNELCLEAN:** evita que elimine los archivos ya compilados que no nec esitan ser recompilados.
- **NO_KERNELDEPEND:** evitar revisar el árbol de dependencias. Utilizar cuando no se haya realizado ningún cambio en un *header*.
- **MODULES_WITH_WORLD:** evita el recompilado de los módulos del *kernel*.