UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Programación de sistemas de tiempo real blando y duro en lenguaje Java: ventajas y desventajas con respecto a C.
Autores: Demian Klc
Director: Fernando Romero
Codirector: Fernando G. Tinetti
Asesor profesional: No aplica
Carrera: Licenciatura en Informática

Resumen

El objetivo de la presente tesina es mostrar que es posible la programación de sistemas de tiempo real hard utilizando el lenguaje Java, gracias a la especificación RTSJ. Se utilizará para tal fin la implementación JamaicaVM y se hará un análisis para demostrar su funcionamiento. Se expondrán y compararán dos plataformas de desarrollo de sistemas de tiempo real: la actual y más utilizada, en el cual implica programación de bajo nivel utilizando C y una alternativa más moderna utilizando Java y una máquina virtual que implemente la especificación RTSJ (Real Time Specification for Java). La comparación buscará presentar al lector las ventajas y desventajas de utilizar la alternativa más moderna.

Palabras Claves

Tiempo Real, Java, RTSJ, JamaicaVM

Conclusiones

Se concluye que C es más rápido y eficiente, con menores latencias que Java (cuatro veces menor). JamaicaVM demuestra que es posible programar aplicaciones de tiempo real duro con el lenguaje Java. A pesar de tener latencias mayores que C, la máquina virtual ofrece muchas facilidades para la programación de sistemas de tiempo real, como el planificador, definición de threads periódicos, aperiódicos y esporádicos, detección de incumplimiento de plazos, entre otros.

Trabajos Realizados

Experimentos utilizando cyclictest (implementación en C) y rtps (implementación propia en Java) sobre un kernel de tiempo real para medir las latencias de cada aplicación para luego compararlos.

Trabajos Futuros

Medir las latencias de JamaicaVM utilizando memorias acotadas para el manejo de la memoria de los threads de tiempo real y compararlos con los resultados obtenidos aquí. Prueba de una ejecución sobre un sistema embebido que ejecute con JamaicaVM con interfaces con el mundo exterior, como lo puede ser un robot. Prueba de comunicación entre dispositivos embebidos con JamaicaVM.

Fecha de la presentación: Octubre 2015

Universidad Nacional de La Plata
Facultad de Informática

Tesina de Grado

Programación de sistemas de tiempo real blando y duro en lenguaje Java

Ventajas y desventajas con respecto a C

Autor: Demian Kĺč
Director: Fernando Romero
Codirector: Fernando G. Tinetti

9 de Octubre de 2015

Índice General

1 Introducción

Los sistemas de tiempo real juegan un papel fundamental en nuestra sociedad porque gran cantidad de los sistemas actuales dependen, en parte o completamente, en la computación en tiempo real. Ejemplos de aplicaciones incluyen plantas nucleares de energía, sistemas de control de vías férreas, sistemas en automóviles y aviación, control de tráfico aéreo, telecomunicaciones, robótica y sistemas utilizados en el ejército. En los últimos años los sistemas de tiempo real encontraron nuevas áreas de aplicación, como en equipo médico, electrónica de consumo, sistemas multimedia, sistemas de simulación de vuelo, realidad virtual y juegos interactivos.

A pesar del amplio dominio de aplicaciones, la mayoría de los sistemas de tiempo real todavía son diseñados e implementados utilizando lenguajes de bajo nivel con técnicas empíricas, sin usar una metodología científica. Este enfoque resulta en una baja confiabilidad, la cual en sistemas críticos pueden provocar grandes daños ambientales e incluso la pérdida de vidas.

Java es uno de los lenguajes más importantes y ampliamente utilizados en el mundo, manteniendo esta distinción por muchos años. A diferencia de a otros lenguajes, cuya influencia fue disminuyendo con el paso del tiempo, Java se hizo más fuerte con cada nueva versión. Ofrece al desarrollador un lenguaje orientado a objetos portable, concurrente y más legible, permitiendo disminuir los tiempos de desarrollo y el mantenimiento del software. Gracias a la especificación RTSJ, es posible utilizar el lenguaje Java para desarrollar sistemas de tiempo real duro (hard) y blando (soft).

El objetivo de este trabajo es validar o refutar la hipótesis acerca de las ventajas del lenguaje Java para programar sistemas de tiempo real sobre el lenguaje C. Para demostrarlo se utilizará sobre un Linux RT la máquina virtual de Aicas de tiempo real JamaicaVM [1] y se correrán distintas pruebas; principalmente se compararán las latencias de tareas periódicas de implementaciones en ambos lenguajes, y adicionalmente se mostrarán los distintos mecanismos provistos por la especificación RTSJ versión 1.0.2 [2], tales como el planificador FPS, threads de tiempo real periódicos, eventos y manejo de excepciones, entre otros.

Se estudiará hasta qué punto se pueden programar sistemas de tiempo real blando y duro con lenguaje Java, permitiendo al desarrollador optar por un lenguaje orientado a objetos y más moderno que C.

2 Conceptos de sistemas de tiempo real

Un sistema de tiempo real (RTS) es un sistema informático que se diferencia de otros sistemas por tener al tiempo como un factor fundamental. En este tipo de sistemas, la correctitud de la respuesta no solo depende de los resultados, sino del instante en el tiempo en que se produjeron. Como suelen ser sistemas que deben responder a eventos externos durante su evolución, el tiempo interno al sistema debe medirse de la misma forma que se mide fuera de él [3][4][5][6][7].

Un sistema de tiempo real no necesariamente debe ser veloz y responder lo más rápido posible; el tiempo de respuesta dependerá de la naturaleza de los eventos y sus posibles restricciones. Lo importante es que responda sin pasarse del tiempo de respuesta máximo permitido.

Si se compara un sistema de tiempo real con un organismo vivo, este último requiere distintos tiempos de respuestas dependiendo de su naturaleza. Por ejemplo, una tortuga puede sobrevivir y existir dentro de su hábitat natural y sus reacciones serán efectivas dentro de este hábitat, a pesar de no ser veloz como otros animales. La tortuga tendrá problemas si se le ofrecen eventos que éste no puede manejar. Otros animales son más veloces pero si se introducen eventos que éstos no pueden manejar, como la trampera a un ratón, la seguridad y supervivencia de esa especia estaría comprometida [4]. Estos eventos suelen ser para esa especie eventos anómalos e inusuales, y puede ocurrir lo mismo con los sistemas de tiempo real. Son sistemas inherentemente concurrentes porque en general están embebidos en un sistema más grande y deben modelar el paralelismo de los objetos del mundo real que están controlando [5].

Los sistemas embebidos de tiempo real, también llamados sistemas ciber-físicos (cyber-physical systems o CPS), forman el segmento de mercado más importante en las tecnologías de tiempo real y en la industria de la computación en general [6].

2.1 Tareas de tiempo real

Generalmente un evento en un sistema de tiempo real está asociado a una tarea de tiempo real, a nivel del procesador. Estas tareas se diferencian de una tarea común por tener definido un máximo tiempo de respuesta permitido para terminar esa tarea y producir los resultados; a este tiempo se le suele llamar *plazo* (deadline). Cuando no se cumple este tiempo para una tarea determinada, se dice que ocurrió un *incumplimiento de plazos* (deadline miss).

Se definen tres clasificaciones sobre los plazos según las consecuencias que impliquen su incumplimiento [6]:

- Un plazo se dice que es *duro* (hard) si su incumplimiento produce resultados catastróficos sobre el sistema y probablemente su entorno.
- Un plazo se dice que es *firme* (firm) si su incumplimiento produce resultados inservibles al sistema, pero no son peligrosos para él.

- Un plazo se dice que es *blando* (soft) si su incumplimiento produce resultados que aún sirven al sistema, aunque resulte en un degradamiento de su desempeño.

Ejemplos de tareas que definan plazos duros pueden encontrarse en sistemas de alta integridad, relacionadas a actividades tales como:

- Adquisición de datos sensoriales
- Filtrado de datos y predicciones
- Detección de condiciones críticas
- Fusión de datos y procesamiento de imágenes
- Servos
- Control de bajo nivel en componentes del sistema crítico
- Planificación de acciones para sistemas con ajustada interacción con el entorno

Ejemplos de tareas que definan plazos firmes pueden encontrarse en aplicaciones en red y sistemas multimedia, en los cuales la pérdida de un paquete o un cuadro de video es menos crítico que un retardo; estas tareas pueden ser:

- Reproducción de video
- Encodeo y decodeo de audio y video
- Procesamiento de imágenes online
- Transmisión de datos sensoriales en sistemas distribuidos

Ejemplos de tareas que definan plazos blandos suelen relacionarse con la interacción entre el sistema el usuario. Éstas incluyen:

- Intérprete de comandos de la interfaz de usuario
- Manejar datos de entrada desde el teclado
- Mostrar mensajes en pantalla
- Representación de variables de estado del sistema

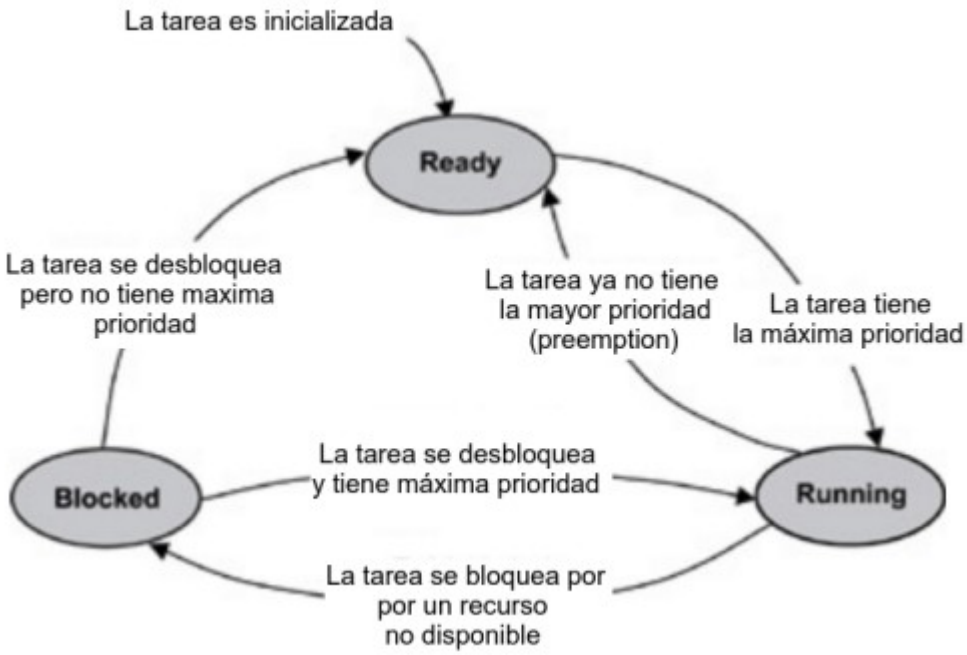
- Actividades gráficas
- Guardado de datos de reportes

Un sistema de tiempo real que define al menos un plazo duro se lo suele llamar sistema de tiempo real duro o sistema de tiempo real crítico para la seguridad (safety-critical real time system). El sistema no necesariamente define plazos únicamente duros, puede definir una mezcla de las tres clasificaciones (duro, firme o blando).

2.2 Estados de las tareas de tiempo real

Sea una tarea del sistema o de la aplicación, en cualquier momento se puede encontrar en alguno de sus estados: listo, ejecutando o bloqueado (ready, running o blocked) [4][7].

Mientras el sistema se encuentra en ejecución, cada tarea va cambiando de estado según la lógica de una máquina de estados finita. Se muestra a continuación un diagrama con sus transiciones:



A pesar de que los kernels pueden definir estados de forma diferente, generalmente se usan 3 estados principales mayormente utilizados en los kernels con apropiamiento (preemption), incluyendo:

- estado listo: la tarea está lista para ejecutarse pero existe otra tarea de mayor prioridad en ejecución, y si tiene la misma prioridad depende de la política de planificación utilizada (ej: FIFO, RR).
- estado bloqueado: la tarea solicitó un recurso que no está disponible, se puso a esperar hasta que ocurra un evento, o se demoró a sí mismo por un tiempo.
- estado ejecutando: la tarea es la que tiene mayor prioridad y está en ejecución.

Notar que algunos kernels, como el de VxWorks, definen estados adicionales, como suspendido (suspended), pendiente (pending) y demorado (delayed). En este caso, pendiente y demorado vienen a ser sub-estados del estado bloqueado. Una tarea en estado pendiente está esperando que se libere un recurso; una tarea demorada está esperando que transcurra una cierta cantidad de tiempo. Independiente de la forma que un kernel implemente su máquina de estados, tiene que mantener el estado actual de todas las tareas del sistema. A medida que se hacen llamadas al kernel para que ejecute tareas, el planificador del kernel primero determina qué tareas necesitan cambiar de estado, para luego aplicar estos cambios.

En algunos casos, un cambio de estado de una tarea puede ocurrir sin tener que hacer un cambio de contexto porque el estado de la tarea con mayor prioridad no cambia (como el cambio de estado desde bloqueado a listo). En otros casos, estos cambios resultan en un cambio de contexto porque la tarea en ejecución pasa a estar bloqueada o apareció una tarea con mayor prioridad, en el cual pasa a estar al estado de ready mientras que la nueva tarea comienza a ejecutarse.

Se describen los estados listo, ejecutando y bloqueado con más detalle. Las siguientes descripciones se basan en sistemas monoprocesador y con un kernel que utiliza un algoritmo de planificación apropiativo (preemptible) basado en prioridades.

2.2.1 Estado listo

Cuando una tarea es creada y está lista para ejecutarse, el kernel le asigna el estado listo. En este estado, la tarea compite con todas las demás tareas que esten en ready para utilizar el procesador. Una tarea en estado listo nunca puede pasar a bloqueado porque primero necesita pasar a ejecutando y luego hacer una llamada bloqueante que la haga pasar al estado bloqueado. Por lo tanto, tareas en estado listo solo pueden pasar al estado ejecutando. Como hay muchas tareas en este estado, el planificador (scheduler) se ocupa de seleccionar una tarea teniendo en cuenta la prioridad definida en cada una, y finalmente le asigna el estado de ejecutando.

Para kernels que soportan una única tarea por nivel de prioridad, el algoritmo de planificación es trivial: la tarea con mayor prioridad es la que debe ejecutarse. En esta implementación, el kernel esta definiendo un límite de tareas por la cantidad de niveles de prioridad.

Sin embargo, la mayoría de los kernels soportan más de una tarea por nivel de prioridad,

permitiendo una mayor cantidad de tareas en el sistema. En este caso, el algoritmo de planificación es más complicado e implica mantener una lista de tareas en estado listo. Algunos kernels mantienen una lista por prioridad y otras una lista combinada.

Se ilustra un ejemplo en 5 pasos de como un planificador de un kernel puede utilizar una lista de tareas en estado listo para mover tareas al estado ejecutando. Asume prioridades de 1 a 99, siendo 99 la máxima prioridad. Para mantener la simplicidad del ejemplo, el ejemplo no muestra tareas del sistema.

En este ejemplo, se tienen 5 tareas (1,2,3,4,5) con prioridades 90,80,80,80,70 respectivamente. Todas las tareas están en estado listo y el kernel las encola por prioridad en una lista de tareas en estado listo. La tarea 1 es la que tiene máxima prioridad (90); luego siguen las tareas 2,3 y 4 con una prioridad menor (80) y la tarea 5 con la menor prioridad (70). Se muestran a continuación los 5 pasos del ejemplo:

1. Las tareas 1,2,3,4 y 5 están listas para ejecutar, esperando en la cola de tareas en estado listo.
2. Como la tarea 1 tiene la máxima prioridad (90), es la primera tarea lista para ejecutar. Si no hay tareas con mayor prioridad ejecutándose, el kernel elimina la tarea 1 de la lista de tareas en estado listo y la mueve al estado ejecutando.
3. Durante la ejecución, la tarea 1 hace una llamada bloqueante. Como resultado, el kernel mueve la tarea 1 al estado bloqueado; toma la tarea 2 que está siguiente en la lista (prioridad 80) de la lista de tareas en estado listo y la mueve al estado ejecutando.
4. Más adelante la tarea 2 hace una llamada bloqueante, el kernel la mueve al estado bloqueado y mueve la tarea 3, siguiente en la lista, al estado de ejecutando como en el paso anterior.
5. Mientras ejecuta la tarea 3, éste libera el recurso que necesitaba la tarea 2. El kernel mueve la tarea 2 al estado listo y lo inserta al final de la lista de tareas en estado listo. Mientras tanto la tarea 3 sigue ejecutando.
6. Se desbloquea la tarea 1; luego el kernel mueve la tarea 1 al estado ejecutando por tener mayor prioridad que la tarea 3 (90 mayor a 80), y la tarea 3 la mueve del estado ejecutando al estado listo.

En la siguiente imagen se muestra la cola de tareas en estado listo para cada paso del ejemplo:

2.2.2 Estado ejecutando

En un sistema monoprocesador, solo una tarea puede ejecutar al mismo tiempo. En este caso, cuando una tarea es movida al estado ejecutando, el procesador carga sus registros con el contexto de la tarea. El procesador luego puede ejecutar las instrucciones de la tarea y manipular su pila asociada.

Como se vio en la sección anterior, una tarea puede moverse al estado listo mientras está en ejecución. Cuando sucede esto también se dice que esta tarea fue apropiada (preempted) por otra tarea de mayor prioridad.

A diferencia de una tarea en estado listo, una tarea en estado ejecutando puede moverse al estado bloqueado por alguna de las siguientes razones:

- Al hacer una llamada que solicita un recurso no disponible
- Al hacer una llamada para esperar por la ocurrencia de un evento
- Al hacer una llamada para demorar la misma tarea por un tiempo

Para cualquier caso, la tarea es movida desde el estado ejecutando a bloqueado, como se describe a continuación.

2.2.3 Estado bloqueado

La posibilidad de tener estados bloqueados es muy importante en sistemas de tiempo real, porque sin este estado las tareas de menor prioridad podrían nunca ejecutarse. Si las tareas de alta

prioridad no se bloquearan, las demás tareas sufrirían inanición por no poder nunca pasar al estado de ejecutando y tomar el control de la CPU.

Una tarea solo puede moverse al estado bloqueado al hacer una llamada bloqueante esperando que una condición de bloqueo se cumpla; hasta que esto no ocurra, la tarea permanecerá en el estado bloqueado. A continuación se muestran ejemplos de cómo estas condiciones pueden ser cumplidas:

- Un semáforo tomado por el cual una tarea está esperando es liberado
- Un mensaje que la tarea está esperando se carga en la cola de mensajes
- Una demora (delay) impuesta por la propia tarea expira

Cuando una tarea se desbloquea, la tarea puede moverse al estado de listo si la tarea no es la que tenía máxima prioridad, cargándose en la lista de tareas en estado listo con el orden impuesto anteriormente.

Sin embargo si la tarea es la que tiene la máxima prioridad, la tarea se mueve directamente al estado ejecutando sin pasar por el estado listo y se apropia de la tarea que estaba ejecutando en ese momento (obliga a que la tarea se mueva al estado listo).

2.3 Características de sistemas de tiempo real

En esta sección se hará una descripción de las características más importantes que debe tener un sistema de tiempo real, las cuales son: temporalidad, predictibilidad, eficiencia, robustez, tolerancia a fallos y mantenibilidad.

2.3.1 Temporalidad

Los resultados deben ser correctos, no solo por sus valores sino también en el instante en que se producen. Por lo tanto el sistema operativo debe proveer mecanismos que permitan agregar restricciones de tiempos con diferente criticalidad. La criticalidad está estrechamente relacionada con las clases de plazos asociadas a las tareas de tiempo real: duros, firmes o blandos.

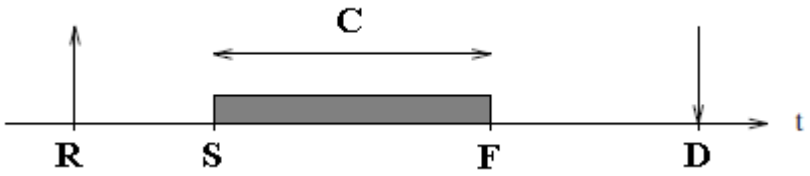
Se modela el pasaje del tiempo como una linea dirigida que se extiende desde el pasado hacia el futuro. Un corte en la línea de tiempo se le llama *instante*. Cualquier ocurrencia que implique un cambio de estado que suceda en un instante se llama *evento*. El presente como punto en el tiempo, el ahora, es un evento especial que separa el pasado del futuro (el modelo presentado de tiempo esta basado en física Newtoniana y no aprueba efectos relativistas). Un intervalo en la línea de tiempo, llamado *duración*, esta definido por dos eventos, el evento de comienzo y de fin del intervalo. Un reloj digital particiona la linea de tiempo en una secuencia de duraciones iguales, llamados gránulos del reloj, delimitados por eventos periódicos especiales, los ticks del reloj.

El parámetro temporal más importante de un sistema de tiempo real asociada a una tarea es el *plazo* (deadline), el cual define el instante de tiempo límite para que una tarea produzca los

resultados. Un *incumplimiento de plazos* (deadline miss) es cuando la tarea no llega a responder antes del plazo.

Otros parámetros importantes de las tareas son los siguientes:

- Tiempo de activación (R) - El instante en el cual una tarea está lista para ejecutar; también llamado instante de liberación (release)
- Tiempo de computación (C) - El tiempo que necesita una tarea para ejecutarse sin interrupción
- Plazo absoluto (D) - El tiempo límite el cual una tarea debe completarse sin causar daño al sistema
- Plazo relativo - Diferencia entre el plazo absoluto y el tiempo de activación
- Tiempo de comienzo (S) - El instante que la tarea comienza a ejecutar
- Tiempo de finalización (F) - El instante que la tarea termina su ejecución
- Tiempo de respuesta - Diferencia entre el tiempo de finalización y el tiempo de activación



También es importante definir dos tipos de tareas con respecto a la regularidad de su activación. Las tareas pueden ser periódicas o aperiódicas.

Las primeras se caracterizan por tener un intervalo de tiempo entre activación constante, llamado período.

Las segundas pueden activarse en cualquier momento, de forma irregular. Si además tiene que haber transcurrido un tiempo mínimo entre activaciones (llamado tiempo de inter arribo), se le suele llamar tarea esporádica.

Para todos los tipos, la secuencia de activaciones suele ser infinita.

2.3.2 Predictibilidad

Una de las propiedades más importantes que debe tener un sistema de tiempo real duro es la predictibilidad. Basado en las características del kernel y la información asociada a cada tarea, el

sistema debería ser capaz de predecir la evolución de las tareas y garantizar por adelantado que todas las restricciones temporales se cumplan [4].

La falta de predictibilidad suele tener como consecuencia una catástrofe o grandes costos. Se enumeran dos ejemplos conocidos:

El primer vuelo de una nave espacial fue demorado con costos considerables por un error en los tiempos, el cual ocurría durante la fase de inicialización, sobre uno de los procesadores redundantes que se ocupan del control de la aeronave. A pesar de los minuciosos test hechos sobre el sistema, no se descubrió el error. Al día siguiente se descubrió el problema y tenía una probabilidad de 1 en 65 de que ocurra durante la inicialización del sistema [4].

El 25 de febrero de 1991, el radar detectó un misil Scud dirigido a Arabia Saudita y el sistema predijo su trayectoria, ejecutó las verificaciones clasificándolo como una falsa alarma; luego el misil cayó sobre la ciudad de Dhahran. Se descubrió luego que a causa del manejo de una interrupción larga que corría con las interrupciones desactivadas, el reloj del sistema perdía algunas interrupciones de reloj, acumulando una demora de 57 microsegundos por minuto. El sistema ese día estuvo encendido más de 100 horas (fue una situación excepcional), por lo cual acumuló una demora de 343 milisegundos. Esta demora hizo que en la fase de verificación se generara un error de predicción de 687 metros. El error se corrigió al día siguiente, insertando código en algunos puntos del código del manejador de la interrupción larga para permitir la apropiación del procesador (preemptible) [4].

Un concepto importante asociado a la predictibilidad es el determinismo. Es una propiedad de una computación que hace posible predecir en el futuro su resultado, conociendo su estado inicial y sus entradas. Una computación es determinada o no determinada [6].

Por ejemplo, consideremos un sistema de frenos en un auto tolerante a fallas. El sistema tiene tres entradas idénticas por canales independientes pero sincronizados, y las entradas son presentadas a cuatro actuadores inteligentes, uno para cada rueda. Los actuadores seleccionarán la entrada correcta y más temprana utilizando una ventana de aceptación, cuyo tamaño está determinado en función del jitter y las velocidades de ejecución de los tres canales, asumiendo que funcionan correctamente. Todos los canales determinísticos cargarán su resultado antes del final de la ventana de aceptación; si un canal falla, el resultado de este canal será diferente al de los otros dos (fallo por valor) o solo dos resultados idénticos llegarán a la ventana de aceptación (fallo temporal). Luego el resultado que esté más veces repetido se tomará como válido, enmascarando los fallos.

El comportamiento determinístico de un componente es deseable por las siguientes razones:

- Una fuerte vinculación entre estado inicial, entrada, salida y tiempo simplifica el entendimiento del comportamiento de una tarea de tiempo real.
- Dos tareas que comienzan con el mismo estado y reciben la misma entrada producirán los mismos resultados casi al mismo tiempo. Si se utiliza un canal que puede tener muchas fallas, se pueden enmascarar los resultados fallidos como se describió en el ejemplo anterior,

utilizando varios canales.

- La testeabilidad de la tarea está simplificada, porque cada caso de prueba se puede reproducir, eliminando la aparición de errores Heisenbug, los cuales son errores que son muy difíciles de detectar porque para que ocurra, la entrada y el instante en que se lee la entrada tienen que ocurrir en un momento exacto, relacionado a los tiempos de otras actividades del sistema [6].

El primer componente que afecta a la predictibilidad de la planificación de los procesos es el propio procesador. Sus características internas, como el pipelining, memoria cache, lectura de instrucciones y el acceso directo a memoria, son las principales fuentes de no determinismo, a pesar de que mejoran el desempeño del procesador. Estas fuentes de no determinismo dificultan la estimación de los tiempos de ejecución del peor caso o WCET (worst case execution time).

Otras causas de no determinismo dependen de las características internas al kernel, como el algoritmo de planificación, los mecanismos de sincronización, los tipos de semáforos, las políticas en el manejo de la memoria y los mecanismos para el manejo de interrupciones.

2.3.3 Tolerancia a fallos

Los requerimientos de confiabilidad y seguridad en los sistemas de tiempo real son mucho más exigentes que en una aplicación estándar. Por ejemplo un programa que computa una solución a algún problema científico, si éste falla puede ser razonable abortar el programa puesto que sólo se ha perdido tiempo de procesamiento, pero en un sistema de tiempo real puede ser inaceptable. El sistema ante la ocurrencia de un fallo debería proveer un servicio degradado para poder evitar el mayor daño posible en su entorno. Si hablamos de un sistema en un avión, ante un fallo debería permitir minimamente al piloto eyectar, por lo cual si el sistema deja de funcionar sería incapaz de hacerlo.

Se pueden definir varios niveles de tolerancia a fallos en un sistema, los cuales son:

- **Tolerancia a fallos completa** - El sistema continúa su ejecución aún con la presencia de fallos sin degradar su funcionalidad o desempeño, aunque sea por un tiempo limitado.
- **Degradación elegante de fallos** (graceful) - El sistema continúa su ejecución luego de la ocurrencia de uno o varios errores, aceptando una degradación parcial de funcionalidad o desempeño durante su recuperación o reparación.
- **A prueba de fallos** (fail safe) - El sistema mantiene su integridad, aceptando una interrupción temporal de sus operaciones.

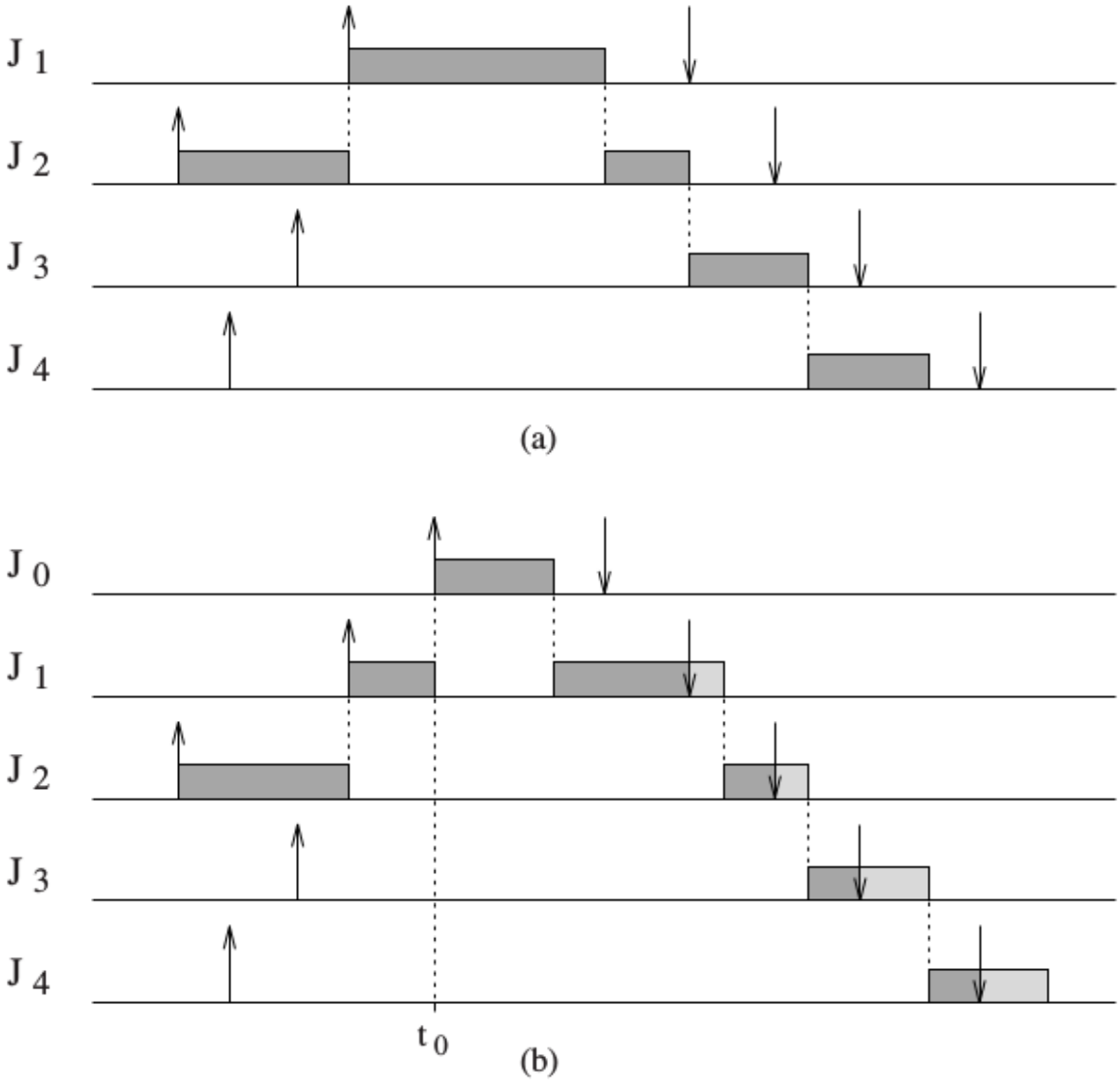
El nivel de tolerancia a fallos requerido dependerá de la aplicación. A pesar de que la mayoría de

los sistemas de tiempo real duro en teoría requieren de una tolerancia a fallos completa, en la práctica suele ser suficiente para algunos de ellos utilizar degradación elegante de fallos. En algunos casos puede ser necesario reiniciar el sistema para dejarlo en un estado seguro.

Un sistema de computación en el cual una tarea no dispone del tiempo suficiente de computación para realizar una tarea, por no tener el procesador disponible el tiempo necesario para completarse, se dice que está sobrecargado.

Un ejemplo de sobrecarga en sistemas de tiempo real es cuando se activan simultaneamente multiples tareas aperiódicas (por la ocurrencia de múltiples eventos) y el sistema no está preparado para manejar todas estas tareas simultáneamente.

Uno de los peores escenarios de sobrecarga es el Efecto Dominó y puede ocurrir utilizando planificación EDF (plazo más próximo primero o earliest deadline first, se tratará en la siguiente sección), en el cual la ejecución de una tarea extra puede ocasionar un incumplimiento de plazos de todas las demás tareas. Se muestra un ejemplo en la siguiente imagen:

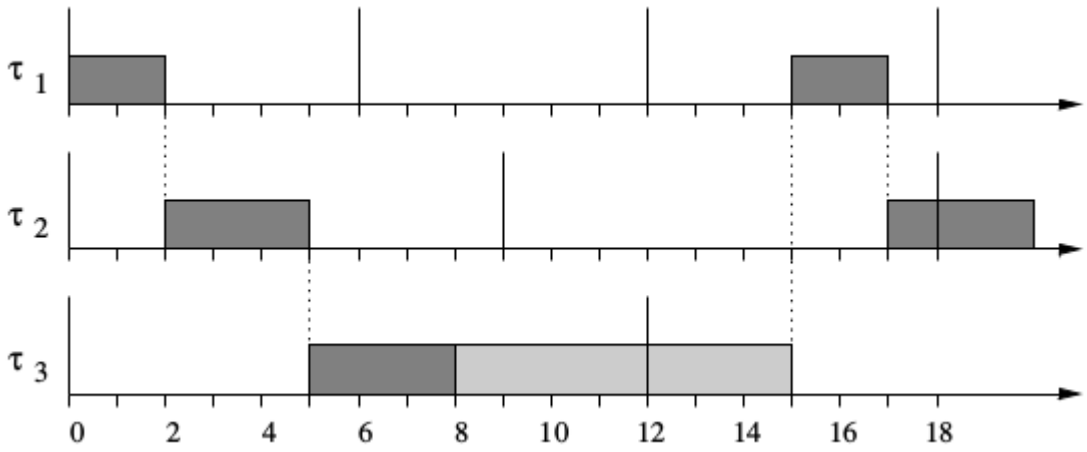


Se observa en la imagen que en (a) las tareas ejecutan casi de forma sobrecargada, puesto que terminan de ejecutar muy cerca del plazo definido para cada una. En (b) la llegada de la tarea 0 (job 0) apropia el procesador a la tarea 1, por tener el plazo más próximo, ocasionando que las tareas 1,2,3 y 4 no cumplan sus plazos.

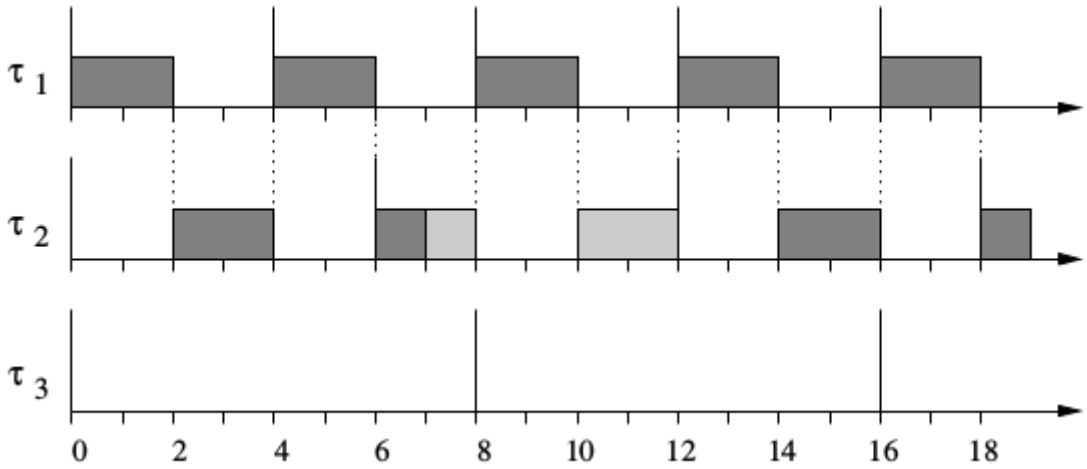
Por otro lado, una tarea que excede el tiempo esperado de ejecución en realizar su trabajo se dice que es una tarea excedida (task overrun) o que tuvo un exceso de costo (cost overrun).

En un sistema que ejecuta varias tareas periódicas de baja prioridad y luego tiene que ejecutar una tarea aperiódica o periódica con mayor prioridad; un exceso de costo esporádico de esta tarea puede provocar una sobrecarga en el sistema y las tareas periódicas no cumplan sus plazos. Utilizando planificación de tasa monotónica (rate monotónic), la tarea excedida solo afectará a tareas que tienen menor prioridad; utilizando planificación EDF, la tarea excedida puede afectar a todas las tareas (los mencionados algoritmos de planificación se verán en la siguiente sección). A este tipo de sobrecarga también es llamada sobrecarga transitoria (transient overload), puesto que ocurre de forma esporádica.

En la siguiente imagen se ilustra un ejemplo con planificación EDF. Se puede observar que la tarea 3 tiene un exceso de costo que hace incumplir los plazos de las tareas 1 y 2.



Otro tipo de sobrecarga es el que ocurre de forma constante, o también llamado sobrecarga permanente. Ocurre cuando hay tareas periódicas que provocan incumplimiento de plazos y hasta la inanición de otras tareas. Esto puede suceder a una mala estimación de los requerimientos de ejecución o por la activación de tareas periódicas no esperadas. En la siguiente imagen se muestra una sobrecarga permanente utilizando planificación de tasa monotónica; la tarea 1 provoca que la tarea 2 incumpla sus plazos y la tarea 3 no llegue a ejecutarse nunca (inanición).



2.4 Algoritmos de planificación

La planificación (scheduling) consiste en la ordenación de la ejecución de tareas (threads o procesos), de manera que los recursos de hardware (procesadores, redes, etc.) y software (objetos de datos compartidos) sean utilizados de una manera eficiente y predecible [4][5].

Un planificador consiste en dos componentes:

- Un algoritmo para ordenar la utilización de los recursos (políticas de planificación)
- Un medio de predecir el comportamiento en el peor caso cuando se aplica la política y el mecanismo (análisis de planificación o factibilidad).

En esta sección se verán los algoritmos de planificación más importantes y populares utilizados en sistemas de tiempo real.

2.4.1 Planificación de prioridad fija (FPS)

La planificación de prioridad fija o FPS (fixed priority scheduling) es una de las planificaciones más populares y utilizadas por los sistemas de tiempo real. Se dice que tiene asignación estática de prioridades puesto que son conocidas antes de la ejecución de la aplicación.

La política de planificación debe cumplir las siguientes reglas:

- Las tareas son asignadas de forma estática a los mismos procesadores, puesto que la ejecución de una tarea es más predecible cuando ejecuta siempre en el mismo procesador
- Las tareas que ejecutan en un mismo procesadores se ordenan de acuerdo a una prioridad

- Las prioridades de cada tarea son asignadas al momento de creación de cada una a criterio del programador
- Se implementa herencia de prioridades para el acceso a los recursos
- Siempre se ejecutará la tarea en estado listo que tenga la mayor prioridad. Por lo tanto siempre que haya una tarea con mayor prioridad que la que está ejecutando, la primera se apropiará de la segunda.

Existe una asignación de prioridades llamada **tasa monotónica**, el cual consiste en asignar una única prioridad a cada tarea basada en su período: cuanto más corto el período, mayor la prioridad. Liu y Layland demostraron que la asignación es óptima puesto que para cualquier conjunto de tareas, si son asignables por algún planificador basado en prioridades fijas, también son asignables por tasa monotónica. Notar que esta asignación de prioridades óptima solo es válida para tareas periódicas.

Existen diversas técnicas para analizar si las tareas de un sistema basado en prioridades cumplirá sus plazos. Una de las más flexibles es el análisis de tiempos de respuesta.

2.4.2 Plazo más próximo primero (EDF)

El algoritmo de planificación del plazo más próximo primero, también llamado EDF (earliest deadline first) es un algoritmo que posee una regla de asignación dinámica que selecciona tareas de acuerdo a los plazos absolutos. A diferencia de la planificación FPS que tiene asignación estática de prioridades, la planificación EDF asigna prioridades de forma dinámica durante la ejecución de la aplicación.

A diferencia de la planificación FPS, EDF no asume nada acerca de si las tareas deben ser periódicas o no; por lo tanto se puede utilizar para planificar tanto tareas periódicas como aperiódicas.

3 El lenguaje C con POSIX de tiempo real

El lenguaje C, desarrollado por Dennis Ritchie en 1972 en los laboratorios Bell, es el lenguaje de programación más utilizado por la comunidad de sistemas embebidos, tanto para programación de sistemas como para el desarrollo de aplicaciones. A pesar de haber sido sobrepasado como lenguaje de propósito general por el lenguaje Java, sigue siendo el lenguaje favorito de los desarolladores de sistemas embebidos por la eficiencia del código generado, la simplicidad y la amplia disponibilidad de compiladores y herramientas de desarrollo, teniendo en cuenta que el lenguaje en sí hace poco por promover la robustez y la confiabilidad del código producido [3][8].

El lenguaje C++, como superconjunto de C con orientación a objetos, también se utiliza para el desarrollo de aplicaciones embebidas, aunque la mayoría prefiere utilizar el lenguaje C.

Sin embargo, tanto C como C++ no proveen soporte para desarrollar sistemas de tiempo real, principalmente porque no ofrecen facilidades para programar aplicaciones concurrentes. Para que sea posible habría que depender de librerías externas que provean servicios para construir aplicaciones concurrentes con comportamiento temporal predecible. Estos servicios pueden depender en kernels modificados o sistemas operativos de tiempo real (RTOS).

Para que el desarrollador pueda preservar su inversión en el desarrollo de software, es importante que los servicios ofrecidos por los sistemas operativos de tiempo real implementen una API estándar que permita portar aplicaciones a diferentes plataformas. Una API que tiene soporte para sistemas de tiempo real, adoptada por los proveedores más importantes de sistemas operativos de tiempo real y con aprobación internacional, es POSIX; es el acrónimo de Portable Operating System Interface (interfaz para sistemas operativos portables) y es una estandarización del sistema operativo UNIX (por eso la X).

POSIX provee una interfaz entre el sistema operativo y el desarrollo de aplicaciones, describiendo sus servicios y la sintaxis y semántica de sus interfaces en términos de sus tipos de datos y prototipos de funciones. La implementación de cada uno de estos servicios es responsabilidad del proveedor de sistemas operativos. Dado que el estándar define las interfaces al nivel del código fuente, la portabilidad obtenida es también al nivel del código fuente.

Se enumeran los servicios más importantes ofrecidos por la API de POSIX:

- Manipulación de threads
- Políticas de planificación basados en prioridades fijas con apropiación (preemptive fixed priority o FPS)
- Sincronización de tareas mediante mutexes y semáforos
- Mecanismos para evitar la inversión de prioridad: prioridad de techo (priority ceiling) y herencia de prioridad (priority inheritance)

- Variables condicionales sincronizadas con señales y esperas (signal and wait) para la implementación de monitores
- Distintos tipos de relojes

El lenguaje C no posee manejo de excepciones, aunque es posible utilizar las facilidades de macros del lenguaje para simularlo. Para implementar un modelo de terminación similar a Java, es necesario guardar el estado de los registros del programa y luego recuperarlos si ocurre una excepción. Para esto se utilizan las facilidades de POSIX `setjmp` y `longjmp`. Se utiliza el siguiente código como estructura [3]:

```
// comienza el dominio de la excepción
typedef char *exception;
// representa una excepción llamada "error"
exception error = "error";


if ((current_exception = (exception) setjmp(save_area)) == 0) {
   // guarda los registros en save_area
   // la región guardada
   // cuando se identifica la excepción "error"
   longjmp(save_area, (int) error);
} else {
   if (current_exception == error) {
         // manejador del error
   } else {
         // relanzar la excepción
   }
}
```

Como el código de arriba es muy complicado de entender, se define el conjunto de macros NEW_EXCEPTION, BEGIN, EXCEPTION, END, RAISE, WHEN y OTHERS para ayudar a estructurar el programa. Luego se pueden utilizar como se hace en el siguiente ejemplo:

```
NEW_EXCEPTION(excepcion1)
NEW_EXCEPTION(excepcion2)
NEW_EXCEPTION(excepcion3)

BEGIN
   // sentencias que pueden causar las excepciones de arriba
   // por ejemplo:
   RAISE(excepcion1)
EXCEPTION
   WHEN(excepcion1)
         // manejador de la excepción "sensor_alto"
   WHEN(excepcion2)
         // manejador de la excepción "sensor_bajo"
   WHEN(OTHERS)
         // manejador de cualquier otra excepción no capturada antes
END;
```

Finalmente, el lenguaje C, gracias a la API de POSIX utilizada en sistemas operativos de tiempo real, se vuelve un lenguaje apto para la programación aplicaciones de tiempo real y uno de los más utilizados. Sin embargo, el lenguaje carece de muchas características del lenguaje que hacen más tediosa y difícil la tarea de desarrollar aplicaciones de tiempo real.

4 El lenguaje Java

Java es un lenguaje de programación de propósito general, orientado a objetos, creado por James Gosling, Patrick Naughton, Chris Warth, Ed Frank y Mike Sheridan en Sun Microsystems en 1991. Se tardaron 18 meses para obtener la primer versión. Inicialmente se lo nombró "Oak" (roble) pero fue renombrado a "Java" en 1995. Después de exponer la primera implementación en 1992 y el lanzamiento al público en 1995, muchas personas contribuyeron al diseño y evolución de este lenguaje [5][9][10].

Inicialmente el propósito y el ímpetu de Java no era Internet, sino que fuera un lenguaje independiente de la plataforma (esto es, neutral a la arquitectura) que pueda utilizarse para crear software embebible en diversos dispositivos electrónicos, como hornos microondas y controles remotos. El problema que tenían el software programado en C/C++ (y en la mayoría de los lenguajes) es que estaban diseñados para ejecutar sobre un sistema (o target) específico. Se comenzó entonces a crear un lenguaje que sea portable e independiente de la plataforma. Mientras se estaba desarrollando, estaba emergiendo lo que jugaría un papel crucial en el futuro de Java, la World Wide Web. Si no fuese por el hecho de que Java se fue desarrollando a medida que la WWW empezaba a tomar forma, hubiese quedado como un lenguaje útil pero oscuro en el mundo de los dispositivos embebidos.

La clave por la cual Java resuelve los problemas de portabilidad es gracias a la utilización de *bytecode*, en lugar de código nativo como hacen otros lenguajes. Un programa fuente de Java se compila generando un bytecode; luego existe lo que se llama máquina virtual de Java (Java Virtual Machine o JVM) y se encarga de ejecutar el código especial de los bytecodes. Luego para que este programa se ejecute en otra arquitectura es necesario que exista una JVM implementada para esa arquitectura. Luego los programas podrán ejecutarse en cualquier JVM con el mismo código. Esta característica suele llamarse "write once, run anywhere" (escribir una vez, ejecutar en cualquier lugar).

En Java, todas las clases que existen heredan de la clase raíz `Object`. Todo es un objeto salvo los tipos primitivos como `int`, `long`, `boolean`, `float`, `double` y `char`. Por este solo hecho se dice que el lenguaje Java no es un lenguaje orientado a objetos puro.

En este capítulo se mostrarán algunas de las características más importantes del lenguaje Java y al final del capítulo se mostrarán los problemas que tiene Java por lo cual no es apto para el desarrollo de aplicaciones de tiempo real; para que sea apto, se creó la especificación RTSJ que será tratado en el siguiente capítulo.

4.1 Interfaces

Java soporta herencia simple de objetos a nivel de clases (solo una superclase posible por clase) y herencia múltiple a nivel de interfaces (múltiples superinterfaces para una clase o interfaz). A continuación se definen varias clases e interfaces.

```
public class A {
   public void metodoUno() { .. }
}
public class B extends A {
   public void metodoDos() { .. }
}
interface I {
   void metodoTres();
}
interface J {
   void metodoCuatro();
}
public C extends B implements I,J {
   public void metodoTres() { .. }
   public void metodoCuatro() { .. }
}
```

En el ejemplo se ve que C hereda todos los métodos de la clase B, y a la vez éste hereda todos los métodos de la clase A. Además, la clase C implementa dos interfaces: I y J. Esto obliga a la clase C a definir los métodos definidos en las interfaces que implementa: `metodoTres()` y `metodoCuatro()`.

Las interfaces son útiles para definir las mismas operaciones a un grupo de clases. Por ejemplo, si se tienen cinco clases que implementan la interfaz I, las cinco clases tendrán definido el mismo método `metodoTres()`. Es un mecanismo ampliamente utilizado que ofrece mayor claridad y flexibilidad, permitiendo utilizar herencia múltiple de interfaces.

4.2 Excepciones

Una excepción puede ser definida como la ocurrencia de un error. Cuando se invoca una operación que resulta en un error y se notifica al invocador, se dice que la operación levanta una excepción; se dice que el invocador maneja la excepción si produce una respuesta al error dado por ejecutar la operación, o de forma similar se dice que el invocador define un manejador de la excepción. Finalmente, un manejador de excepciones provee un mecanismo de recuperación de errores, puesto que define el comportamiento a tomar cuando ocurre un error. El manejo de

excepciones provee un mecanismo para que el sistema sea tolerante a fallos.

El lenguaje Java ofrece de forma nativa el manejo de excepciones y está integrado al modelo de objetos. Todas las excepciones en Java son objetos que heredan de la clase `Throwable`. Existen distintos tipos de excepciones representadas por las siguientes clases: `Error`, `Exception` y `RuntimeException`. Las clases `Error` y `Exception` heredan de `Throwable`, y la clase `RuntimeException` hereda de `Exception`.

Las excepciones representadas por la clase `Error` son generalmente errores internos a la máquina virtual de Java o errores por el agotamiento de recursos, por ejemplo: `OutOfMemoryError`, `StackOverflowError`. Cuando ocurre uno de estos errores, poco se puede hacer para mantener la integridad del sistema.

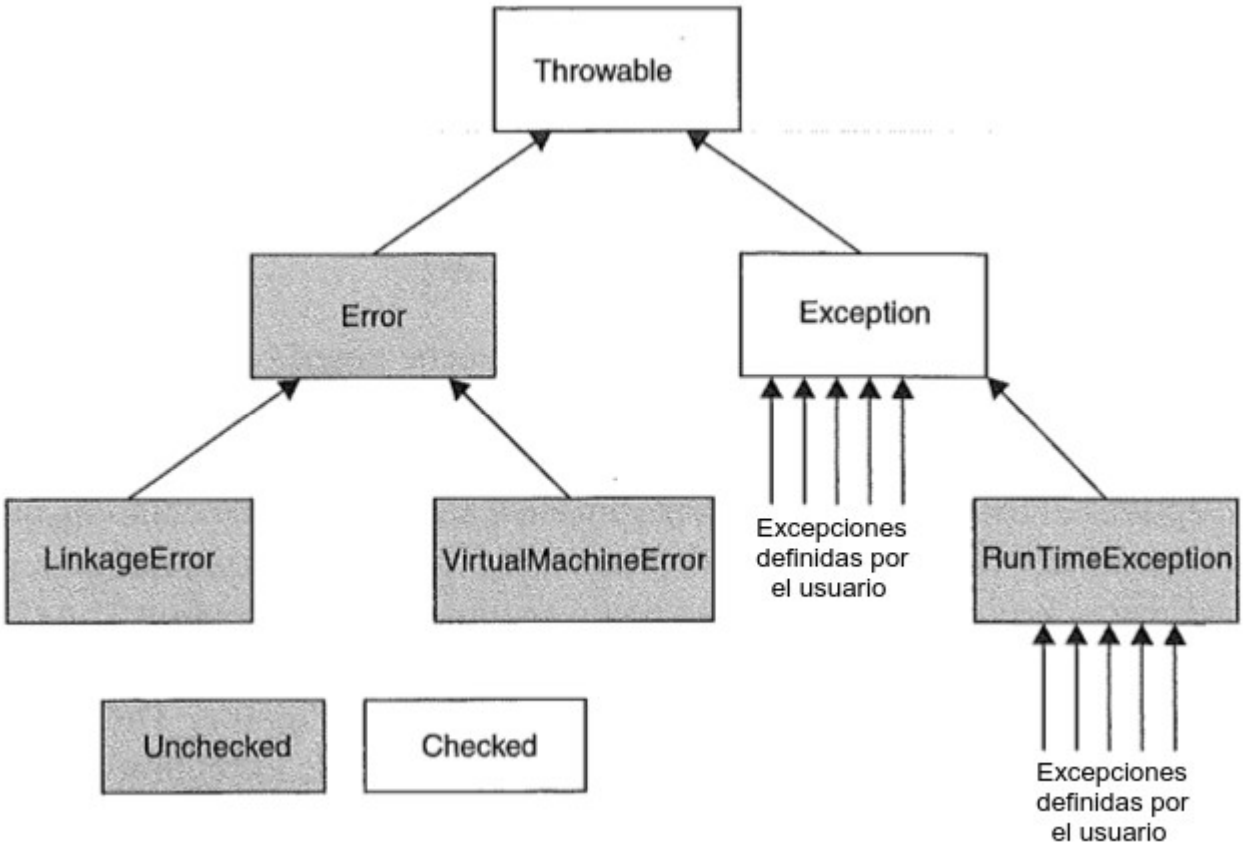
Las clases `Exception` y `RuntimeException` representan a las llamadas excepciones comprobadas y no comprobadas respectivamente. Las comprobadas son excepciones que obligan al programador a definir un manejador de excepciones para cada una de ellas, resultando en un error de compilación si no se define, y si no se maneja se debe indicar al método que éste puede levantar una excepción.

Para levantar una excepción en Java, se utiliza la cláusula `throw`; por ejemplo la siguiente línea de código indica que se debe levantar una excepción de tipo `Exception`:

```
throw new Exception();
```

Luego el usuario puede crear sus propias excepciones extendiendo de las clases `Exception` y `RuntimeException`. Toda clase que extienda a `Exception` o a una de sus subclases será una excepción comprobada (checked); si extiende de la clase `RuntimeException` o una de sus subclases será una excepción no comprobada (unchecked). En ámbos casos se pueden definir constructores de excepciones que reciban parámetros.

En la siguiente imagen se ilustra la jerarquía de las excepciones de Java:



Si un método levanta explícitamente una excepción con la clausula `throw`, se pueden hacer dos cosas: capturar la excepción con un manejador mediante las cláusulas `try/catch` o indicar que el método va a levantar la excepción con la cláusula `throws`. También se puede definir un comportamiento común, independiente si ocurrió una excepción o no, con la cláusula `finally`.

Veamos el siguiente ejemplo:

```
try {
   <código que puede lanzar una excepción>
} catch (MyException1 e) {
   <código del manejador de la excepción>
} catch (MyException2 e) {
   <código del manejador de la excepción>
} catch (Exception e) {
   <código del manejador de la excepción>
}
```

```
finally {
   <código que se ejecuta siempre, haya o no ocurrido una excepción>
}
```

Cuando se ejecuta un bloque `try/catch`, se asume que dentro de este bloque pueden ocurrir una o varias excepciones. Si ocurriese una excepción, se ejecuta el manejador que corresponda a la excepción lanzada. En el ejemplo se muestran 3 manejadores. Cada manejador define la clase de la excepción a capturar, y como norma capturará también a las subclases de la excepción; por lo tanto un manejador que capture `Exception` capturará todas las excepciones. Finalmente, dentro del bloque `finally` habrá código que siempre se va a ejecutar.

Java permite propagar las excepciones a través de los métodos según la pila de ejecución; por ejemplo si un método A llama a un método B y éste a un método C, y C levantan la excepción E, ésta puede ser capturada en el método A o B. Si la excepción E es comprobada y la excepción se captura en el método A, el lenguaje obliga a definir a los métodos B y C en su firma que pueden levantar la excepción E, mediante la cláusula `throws`. Si es no comprobada no es necesario.

A continuación se ilustra el ejemplo:

```
void metodoA() {
   try {
         this.metodoB();
   } catch (ExcepcionE e) {
         <manejador de la excepción>
   }
}
void metodoB() throws ExcepcionE {
   this.metodoC();
}
void metodoC() throws ExcepcionE {
   <codigo>
   if (<condición de error>) {
         throw new ExcepcionE();
   }
```

```
}
```

En el ejemplo la excepción se propaga desde el método C al método A donde se captura la excepción E.

Por último, las excepciones, al ser objetos, permiten ser creadas con parámetros, puesto que se pueden definir constructores como a cualquier otra clase.

En el siguiente ejemplo se muestra la definición de una excepción con parámetros:

```
public class MiExcepcion extends Exception {
    private int miValor;

    public MiExcepcion(int miParametro) {
        this.miValor = miParametro;
    }

    public int getValor() { return miValor; }
}
```

Luego, para lanzar esta excepción:

```
int a = ..
...
throw new MiExcepcion(a);
```

Finalmente, se puede acceder al parámetro mediante un manejador de esta excepción:

```
try {
    ...
} catch (MiExcepcion e) {
    System.out.println(e.getValor());
```

```
}
```

Las excepciones ofrecen una facilidad al programador que le permite separar la lógica del programa por un lado, y por otro manejar los errores ocurridos en la mencionada lógica.

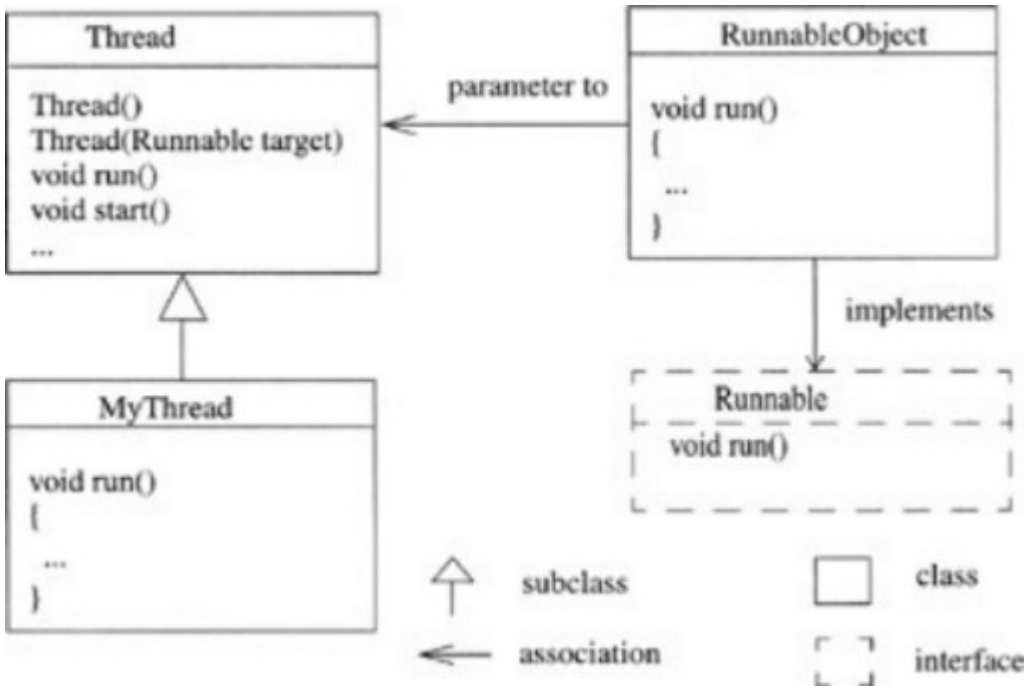
Más adelante, se verá que el modelo de interrupciones explicado aquí requiere de algunas modificaciones para soportar el mecanismo de transferencia de control asincrónica definido en la especificación RTSJ. La modificación también implica cambios en la manera que se interrumpe un thread. En la siguiente sección se verá el modelo de threads de Java, para luego entender las modificaciones propuestas por la especificación.

4.3 Threads y sincronización

El modelo de concurrencia de Java está integrado dentro de su framework orientado a objetos, por lo cual el manejo de threads de Java es nativo del lenguaje. Existe la clase `Thread` para representar threads o tareas; toda instancia de `Thread` o cualquiera de sus subclases creará un thread al momento de crearse la instancia.

El código que ejecutará un thread se puede especificar de dos maneras: a través del método `run()` de la clase `Thread` por lo cual se deberá subclasificar la clase e implementar este método, o mediante un objeto separado que implemente la interfaz `Runnable`, el cual define un único método `run()`; luego este objeto se debe pasar como parámetro al constructor de la clase `Thread` o alguna de sus subclases. Finalmente, para que un thread comience su ejecución se utilizará el método `start()`.

A continuación se muestra el modelo de clases relacionado a los threads:



La comunicación entre threads se realiza leyendo y escribiendo en objetos compartidos, por lo cual hay que tener cuidado de proteger estos objetos para evitar inconsistencias cuando varios threads quieren escribir sobe ellos en forma simultánea.

Para lograrlo, Java maneja el concepto de *monitor*. Un monitor es una entidad que provee un mecanismo de sincronización llamado *cerrojo* (lock) para implementar exclusión mutua para el acceso a un recurso compartido. El cerrojo solo puede ser apropiada por un único thread a la vez. Cada uno de los objetos de Java tiene asociado un cerrojo, y puede ser apropiado y liberado por los threads. Al proceso que permite restringir el acceso a recursos compartidos a un único thread al mismo tiempo se le llama sincronización.

Java define la palabra clave `synchronized` y se puede utilizar de dos maneras: sobre un método de una clase o definiendo un bloque de código con una clase. A cada uno se lo llama método sincronizado y bloque sincronizado respectivamente.

Cuando un thread quiere ejecutar un método sincronizado, primero verificará que *el cerrojo asociado a la clase que contiene el método sincronizado* esté libre (no esté apropiada por ningún otro thread). Si está libre, se apropia del cerrojo y ejecuta el método; cuando termina de ejecutar este método, libera el cerrojo y desbloquea a los threads que estaban esperando para apropiarse de este cerrojo. De forma similar, cuando un thread quiere ejecutar un bloque sincronizado, el cerrojo que deberá apropiar está asociada al objeto asociado al bloque sincronizado.

Se muestra un ejemplo a continuación, con tres métodos sincronizados y un bloque sincronizado. Tener en cuenta que si un thread intenta ejecutar *cualquiera de los tres métodos sincronizados*, sólo uno ejecutará a la vez, porque es un cerrojo por objeto. El bloque sincronizado define al objeto miCerrojo para utilizarlo como llave.

```
public class A {
   private Object miCerrojo = new Object();
   private int a = 10;
   private int b = 20;
   public synchronized int obtenerSuma() {
         return variableCompartidaA + variableCompartidaB;
   }
   public synchronized void setearA(int valor) { this.a = valor; }
   public synchronized void setearB(int valor) { this.b = valor; }
   public void otroMetodo() {
         // codigo no protegido
         synchronized(miCerrojo) {
               // codigo protegido
         }
   }
}
```

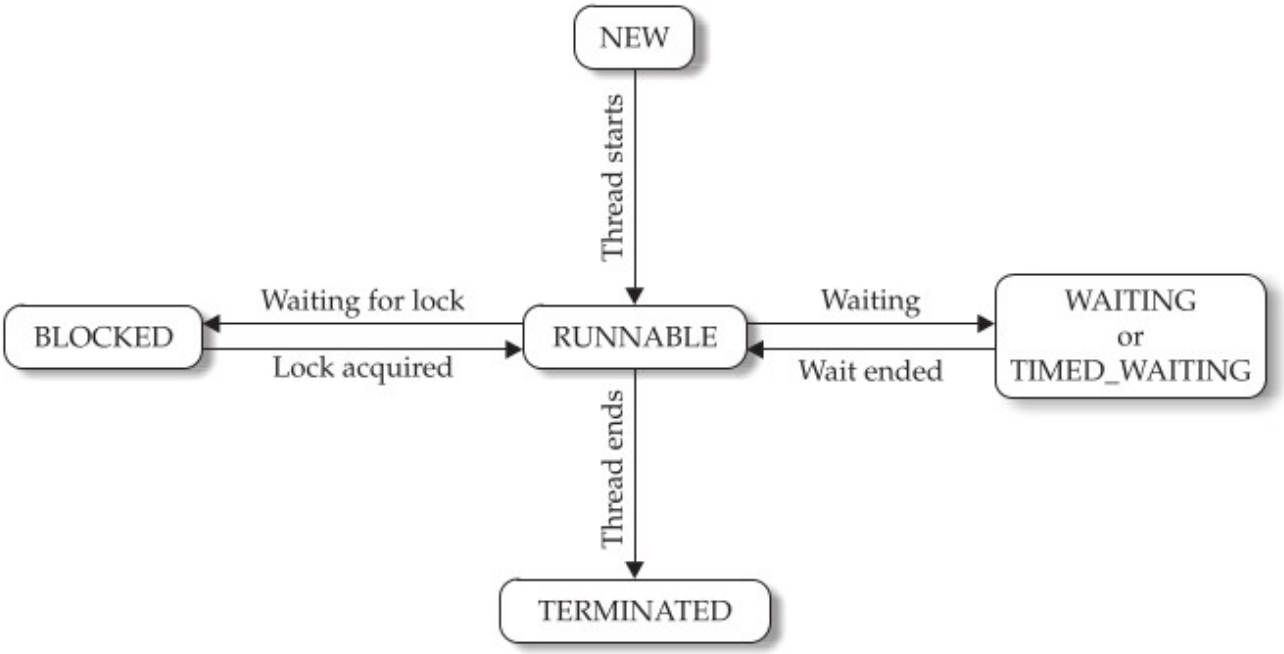
Otros métodos de la clase Thread importantes con respecto al flujo de la ejecución:

- `join()` y `join(long millis)` - Si el thread t1 invoca este método sobre otro thread t2, t1 se bloqueará hasta que t2 haya terminado su ejecución. Es posible pasarle un tiempo máximo de espera como parámetro si se desea este comportamiento (timeout).
- `sleep(long millis)` - Es un método estático que bloquea el thread por una cantidad de milisegundos.
- `wait()` - Utilizable dentro de un método o bloque sincronizado. Provoca que el thread que lo ejecuta se bloquee y libere el cerrojo asociado al monitor
- `notify()` y `notifyAll()` - Ídem anterior. El primer método desbloquea algún thread que haya quedado bloqueado con `wait()` y el segundo desbloquea a todos los threads si los hay.

Se puede obtener el estado de un thread mediante un enumerativo accesible desde la clase thread y el método `getState()`:

```
Thread.State.getState();
```

Los posibles estados de un thread se muestran en el siguiente diagrama:



Se explica a continuación cada estado:

- NEW - El estado inicial de un thread, al momento de crear la instancia.
- RUNNABLE - Un thread pasa a este estado cuando se invoca al método start() y el thread estaba en estado NEW. Puede estar ejecutando o próximo a ejecutar cuando tenga control de la CPU.
- BLOQUED - Pasa a este estado cuando está esperando obtener un cerrojo; la ejecución queda suspendida. Una vez obtenido el cerrojo, el thread pasa al estado RUNNABLE.
- TIMED_WAITING - Cuando el thread se bloquea por la llamada de los métodos `sleep()`, `join()` y `wait()` por un tiempo definido al momento de invocar el método (métodos con timeout). Cuando haya pasado el tiempo, y en el caso de `join()` y `wait()` si el evento que están esperando ocurre antes del timeout, el thread pasa al estado RUNNABLE.
- WAITING - Cuando el thread está esperando por un evento a partir de los métodos wait() y join(). Al cumplirse el evento el thread pasa al estado RUNNABLE.

Los threads de Java tienen asociada una prioridad. Por defecto, tomará la prioridad intermedia 5 o la prioridad del thread padre que la creó. Posteriormente se puede modificar la prioridad, en un

rango de 1 a 10. También existen constantes para obtener las prioridades mínima, máxima y la intermedia por defecto. Un thread puede ceder explícitamente el control del procesador para que el planificador elija otra tarea para ejecutarse.

Se listan a continuación los métodos de la clase `Thread` relacionados a lo recientemente explicado:

- `int getPriority()` - Obtiene la prioridad actual
- `setPriority(int priority)` - Asigna una nueva prioridad
- `yield()` - Cede el control del procesador
- `int Thread.MIN_PRIORITY` - Devuelve la mínima prioridad (1)
- `int Thread.MAX_PRIORITY` - Devuelve la máxima prioridad (10)
- `int Thread.NORM_PRIORITY` - Devuelve la prioridad por defecto (5)

Tener en cuenta que el planificador de Java no garantiza que el thread con mayor prioridad esté siempre ejecutándose, threads con igual prioridad puede que no estén ejecutando con time-slice, y puede que diferentes prioridades estén mapeadas a una misma prioridad en el sistema operativo.

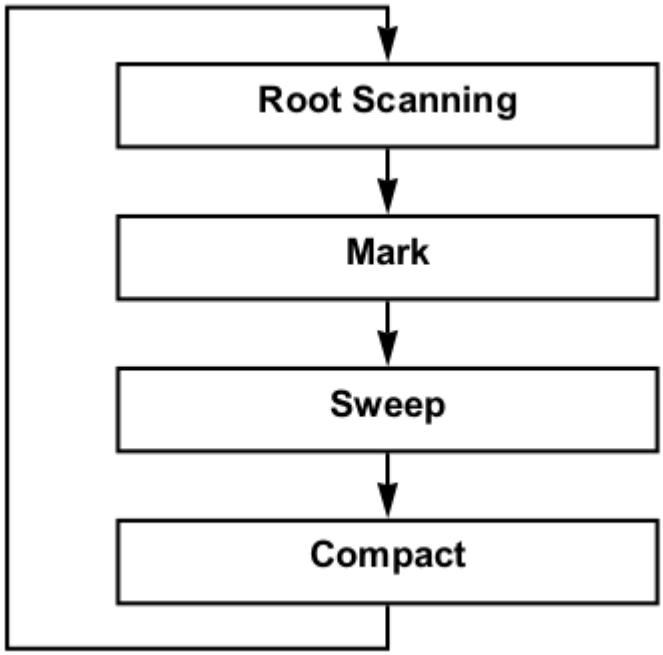
Fortalecer el planificador y el modelo de prioridades son algunos de los objetivos de la especificación RTSJ, vista más adelante.

4.4 Recolección de basura

El recolector de basura (garbage collector) es un mecanismo automático que tienen los lenguajes orientados a objetos como Java para liberar la memoria no utilizada. Protege al usuario de producir código con errores por el manejo de memoria y código difícil de mantener, como ocurre con el lenguaje C y también permite la ejecución segura de código no confiable. Uno de los mayores obstáculos de la programación orientada a objetos es la asignación y liberación explícita de la memoria en los lenguajes como C [11].

4.4.1 Algoritmo de marcar-barrer-compactar

Los algoritmos utilizados por los recolectores de basura de lenguajes orientados a objetos como Java se basan en algoritmos marcar-y-barrer (mark-and-sweep) o marcar-barrer-compactar (mark-sweep-compact). Los recolectores de basura que utilizan estos algoritmos utilizan un proceso cíclico. Un ciclo del recolector de basura consiste en varias fases como se muestra a continuación:



Las fases de cada ciclo del recolector de basura son escaneo de raíz (root scanning), de marcado (mark), de barrido (sweep) y de compactación (compact).

Fase de escaneo de raíz

Todos los objetos en la heap que son referenciados por variables raíz se marcan. Las variables raíz (root variables o variables de raíz) son variables que no están directamente almacenadas en la heap, como lo pueden ser variables de las pilas de ejecución de los threads, registros del procesador o variables globales almacenadas fuera de la heap.

Fase de marcado

Durante la fase de marcado, todos los objetos que se referencien desde los objetos marcados en la fase anterior se marcan también. Para cada objeto marcado, se marcan sus referencias hasta no encontrar más objetos para marcar. Finalmente quedarán marcados todos los objetos a partir del grafo de referencias que nace de las variables raíz.

Fase de barrido

Despues de la fase de marcado, todos los objetos que no fueron marcados se conocen como objetos inalcanzables (unreachable object). Estos objetos no se pueden usar más, por lo tanto son basura y el espacio de memoria que ocupan se puede reutilizar. Durante la fase de barrido, se lee toda la heap y todos los objetos inalcanzables se liberan, sumando memoria libre a la heap.

Fase de compactación

Identificar objetos no utilizados y liberar su memoria no es suficiente. Luego de la fase de barrido, habrá memoria liberada pero no necesariamente contigua físicamente, produciendo fragmentación en la memoria. Estos pequeños huecos de memoria imposibilitan alocar un objeto que sea mas grande que estos huecos. La solución a este problema implica agregar una fase más de compactación, en el cual la memoria que se sigue utilizando se mueva a otro espacio de memoria, de forma que quede contigua sin dejar huecos.

4.4.2 El recolector de basura de Java

La máquina virtual Hotspot JVM de Oracle utilizada en la versión Java SE 7 y 8 (Java Standard Edition) dispone de cuatro recolectores de basura diferentes: recolector serial (pensado para computadoras con una única CPU), recolector paralelo (utilizado en servidores con múltiples CPUs), recolector concurrente (diseñado para evitar pausas largas) y el recolector G1 (para heaps de más de 4 GB) [12].

Cuando un programa Java está en ejecución, la máquina virtual de Java ejecutará regularmente el recolector de basura para liberar y ordenar la memoria heap a través de uno o más threads especiales. Por lo tanto existirán dos grupos diferentes de threads ejecutando en la máquina virtual: threads de la aplicación y threads del recolector de basura.

En cualquiera de los cuatro recolectores mencionados antes, cuando los threads del recolector están ejecutando, tiene que poder acceder a la memoria que está siendo referenciada por los threads de la aplicación para poder cumplir la etapa de compactación (requiere mover los datos de un lugar a otro). Por lo tanto, los threads de la aplicación deben detener su ejecución temporalmente para que la recolección se pueda llevar a cabo, resultando en una pausa general de la aplicación. A estas pausas se las conoce como *pausas que detienen el mundo* (stop-the-world pauses). Las pausas más largas ocurren cuando la heap se queda sin memoria, forzando a la máquina virtual a realizar una recolección completa de basura (full garbage collection).

Los recolectores concurrentes y G1 son recolectores que ejecutan de forma concurrente utilizando varios threads, y minimizan las pausas largas que suelen tener los otros dos recolectores. Sin embargo, a pesar de ser pausas cortas siguen siendo pausas que detienen al mundo. Las minimizan porque mantienen más tiempo la memoria sin llenarse completamente realizando recolecciones más pequeñas, a pesar que requieren más tiempo de la CPU para ejecutar.

Ninguno de estos recolectores, por las pausas que generan en la ejecución de la aplicación, son recolectores de basura determinísticos y convierte al lenguaje Java estándar en un lenguaje no apto para el desarrollo de aplicaciones de tiempo real. Se verá más adelante que la especificación define áreas de memoria especiales que no están sujetas a la recolección de basura para permitir obtener el determinismo que no tiene Java estándar. También se verá el recolector de basura utilizado por JamaicaVM que sí es determinístico, también llamado *recolector de basura de tiempo real duro*.

4.5 Problemas de utilizar Java estándar para el desarrollo de sistemas de tiempo real

El problema más importante presente en el lenguaje Java estándar es la falta de predictibilidad de sus programas a causa del recolector de basura, que introduce demoras en la ejecución de la aplicación arbitrarias. En una aplicación de tiempo real estas demoras pueden ocasionar incumplimiento de plazos de forma regular, puesto que las demoras son muy largas (desde varios milisegundos a varios segundos).

Otro problema es el planificador que utiliza la máquina virtual de Java estándar: no garantiza que un thread de mayor prioridad vaya a ejecutarse inmediatamente, siendo éste el de mayor prioridad. Además, Java estándar define únicamente 10 niveles de prioridades, lo cual pueden llegar a ser niveles insuficientes.

Java estándar no provee mecanismos para obtener la hora o manejar tiempos menores al milisegundo, limitando cualquier plazo que se desee definir al mínimo de milisegundo, cuando muchas aplicaciones de tiempo real requieren plazos que rondan en los microsegundos.

Por último, Java estándar no provee mecanismos para acceder a la memoria física.

En el siguiente capítulo veremos la especificación para Java de tiempo real, que contempla todos estos problemas.

5 La especificación RTSJ para Java

El grupo de expertos de tiempo real para java (RTJEG), a través de la solicitud JSR-000001 del JCP (Java Community Process), tuvieron la responsabilidad de producir una especificación para extender las especificaciones del lenguaje Java y la máquina virtual, y proveer una interfaz de programación de aplicaciones (API) para hacer posible el desarrollo de aplicaciones de tiempo real sobre Java [5][2].

La especificación hace posible el uso del lenguaje Java y su máquina virtual para programar aplicaciones en sistemas de tiempo real blando y duro.

La versión de Java actual tiene algunos problemas importantes que restringen su uso en estos sistemas, como el no ser predecible su ejecución.

Un requerimiento de la especificación es que mantenga la sintaxis original de java (en este caso implica una API de java ampliada con semánticas bien definidas). Quiere decir que un programa corriente de Java puede ser ejecutada por una máquina virtual (JVM) con el RTSJ.

Uno de los problemas más críticos se encuentra en el recolector de basura (GC), que se ejecuta de forma no predecible; si en el instante que se está ejecutando el GC ocurre un evento de alta prioridad y con criticidad alta, el programa deberá esperar a que termine primero de ejecutar el GC (porque no es apropiable) y en este caso probablemente el evento será atendido demasiado tarde, ocasionando un incumplimiento de plazos. Esto significaría una catástrofe para un sistema de tiempo real duro.

5.1 Principales cambios a la especificación estándar de Java.

Las JVMs que cumplen la especificación RTSJ tienen las siguientes 8 mejoras con respecto a las JVM comunes (con sus respectivas semánticas):

- Planificación de prioridad fija (FPS)
- Relojes monotónicos
- Manejo de memoria
- Thread de tiempo real
- Manejo de Eventos Asincrónicos
- Transferencia de Control Asincrónico
- Sincronización y recursos compartidos
- Acceso a Memoria Física

La versión actual de la especificación (1.0.2) está pensada para la ejecución de programas en sistemas monoprocesador; no prohíbe su uso en sistemas multiprocesador, pero no provee ningún mecanismo que provea control directo, como por ejemplo, asignación de threads a procesadores. Cada implementación particular tendrá su manera de tratar este tema.

Al momento de la escritura existe un borrador de la versión 2.0 de la especificación, el cual ataca los problemas encontrados en la especificación actual, como lo son el recolector de basura, acceso a los dispositivos, control de múltiples procesadores como la configuración de afinidad y acceso directo a memoria (DMA).

5.2 Planificador

Java soporta la planificación de threads con un máximo de 10 niveles de prioridades, pero no garantiza que el thread con máxima prioridad vaya a ejecutarse, reduciendo de esta forma la predictibilidad de la ejecución de un determinado thread.

El los sistemas de tiempo real, es necesario asegurar la ejecución de una secuencia de instrucciones en un tiempo predecible. Distintos esquemas de planificación nombran estas secuencias de instrucciones de forma diferente, por ejemplo, tareas, threads, bloques o módulos. El planificador definido por RTSJ reconoce a los objetos llamados *objetos planificables* (schedulables).

El nuevo planificador hace un análisis de factibilidad sobre los planificables para saber si pueden cumplir con las métricas de tiempo definidas para cada uno. Cada planificable debe ser configurado previamente mediante distintos parámetros (por ejemplo, prioridad, periodicidad de la ejecución del objeto). Dependiendo de estos parámetros, el análisis de factibilidad permite saber si los planificables podrán, por ejemplo, cumplir sus plazos.

Por defecto RTSJ define un único tipo de planificador, basado en en el enfoque llamado planificación de prioridad fija apropiativa (preemptive fixed priority scheduling o FPS) y tiene 28 niveles de prioridad. Al ser apropiativo puede interrumpir la ejecución de un planificable de menor prioridad para que ejecute otro de mayor prioridad.

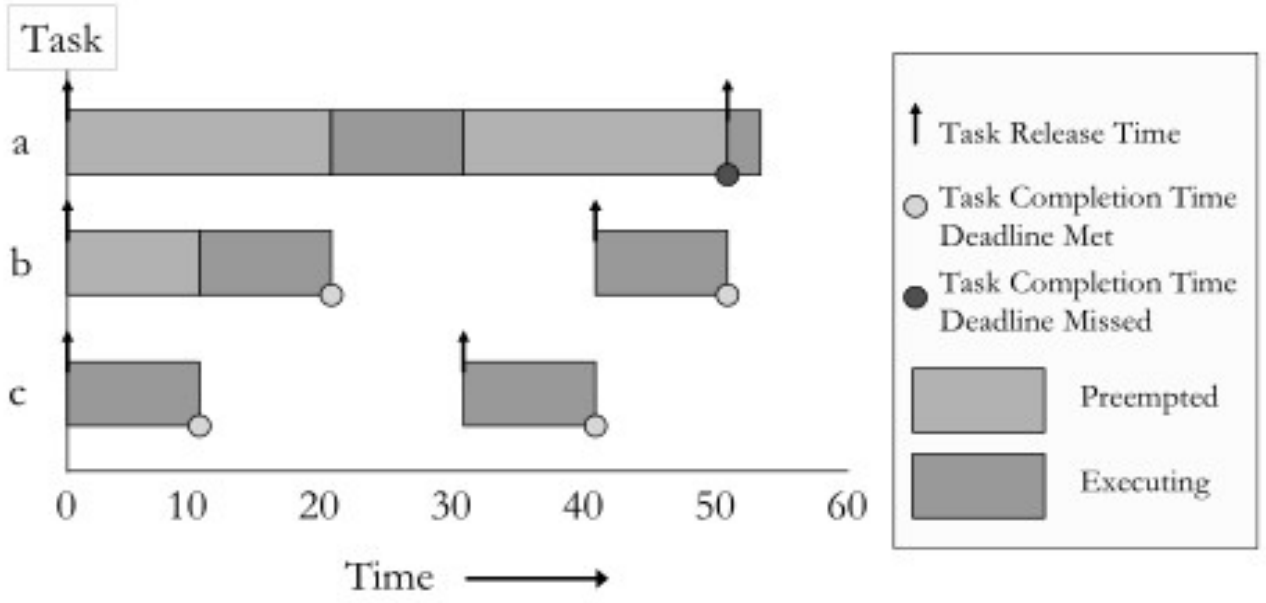
Una función fundamental del planificador es detectar la ocurrencia de un incumplimiento de plazos o de un exceso de costos (cost overrun) de un planificable y activar su manejador correspondiente. Los planificables y sus configuraciones se verán en las siguientes secciones.

Los objetos planificables pueden ser objetos de las clases `RealtimeThread` (y sus subclases) y `AsyncEventHandler` (y sus subclases) e implementan la interfaz `Schedulable`. El planificador está representado por la clase abstracta `Scheduler` y una subclase concreta `PriorityScheduler`. La especificación permite definir otros tipos de planificadores, subclasificando la clase `Scheduler`.

A continuación se muestra un ejemplo con 3 planificables (threads) de distintas prioridades (**a** con prioridad 10, **b** con prioridad 20 y **c** con prioridad 30). Los threads están configurados para

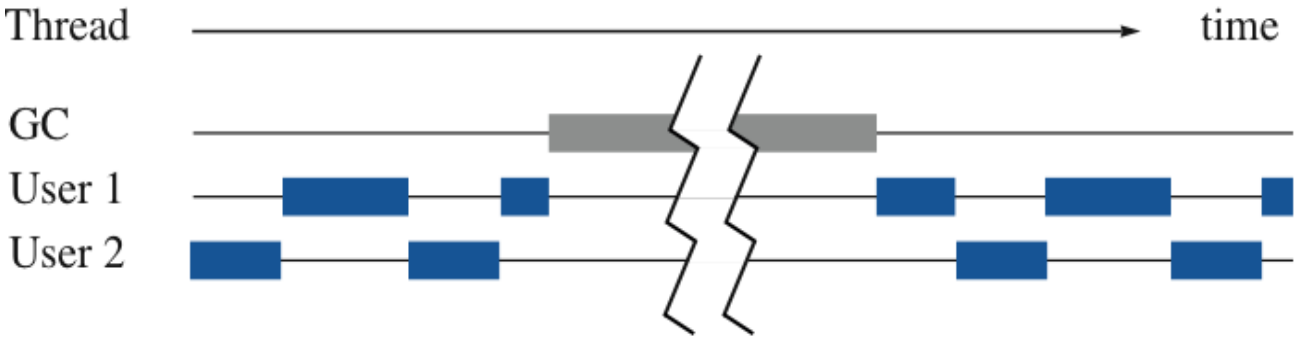
activarse de forma periódica (*periodic release*). En el tiempo 0 se activan los 3 threads al mismo tiempo; como **c** tiene mayor prioridad, se ejecuta primero (los otros dos threads están apropiados). Termina **c** antes de su plazo. Luego sigue **b** y también cumple con su plazo. Luego ejecuta **a** pero antes que termine **c** le apropia el procesador; **b** tampoco lo deja progresar. Ocurre un incumplimiento de plazos sobre **a** porque se le terminó el tiempo y tiene que volver a ejecutar (tiempo 50). Notar que en este ejemplo, **a** nunca podrá cumplir su plazo si su ejecución toma más de 10 unidades de tiempo.

Tarea	Prioridad	Período/Plazo
a	10	50
b	20	40
c	30	30

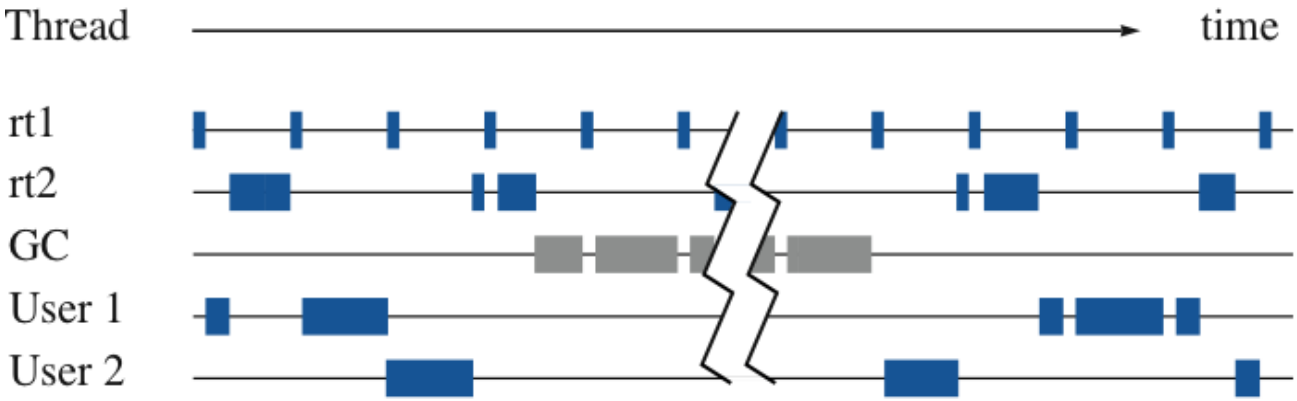


Con respecto al recolector de basura, se muestran dos imágenes comparativas de Java tradicional y Java con RTSJ.

En la primera imagen, el recolector de basura toma el control del procesador por un tiempo arbitrario no predecible, creando demoras inesperadas a los threads de alta prioridad.



En la segunda imagen, el recolector de basura se ejecuta, pero se interrumpe para permitir la ejecución de threads prioritarios.



A diferencia de los threads de C cuya ejecución y planificación están a cargo del planificador del sistema operativo de tiempo real, una aplicación de Java de tiempo real posee internamente en su máquina virtual su propio planificador, o planificador de alto nivel. Por lo tanto la ejecución de los planificables de Java (threads y eventos) serán administrados por el planificador de la máquina virtual (con su propio algoritmo de planificación y de factibilidad). También se ocupa de detectar incumplimiento de plazos y exceso de costos de los planificables.

Depende de la implementación particular de la máquina virtual cómo se relacionarán los planificables y sus 28 nuevas prioridades con el sistema operativo de tiempo real. Por ejemplo, JamaicaVM (la implementación utilizada en este trabajo, vista en un capítulo más adelante) permite configurar un mapeo entre las prioridades de Java y la del sistema operativo de tiempo real.

5.3 Relojes

Java ofrece relojes que ofrecen la hora en UTC (Coordinated Universal Time) para representar una hora según el calendario, y permiten obtener resolución de hasta milisegundos, o decenas de milisegundos.

Los sistemas de tiempo real generalmente necesitan de funcionalidad adicional como las siguientes:

- Mayor resolución
- Relojes monotónicos cuyos tiempos progresen de forma constante. Por ejemplo, un reloj con hora UTC agrega segundos extras llamados segundos intercalares (leap seconds) para reflejar con más exactitud una fecha calendario y no son monotónicos. Los relojes monotónicos son útiles para medir el pasaje del tiempo de forma exacta y suelen tener una hora base el cual no cambia nunca.
- Mecanismos para pausar, resetear o resumir el pasaje del tiempo como las que ofrece un cronómetro.
- Relojes que registren la cantidad de tiempo que ejecuta un proceso o un thread (tiempo de ejecución de CPU).

RTSJ define la clase `Clock`; representa un reloj monotónico del que se puede obtener la hora actual. La mínima fracción de tiempo es el nanosegundo. Se utilizan 64 bits para representar los milisegundos y 32 bits para los nanosegundos. La clase base es `HighResolutionTime` y representa una hora que puede ser relativa o absoluta.

Un reloj monotónico es un reloj que se ejecuta de forma constante y uniforme, sin perder ticks, para evitar el no determinismo.

Asociado al reloj, existen dos objetos para definir tiempos absolutos o relativos: `AbsoluteTime` y `RelativeTime`, ámbos subclases de `HighResolutionTime`.

Los componentes de un objeto de la clase `HighResolutionTime` son los milisegundos y nanosegundos; se pueden obtener con los métodos `getMilliseconds()` (devuelve un `long`) y `getNanoseconds()` (devuelve un `int`). Existen tambien los setters para cambiar el valor de los milisegundos y nanosegundos a un tiempo relativo o absoluto.

Para obtener la hora actual, se utiliza la clase `Clock` como se muestra a continuación:

```
Absolute now = Clock.getRealtimeClock().getTime();
```

El método `getRealtimeClock()` devuelve un objeto de la clase `Clock`, y `getTime()` devuelve el tiempo actual almacenado en un tiempo absoluto.

Existen dos métodos definidos en la clase `HighResolutionTime` que permiten convertir tiempos absolutos en relativos y viceversa. Los métodos son:

- `AbsoluteTime absolute(Clock clock)`

- `RelativeTime relative(Clock clock)`

Si el parámetro `clock` es nulo, se utiliza el reloj definido por defecto.

Dependiendo si el objeto es un `RelativeTime` o `AbsoluteTime`, se obtienen diferentes resultados. A continuación se muestran las cuatro combinaciones posibles

1. absoluto a absoluto - se obtiene un tiempo absoluto con los mismos valores que el tiempo absoluto original
2. relativo a relativo - se obtiene un tiempo relativo con los mismos valores que el tiempo relativo original
3. absoluto a relativo - se obtiene un tiempo relativo que es la diferencia entre el tiempo absoluto y el tiempo actual
4. relativo a absoluto - se obtiene un tiempo absoluto igual al tiempo actual más el tiempo relativo

Por ejemplo, si creamos un tiempo absoluto que contiene la fecha actual (como se mostró mas arriba) y cinco segundos después a este tiempo lo convertimos a un tiempo relativo, este nuevo tiempo representará un tiempo de cinco segundos relativo al tiempo absoluto calculado antes y al tiempo actual. A continuación se ilustra el ejemplo en código:

```
Absolute now = Clock.getRealtimeClock().getTime();
// se esperan 5 segundos
RelativeTime rel = now.relative(null);
System.out.println(rel);       // imprime (5000ms, 0ns)
```

Se aclara que en el ejemplo se asume que pasan 5 segundos exactos; en la realidad se obtendrá un tiempo aproximado muy cercano a 5 segundos.

Un tiempo absoluto define la hora calendario, utilizando como referencia una época. Al obtener sus milisegundos y nanosegundos mediante los métodos `getMilliseconds()` y `getNanoseconds()` obtenemos el tiempo que transcurrió desde la época hasta el tiempo que representa.

Un tiempo relativo define una duración, como puede ser la duración del período de un thread periódico o la diferencia entre dos tiempos absolutos.

Existen métodos para hacer operaciones de suma y resta entre tiempos, mediante los métodos `add()` y `subtract()`. A continuación se muestra un ejemplo con operaciones entre tiempos absolutos para obtener el mismo resultado que el ejemplo anterior:

```
Absolute now = Clock.getRealtimeClock().getTime();
// se esperan 5 segundos
Absolute now2 = Clock.getRealtimeClock().getTime();
RelativeTime rel = now2.subtract(now);
System.out.println(rel);       // imprime (5000ms, 0ns)
```

Gracias a la nueva definición de tiempos absolutos y relativos, así como la utilización de relojes que representen el tiempo con resolución de nanosegundo, es posible definir restricciones de tiempos para los plazos con resoluciones menores al milisegundo. Esto hace posible el desarrollo de programas de tiempo real duro con plazos que rondan los microsegundos.

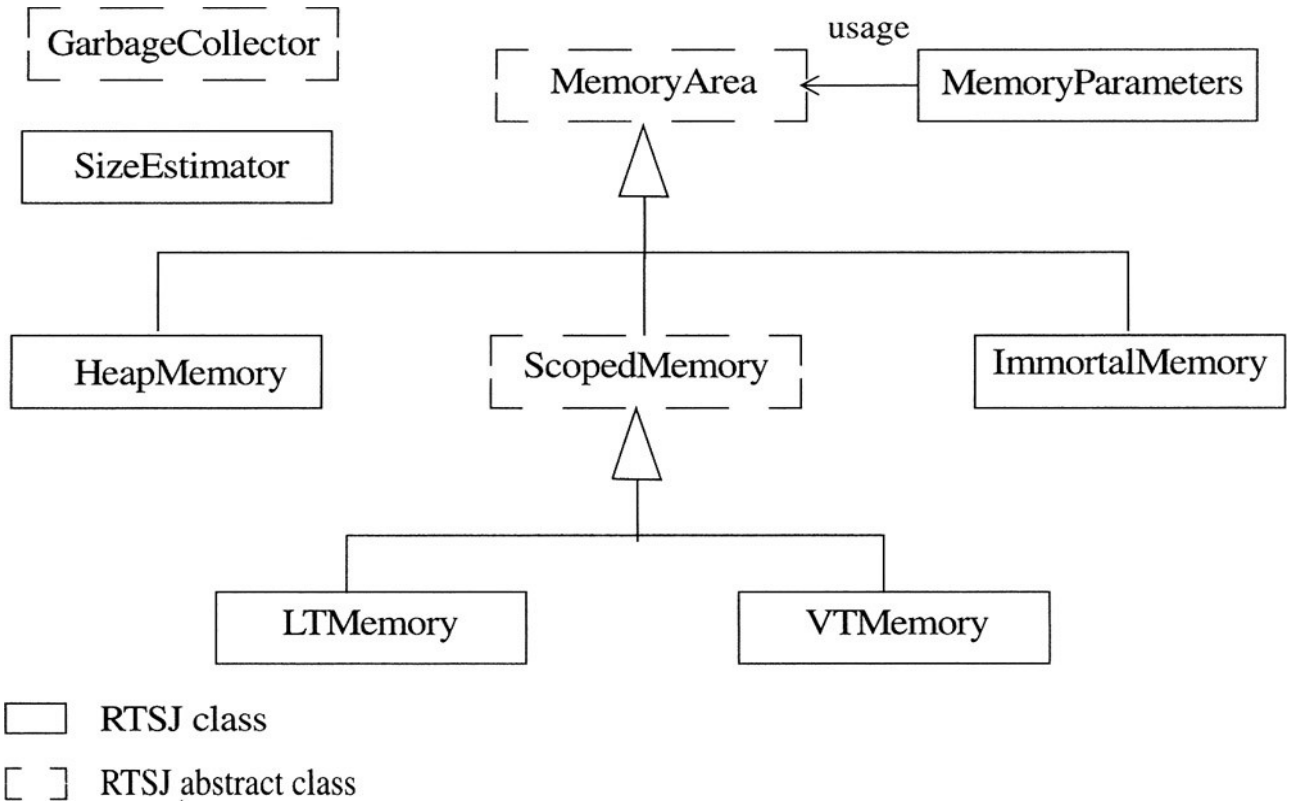
5.4 Manejo de Memoria

Los recolectores de basura siempre fueron un obstáculo en la programación en tiempo real por sus latencias impredecibles. Java tiene definida una única área de memoria donde almacenar los objetos llamada *heap*. La RTSJ aborda el problema definiendo nuevas áreas de memoria además de la heap.

Se definen cuatro tipos de áreas de memoria:

- Memoria acotada (scoped memory): utilizada para objetos que tienen un ciclo de vida dado por el alcance de éste (scope). Los objetos que tienen un alcance reducido se ponen en esta memoria. Internamente se mantiene un contador de referencias a esta memoria; si el contador llega a 0, se libera la memoria con todos sus objetos.
- Memoria física: permite crear objetos en regiones físicas específicas y con distintas características, por ejemplo, una memoria más rápida.
- Memoria inmortal: es una memoria en la cual los objetos creados nunca son sujetos al recolector de basura; o sea el espacio que utilizan los objetos nunca se libera. La memoria es compartida por todos los planificables. Solo hay una memoria definida de este tipo, y su tamaño está definido por la JVM.
- Memoria heap: también hay una memoria sola y representa la memoria estándar de Java.

A continuación se muestra el modelo de clases:



Existe una clase base `MemoryArea` en el cual se situa el comportamiento común a todos los tipos de memoria disponibles. Se definen constructores que especifican cuanto espacio en bytes ocupara el área de memoria y métodos para saber la memoria utilizada y disponible. Para ejecutar código dentro de las áreas de memoria se debe indicar un objeto `Runnable`.

Se verá a continuación el área de memoria más importante definida por la especificación RTSJ, puesto que hace posible una ejecución predecible de los planificables por no estar sujetos al recolector de basura.

Memoria acotada

Para crear un espacio de memoria acotada hay que crear un objeto de la clase `LTMemory` (linear time scoped memory) o `VTMemory` (variable time scoped memory), subclases de `ScopedMemory`; además de especificar la cantidad de espacio que contendrá el área de memoria. Luego el código que se ejecutará dentro de esta memoria (junto con las variables y objetos que se precisen) se definirá en un objeto `Runnable`. Finalmente con el método `enter(runnable)` del `ScopedMemory` se le indica que se ejecute el código del objeto `runnable` dentro de la memoria acotada.

Por ejemplo, sin utilizar `ScopedMemory`, el espacio del nuevo objeto referenciado por la

variable `a` estará listo para ser recolectado por el recolector de basura cuando la variable a salga de su área de memoria.

```
MiObjeto a = new MiObjeto();
<código en el cual utilizo a>
```

Si queremos que este objeto se utilice en la memoria acotada, debemos adaptar el código como se muestra a continuación:

```
// define una nueva area de memoria acotada con 100 bytes de
// memoria mínima y 400 bytes de memoria máxima
ScopedMemory memoriaScoped = new LTMemory(100, 400);
memoriaScoped.enter(new Runnable() {
      public void run() {
            MiObjeto a = new MiObjeto()
            <código en el cual utilizo a>
      }
}
```

En este último ejemplo la memoria del objeto contenido en la variable `a` se libera automáticamente al terminar la ejecución del método `run` (por lo tanto no será liberado por el recolector de basura). En este ejemplo se asume que el contador de referencias era igual a 1 cuando entró al método `run()`.

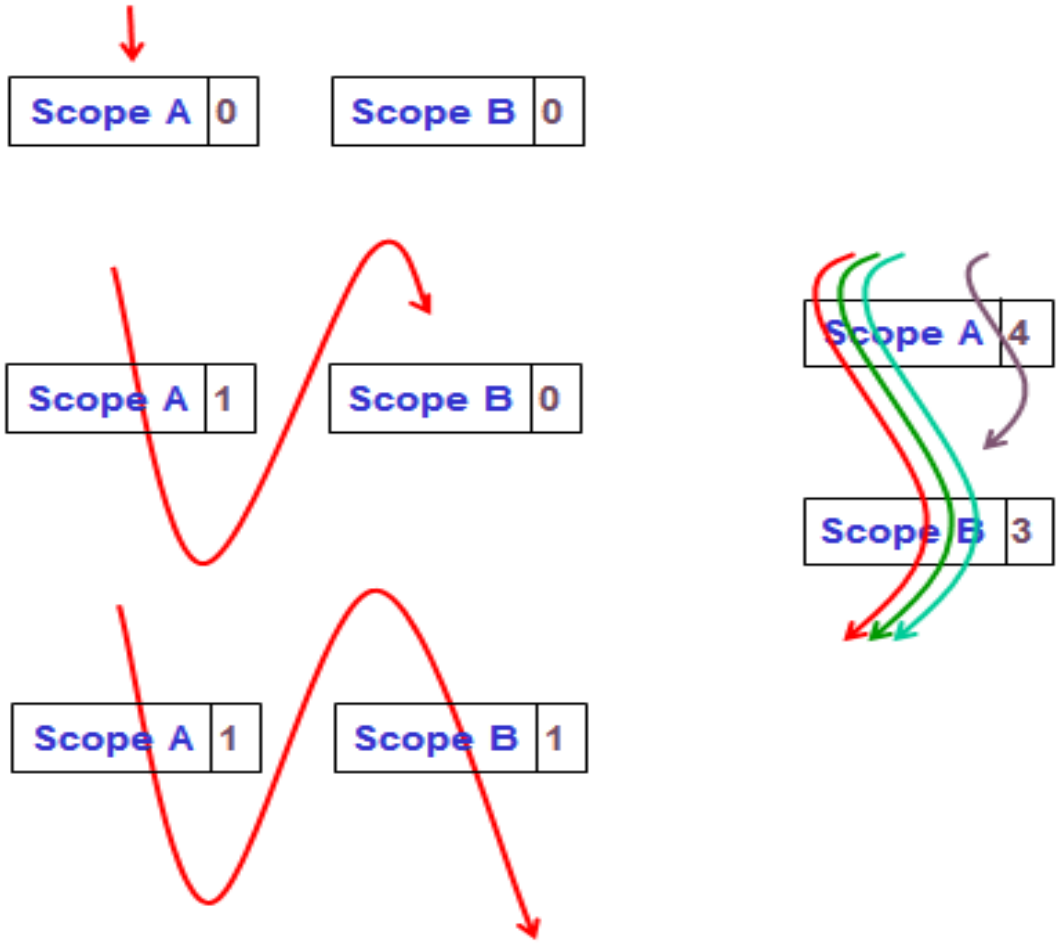
Puede ocurrir que haya varios planificables que están ejecutando en la memoria acotada (puede haber varios objetos del tipo `Runnable` que ejecuten dentro del método `enter()` utilizando la misma memoria). Si por ejemplo se tienen 3 planificables ejecutando en la memoria, el contador de referencias será igual a 3; hasta que los tres planificables no salgan del método `run()`, el contador no llegará a 0 y no se liberará la memoria.

Se imponen restricciones con respecto a referencias entre variables de distintas áreas de memoria. Supongamos que haya un objeto en la heap que referencia a otro objeto en un área acotada; si dicha área se llega a liberar (contador de referencias igual a 0), la variable de la heap quedará referenciando a un objeto que ya no existe (dangling reference). Por lo tanto no se admiten este tipo de referencias.

El contador de referencias se refiere a cuantos planificables están dentro de esta memoria ejecutándose (mediante el método `enter()`), y no a las referencias que puedan tener otros objetos a objetos de la memoria acotada.

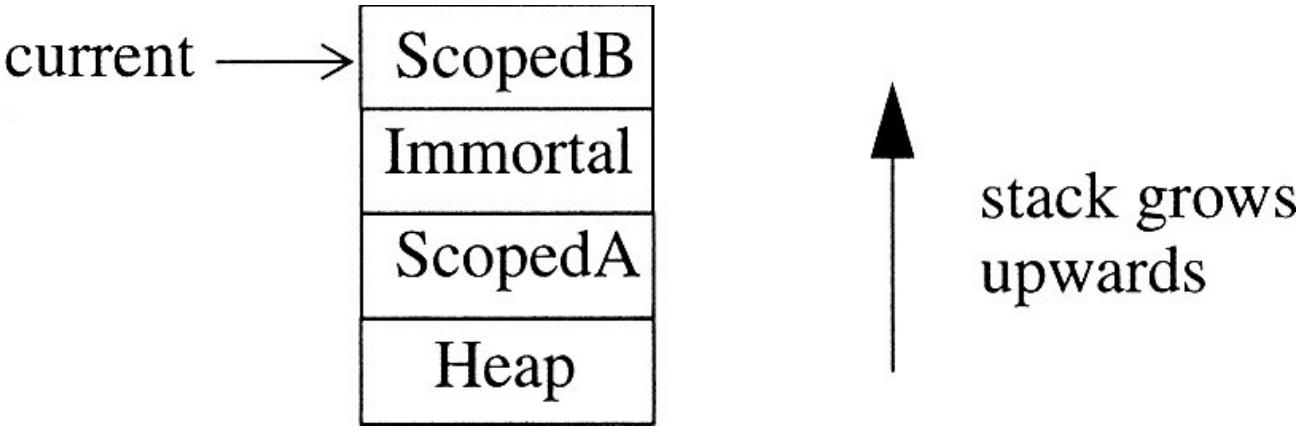
Con respecto al anidamiento de áreas de memoria acotada, es posible tener una memoria que tenga objetos que referencien a objetos de otra, siempre y cuando sean del tipo acotada, y los planificables que entren a las memorias lo hagan en el orden correcto.

En la siguiente imagen se ve a un planificable, expresado con una línea roja, que comienza a ejecutar dentro de la memoria acotada A (se incrementa el contador de 0 a 1). Mientras está dentro de esta memoria, entra a la memoria acotada B. En este caso puede haber referencias desde la memoria B a la memoria A, siempre y cuando los planificables liberen primero la memoria B y luego la memoria A.



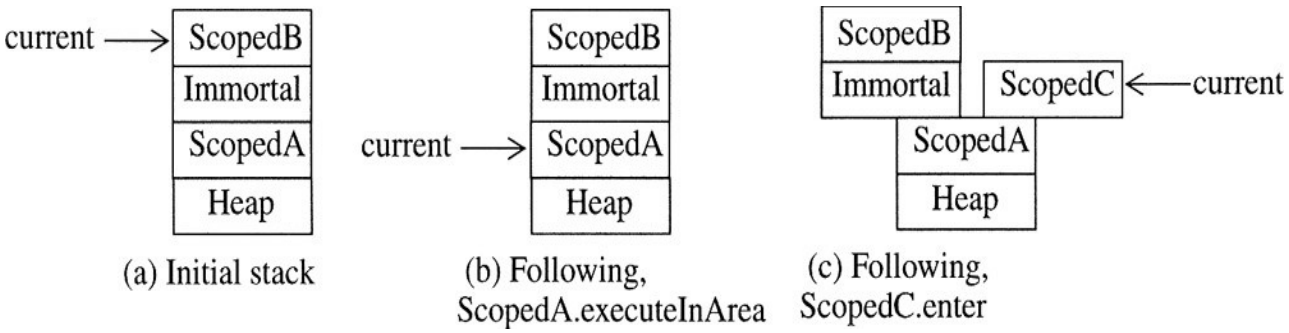
Internamente, un planificable mantiene una pila con las áreas de memoria que está accediendo. En la siguiente imagen se puede ver un planificable que inicialmente se creó en la heap; luego

siguió ejecutando dentro de la memoria acotada A, y después la memoria inmortal y la memoria acotada B. Cuando termine de ejecutar en la memoria acotada B, estará ejecutando en la memoria inmortal, y así siguiendo.

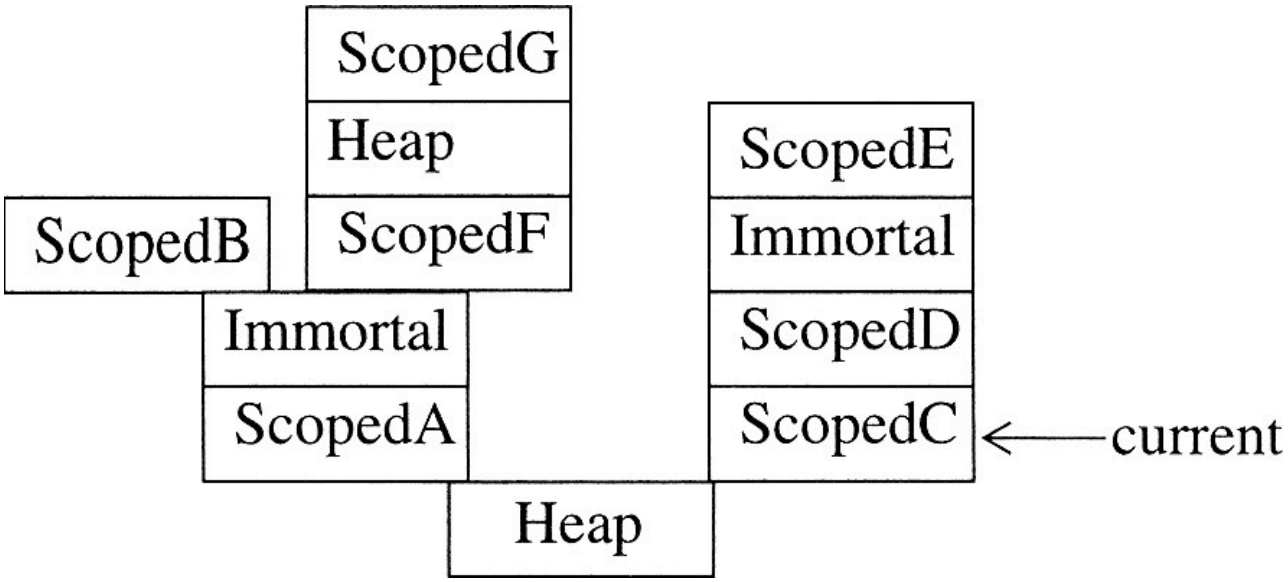


En este ejemplo, sería inválido que el planificable vuelva a entrar a la memoria acotada A puesto que ya está en la pila.

Existe un método especial de la clase `MemoryArea` que permite ejecutar código en un área de memoria que esté en la pila. En el ejemplo anterior, estando en el área acotada B se puede pasar al área acotada A con el método `executeInArea()`. Se ilustra en la siguiente imagen.



Se pueden tener pilas más complicadas (también tienen el nombre de *pilas cactus*). Estas estructuras son validas siempre y cuando las áreas acotadas tengan un único padre (en el ejemplo, el padre de la memoria acotada F es la memoria A, el padre de la memoria D es la memoria C, y las memorias C y A no tienen padre; las memorias inmortales y la heap se ignoran).



La memoria acotada permite aislar los objetos de la heap para que el recolector de basura no los tenga en cuenta, permitiendo la ejecución predecible de planificables. Sin embargo, la programación se vuelve un poco más complicada. El borrador de la especificación RTSJ versión 2.0 tiene en cuenta de incluir la posibilidad de definir recolectores de basura de tiempo real duro en las aplicaciones Java, como el utilizado por la implementación de Jamaica VM (utiliza su propio recolector independiente de la especificación), para evitar la programación de áreas acotadas y tener un código más claro.

5.5 Threads de tiempo real

Un posible tipo de planificable en RTSJ es el thread de tiempo real (real-time thread) representada por la clase `RealtimeThread`. Extiende de la clase Java `Thread` y agrega nuevos métodos para configurar el thread: se pueden definir parámetros de activación (release), en qué memoria ejecutar y qué prioridad tendrá asignada.

Hay tres tipos de activaciones asignables, según la naturaleza del thread que se desee representar:

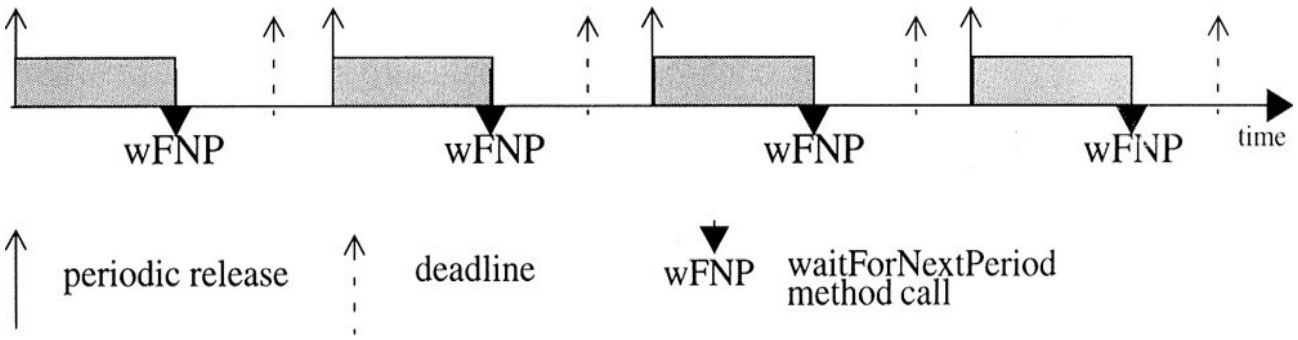
- Activación periódica - Para threads que necesitan ejecutarse de forma periódica y regular (por ejemplo, una vez por milisegundo). Dentro del thread se puede ejecutar el método `waitForNextPeriod()` para indicar que ya se terminó de ejecutar el código para la activación actual y para que espere a la siguiente activación.
- Activación aperiódica - El thread se puede ejecutar en cualquier momento. El mecanismo para que vuelva a ejecutarse una activación no está provisto en la especificación (se tiene que hacer a mano).

- Activación esporádica - El thread se puede ejecutar en cualquier momento, pero entre activaciones tiene que haber transcurrido un tiempo mínimo (llamado también *minimum interarrival time* o mit).

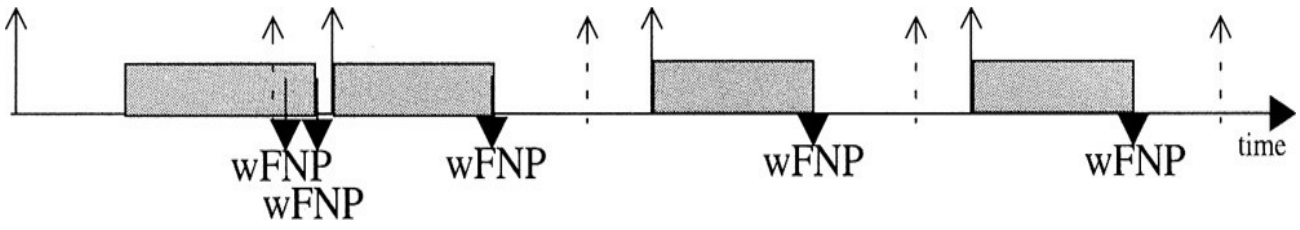
Para los threads periódicos, se puede asociar un manejador que se ejecutará si ocurre un incumplimiento de plazos. Si hay un manejador asociado y ocurre un incumplimiento, el thread no volverá a activarse hasta que se le haya indicado con el método `schedulePeriodic()` que puede seguir con su ejecución cuando llegue el próximo período. Si no tiene manejador y hay un incumplimiento, el thread se activará en el próximo período. Por defecto el plazo de los threads es igual al período.

De forma equivalente se puede asignar un costo al thread. El costo es el máximo tiempo permitido de ejecución de un thread. Si el thread utiliza más tiempo que el definido por el costo, se está violando el requerimiento de tiempo configurado para ese thread; a este incumplimiento se le llama exceso de costo (cost overrun). Así como para el plazo, se puede configurar un manejador para que se dispare al ocurrir un exceso de costo.

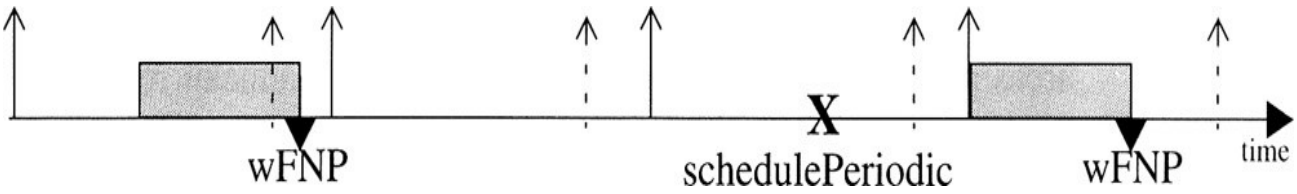
En la siguiente imagen se observa un caso sin incumplimiento de plazos; como se dijo antes, cuando el thread termina de hacer su computación, llama al método `waitForNextPeriod()`, indicado como wFNP:



Si ocurre un incumplimiento de plazo y no se asoció ningún manejador:



Ejemplo con manejador (notar que no hace nada hasta que se le indica que puede seguir ejecutando con el método `schedulePeriodic()`. Sin embargo no ejecuta inmediatamente; debe esperar a su próxima activación.



Para asociar un manejador al thread cuando ocurre un incumplimiento de plazos, se debe configurar el parámetro de activación. Como solo aplica a activaciones periódicas, cuando creamos el thread debemos utilizar el método `setDeadlineMissHandler(handler)` a la clase que representa a los parámetros de activación, o sea: `PeriodicRelease`. Otra opción es pasarle un manejador al constructor de `PeriodicRelease` antes de crear el thread.

Finalmente el manejador es un objeto de la clase `AsyncEventHandler` que se verá en la siguiente sección.

5.6 Manejo de Eventos Asincrónicos

RTSJ define a los eventos asincrónicos, que representan tanto eventos internos (disparados desde la aplicación) o externos (interrupciones o señales externas al sistema). Estos eventos se pueden asociar a uno o más manejadores. Está representado por la clase `AsyncEvent`.

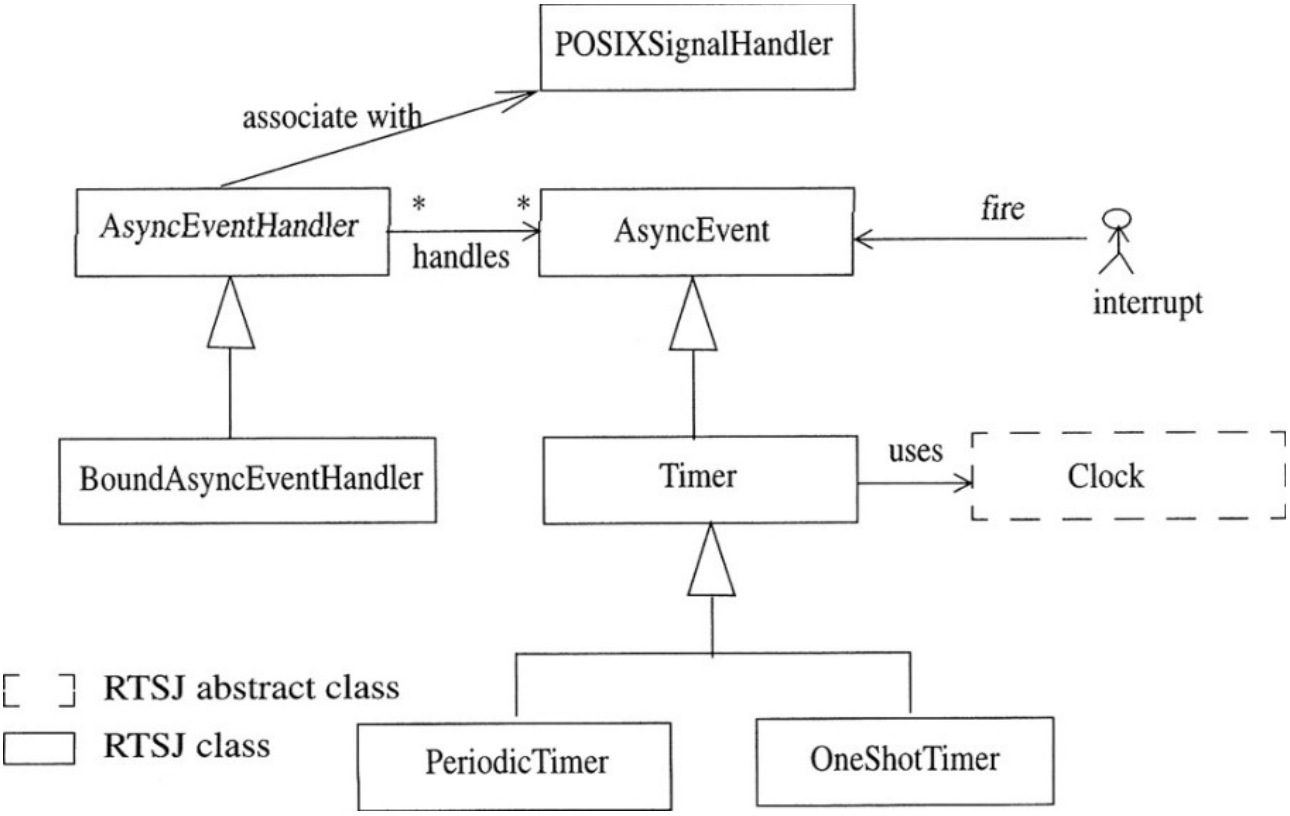
Un evento se puede disparar mediante el método `fire()` desde la aplicación. El evento incrementa en uno un contador `fireCount` que reside en cada manejador. Inmediatamente todos los manejadores asociados comenzarán su ejecución; cada uno deberá luego limpiar el contador `fireCount` para indicar que ya se atendió el evento.

Para indicar que un evento se dispare por un estimulo externo, existe el método `bind()` que recibe como parámetro el código de la señal externa (llamada también *happening*). Cuando se detecta la señal, ejecuta automáticamente el método `fire()`. Un evento que está atado (binded) a una señal externa también puede ser invocado desde la aplicación.

Hay eventos que dependen del reloj llamados cronómetros (timers), y se disparan dependiendo de como se configure el reloj. Existen dos clases que dependen del reloj: `PeriodicTimer` (se dispara de forma periódica) y `OneShotTimer` (solo se dispara una única vez). Ámbas son subclases de la clase abstracta `Timer`.

Existe una clase especial llamada `POSIXSignalHandler` que sirve para asignar manejadores a señales POSIX (siempre y cuando el sistema anfitrión sea capaz de emitir dichas señales). Sin embargo, no soporta las señales de POSIX definidas para tiempo real.

Los manejadores de eventos asincrónicos son objetos de la clase `AsyncEventHandler` (y sus subclases). Corresponden al segundo tipo de planificable junto con los threads de tiempo real. Esto implica que los manejadores tienen los mismos parámetros aplicables a un thread (prioridad, parámetros de activación y configuración de memoria). Internamente un manejador es ejecutado por un thread especial para manejadores, pero ofrece la sensación de estar ejecutando en un thread propio y exclusivo.



La acción de un manejador se define redefiniendo el método `handleAsyncEvent()`. En su ejecución se puede manipular el `fireCount` con los métodos `getAndClearPendingFireCount()`, `getAndDecrementPendingFireCount()`, `getAndIncrementPendingFireCount()` y `getPendingFireCount()`.

Un tipo especial de manejador es aquel que está ligado a un thread de forma permanente. Para ello existe la clase `BoundAsyncEventHandler`, subclase de `AsyncEventHandler`. Permite disminuir el tiempo de latencia que puede haber entre un thread y la ejecución de su manejador.

Los manejadores de eventos asincrónicos, junto a los threads de tiempo real, permiten el modelado de tareas de tiempo real periódicas, aperiódicas y esporádicas, muy utilizados en las aplicaciones de tiempo real. Ambos permiten definir manejadores ante la ocurrencia de incumplimientos de plazos o excesos de costos.

5.7 Transferencia de Control Asincrónico

Es necesario que exista un mecanismo para interrumpir un planificable cuando ocurre un evento asincrónico para que deje de hacer lo que estaba haciendo y ejecute un algoritmo alternativo. Para implementar este mecanismo en Java, la única alternativa es usar el mecanismo de interrupciones (un thread hace una verificación regular o polling para saber si fue interrumpido); el problema es que este mecanismo es sincrónico y puede introducir demoras no permitidas en los sistemas de tiempo real.

RTSJ introduce un nuevo mecanismo, llamado *transferencia de control asincrónica* (asynchronous transfer of control o ATC). A un planificable se le debe indicar si está preparado para ser interrumpido (por defecto no se pueden interrumpir). Cuando un planificable es interrumpido, deja de hacer lo que estaba haciendo sin forma de resumir la ejecución desde el punto que fue interrumpido.

La interrupción se implementa mediante el levantamiento de una excepción sobre el planificable interrumpido. Un planificable puede ser interrumpido marcando sus métodos (o métodos de otros objetos llamados por él) como interrumpibles. Se indican con la cláusula `throws` y se debe especificar la excepción `AsynchronouslyInterruptedException`.

Si se lanza una excepción mientras el planificable está ejecutando un método que no tiene la cláusula `throws`, la excepción quedará pendiente hasta que el planificable ejecute alguno de los métodos interrumpibles (o retorne el control a un método interrumpible); si el planificable está ejecutando inicializadores estáticos o bloques `synchronized` de Java, también quedará pendiente la interrupción.

Cuando se levanta la excepción, ésta será atendida por un manejador de interrupciones (clausula `try-catch`). Para que no se siga propagando la interrupción, hay que ejecutar el método `clear()` sobre la excepción; además este método devuelve `true` si estaba pendiente o `false` si ya fue liberada la excepción.

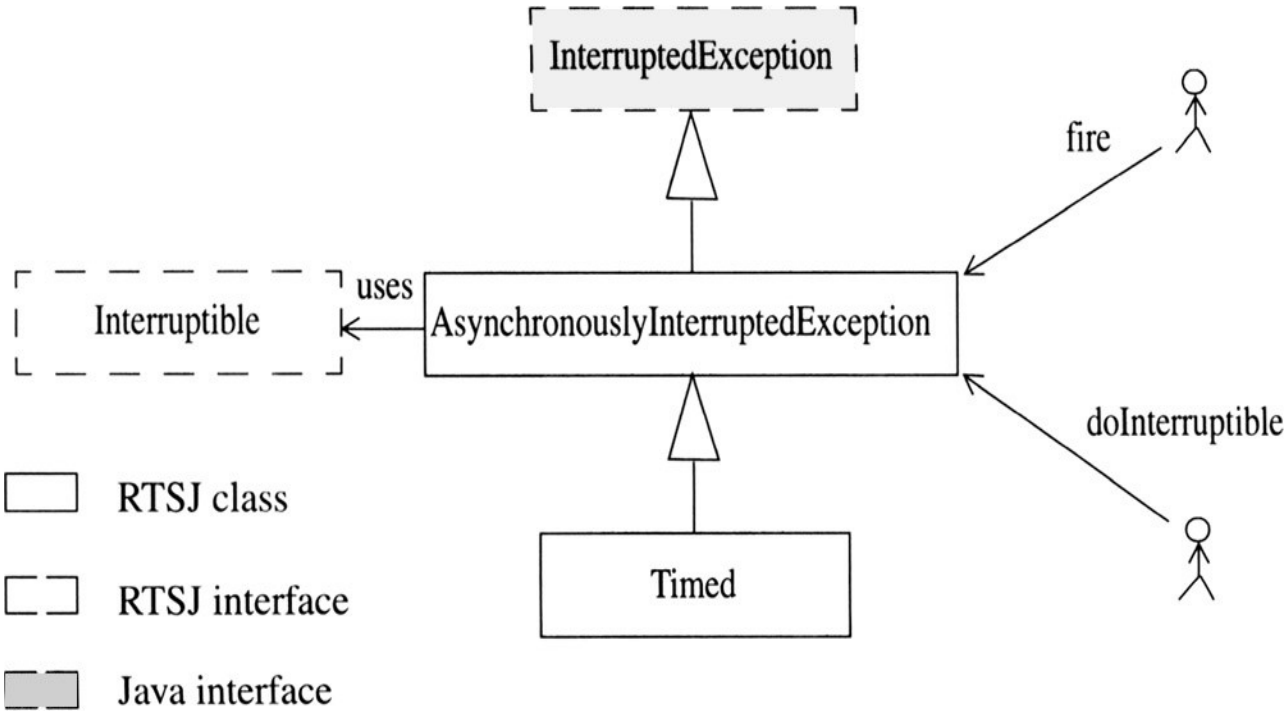
Una primer forma de disparar un ATC es ejecutando el método `interrupt()` sobre un thread de tiempo real; esto genera una excepción genérica que puede ser capturada por cualquier thread que haya sido interrumpido de esta forma. Para obtener el ATC genérico se debe ejecutar el método estático `AsynchronouslyInterruptedException.getGeneric()`.

Otra forma es creando nuevas instancias de la excepción ATC y asignándole planificables que implementen la interfaz `Interruptible`, a través del método `doInterruptible()`. Esto hará que el planificable ejecute su código y esté habilitado para ser interrumpido; mientras dure la ejecución del método `run()`, es posible interrumpirlo si al ATC se le aplica el método `fire()`. Si justo no está ejecutando, el método no tiene efecto. La acción que se ejecuta luego de la interrupción está definida en el método `interruptAction()` definido por la interfaz `Interruptible`.

A continuación se muestra un ejemplo de ejecución: se tiene un planificable que implementa la

interfaz `Interruptible`, otro planificable y un ATC conocido por ambos; el otro planificable intenta disparar el ATC pero en ese momento no hay ningun planificable ejecutando. A continuación el planificable interrumpible ejecuta sobre el ATC y pasa a un estado interrumpible (mientras dure el método `run`). Nuevamente el otro planificable dispara el ATC, provocando la interrupción asincrónica del primer planificable; el ATC espera a que se eleve la excepción y luego ejecuta el método `interruptAction()`. La espera depende de las reglas explicadas anteriormente.

Finalmente existe un tipo específico de interrupción asincrónica que es temporizada (timed). Está representada por la clase `Timed`. Sirve para interrumpir un `Interruptable` si el método `run()` excede un límite de tiempo.



El mecanismo de transferencia de control asincrónico suele ser controversial, puesto que utiliza una semántica complicada y puede dificultar la escritura de código correcto. Sin embargo es un requerimiento clave dentro de la comunidad de tiempo real poseer un mecanismo para llamar la atención de un planificable de una forma rápida.

5.8 Sincronización y recursos compartidos

Existe un problema conocido en la sincronización de threads por un recurso compartido llamado *inversión de prioridad* en el cual ocurre de forma siguiente: supongamos que existen 3 threads, LP, MP y HP, con baja, media y alta prioridad respectivamente. El thread LP adquiere el cerrojo de un recurso y mientras está tomando el cerrojo, MP le quita el control a LP (tiene mayor prioridad). Mientras ejecuta MP, HP le quita el procesador y empieza a ejecutar. En un momento HP requiere

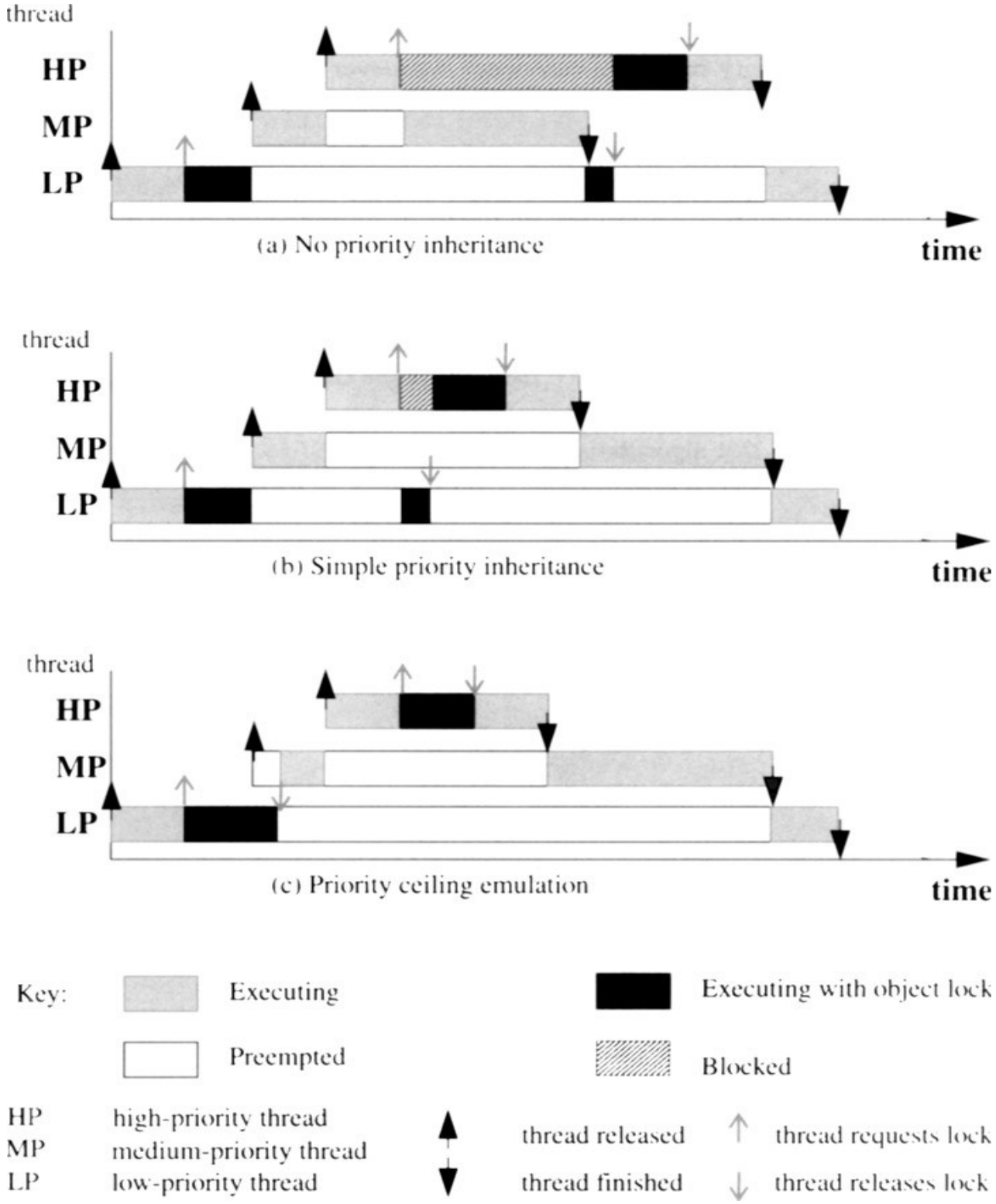
utilizar el mismo recurso que está leyendo LP, pero como no tiene el cerrojo, se bloquea y continúa ejecutando MP. HP recién se podrá desbloquear cuando LP libere el cerrojo, pero LP no puede hacer nada hasta que termine MP. Es como si MP tuviese más prioridad que HP porque lo bloquea de forma indirecta.

Existen dos algoritmos para evitar este problema: *herencia de prioridades* (priority inheritance) y *techo de prioridades* (priority ceiling emulation).

En el primer algoritmo, utilizando el ejemplo anterior, cuando HP intenta obtener el cerrojo, LP hereda la prioridad de HP (incrementando su prioridad) para dejar que LP termine y así liberar el cerrojo. Cuando LP libera el cerrojo vuelve a tener la prioridad original.

En el segundo algoritmo, cuando un objeto toma un cerrojo, éste inmediatamente pasa a tener la máxima prioridad entre todos los threads existentes (prioridad techo). Esto permite que concluya la ejecución y libere el cerrojo sin que nadie le saque el procesador. En este caso LP nunca es interrumpido mientras retiene el cerrojo.

Se muestra una imagen que ilustra los tres casos posibles: sin herencia de prioridades (a), con herencia de prioridades (b) y con techo de prioridades (c). En el primer caso, ocurre una inversión de prioridad puesto que HP se bloquea esperando el cerrojo que tiene LP, y MP no deja progresar a LP para que éste la libere. En el segundo caso cuando HP se bloquea, se incrementa la prioridad de LP para que pueda terminar su ejecución y liberar el cerrojo. En el tercer caso, LP no es apropiado por MP porque cambia su prioridad a la máxima por poseer el cerrojo del recurso compartido.



RTSJ provee métodos y clases para utilizar ámbos algoritmos. Existe la clase abstracta `MonitorControl` y dos subclases `PriorityInheritance` y `PriorityCeilingEmulation`. Llamando a los métodos estáticos de `MonitorControl` se puede configurar el algoritmo que se utilizará por defecto (se aplica a todos los objetos) o se puede configurar el algoritmo para un objeto en particular.

Por último, existe un mecanismo alternativo a la sincronización estándar de Java para poder comunicar datos entre threads de tiempo real que están en la memoria heap y en memoria no heap (memoria acotada o memoria inmortal) de una manera segura y libre de inversiones de prioridad. Para ello existen las colas libres de espera (wait free queue), representadas por dos clases:

- `WaitFreeWriteQueue` - Los threads que escriban en esta cola no se verán sujetos a esperas y los threads lectores se bloquearan si la cola está vacía. Utilizada para que los threads que están en memoria no heap puedan mandar datos a threads en la heap.
- `WaitFreeReadQueue` - Los threads que escriban se bloquean si la cola está llena, y los lectores podrán leer sin bloquearse. Utilizado para que los threads de tiempo real que no están en la heap puedan recibir datos de otros threads que están en la heap.

Dado el impacto que producen las inversiones de prioridad, la inclusión de los algoritmos de herencia de prioridades y techo de prioridades al lenguaje Java son fundamentales para limitar estos impactos.

5.9 Acceso a Memoria Física

Existe un administrador de memoria física representada por la clase `PhysicalMemoryManager`; contiene métodos estáticos que permiten registrar regiones de memoria y los tipos de memoria definidos (ej: DMA, Pagina de E/S). Éstos métodos sirven como mecanismo para que una implementación particular de la RTSJ registre sus propias secciones de memoria. Las secciones son llamadas *filtros* y deben implementar la interfaz `PhysicalMemoryTypeFilter`. Solo existe un único filtro por tipo de memoria. Al administrador se le pueden asignar eventos a las acciones de eliminación o inserción de distintos tipos de memoria. Generalmente son programadas por el programador del sistema; el programador de aplicaciones solo necesita saber qué tipos de memoria hay en el sistema.

Es posible crear objetos en la memoria física; para esto hay que crear las áreas de memoria a través de las clases `ImmortalPhysicalMemory`, `LTPhysicalMemory` y `VTPhysicalMemory`. A diferencia de la memoria inmortal vista antes (única area de memoria), puede haber múltiples memorias inmortales definidas. Las áreas físicas se definen especificando el tipo de memoria y un tamaño en bytes; es posible especificar también la dirección base del área de memoria.

Por último, se puede acceder a la memoria física fuera del modelo de objetos (lectura a nivel de byte). Existen dos clases: `RawMemoryAccess` (lectura de variables de tipo `int` y `short`) y `RawMemoryFloatAccess` (de tipo `float` y `double`). Por ejemplo se podría implementar el acceso a un ADC (conversor analógico digital), especificando las direcciones de memoria de entrada salida; luego a través de una interrupción de hardware mapeada a un evento asincrónico, se

puede iniciar la lectura y conversión de los datos para luego procesarlos.

La especificación provee un mecanismo para poder acceder a los registros de dispositivos (utilizando la clase `RawMemoryAccess`), pero de una manera limitada puesto que solo se pueden usar tipos primitivos para acceder a ellos, y operadores de bits para manipularlos.

5.10 Comparación con C/C++ con POSIX

Se resume en el siguiente cuadro las diferentes facilidades que tiene Java con RTSJ con respecto a C/C++ con POSIX de tiempo real.

Facilidad	C/C++ con POSIX	Java con RTSJ
Manejo de memoria	Asignación y liberación de memoria explícito.	Asignación explícita y liberación automática de memoria mediante el recolector de basura. Utilización de espacios de memoria acotados.
Herencia	Solo aplicable a C++: herencia múltiple de clases.	Herencia simple de clases y herencia múltiple mediante interfaces.
Manejo de excepciones	No posee. Simulable mediante la utilización de directivas.	Manejo de excepciones sincrónicas y asincrónicas.
Threads	API de threads "pthread" de POSIX.	Define las clases `RealtimeThread` y `AsyncEventHandler`. Permite configurar plazos y excesos de costo y asociarles un manejador de errores. Se pueden configurar threads o eventos periódicos, aperiódicos o esporádicos.
Acceso a la memoria física	Mediante punteros. Alta flexibilidad.	Mediante objetos especiales definidos por la especificación RTSJ. Poco flexible y con acceso limitado.
Planificación	La utilizada por sistemas operativos POSIX: planificación FPS y políticas FIFO, RR y OTHER.	Posee su propia planificación. Por defecto FPS, luego cada implementación puede definir otras planificaciones y la forma de relacionar las prioridades al sistema operativo de tiempo real.
Sincronización	API de POSIX: semáforos, mutexes.	Utilización de monitores: métodos y bloques sincronizados.
Tratamiento de inversión de prioridad	Algoritmos de prioridad de techo y herencia de prioridades provistos por el sistema operativo POSIX.	Algoritmos de prioridad de techo y herencia de prioridades provistos por la máquina virtual.

La especificación RTSJ hace posible que el lenguaje Java se pueda utilizar para el desarrollo de las aplicaciones de tiempo real. La versión 1.0.2 tiene algunos problemas que están siendo revisados en el nuevo borrador de la versión 2.0 de la especificación. En el siguiente capítulo se verá una implementación particular de la especificación: JamaicaVM de Aicas.

6 Aicas JamaicaVM

JamaicaVM (Jamaica Virtual Machine) de Aicas (versión 6.3 al momento de escritura), es una implementación de la máquina virtual de Java (JVM) que permite la ejecución de aplicaciones escritas para Java 6 Standard Edition y aplicaciones con las nuevas funcionalidades y clases definidas en la especificación RTSJ. Está diseñada para sistemas de tiempo real y ofrece las siguientes características [11][13]:

- Garantías de ejecución en sistemas de tiempo real *duro*
- Implementa la especificación RTSJ 1.0.2
- Tamaño reducido de la aplicación
- Capacidad para generar un único archivo compilado y ejecutable
- Enlace estático y dinámico de clases
- Múltiples plataformas soportadas
- Recolector de basura de tiempo real
- Soporte para JNI (Java Native Interface)
- Soporte para múltiples procesadores

Es posible reducir el tamaño y la utilización de memoria de una aplicación compilada de forma considerable; se puede llegar a tener un ejecutable de menos de 1 Mb (incluye clases y JVM todo en un archivo).

Actualmente soporta las siguientes plataformas:

- De desarrollo: Linux, Windows, SunOS/Solaris
- Plataformas (RTOS): QNX, WinCE, VxWorks, Linux-RT
- Plataformas (SO no RT): Windows, SunOS/Solaris, Linux
- Arquitecturas de procesadores: ARM, Nios, PowerPC, Sparc, x86

Para el desarrollo existe un plugin para utilizar JamaicaVM desde el IDE de Eclipse descargable desde el sitio de actualización de Aicas. Éste facilita la construcción de los ejecutables y permite la depuración de la aplicación utilizando el intérprete Java de Jamaica.

Todos los threads creados por la implementación son threads de tiempo real; no hay necesidad de diferenciarlos entre threads comunes o threads no-heap puesto que el recolector de basura no interrumpe la ejecución de los threads (ver siguiente sección). Se puede utilizar la memoria heap como si fuese una memoria no sometida a la recolección de basura. Por lo tanto, los real time threads comparten la memoria heap por defecto (distinto a lo definido en la especificación RTSJ). Si se quiere hacer más portable la aplicación, se puede utilizar la opción "`-strictRTSJ`" para que la JVM ejecute cumpliendo de manera estricta la especificación RTSJ.

6.1 Recolector de basura para sistemas de tiempo real duro

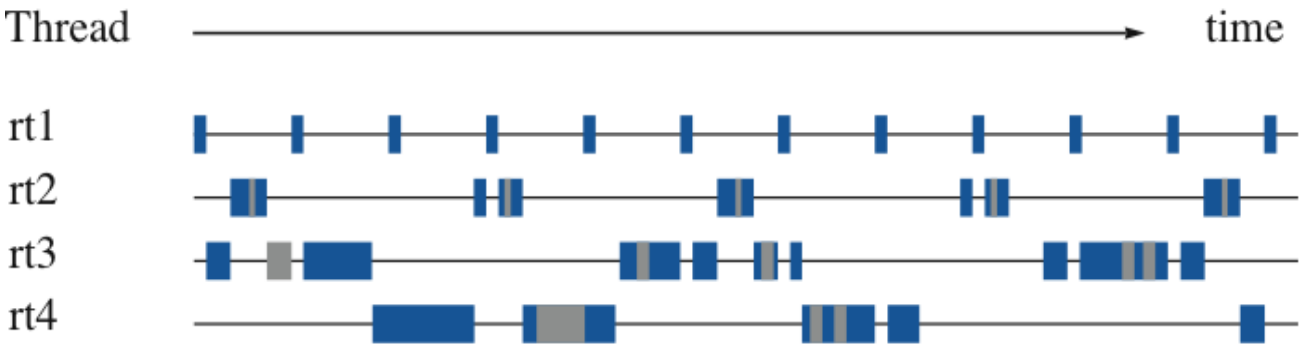
La RTSJ provee soluciones a la programación de tiempo real en Java, pero también introduce algunas dificultades; un problema importante es que el desarrollador debe dividir estrictamente el programa en dos partes: una parte para tiempo real y otra no tiempo real con respecto a la memoria, para evitar que el recolector de basura se ejecute sobre los threads de tiempo real. La sincronización entre threads de tiempo real y no tiempo real se encuentra restringida, para prevenir que los threads de tiempo real se bloqueen por el recolector de basura por inversión de prioridad.

JamaicaVM elimina estos problemas y restricciones gracias a su recolector de basura de tiempo real. Por lo tanto no hace falta dividir el programa en dos como se mencionó antes, puesto que el recolector de basura es determinístico y se puede ejecutar para áreas de memoria de threads te tiempo real o no tiempo real.

JamaicaVM utiliza un recolector de basura incremental, en el cual las cuatro fases de la recolección (ver sección 4.4) deben ser ejecutadas en pequeños incrementos de recolección de basura mientras la aplicación continúa su ejecución y modifica la heap. Para que el recolector de basura sea de tiempo real, estos incrementos deben ejecutarse de forma predecible y al mismo tiempo garantizar que se liberará la cantidad de memoria suficiente para que el programa pueda alocar más objetos.

El recolector de basura de Jamaica se ejecuta directamente sobre el thread que está ejecutando cuando éste necesita alocar memoria para los objetos que está creando. El thread es apropiable (preemptible) incluso cuando está ejecutando el recolector de basura. Utilizan una muy pequeña fracción de tiempo para ir limpiando la memoria, en incrementos muy pequeños de trabajo. En JamaicaVM, un incremento consiste en procesar y posiblemente reclamar solo 32 bytes de memoria. Cada vez que un thread necesita alocar memoria, se ejecutan uno o más de estos incrementos. El número de incrementos se puede analizar para determinar comportamientos de peor caso en código de tiempo real.

En la siguiente imagen se muestra la ejecución del recolector de basura distribuida en los threads (en vez de tener un thread dedicado). El color gris simboliza actividad del recolector de basura.



Para que la aplicación funcione de forma óptima, hay tres formas de configurar el recolector de basura, según las necesidades de la aplicación:

- recolector de basura que "detiene el mundo" (stop-the-world*)* - El utilizado por Java estándar. La aplicación pierde determinismo porque detiene la ejecución de todos los threads para ejecutar la limpieza, por lo cual no debería utilizarse para aplicaciones de tiempo real.
- recolector de basura de tiempo real dinámico - El costo de la limpieza es dinámico, con un promedio de latencia bajo pero con posibles picos de latencia. Es suficiente para aplicaciones de tiempo real blando.
- recolector de basura de tiempo real estático - El costo de la limpieza es siempre el mismo siempre que se haga una asignación de memoria. Este hecho hace que el costo de ejecución sea menor, pero en promedio mayor al modelo dinámico. Al tener un costo constante, se vuelve predecible y adecuado para sistemas de tiempo real duro.

En la siguiente sección se mostrará como compilar una aplicación de JamaicaVM y también ajustarlo para que funcione lo más predecible y con el mayor desempeño posible.

6.2 Jamaica Builder

Para construir el ejecutable con la aplicación compilada en C, existe la herramienta Jamaica Builder, que utiliza las librerías de compilación y enlace de C/C++ disponibles del sistema operativo para armar el ejecutable a partir de código Java.

Para configurar Jamaica Builder, se puede hacer por línea de comandos y mandar los parámetros; también se puede leer de un archivo de configuración, generable desde la misma utilidad o una combinación de los dos, donde los parámetros por linea de comandos toman prioridad sobre los encontrados en el archivo de configuración. Existe una opción más, que es a través del plugin para Eclipse, en el cual se define un archivo build.xml de Ant extendido y permite agregar los parámetros, ofreciendo por cada uno de ellos una explicación.

Si no se especifica ningún archivo de configuración, utiliza la configuración por defecto dentro de la carpeta de Jamaica (*jamaica-home*/etc/global.conf).

La forma que tienen los parámetros es *opción=valor*. A continuación se muestra un ejemplo con parámetros con y sin valor:

```
jamaicabuilder -opción1=valor -opcion2
```

Existe una numerosa cantidad de opciones de compilación. Las opciones más importantes y utiles se marcarán en **negrita**. Las siguientes opciones sólo son utilizables desde la línea de comandos:

Opción	Descripción
-help	Muestra como utilizar el builder y muestra las opciones estándar con una breve descripción
-version	Muestra la versión del builder y termina
-verbose=*n*	1 por defecto. n=1 : muestra advertencias, n=2 : muestra más información; n=0 : no muestra advertencias
-showSettings	Muestra las opciones de configuración
-saveSettings=*archivo*	Igual que la anterior pero guarda las opciones en un archivo; posteriormente se puede utilizar.
-configuration=*archivo*	Permite cargar las opciones desde un archivo.
-XavailableTargets	Muestra una lista de plataformas disponibles para utilizar como plataforma destino (en qué plataforma correrá el ejecutable). Ver la opción *-target* más abajo.

Las siguientes opciones pueden ser tanto configuradas desde línea de comandos como en el archivo de configuración:

Opción	Descripción
-classpath=*classpath*	Permite especificar clases y carpetas donde buscar clases utilizadas por el Builder, cada valor separado por ":". Solo para línea de comandos, si se usa "-classpath+=*classpath*" se agregan rutas de archivo y clases a las que ya están configuradas. También se puede utilizar "-cp=*classpath*"
-main	Especifica la clase que contiene el método estático *main*, el cual es el punto de entrada de la aplicación.

-jar	Permite utilizar un .jar cerrado con una aplicación; el archivo MANIFEST debe tener especificado una clase *main*.
-lazy=*boolean*	***true*** por defecto. Como cualquier JVM de Java, JamaicaVM carga sus clases dinámicamente, pero esto puede provocar demoras imprevisibles inaceptables para la ejecución de aplicaciones de tiempo real; por esto conviene configurarlo en ***false*** para que cargue todas las clases antes de empezar a ejecutar la aplicación.
-destination=*nombre*	Permite especificar el nombre del ejecutable a generar. Si no se especifica, tomará el nombre de la clase *main*.
-smart	Busca a nivel de campos y métodos de las clases cuales no se utilizan, para excluirlas del ejecutable. Sin la opción, las clases se excluyen si no son utilizadas en absoluto. Tener cuidado con activar esta opción y al mismo tiempo utilizar *reflection* de Java.
-closed	Si la aplicación no va a necesitar cargar clases dinámicamente, esta opción permite al builder y al compilador hacer optimizaciones por no tener carga dinámica de clases. Tener en cuenta que tienen que estar cargadas todas las clases que se van a utilizar.
-profile	El ejecutable generado con esta opción, al ejecutarse creará un archivo de perfil, con extensión **.prof**, donde se registrará información de la ejecución, por ejemplo cuales fueron las rutinas más utilizadas. Este archivo posteriormente se utilizará para generar un nuevo ejecutable optimizado.
-useProfile=*archivo*	Instruye al builder para generar la aplicación optimizada, compilando sólo los métodos más utilizados según el archivo .prof. Es la mejor forma de optimizar una aplicación con Jamaica Builder. Si no se utiliza archivo .prof, el builder utilizará un perfil por defecto.
-percentageCompiled=*n*	Por defecto n=10. Indica el porcentaje de métodos más importantes, según el archivo .prof, que debe compilar. Se usa en conjunto con la opción anterior.
-inline=*n*	Por defecto n=5. Cuando los métodos están compilados, indica el nivel de *inlining* de 0 a 10. Cuanto más alto el número, mayor desempeño tendrá la aplicación, pero también será más grande por tener mayor duplicación de código.
-optimize	Permite instruir al compilador de C para optimizar el codigo nativo, de una manera independiente de la plataforma, según las siguientes opciones: *none*, *size*, *speed*, o *all*.
-target=*plataforma*	Indica para qué plataforma se va a utilizar el ejecutable generado.
-heapSize=*n*	Configura la cantidad de memoria inicial de la heap, en bytes. Se puede utilizar nK para n KBytes, y nM para n MBytes.
-maxHeapSize=*n*	Indica el máximo de memoria utilizada por la heap. Si es mayor a

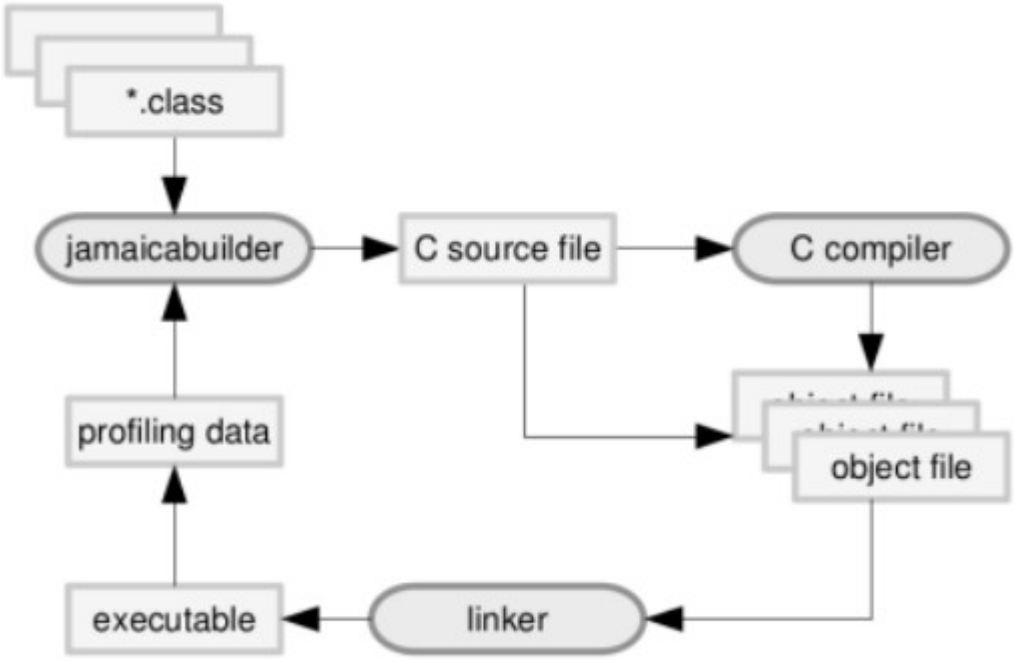
	la memoria inicial, ésta irá ajustando su tamaño según la demanda de memoria. Para aplicaciones que precisan un mayor desempeño, utilizar los mismos valores para heapSize y maxHeapSize, especialmente cuando está configurado el recolector de basura según la opción -analyze (ver más adelante).
-numThreads	Indica la cantidad inicial de threads soportada por la aplicación. Mínimo: 2
-maxNumThreads	Indica la máxima cantidad de threads soportada por la aplicación. El límite máximo de threads de una aplicación de JamaicaVM es 511.
-priMap=*configuracion*	Hace un mapeo de prioridades de Java con las prioridades del sistema operativo. También permite seleccionar la política de planificación para cada prioridad. Por ejemplo, la siguiente configuración `1..10=5/RR,sync=6,11..38=7..34/FIFO` configura las prioridades de Java de 1 al 10 con la prioridad 5 en el SO y utilizan política *round robin*; el thread de sincronización con prioridad 6 del SO (thread utilizado para sincronizar threads con prioridades nativas entre 1 a 10), prioridades 11 a 38 se distribuyen entre las prioridades 7 y 34 del SO utilizando política FIFO.
-analyze=n	La aplicación compilada con esta opción ejecutará en un modo especial: recolectará información acerca de la utilización de la memoria, y al terminar su ejecución mostrará un informe con los valores óptimos de configuración del recolector de basura dinámico y estático según la cantidad de memoria heap utilizada, y la cantidad máxima de memoria utilizada en esa ejecución. n indica la precisión relacionada al valor máximo de memoria utilizado. Se recomienda utilizar n=5 para los análisis. n=0 para no utilizar el análisis.
-constGC=n	Instruye al builder a construir una aplicación que utilice el recolector de basura en modo constante. n es el valor seleccionado según el análisis. n debe ser coherente con las opciones **heapSize** y **maxHeapSize** según el análisis. Si no se utiliza este parámetro (o es igual a 0), se utilizará el recolector de basura en modo dinámico. En este caso también las opciones **heapSize** y **maxHeapSize** deben ser coherentes según el análisis.

El proceso de construcción de un ejecutable utilizando el builder consiste en indicar el classpath con las clases compiladas (.class) a incluir en la aplicación. Luego el builder convierte las clases compiladas java (bytecode) a código fuente en C. Posteriormente se utilizará el compilador de C que esté configurado en el sistema operativo para generar los objetos y a través del proceso de enlace generar un ejecutable.

Si el builder estaba configurado para generar perfiles (opción -profile), el ejecutable generado al ejecutarse generará un archivo de perfil que contiene información acerca de los métodos más utilizados en la aplicación. Luego este archivo podrá ser utilizado para configurar el builder y generar un segundo ejecutable optimizado según el archivo de perfil.

A continuación se muestra el proceso de construcción del ejecutable mediante Jamaica Builder:

Para



optimizar aún más la aplicación, se debe configurar el recolector de basura modificando las opciones **heapSize**, **maxHeapSize** y **constGC**. Las dos primeras opciones deberían tener el mismo valor. El último parámetro solo se indica si se desea utilizar el recolector de basura en modo constante.

Para saber qué valores deben tener estas opciones, previamente debemos generar un ejecutable con la opción **analyze=5** (se recomienda este número). Luego de ejecutar la aplicación, imprimirá los resultados del análisis que nos servirán para completar las opciones vistas antes.

Un ejemplo de resultado de un análisis se muestra a continuación. Se muestran 3 columnas: la memoria máxima a utilizar y los peores costos de recolección de basura según sea dinámico o constante. También sugiere una cantidad de memoria recomendada según el análisis.

```
### Recommended heap size: 9427K (contiguous memory).
### Application used at most 5704832 bytes for reachable objects on the Java
### heap (accuracy 5%).
###
### Reserved memory is set to 10%.  To obtain lower memory bounds
### or worst-case GC overhead, set reserved memory to 0.
###
### Worst case allocation overhead:
### heapSize   dynamic GC  const GC work
### 29477K     6           3
### 22647K     7           4
### 18822K     8           4
### 16434K     9           4
### 14778K     10          4
### 12662K     12          5
### 11393K     14          5
### 10512K     16          6
### 9896K      18          6
### 9427K      20          7
### 8774K      24          8
### 8340K      28          9
### 8040K      32          10
### 7803K      36          11
### 7622K      40          12
### 7360K      48          14
### 7180K      56          17
### 7053K      64          19
### 6753K      96          27
### 6609K      128         36
### 6463K      192         53
### 6397K      256         69
### 6331K      384         100
```

Luego si decidimos utilizar el recolector de basura dinámico con la memoria recomendada, debemos configurar las siguientes opciones:

```
-heapSize=9427K

-maxHeapSize=9427K
```

Con estas opciones, el valor que indica el costo de la recolección de basura en modo dinámico sería de 20.

Si se quiere utilizar el recolector de basura en modo constante, las dos opciones anteriores se configuran igual, pero se agrega el tercer parámetro:

```
-constGC=7
```

De esta manera, la configuración está consistente con el análisis anterior.

Para el desarrollo en Eclipse, existe un plugin que permite configurar el build a través de un archivo de construcción de Jamaica (Jamaica build file), que consiste en un archivo build.xml de Ant extendido que permite configurar las mismas opciones de compilación vistas anteriormente desde un editor más amigable y con una descripción de cada opción. Además permite configurar la máquina virtual de JamaicaVM para poder ejecutar código desde el Eclipse y usar la herramienta de debug.

JamaicaVM es una implementación de la especificación RTSJ 1.0.2 que demuestra tener un gran potencial para el desarrollo de aplicaciones de tiempo real duro, tanto en sistemas embebidos como en sistemas de tiempo real. Más adelante se mostrarán los experimentos hechos con la máquina virtual de Jamaica y sus resultados.

7 Patch rt-preempt para Linux

El kernel estándar de Linux solo cumple requerimientos de sistemas de tiempo real blando. Provee operaciones básicas de POSIX para manejo del tiempo pero no ofrece garantías para el cumplimiento de plazos duros. Con el parche de apropiación de tiempo real de Ingo Molnar, también llamado "rt-preempt", y la capa de eventos del reloj de alta resolución genérica de Thomas Gleixner, el kernel convierte al sistema operativo a tiempo real duro [14].

El parche fue ganando interés en la industria gracias a su limpio diseño y su objetivo de mantener las lineas de desarrollo bien integradas. Lo hace una buena opción para producir aplicaciones de tiempo real duro y firme, las cuales van desde aplicaciones de audio profesional hasta programas de control industrial.

A medida que el parche se vuelve cada vez más utilizable, se va incorporando más funcionalidad al kernel original.

El parche convierte al kernel del Linux en un kernel con apropiación total (fully preemptable kernel). Lo logra aplicando los siguientes cambios al kernel original:

- Las primitivas ejecutadas dentro del kernel para la exclusión mutua (cerrojos) se vuelven apropiables gracias a su reimplementación con mutex de tiempo real (rtmutexes).
- Las secciones críticas protegidas por spinlocks se vuelven apropiables y se agregan primitivas nuevas para la creación de secciones críticas no apropiables.
- Implementación de herencia de prioridades para los mutex, semáforos y spinlocks del kernel para evitar problemas de inversión de prioridades.
- Los manejadores de interrupciones se convierten en threads del kernel apropiables.
- Se amplia la API de Linux para utilizar relojes y cronómetros POSIX de alta resolución.

En el siguiente capítulo se harán experimentos utilizando un kernel con el patch rt-preempt como sistema operativo de tiempo real (RTOS) y se mostrará que tiene un comportamiento de sistema de tiempo real duro.

8 Experimentos

A través de experimentos, y mediante distintos escenarios, se compararán las latencias entre programas implementados en C y Java. El programa en C a utilizar es el programa **cyclictest** y los programas de Java a utilizar serán los siguientes: **jittertest** de Aicas y una aplicación propia llamada **rt-period-simple**. De ahora en adelante nos referiremos a esta última como **rtps**.

Como primer paso, se propone mostrar que el sistema utilizado es de tiempo real, utilizando un kernel de tiempo real optimizado. Más abajo se mostrará la configuración utilizada.

Luego se compararán las ejecuciones de programas sobre un sistema operativo sin tiempo real y otro con tiempo real, utilizando implementaciones en los dos lenguajes para mostrar que son aplicaciones con características de tiempo real.

Posteriormente se compararán las latencias obtenidas entre cyclictest y rtps utilizando distintos intervalos, realizando pruebas de larga duración.

Previo a las pruebas se dará una descripción de las aplicaciones utilizadas.

El entorno de ejecución utilizado consiste en un sistema operativo Linux Ubuntu 14.04 LTS de 64 bits con un kernel vanilla 3.14.39 con el parche **rt-preempt version 37** (3.14.39-rt37) corriendo en una CPU AMD A10-6800K APU con 4 cores de 4120 MHz con 16 GB de RAM. También se utilizará el mismo kernel pero sin el parche rt-preempt. En todas las pruebas se utilizará la misma arquitectura y sistema operativo y mismo kernel.

En las secciones siguientes se mostrarán los siguientes experimentos y comparaciones:

- cyclictest sobre un sistema operativo sin soportar tiempo real - con y sin stress
- cyclictest sobre un sistema operativo de tiempo real - con y sin stress
- Comparación de las pruebas de cyclictest
- jittertest sobre un sistema operativo sin soportar tiempo real - con y sin stress
- jittertest sobre un sistema operarivo de tiempo real - con y sin stress
- Comparación de las pruebas de jittertest
- cyclictest: pruebas de 24 hs sobre SO de tiempo real - con y sin stress
- rtps: pruebas de 24 hs sobre SO de tiempo real - con y sin stress
- Resumen de los resultados obtenidos

8.1 Configuración optimizada del kernel de tiempo real

El kernel se compiló utilizando las siguientes opciones en la configuración [14]:

- Se desactivaron los ticks dinamicos (NO_HZ)
- Se desactivaron opciones de tracing
- Se desactivaron todas las funciones de ahorro de energía (salvo el módulo ACPI)
- Se habilitaron los relojes de alta resolución
- Se configuró para que sea un kernel completamente apropiable (fully preemptable kernel)

Además del kernel, para tener un comportamiento del procesador predecible con las menores latencias, hay que configurar el BIOS. Se hicieron los siguientes cambios en su configuración:

- Se desactivó la aceleración (throttling) del procesador
- Se configuró el reloj para que funcione siempre al máximo (extreme mode)
- Se disminuyó el voltaje (para evitar sobrecalentamiento del procesador, puesto que funciona al 100% de forma constante)

Para evitar que el procesador disminuya su velocidad cuando éste se encuentra ocioso, hay que agregar a las opciones de arranque la opcion "idle=poll" como se muestra a continuación:

```
  linux    /boot/vmlinuz-3.14.39-rt37-notrace-nohzoff  root=UUID=03e177f4-1569-
4615-aa07-97ec5324dfd6 ro  quiet splash $vt_handoff idle=poll
```

Explicar cada una de las opciones utilizadas está más allá del alcance de la presente tesina.

8.2 Aplicaciones utilizadas

En esta sección se hará una explicación de las aplicaciones utilizadas para los experimientos. La lista de aplicaciones es: cyclictest, jittertest, rtps, stress y taskset.

8.2.1 Cyclictest

El programa cyclictest mide la latencia ejecutando la llamada al sistema *clock_nanosleep* e inmediatamente una llamada al sistema *clock_gettime* (ambas llamadas forma parte de POSIX). El primer comando bloquea al thread que está ejecutando por un tiempo configurable en milisegundos y nanosegundos, y el segundo comando obtiene el tiempo actual. La latencia se obtiene calculando el tiempo transcurrido entre el momento que despierta el thread hasta el momento que se calcula el tiempo actual.

El programa permite configurar su ejecución de diferentes maneras, como el intervalo entre cada iteración, cantidad de threads, iteraciones, etc.

A continuación se enumeran los parámetros más importantes:

Parámetro	Descripción
-a	Cada thread se ejecutará con afinidad sobre un único procesador
-n	Utiliza nanosleep (sleep de alta resolución)
-pX	Indica la prioridad X
-iX	Tiempo X en microsegundos entre cada iteración
-hX	Arma un histograma, con muestras de latencias entre 0 y X microsegundos
-t	Crea tantos threads como procesadores haya
-N	Los resultados se imprimen en nanosegundos en lugar de microsegundos
-DX	Indica cuanto tiempo se ejecutará el test, pueden ser X segundos, Xm minutos, Xh horas o Xd días.
-bX	Interrumpe la ejecución cuando se alcanzan los X microsegundos - El kernel debe tener tracing activado - Utilizado para depurar el kernel y encontrar problemas de latencia
-TX	Indica el tracer utilizado - El kernel debe tener tracing activado
-f	Utiliza la función ftrace para el tracing. Se usa con el parámetro -b. El kernel debe tener activado el tracing y ftrace. Produce el mismo efecto que usar -TFUNCTION.

Se muestra un ejemplo de uso del programa y sus resultados:

```
$ sudo ./cyclictest -a -n -p99 -i500
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 10.23 10.35 8.66 11/539 3889

T: 0 ( 3345) P:99 I:500 C:2972767 Min:   2 Act:    3 Avg:    4 Max:     393
```

En el ejemplo se ejecutó cyclictest con la opción de afinidad (-a), utiliza nanosleep (-n), la prioridad máxima 99 (-p99) y 500 microsegundos de duración para el intervalo (-i500).

La información mostrada en la última línea se actualizará según el intervalo indicado. Se enumeran las leyendas a continuación:

- T: 0 (3345) - Indica el número de thread de la ejecución (ej: si ejecuto sobre 4 threads tendré los números 0,1,2,3). Entre paréntesis se indica el id del thread.
- P: 99 - Indica la prioridad del thread
- C: 2972767 - Indica la cantidad de iteraciones hechas durante la ejecución.
- Min: 2 - La mínima latencia durante la ejecución. En el ejemplo, todos los tiempos se expresan en microsegundos.
- Act: 3 - La ultima latencia o latencia actual registrada. Durante la ejecución este número es el que más cambia.
- Max: 393 - La máxima latencia registrada durante la ejecución.
- loadavg: 10.23 - Indica la carga que tiene el sistema. Un número cercano a 0 indica cpus ociosas.

El programa cyclictest es uno de los programas más utilizados para medir las latencias de un sistema operativo y saber si es apto para tiempo real. Se puede obtener, con una ejecución larga del programa, el tiempo de ejecución del peor caso, el cual es la latencia máxima obtenida en esa ejecución. Se utilizará como referencia de una aplicación hecha en C para luego comparar los tiempos de latencias obtenido con aplicaciones Java de tiempo real.

8.2.2 Jittertest

El programa jittertest es una aplicación de JamaicaVM que ejecuta un test periódico por una cantidad fija de tiempo, y va reduciendo el periodo hasta que ocurra un incumplimiento de plazos o el jitter exceda el 50%. Son cuatro test separados que se ejecutan de esta manera pero de cuatro formas distintas: utilizando un thread de tiempo real, un evento asincrónico con su manejador, un thread estándar de Java y un timer de la API de Java estándar, en este orden.

Cada test comienza su ejecución con un período de 500 milisegundos durante 1.5 segundos (3 iteraciones) y recoge estadísticas del tiempo de ejecución promedio, máximo y el jitter sobre el total de iteraciones para ese período seleccionado y muestra dos gráficos: la duración de cada período y los tiempos de ejecución.

Cuando termina esta ejecución, si no hubo incumplimiento de plazos o el jitter no excedió el 50%, se disminuye el período a 400 ms, 300 ms, 250 ms, 200 ms y así sucesivamente, hasta que

falle el test. Tener en cuenta que para cada período utilizado, siempre ejecuta por 1.5 segundos, por lo cual la cantidad de iteraciones aumentará al verse reducido el período.

Concluido el primer test (utilizando threads de tiempo real), se ejecutan los restantes tres tests. Cuando hayan terminado los 4 tests, se muestra un resumen de la ejecución de los cuatro períodos más pequeños que no hayan fallado para cada uno de los cuatro tests.

Se muestra como ejemplo los gráficos producidos por una ejecución para un período igual a 400 microsegundos y 3750 iteraciones:

```
   Period  400.00us: min= 339.14us ( 85%) max= 463.14us (116%) average= 399.99us sigma=
6.82us (  2%) (3750 iterations)
   count                         ---  actual periods  ---
     4K                                 *
     2K                                 *
     1K                                 *
   512                                  *
   256                                  *
   128                                  *
    64                                 ***
    32                                * *** *
    16                               *********
     8                               *********
     4                               *********
     2                               *********
     1                              ***********
        |       |       |       |       |       |       |       |       |       |
      0.00ns        200.00us        400.00us        600.00us        800.00us   >

   count                         ---  execution times  ---
     2K                          *
     1K                         ***
   512                         *****
   256                         *****
   128                        ********
    64                        ********
    32                        *********
    16                        *********
     8                        *********
     4                        *********
     2                        **********
     1                        ***********
        |         |         |         |         |         |         |           |
      1ns       32ns      1us       32us      1ms       32ms      1s          >
```

El gráfico de arriba muestra que hubo entre 2049 y 4096 iteraciones cuyo período exacto fueron 400 microsegundos. Abajo se muestra un gráfico con escala logarítmica en ambos ejes. Entre cada asterisco, de forma horizontal se incrementa aproximadamente 1.41 del valor anterior; por lo tanto, el pico observado indica que hubo entre 1025 y 2048 iteraciones que tuvieron un tiempo de ejecución de 3.95 microsegundos (1.41 * 4).

Finalmente se muestra el resumen de los cuatro tests de una ejecución. Tener en cuenta que los dos primeros tests utilizan `RealtimeThread` y `AsyncEventHandler` que son de tiempo real. Los otros dos tests no son de tiempo real.

```
                  ---  RESULT SUMMARY ---
-----------------------+-----------------------------------+---------------
         period        |          execution time           |
                       |                                   | Test
 selected  |  average  |  average  |    max    |  jitter   |
-----------+-----------+-----------+-----------+-----------+---------------
  200.00us |  199.99us |    4.20us |   25.49us |    2.08us | RealtimeThread
  150.00us |  149.99us |    4.71us |   28.22us |    2.48us | AsyncEventHandler
  400.00us |  399.99us |    4.98us |   27.63us |    2.47us | Java Thread
    1.00ms |    1.00ms |    6.06us |   52.21us |    3.08us | java.util.Timer
```

Este programa se mostró útil para demostrar que JamaicaVM tiene características de un programa de tiempo real y para medir el jitter de un sistema utilizando threads y eventos de Java. También muestra la diferencia de predictibilidad entre los threads de tiempo real y los threads comunes de Java, como se verá en los experimentos más adelante.

8.2.3 Real Time Period Simple (rtps)

El programa rtps es una aplicación propia escrita en Java que consiste en la ejecución de un thread periódico que mide la latencia de planificación entre el estado listo y ejecutando. Se ejecutará sobre JamaicaVM para medir las latencias entre la activación de un thread periódico y su ejecución.

Dado que ni JamaicaVM ni la especificación RTSJ ofrecen una forma directa de obtener el instante en que comienza a ejecutar un thread periódico desde su activación, para calcular la latencia se asigna a una variable un tiempo absoluto de comienzo de ejecución del thread y un período configurable. De esta forma se puede saber para cada activación el instante justo en el cual el thread pasa al estado de listo a partir del tiempo de comienzo y el numero de activación. Escrito en una formula:

*instante de activación del thread = tiempo de comienzo + (periodo * numero de activación)*

Para obtener la latencia hay que obtener el tiempo en el instante que el thread comienza a ejecutar y obtener la diferencia entre este tiempo y el instante de activacion del thread:

latencia = instante del comienzo de ejecución - instante de activación del thread

El programa además registra cada tiempo y al final de la ejecución guarda en disco varias

estadísticas: el tiempo máximo y promedio de latencia, un histograma con la cantidad de activaciones ocurridas por microsegundo y un resumen de los incumplimientos de plazos que ocurrieron si los hubo.

Para ejecutarlo, se cuenta con un archivo de configuración rtps.properties y con parámetros en la línea de comandos. La sintaxis del programa es:

```
$ rtps p ms us.
```

Los parámetros se explican a continuación:

- p - Prioridad de Java según RTSJ: un número entre 1 y 38. 38 es la máxima prioridad, y está mapeada a la prioridad de tiempo real máxima 99 de Linux.
- ms, us - Estos dos campos definen el período utilizado para el thread periódico; ms son los milisegundos y us los microsegundos.

Un ejemplo de ejecución

```
$ ./rtps 37 0 500
```

La ejecución anterior hará que rtps utilice un thread de tiempo real periódico con prioridad 37 y periodo de 500 microsegundos.

El archivo de configuración rtps.properties permite configurar las siguientes propiedades:

Propiedad	Descripción
`terminateOnFirstMiss`	Configurar en `true` si se desea que la ejecución termine cuando ocurre un incumplimiento de plazos. `false` en caso contrario.
`numberOfSecondsToEndProgram`	Indica la cantidad de segundos de estará ejecutando el test.
`maxMicrosecondsHistogram`	El histograma construido contabilizará latencias por microsegundos hasta un máximo indicado por este parámetro (ej: 300 ms).

`groupOfReleasesWithMaxAvg`	Indica cuantos grupos se utilizarán en la corrida. La cantidad de grupos define la cantidad de fracciones de tiempo sobre el total dado por el parámetro `numberOfSecondsToEndProgram`. Un grupo contiene información independiente de otro grupo, e informa la máxima latencia y la latencia promedio para ese grupo o fracción de tiempo. Al finalizar la ejecución, cada grupo aparecerá en el resultado con un promedio y máximo.

Con este programa, comparándolo con cyclictest con ejecuciones largas, se mostrará que Java es un lenguaje apto para desarrollar aplicaciones de tiempo real duro.

8.2.4 stress

El programa "stress" permite crear threads para poder utilizar al 100% los procesadores disponibles en el sistema.

Por ejemplo, ejecutando el siguiente comando se crearán 10 threads que constantemente están haciendo cálculos (calculan indefinidamente la función de raíz cuadrada):

```
$ stress -c 10
```

El propósito de la utilización de este programa es para poder probar las aplicaciones de tiempo real en un sistema con carga. De aquí en adelante diremos que el sistema (o la prueba) es con stress cuando se está ejecutando este programa. Cuando no se utiliza diremos que el sistema está sin stress.

8.2.5 taskset

Es una aplicación que permite ejecutar otros programas pero de manera restringida: el programa correrá únicamente en uno o más procesadores indicados por el comando, o en otras palabras, se asigna la afinidad de los procesos y threads para que ejecuten en uno o varios procesadores dados.

Esta aplicación es útil para aislar la ejecución de los programas, de forma de tener por un lado los procesos/threads de aplicaciones que no son tiempo real en un grupo de procesadores, y aplicaciones de tiempo real en otro grupo.

Un proceso o grupo de threads cuando ejecutan siempre en el mismo procesador, suele ser más predecible puesto que mantiene la localidad de los datos e instrucciones en la caché.

Utilizando el parámetro -c se puede indicar qué procesador o procesadores ejecutarán el comando. Por ejemplo la siguiente llamada ejecuta un script en el procesador 1:

```
$ taskset -c 1 script.sh
```

En las pruebas se utilizará este comando para ejecutar los procesos siempre sobre un único procesador para tener la máxima predictibilidad.

8.3 Preliminares

Esta sección explicará como se organizaron los experimentos.

Primero se ejecutará cyclictest a fin de mostrar las latencias que se obtienen sobre un sistema operativo que no soporta tiempo real y un sistema con tiempo real, a través de pruebas de corta duración, y luego se harán pruebas más extensivas de 24 hs sobre el sistema de tiempo real para obtener un valor máximo de latencia más fiel.

En particular se utilizará el kernel 3.14.39 para las pruebas, sin el parche rt-preempt, y otro kernel con el parche (3.14.39-rt37).

En todas las pruebas (salvo las de 24 hs), cyclictest se ejecutará con solo un thread con afinidad a una CPU, utilizando la opción de nanosleep, prioridad 99 (la máxima), iteración cada 500 microsegundos y guardando muestras de cada latencia en un histograma hasta un máximo de 500 microsegundos. En linea de comando:

```
$ sudo ./cyclictest -a -n -p99 -i500 -h500
```

El programa jittertest, en cada experimento particular, se ejecutará 9 veces y se mostrara un resumen con los 9 resultados. Solo alcanza con ejecutar el siguiente comando:

```
$ ./jittertest
```

Para ambos programas, se ejecutará con y sin el programa stress. Con el mencionado comando se crearan 10 threads que tratarán de utilizar todas las CPUs. Tener en cuenta que estos threads no tienen prioridad de tiempo real.

```
$ stress -c 10
```

Por último, el programa rtps se utilizará solo para las pruebas de 24 hs para detectar el peor caso de tiempos de latencias utilizando JamaicaVM en el sistema utilizado, para luego seleccionar el período más pequeño posible y más apropiado para una tarea periódica de tiempo real. Finalmente se compararán con las pruebas de cyclictest y se obtendrán conclusiones.

8.4 cyclictest sobre un SO sin soportar tiempo real

Se ejecutará el programa cyclictest sobre un sistema operativo sin el parche rt-preempt, con y sin stress para luego comparar sus resultados con lo obtenidos con el mismo programa pero sobre un kernel con el parche.

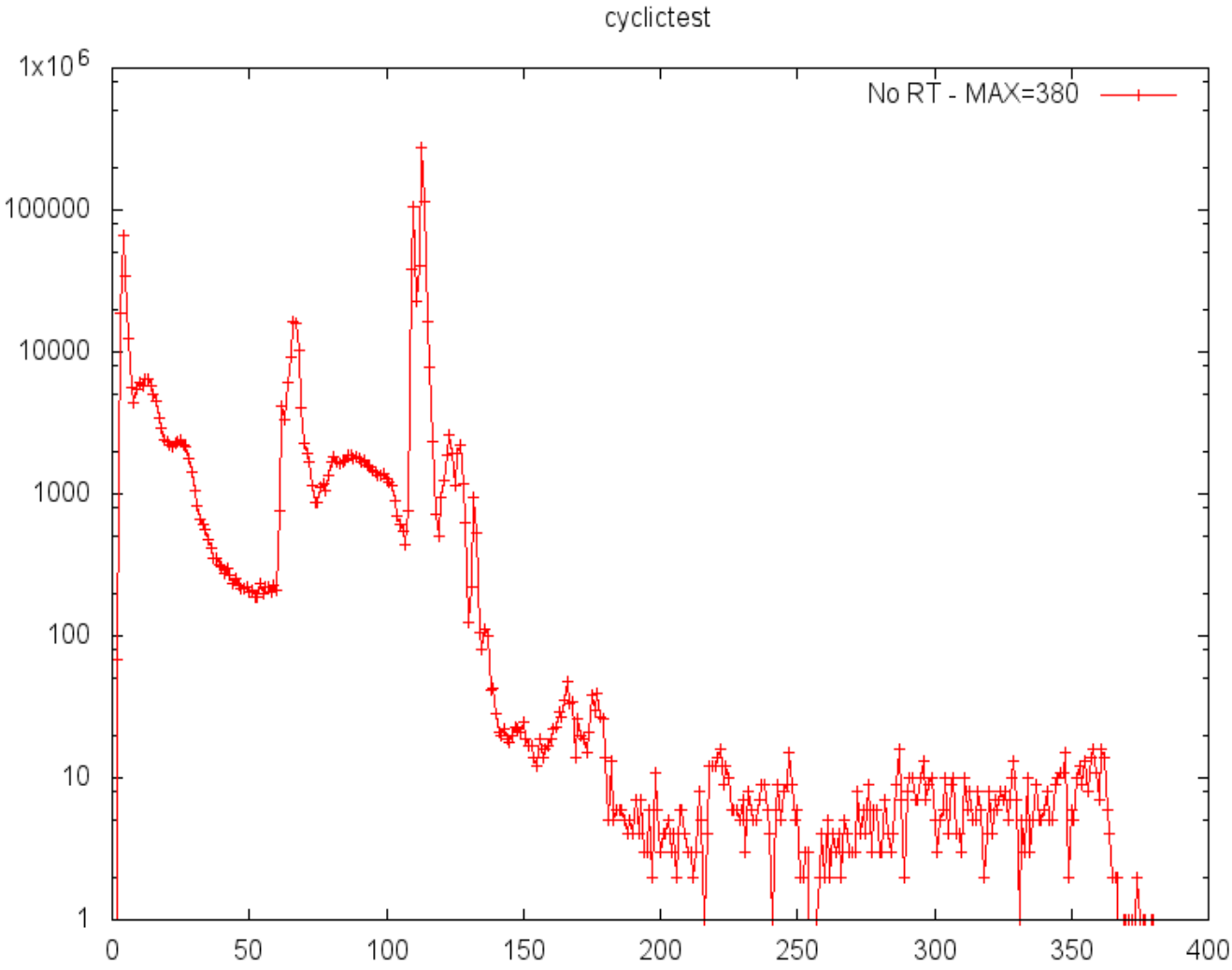
Sin stress

Se ejecuta cyclictest utilizando el kernel sin el patch de tiempo real.

Primero se ejecutará sin stress, por lo tanto el proceso tiene disponible el procesador la mayoría del tiempo, con poca competencia con otros procesos por estar mayormente en estado ocioso (idle).

A continuación se muestran los resultados, junto con un gráfico:

```
T: 0 ( 2493) P:99 I:500 C:1003241 Min: 2 Avg:   84 Max: 380
```



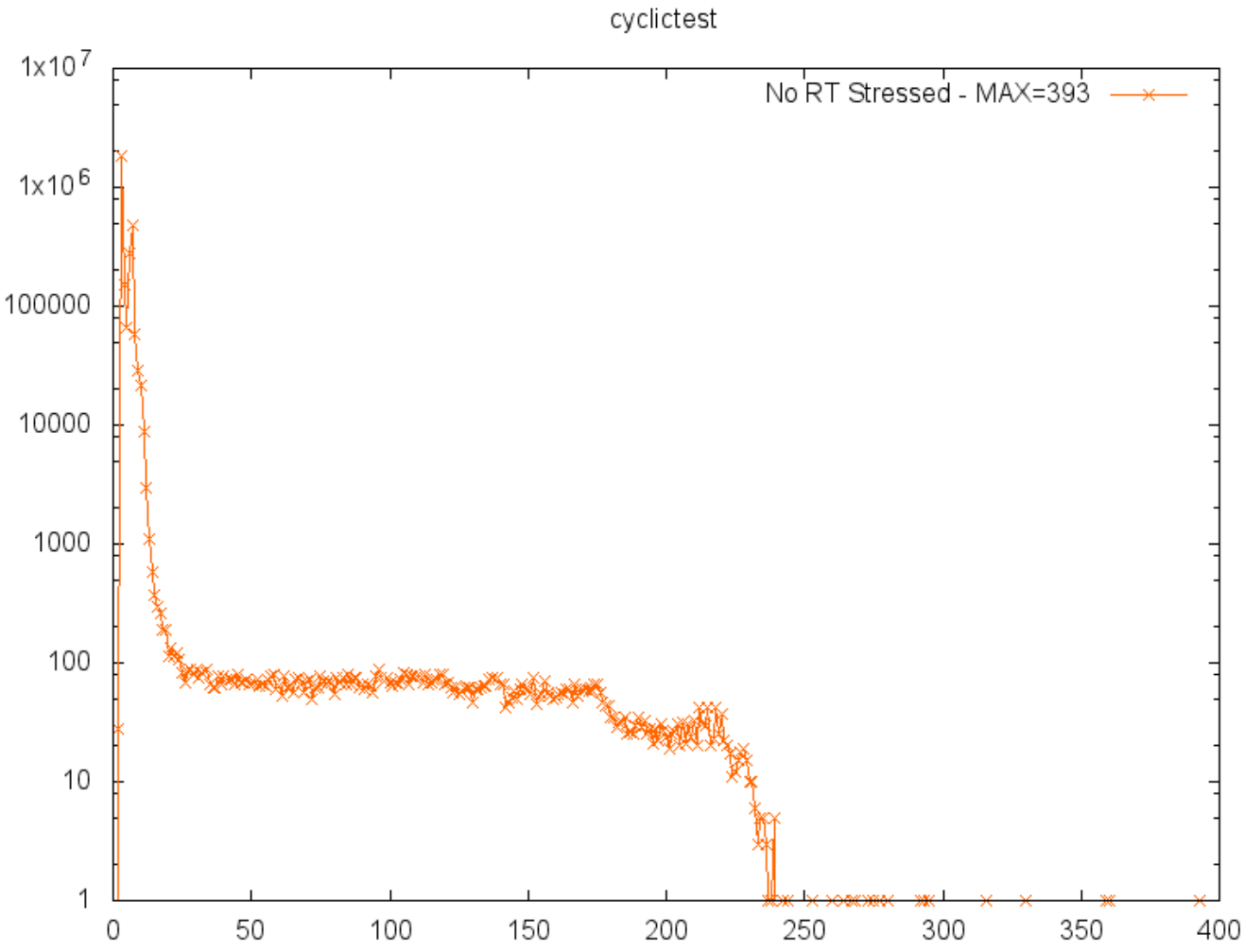
Se observa una gran variabilidad en las latencias, con un promedio de 84 microsegundos y alcanzando un máximo de 380 microsegundos (casi 5 veces más que el promedio).

Con stress

Se corre el mismo test, pero con 10 threads que compiten constantemente por el procesador, aunque tienen menor prioridad que cyclictest.

A continuación los resultados:

```
T: 0 ( 3345) P:99 I:500 C:2972767 Min: 2 Avg: 4 Max: 393
```



Notar que el promedio mejoró considerablemente (4 microsegundos contra 84); se debe a que en el primer ejemplo las CPUs están la mayoría del tiempo ociosas y suelen bajar su desempeño por tener activadas opciones de ahorro de energía; por lo tanto la latencia sube en promedio.

Con stress se muestra una mayor latencia que en el ejemplo sin stress (393 microsegundos contra 380).

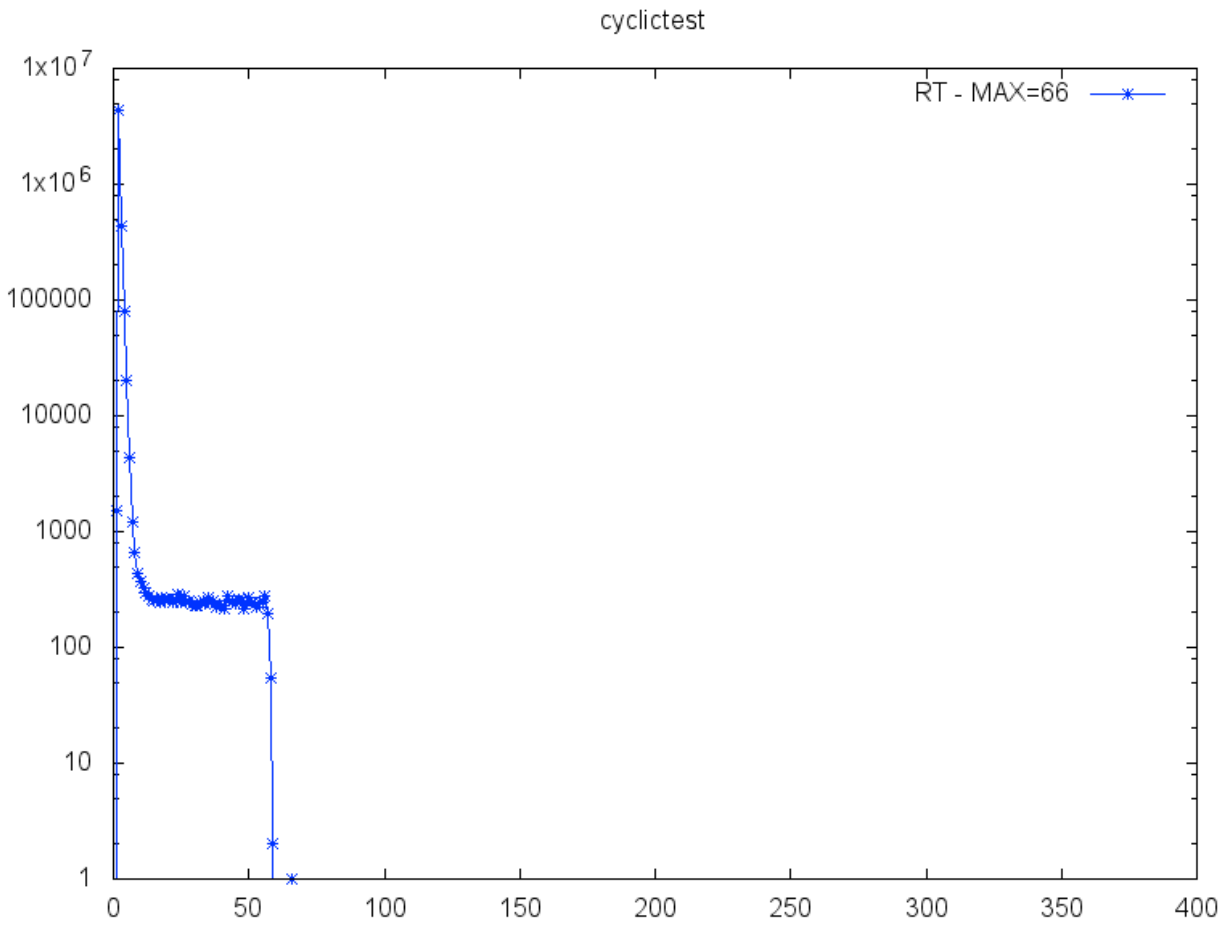
8.5 cyclictest sobre un SO de tiempo real

A continuación se mostrarán corridas de cyclictest sobre un sistema operativo, con el parche de rt-preempt. Se muestran los resultados junto con su gráfico para pruebas sin stress y con stress.

Sin stress

Se muestra a continuación el resultado de la ejecución de cyclictest junto a su gráfico:

```
T: 0 ( 2593) P:99 I:500 C:4860337 Min:  1 Act:    2 Avg:    2 Max:      66
```

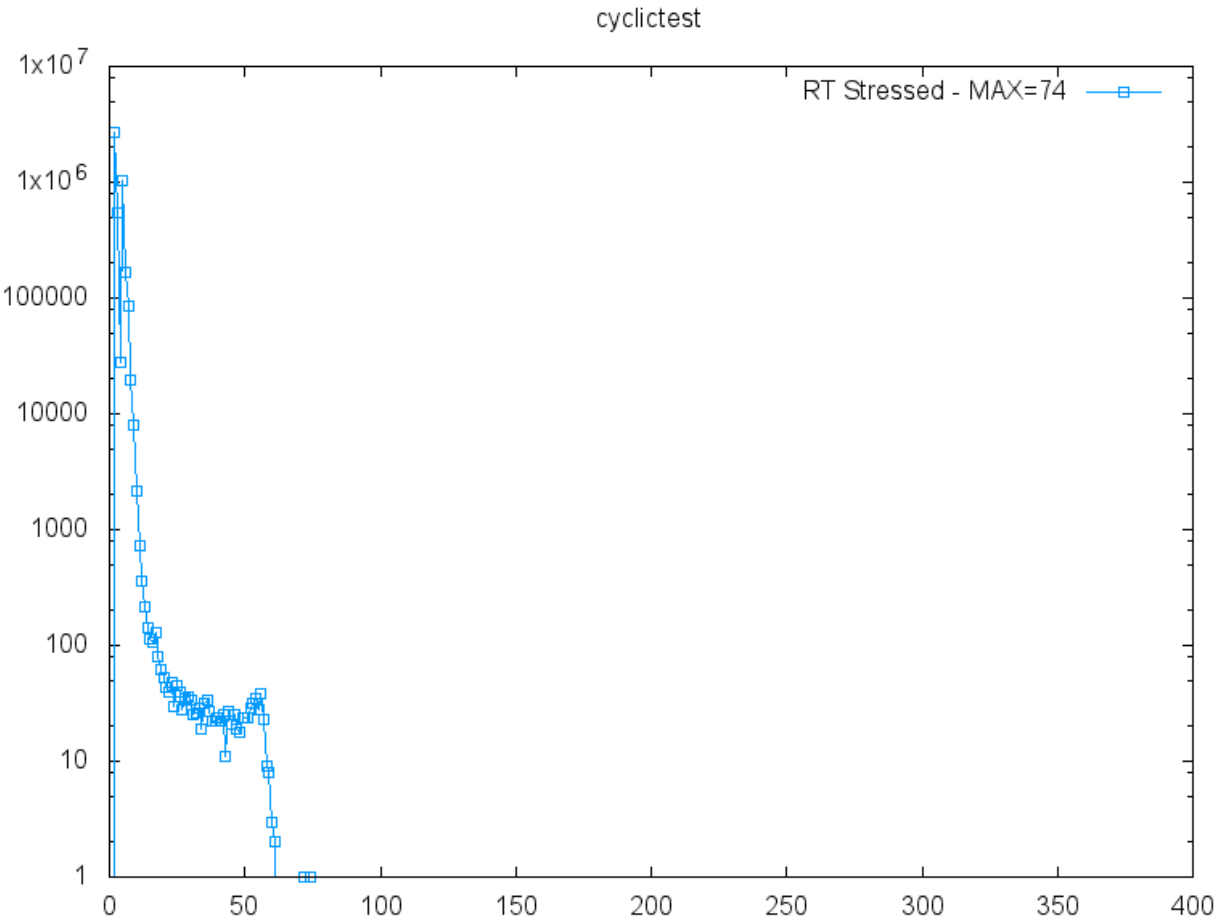


Se puede observar una gran diferencia en la mejora de los tiempos: del máximo obtenido en la prueba anterior sin tiempo real, que fue igual a 380us, bajó a 66us. También se observa un gráfico más estable, con un pico alto que ronda en los 2 us.

Con stress

A continuación los resultados de cyclictest y su gráfico con stress:

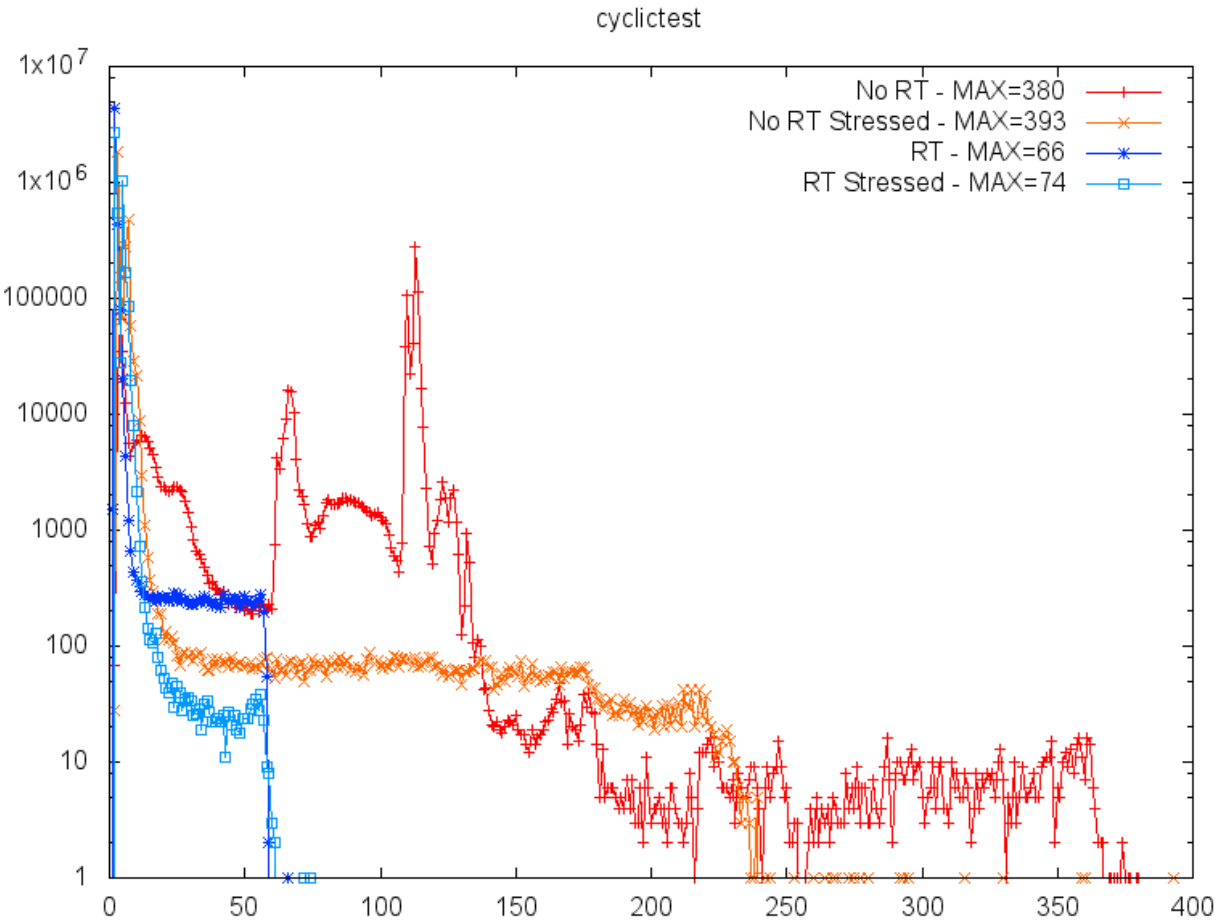
```
T: 0 ( 2854) P:99 I:500 C:4565136 Min:  2 Act:    3 Avg:    3 Max:      74
```



Comparada con la prueba anterior, se observa una leve desmejora cuando el sistema está sobrecargado: el promedio subió un microsegundo y el máximo subió a 74 us.

8.6 Comparación de las pruebas de cyclictest

A continuación se mostrará un gráfico que muestra las cuatro pruebas anteriores representadas en cuatro curvas:



En el gráfico se puede visualizar claramente las diferencias entre correr cyclictest en un sistema sin tiempo real y un sistema con tiempo real. Las curvas azul y celeste representan las pruebas con tiempo real y las curvas naranja y rojo las pruebas sin tiempo real. Las primeras tienen gráficas que muestran latencias no mayores a 74 microsegundos y con una curva más predecible y suave. Las últimas muestran latencias mucho mayores, especialmente la curva roja (prueba sin stress sin tiempo real).

Sin embargo, estos resultados se obtuvieron con pruebas de muy corta duración (no más de 5 minutos) y los máximos obtenibles pueden ser aún mayores, pero para el propósito de comparar latencias entre sistemas de tiempo real y sin tiempo real es más que suficiente.

En secciones posteriores se harán pruebas de alta duración y también se configurará el kernel y el BIOS para mejorar aún más la latencia del sistema, para luego compararlas con las latencias obtenidas con JamaicaVM a través de rtps.

8.7 jittertest sobre un SO sin soportar tiempo real

Se ejecutará el test jittertest sobre un sistema sin el parche rt-preempt un total de 9 veces sin stress y 9 veces con stress. Cada uno de los 9 resúmenes generados se unen en un único archivo. Se ejecutaron 9 veces porque cada uno fue arrojando diferentes resultados.

Estos resúmenes solo muestran los resultados de los dos tests de tiempo real (`RealtimeThread` y `AsyncEventHandler`). Los tests que no son de tiempo real (únicamente threads estándar de Java) se mostrarán utilizando un kernel de tiempo real más adelante.

Sin stress

Los resultados obtenidos se muestran a continuación:

period		execution time			
selected	average	average	max	jitter	Test
200.00us	199.99us	4.16us	28.56us	2.00us	periodic RealtimeThread
200.00us	199.99us	4.11us	25.51us	1.96us	periodic RealtimeThread
200.00us	199.99us	4.20us	27.29us	2.08us	periodic RealtimeThread
200.00us	199.99us	4.55us	26.18us	2.61us	periodic RealtimeThread
200.00us	199.99us	4.14us	27.45us	2.04us	periodic RealtimeThread
200.00us	199.99us	4.06us	26.29us	1.96us	periodic RealtimeThread
200.00us	199.99us	4.15us	24.87us	1.97us	periodic RealtimeThread
200.00us	199.99us	4.32us	25.37us	2.06us	periodic RealtimeThread
200.00us	199.99us	4.16us	24.75us	2.01us	periodic RealtimeThread
150.00us	149.99us	4.55us	30.97us	2.10us	periodic AsyncEventHandler
150.00us	149.99us	4.65us	25.33us	2.23us	periodic AsyncEventHandler
150.00us	149.99us	4.49us	26.49us	2.10us	periodic AsyncEventHandler
150.00us	149.99us	4.50us	**25.98us**	2.06us	periodic AsyncEventHandler
150.00us	149.99us	4.49us	26.01us	2.09us	periodic AsyncEventHandler
150.00us	149.99us	4.51us	34.30us	2.12us	periodic AsyncEventHandler
125.00us	124.99us	4.83us	**84.74us**	2.43us	periodic AsyncEventHandler
150.00us	149.99us	4.55us	**35.39us**	2.14us	periodic AsyncEventHandler
150.00us	149.99us	4.54us	26.13us	2.08us	periodic AsyncEventHandler

Se puede observar que los tests fueron muy parejos utilizando threads de tiempo real: periodo con duración de 200 microsegundos sin incumplimiento de plazos ni jitter excedido las 9 ejecuciones. Utilizando eventos asincrónicos se obtuvieron mejores tiempos y en una ejecución el período alcanzó los 125 microsegundos, a pesar de que por lo menos una iteración (entre un total de 12000 iteraciones) estuvo ejecutando 84.74 microsegundos. Las restantes 8 ejecuciones fueron muy parejas y similares entre si con un período de 150 microsegundos y con una ejecución máxima de 35.39 microsegundos entre las 8 ejecuciones.

Observando estos resultados se observa que el kernel sin el parche de tiempo real funciona muy bien cuando el sistema operativo está con la CPU mayormente ociosa. Sin embargo hay que

observar los resultados obtenidos en la siguiente prueba con la CPU ocupada para poder sacar conclusiones.

Con stress

Se ejecutaron las 9 ejecuciones nuevamente, pero esta vez con la CPU ejecutando la aplicación stress, ocupando el 100% de la CPU con threads de aplicación de baja prioridad.

Se muestra a continuación el resumen de las 9 ejecuciones:

period		execution time			Test
selected	average	average	max	jitter	
300.00us	299.99us	4.86us	33.27us	2.32us	periodic RealtimeThread
1.00ms	999.96us	5.27us	22.74us	2.21us	periodic RealtimeThread
250.00us	249.99us	5.12us	29.75us	2.27us	periodic RealtimeThread
400.00us	399.99us	5.16us	34.15us	2.40us	periodic RealtimeThread
300.00us	299.99us	5.08us	37.85us	2.30us	periodic RealtimeThread
250.00us	249.99us	5.24us	28.51us	2.37us	periodic RealtimeThread
300.00us	299.99us	5.09us	31.17us	2.31us	periodic RealtimeThread
250.00us	250.00us	5.23us	23.72us	2.25us	periodic RealtimeThread
250.00us	249.99us	5.20us	37.00us	2.31us	periodic RealtimeThread
300.00us	300.00us	5.90us	29.84us	3.03us	periodic AsyncEventHandler
200.00us	200.00us	5.43us	30.63us	2.40us	periodic AsyncEventHandler
250.00us	249.99us	5.40us	36.31us	2.40us	periodic AsyncEventHandler
250.00us	250.00us	5.65us	**36.39us**	3.08us	periodic AsyncEventHandler
250.00us	250.00us	5.51us	34.39us	2.49us	periodic AsyncEventHandler
250.00us	249.99us	5.45us	27.93us	2.44us	periodic AsyncEventHandler
200.00us	200.00us	5.70us	**136.46us**	3.20us	periodic AsyncEventHandler
250.00us	250.00us	5.43us	26.07us	2.40us	periodic AsyncEventHandler
250.00us	250.00us	5.38us	36.30us	2.44us	periodic AsyncEventHandler

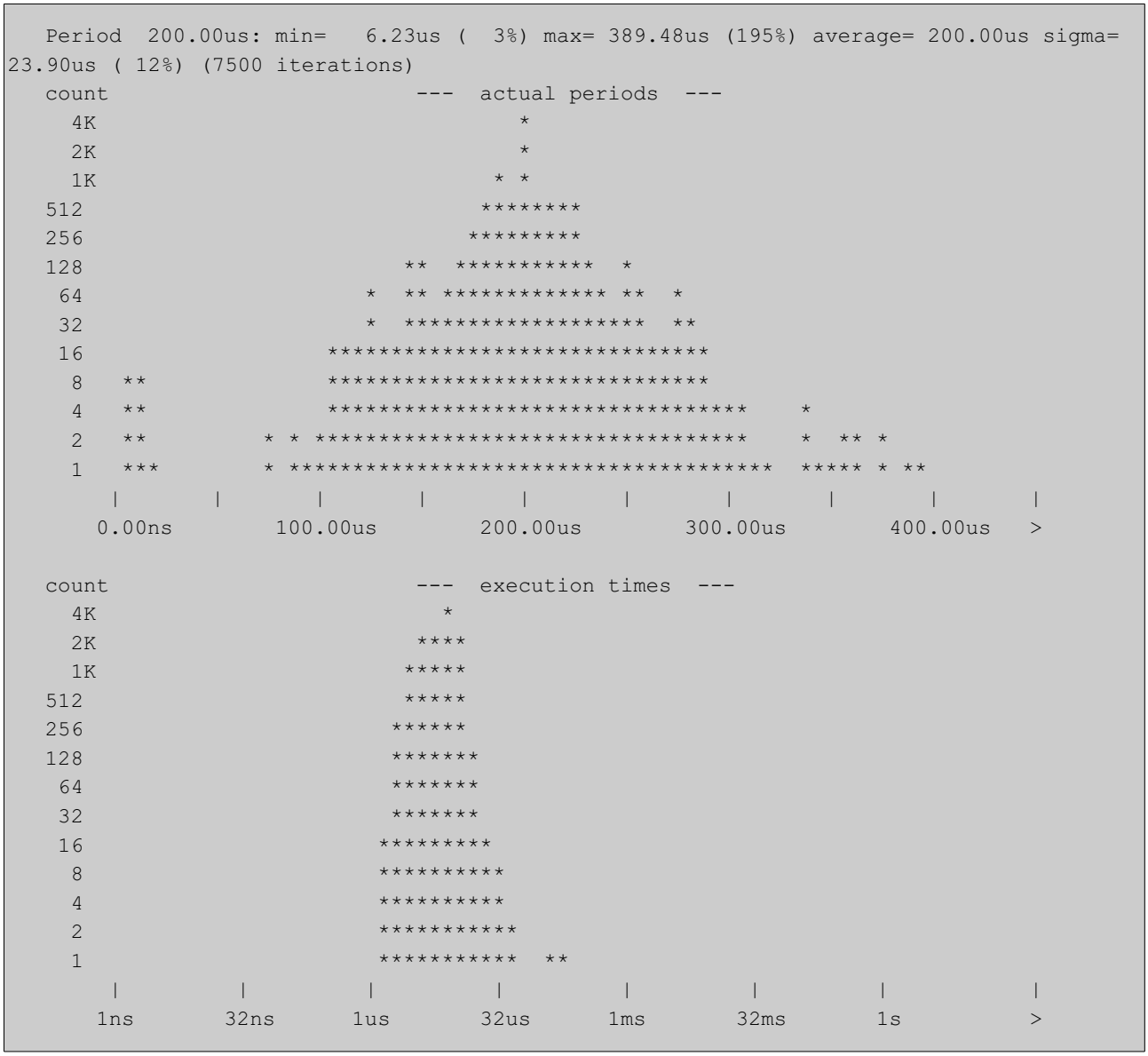
Aquí se pueden observar resultados totalmente diferentes a los anteriores. Se observa tanto para los tests hechos con threads de tiempo real y con eventos asincrónicos que hay mayor variabilidad entre cada ejecución.

La ejecución que más llama la atención es la que indica 1 milisegundo con el menor período encontrado en la cual no ocurrió un incumplimiento de plazos (la ejecución falló cuando quiso utilizar un período de 750 microsegundos en la primera iteración); el mayor entre todas las ejecuciones con threads de tiempo real. La segunda ejecución más grande fue con un período de 400 milisegundos.

En cuanto a las ejecuciones utilizando eventos asincrónicos, se obtuvo una ejecución con un período máximo de 300 microsegundos, el doble de tiempo que las pruebas sin stress. También

ocurrió en una de las ejecuciones que hubo una iteración que ejecutó 136.46 microsegundos, un tiempo mayor a tres veces más comparado contra el segundo máximo que fue 36.39 microsegundos.

Veremos por último los gráficos generados para la ejecución con el menor período exitoso, el cual fue 200 milisegundos:



El primer gráfico muestra un gráfico muy irregular y con mucho jitter, a pesar de no haber ocurrido ningún incumplimiento de plazos.

El segundo gráfico es más parejo, aunque se ven dos ejecuciones con un exceso de tiempo de ejecución (el máximo fue de 136.46 microsegundos como se vio antes).

Estas pruebas alcanzan para mostrar la falta de predictibilidad de un kernel que no es de tiempo real con una máquina virtual de JamaicaVM de tiempo real ejecutando jittertest.

8.8 jittertest sobre un SO de tiempo real

Continuando con el programa jittertest, ahora se mostraran los resultados obtenidos utilizando un kernel de tiempo utilizando el patch rt-preempt.

Los resultados obtenidos utilizando threads estándar de Java que no son de tiempo real se mostrarán aparte al final de la sección para mostrar que Java Standard Edition no es apropiado para desarrollar aplicaciones de tiempo real.

Sin stress

Se muestra el resumen de las 9 ejecuciones sobre un kernel de tiempo real sin stress:

period		execution time			Test
selected	average	average	max	jitter	
150.00us	149.99us	5.13us	29.53us	2.37us	periodic RealtimeThread
200.00us	199.99us	4.95us	28.36us	2.22us	periodic RealtimeThread
150.00us	149.99us	4.99us	32.52us	2.32us	periodic RealtimeThread
200.00us	199.99us	5.01us	23.29us	2.21us	periodic RealtimeThread
150.00us	149.99us	5.12us	28.12us	2.29us	periodic RealtimeThread
200.00us	200.00us	5.32us	34.39us	2.51us	periodic RealtimeThread
150.00us	149.99us	5.11us	27.49us	2.35us	periodic RealtimeThread
200.00us	199.99us	4.90us	28.19us	2.27us	periodic RealtimeThread
150.00us	149.99us	5.22us	27.03us	2.44us	periodic RealtimeThread
200.00us	199.99us	5.00us	29.03us	2.28us	periodic AsyncEventHandler
200.00us	199.99us	5.10us	30.05us	2.28us	periodic AsyncEventHandler
150.00us	149.99us	5.21us	29.80us	2.35us	periodic AsyncEventHandler
150.00us	150.00us	5.34us	34.47us	2.44us	periodic AsyncEventHandler
200.00us	199.99us	5.36us	**52.26us**	2.54us	periodic AsyncEventHandler
150.00us	149.99us	5.19us	28.03us	2.30us	periodic AsyncEventHandler
150.00us	149.99us	5.25us	30.31us	2.30us	periodic AsyncEventHandler
200.00us	199.99us	5.03us	29.02us	2.34us	periodic AsyncEventHandler
150.00us	149.99us	5.36us	29.38us	2.34us	periodic AsyncEventHandler

Se pueden observar que tanto para threads de tiempo real como para eventos asincrónicos, los resultados son muy similares, con un período que no excede los 200 microsegundos.

De forma similar, los tiempos de ejecución fueron bajos y con poca variabilidad, excepto una iteración de una ejecución el cual estuvo ejecutando 52.26 microsegundos, a diferencia de los otros máximos encontrados en las demás ejecuciones que fue un poco menor.

Con stress

A continuación se muestra el resumen de las 9 ejecuciones con un kernel de tiempo real y con stress:

period		execution time			Test
selected	average	average	max	jitter	
200.00us	200.00us	5.61us	34.36us	2.56us	periodic RealtimeThread
200.00us	199.99us	5.22us	26.55us	2.26us	periodic RealtimeThread
200.00us	199.99us	5.33us	25.70us	2.29us	periodic RealtimeThread
200.00us	200.00us	5.44us	45.60us	2.55us	periodic RealtimeThread
150.00us	149.99us	5.46us	33.21us	2.36us	periodic RealtimeThread
200.00us	200.00us	5.26us	28.06us	2.33us	periodic RealtimeThread
200.00us	199.99us	5.31us	29.86us	2.31us	periodic RealtimeThread
150.00us	149.99us	5.49us	34.20us	2.50us	periodic RealtimeThread
200.00us	199.99us	5.39us	30.49us	2.46us	periodic RealtimeThread
150.00us	149.99us	5.65us	**81.85us**	2.68us	periodic AsyncEventHandler
150.00us	149.99us	5.65us	27.73us	2.44us	periodic AsyncEventHandler
150.00us	149.99us	5.50us	31.59us	2.44us	periodic AsyncEventHandler
200.00us	200.00us	5.51us	27.21us	2.37us	periodic AsyncEventHandler
200.00us	199.99us	5.47us	27.54us	2.33us	periodic AsyncEventHandler
150.00us	149.99us	5.65us	28.72us	2.46us	periodic AsyncEventHandler
150.00us	149.99us	5.69us	30.29us	2.59us	periodic AsyncEventHandler
125.00us	125.00us	5.84us	31.74us	2.48us	periodic AsyncEventHandler
150.00us	149.99us	5.69us	33.24us	2.62us	periodic AsyncEventHandler

Los resultados son muy similares a las ejecuciones hechas sin stress. Hubo una ejecución con un período menor a los encontrados sin stress, el cual fue de 125 microsegundos. En este caso hubo una ejecución que tuvo una iteración que estuvo ejecutando 81.85 microsegundos, más del doble que los demás máximos de las otras ejecuciones.

Se muestra el detalle de esta ejecución particular (solo el primer gráfico):

```
   Period  150.00us: min=   9.33us (  6%) max= 271.13us (181%) average= 149.99us sigma=
16.45us ( 11%) (10000 iterations)
   count                         ---  actual periods  ---
     8K                                 *
     4K                                 *
     2K                                 *
     1K                                ***
   512                           *       ****      *
   256                           *  * * ***** * *  *
   128                           ** *************  *
    64                           ********************
    32                          **********************
    16                          **********************   *
     8                       * ******************************
     4                      ** ******************************
     2    *                 ********************************
     1    *       *  ** ************************************ **    *
        |       |       |       |       |       |       |       |       |       |
      0.00ns          75.00us         150.00us         225.00us         300.00us   >

   count                         ---  execution times  ---
     4K                           **
     2K                          ****
     1K                          *****
   512                          ******
   256                          *******
   128                          *******
    64                          *******
    32                         *********
    16                         **********
     8                         **********
     4                         **********
     2                         **********
     1                         *********** *
        |         |         |         |         |         |         |           |
      1ns       32ns      1us       32us      1ms       32ms       1s          >
```

El primer gráfico es muy similar al gráfico equivalente a la prueba con un kernel estándar y con stress. Tener en cuenta que el período utilizado en esta ejecución es menor a su equivalente sin tiempo real (150 microsegundos contra 200 microsegundos).

Finalmente mostraremos los gráficos de la primera ejecución con threads de tiempo real (período 200 ms) para compararlo con el gráfico de la ejecución con el kernel sin tiempo real:

```
   Period  200.00us: min= 137.09us ( 69%) max= 260.77us (130%) average= 200.00us sigma=
7.28us (  4%) (7500 iterations)
   count                         ---  actual periods  ---
     8K                                 *
     4K                                 *
     2K                                 *
     1K                                 *
   512                                 ***
   256                                 ****
   128                                ******
    64                              ** ********
    32                              *************
    16                             ***************
     8                             ****************
     4                            ******************
     2                          * ******************
     1                          **********************
        |       |       |       |       |       |       |       |       |       |
      0.00ns        100.00us        200.00us        300.00us        400.00us   >

   count                         ---  execution times  ---
     4K                              *
     2K                             ****
     1K                            *****
   512                             *****
   256                            *******
   128                            *******
    64                            *******
    32                           ********
    16                           *********
     8                           **********
     4                           **********
     2                           ***********
     1                           ***********
        |         |         |         |         |         |         |           |
      1ns       32ns      1us       32us      1ms       32ms       1s          >
```

Claramente se observa un gráfico que muestra tiempos más cercanos al promedio, sin mucha variabilidad como su gráfico equivalente sin tiempo real.

Comparando los dos gráficos relativas a los períodos 150 ms y 200 ms, se puede observar que si se utiliza un período muy pequeño, la variabilidad de cada activación aumenta.

Java Standard con stress

Se mostrará a continuación el resumen de las 9 ejecuciones hechas con stress en la prueba anterior (con tiempo real), teniendo en cuenta únicamente los tests de threads estándar de Java:

period		execution time			Test
selected	average	average	max	jitter	
4.00ms	3.99ms	9.55us	22.10us	3.17us	(not realtime) Java Thread
400.00us	399.99us	6.33us	28.05us	2.69us	(not realtime) Java Thread
1.00ms	1.00ms	6.93us	20.01us	2.75us	(not realtime) Java Thread
750.00us	750.01us	6.41us	30.18us	3.21us	(not realtime) Java Thread
7.50ms	7.49ms	9.83us	23.03us	3.82us	(not realtime) Java Thread
1.00ms	1.00ms	7.18us	21.35us	2.89us	(not realtime) Java Thread
750.00us	749.99us	6.78us	22.53us	2.88us	(not realtime) Java Thread
1.25ms	1.25ms	7.50us	25.81us	2.90us	(not realtime) Java Thread
1.25ms	1.24ms	7.77us	28.00us	3.02us	(not realtime) Java Thread

Los resultados muestran ejecuciones muy desparejas, con una gran variabilidad en el período para cada ejecución. Una ejecución tuvo un período de 7.5 milisegundos contra un mínimo de 400 microsegundos.

Se puede concluir con estos resultados que Java Standard no es apropiado para el desarrollo de aplicaciones de tiempo real.

8.9 Comparación de las pruebas de jittertest

Las pruebas hechas con jittertests mostraron que si se utiliza JamaicaVM sobre un sistema operativo que no es de tiempo real, entonces una aplicación de JamaicaVM tampoco lo será.

Los resultados mostraron buenos tiempos, tanto para un kernel estándar y otro de tiempo real, cuando la CPU se encuentra mayormente ocioso. En cambio cuando la CPU está ocupada la mayor parte del tiempo, las ejecuciones sobre el kernel de tiempo real no mostraron cambios en los tiempos obtenidos, a diferencia de las ejecuciones con kernel sin tiempo real que fueron mucho mayores.

También se observó que los gráficos muestran que las pruebas hechas sobre el kernel de tiempo real muestran valores mucho más predecibles que las pruebas equivalentes con el kernel sin tiempo real.

Por último, se mostró que Java estándar no es apto para desarrollar aplicaciones de tiempo real.

8.10 cyclictest: pruebas de 24 hs sobre SO de tiempo real

En esta sección se mostraran pruebas de larga duración hechas con cyclictest y así obtener la latencia más larga obtenida por un programa C.

Sin stress

Primero se muestra una prueba con el CPU en su mayoría del tiempo en estado ocioso. El programa cyclictest se ejecutó con el siguiente comando:

```
$ taskset -c 2 ./cyclictest -a -n -p95 -i100 -h100 -D24h
```

Es una ejecución con un intervalo de 100 microsegundos, de 24 horas de duración, ejecutando con nanosleep, con prioridad 95 (la máxima es 99) y ejecutando siempre en la CPU con id 2.

Como una aclaración, NO se debería ejecutar una aplicación de tiempo real utilizando la máxima prioridad del sistema operativo para no comprometer su estabilidad por alguna falla en el programa. Además el sistema operativo mantiene algunos procesos corriendo en la máxima prioridad, como los llamados watchdogs, que sirven para interrumpir cualquier programa que se haya apropiado de la CPU por mucho tiempo y mantener la integridad del sistema operativo.

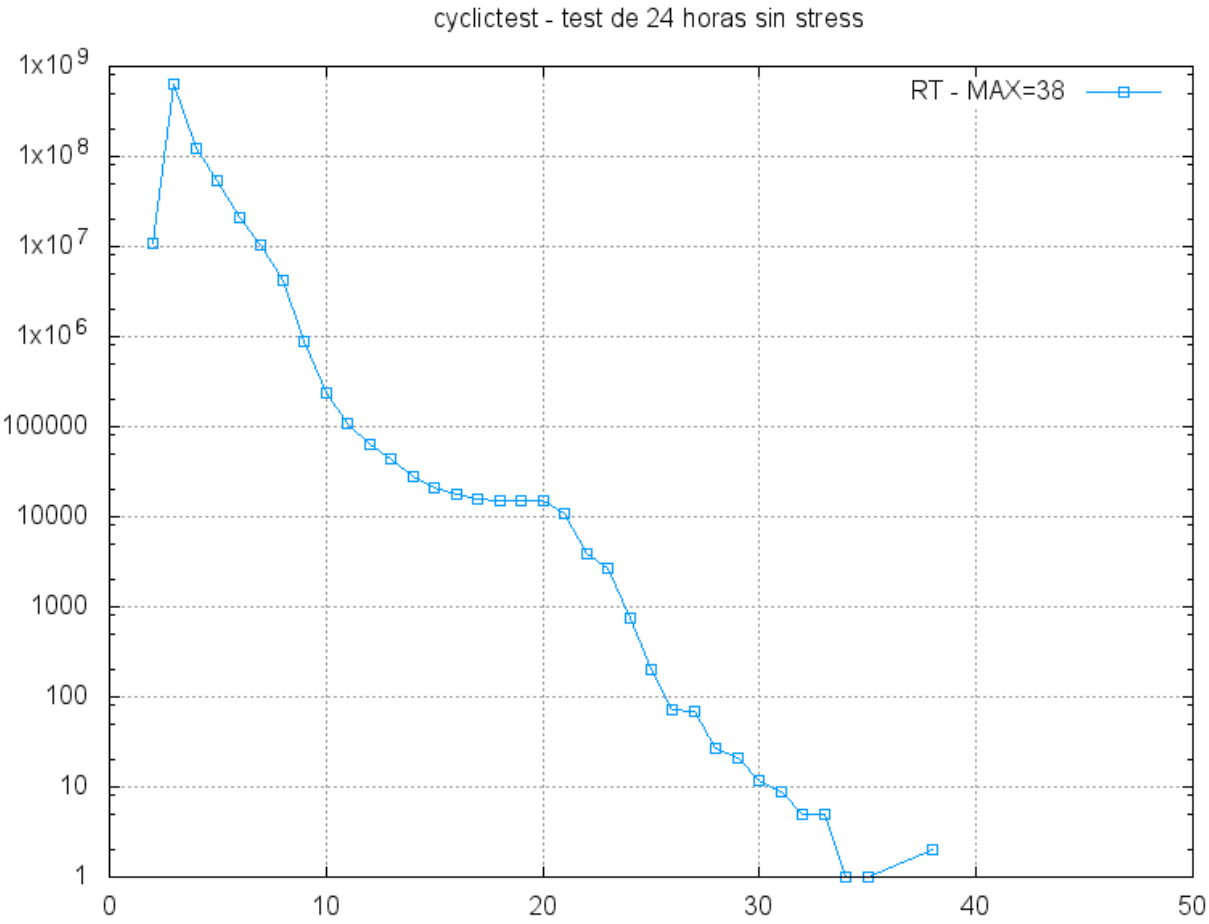
Se muestra a continuación la salida obtenida por el comando:

```
# Total: 864000000
# Min Latencies: 00002
# Avg Latencies: 00003
# Max Latencies: 00038
# Histogram Overflows: 00000
```

El test tuvo 864000000 iteraciones en total; la latencia mínima fue de 2 microsegundos y el promedio de 3 microsegundos. La latencia máxima en toda la ejecución fue de 38 microsegundos.

El indicador "histogram overflows" contabiliza la cantidad de latencias superiores a 100 microsegundos (según el parámetro -h utilizado arriba). En el ejemplo no hubo ninguna latencia que haya superado los 100 microsegundos.

Se muestra un gráfico con la cantidad de ocurrencias por microsegundo:



El gráfico muestra un pico que representa que la mayor cantidad de latencias duraron 3 microsegundos. Luego la curva muestra menos ocurrencias a mayor latencia, hasta llegar al máximo de 38 microsegundos.

Con stress

Este test se ejecutó igual al anterior, pero al mismo tiempo que el programa stress, por lo cual cyclictest compite con el procesador con este programa. Como stress se ejecuta en una prioridad mucho menor (prioridad de aplicaciones de escritorio), cyclictest apropiará a stress cuando necesite ejecutar.

El resultado es el siguiente:

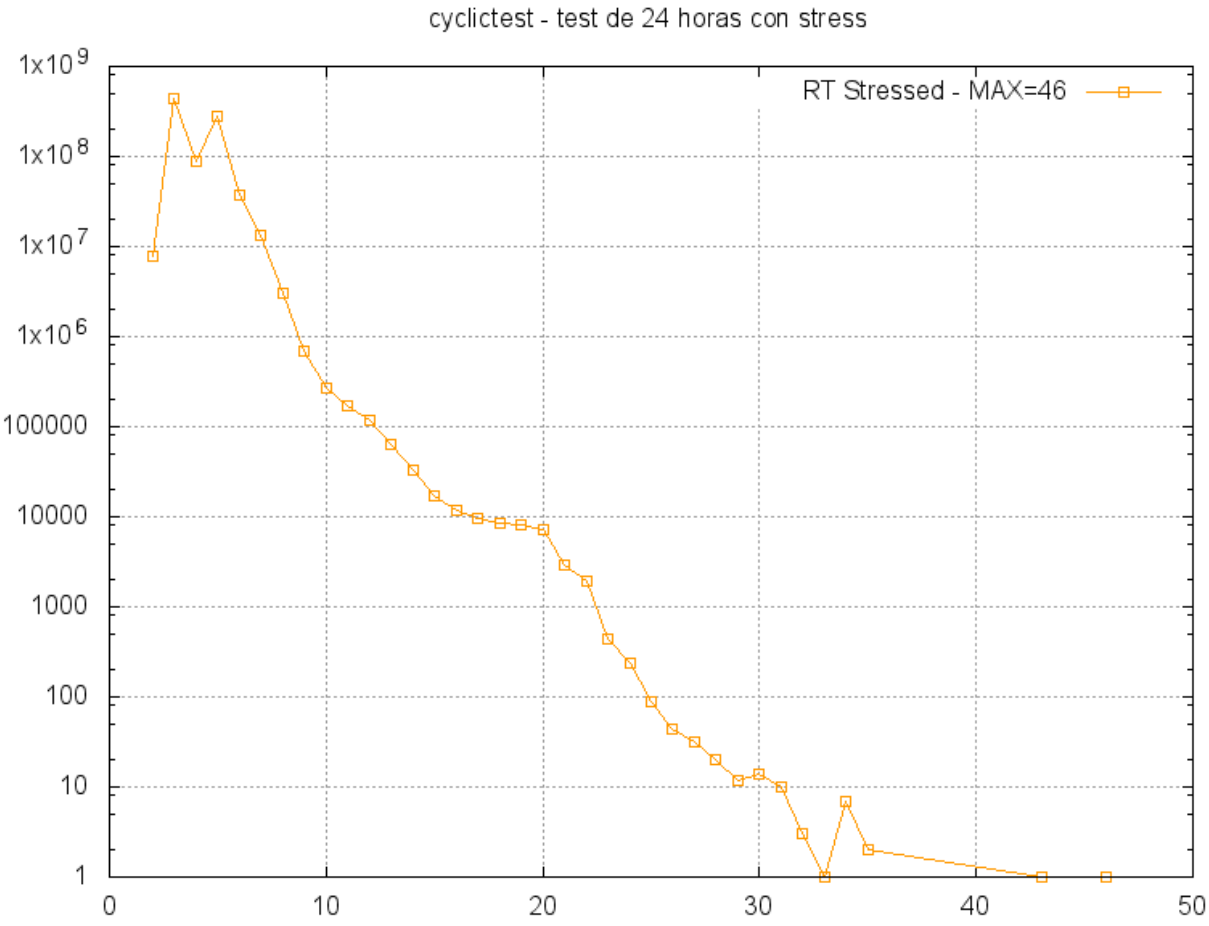
```
# Total: 864000000
# Min Latencies: 00002
# Avg Latencies: 00003
```

```
# Max Latencies: 00046

# Histogram Overflows: 00000
```

Se observa un leve incremento en la máxima latencia con respecto al test anterior: 46 microsegundos contra 38. El promedio y la mínima latencia son las mismas. Tampoco hubo una latencia mayor a 100 microsegundos.

A continuación un gráfico con características similares al anterior:



La diferencia con el gráfico anterior es que se ven dos picos, con latencias de 3 y 5 microsegundos. Luego es similar con una curva suave. El máximo fue de 46 microsegundos.

8.11 rtps: pruebas de 24 hs sobre SO de tiempo real

Los siguientes experimentos consisten en encontrar un período mínimo en el cual la ejecución de rtps pueda ejecutar durante 24 hs sin incumplir ningún plazo en su thread periódico para un período mínimo. La idea es empezar utilizando un período pequeño e ir incrementándolo en 10 microsegundos, hasta pasar la prueba de 24 hs.

La aplicación rtps se construyó con opciones óptimas utilizando el recolector de basura estático, para tener la mayor predictibilidad posible.

También se construyó una versión que utiliza el recolector de basura dinámico y se mostrará que las pruebas fallan cuando se utiliza este tipo de recolector de basura.

Sin stress

Se comenzó ejecutando el test con un período de 120 microsegundos, y las pruebas fueron fallando y con incrementos de 10 microsegundos, como se dijo antes, se encontró que rtps ejecutó exitosamente sin un incumplimiento de plazos utilizando un período de 160 microsegundos. Con períodos más grandes tampoco fallaron.

Un resumen de cada prueba indica cuantas activaciones logró realizar, y el máximo de latencia de cada uno (MISS indica incumplimiento de plazos y OK éxito):

```
120 us: MISS -- Activaciones: 200693    Latencia máxima: 446078 ns
130 us: MISS -- Activaciones: 39534810  Latencia máxima: 603831 ns
140 us: MISS -- Activaciones: 2593316   Latencia máxima: 411195 ns
150 us: MISS -- Activaciones: 1321917   Latencia máxima: 512255 ns
160 us: OK   -- Activaciones: 539993711 Latencia máxima: 140665 ns
170 us: OK   -- Activaciones: 508229377 Latencia máxima: 146257 ns
200 us: OK   -- Activaciones: 431994971 Latencia máxima: 134029 ns
500 us: OK   -- Activaciones: 172797982 Latencia máxima: 149603 ns
  1 ms: OK   -- Activaciones: 86398995  Latencia máxima: 165317 ns
```

Se observa que la latencia máxima para cada prueba excede considerablemente el período utilizado para cada una. Por ejemplo en la prueba de 130 microsegundos, la latencia fue de 603 microsegundos por lo cual incumplió el plazo por 473 microsegundos.

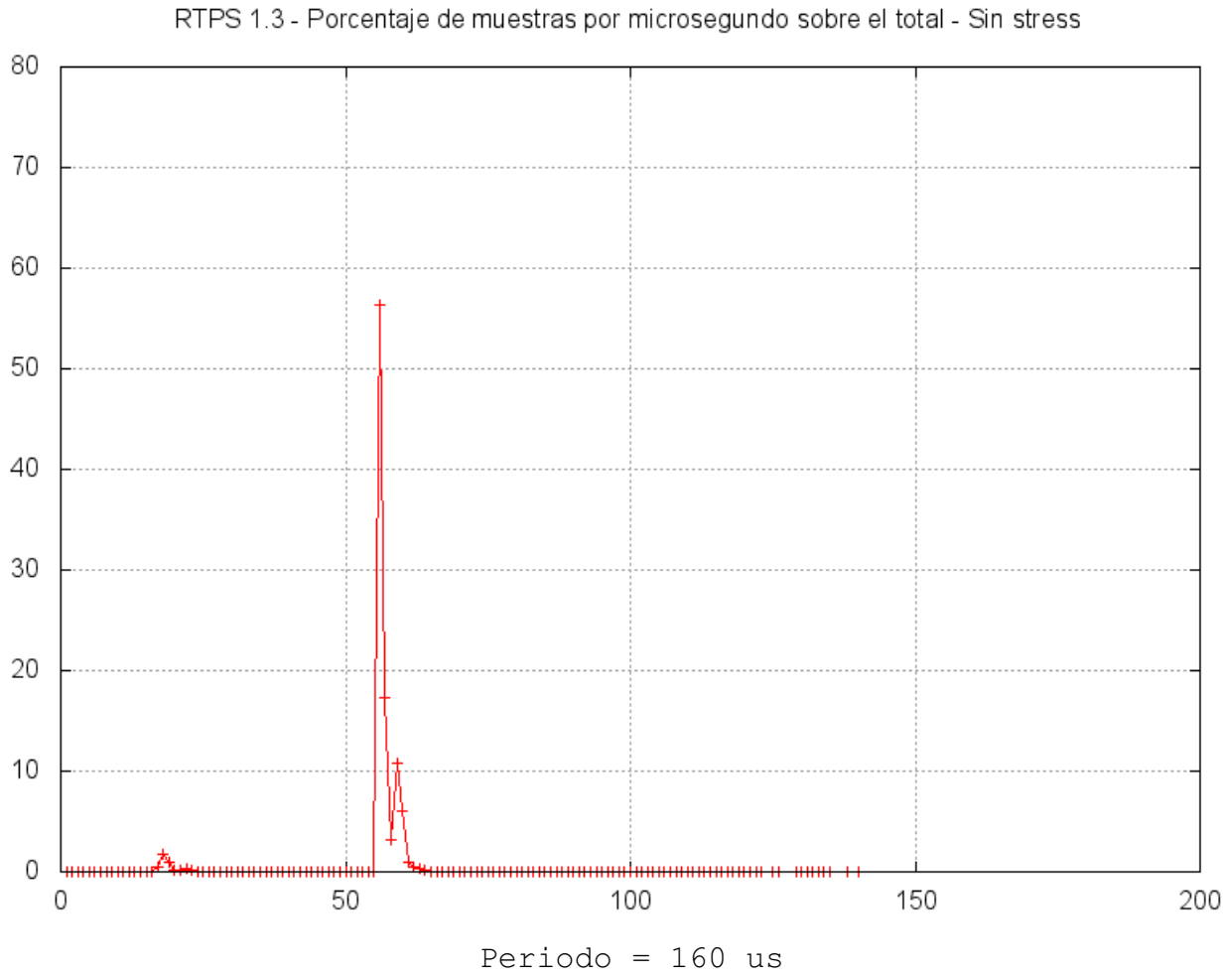
Las pruebas exitosas muestran un máximo similar que ronda los 140 microsegundos (salvo la prueba con periodo de 1 milisegundo que fue levemente mayor). En cambio las pruebas que fallaron

tuvieron un pico mucho mayor al período, cercano al medio milisegundo y con muy pocas activaciones en relación a las exitosas.

Notar que la prueba exitosa con periodo igual a 170 microsegundos tuvo una cantidad de activaciones menor con respecto a la respectiva prueba de 160 microsegundos; esto es así porque en 24 horas ocurren menos activaciones para un período más grande.

Ahora observaremos con más detalle la prueba exitosa: utilizando un período de 160 microsegundos, la latencia promedio fue de 55 microsegundos y la máxima fue de 140 microsegundos. La cantidad total de activaciones fue de 539993711, según se puede observar en el resultado de arriba.

En el gráfico a continuación se muestran 3 picos, el cual el más alto indica que casi el 56.42% de las activaciones del thread periódico tuvieron una latencia de 56 microsegundos. El porcentaje es equivalente a 304656424 activaciones.



Se detallan algunos datos obtenidos por la corrida, puesto que no llegan a ser visibles en el gráfico. La primer columna indica la latencia en microsegundos y la segunda la cantidad de

activaciones que tuvieron la latencia indicada en la primer columna.

Por ejemplo, hubo solo 2 activaciones que tuvieron 1 microsegundo de latencia; ésta fue la latencia mínima registrada por la corrida; hubo 4143 activaciones que tuvieron 3 microsegundos de latencia. A continuación los datos:

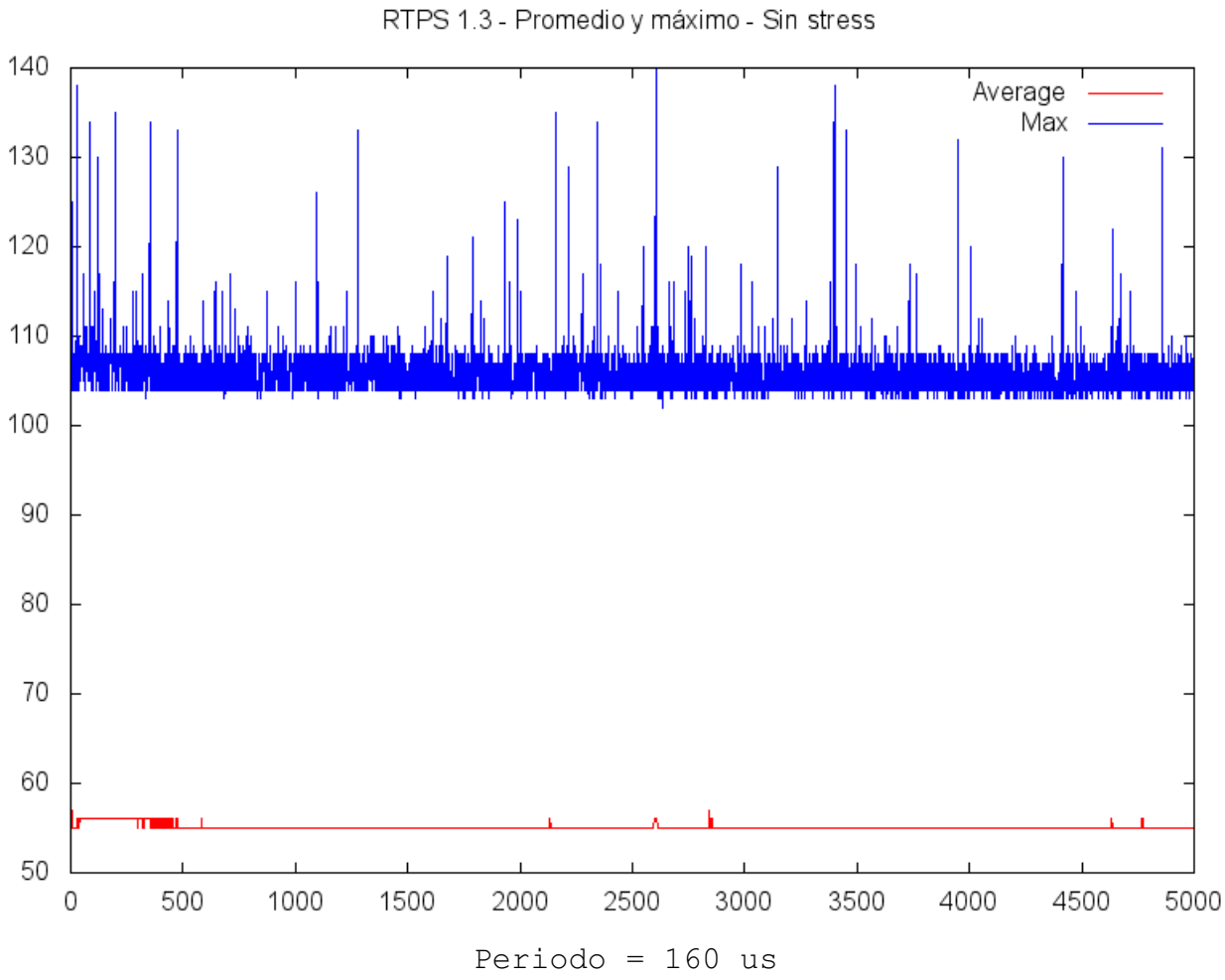
```
00001 2
00002 1506
00003 4143
00004 3445
00005 2162
```

Viendo los siguientes datos se puede observar un pico de latencias de 18 microsegundos que corresponde al pico más pequeño encontrado en el gráfico anterior. No visible en éste se puede encontrar un pico más pequeño aún, correspondiente a latencias de 22 microsegundos. Los picos se resaltan en negrita:

```
00016 30557
00017 2240972
00018 9713846
00019 5018892
00020 736376
00021 1142111
00022 1670462
00023 950889
00024 245344
00025 81832
00026 58155
00027 50144
```

El segundo gráfico más abajo (gráfico temporal) muestra la latencia promedio y máxima entre intervalos de tiempo. Cada intervalo tiene una duración de 17.28 segundos (equivalen a 5000 intervalos dentro de 24 hs). Se observan diversos picos, en el cual el máximo ocurrió en el intervalo

número 2603 con promedio de 56 microsegundos y un máximo de 140 microsegundos, como se mencionó más arriba. Se observa que todos los máximos son mayores a un número cercano a 100, lo cual es casi el doble que el promedio en general.



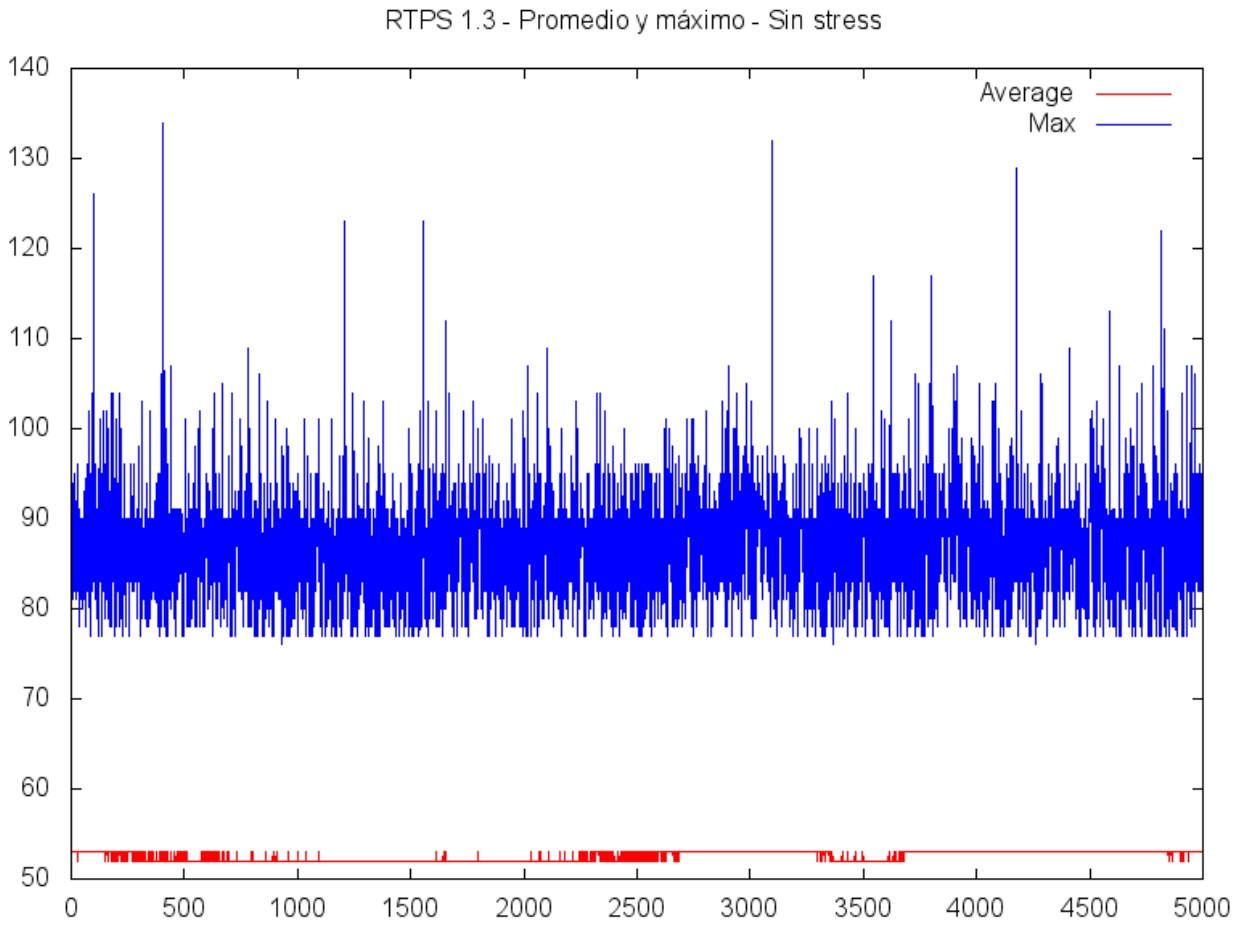
Llama la atención, si se compara con el primer gráfico, que los máximos del segundo gráfico excedan los 100 microsegundos cuando el promedio es de 55 segundos. A continuación mostramos un detalle de las activaciones ocurridas cerca de los 100 microsegundos:

```
00098 81
00099 99
00100 129
00101 177
00102 1268
```

```
00103 21253
00104 13071
00105 1030
00106 968
00107 1394
00108 779
00109 116
```

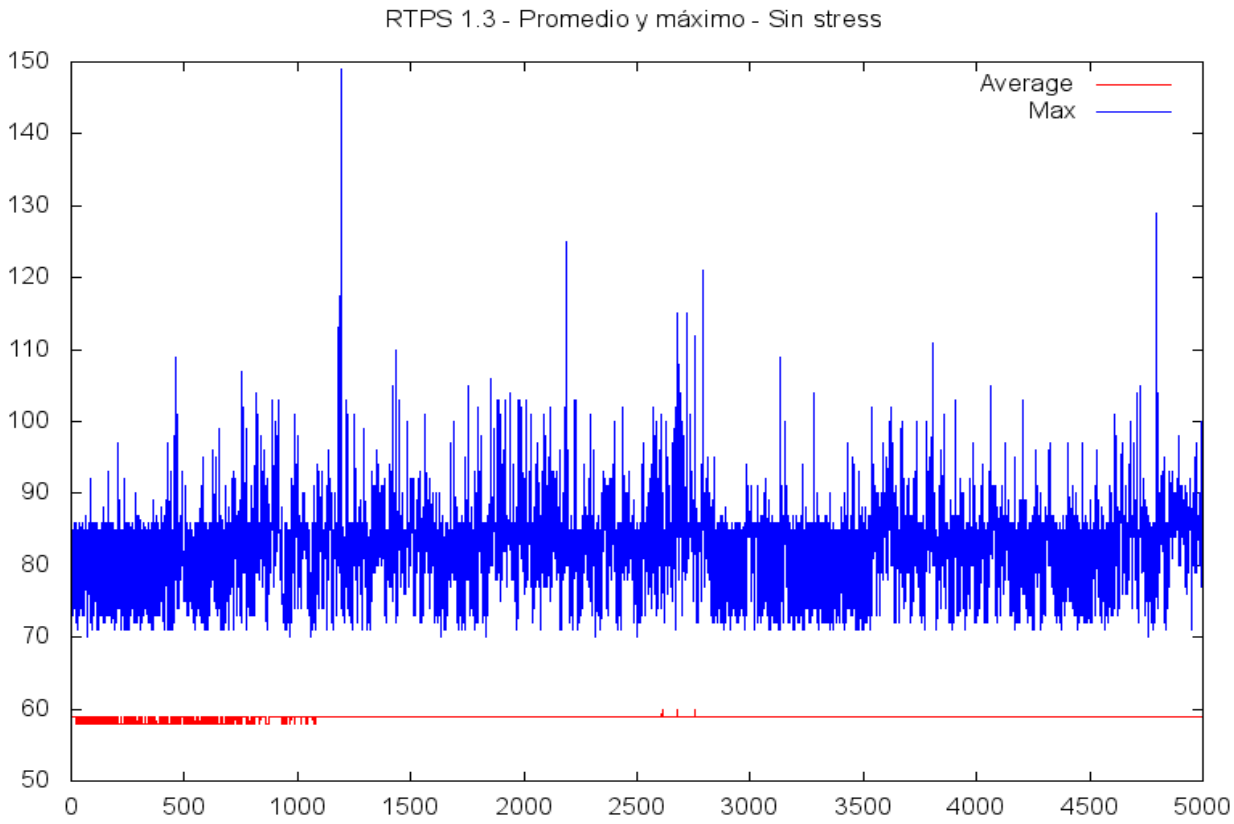
Rondando los 103 microsegundos de latencia, se pueden contabilizar muy pocas activaciones comparado al total; sin embargo al observar el segundo gráfico se puede concluir que estas latencias de 103 microsegundos ocurrieron regularmente, por lo menos una vez cada 17.28 segundos durante las 24 horas de ejecución.

Ahora se mostrará un gráfico temporal del test con período de 200 microsegundos para compararlo con el anterior:

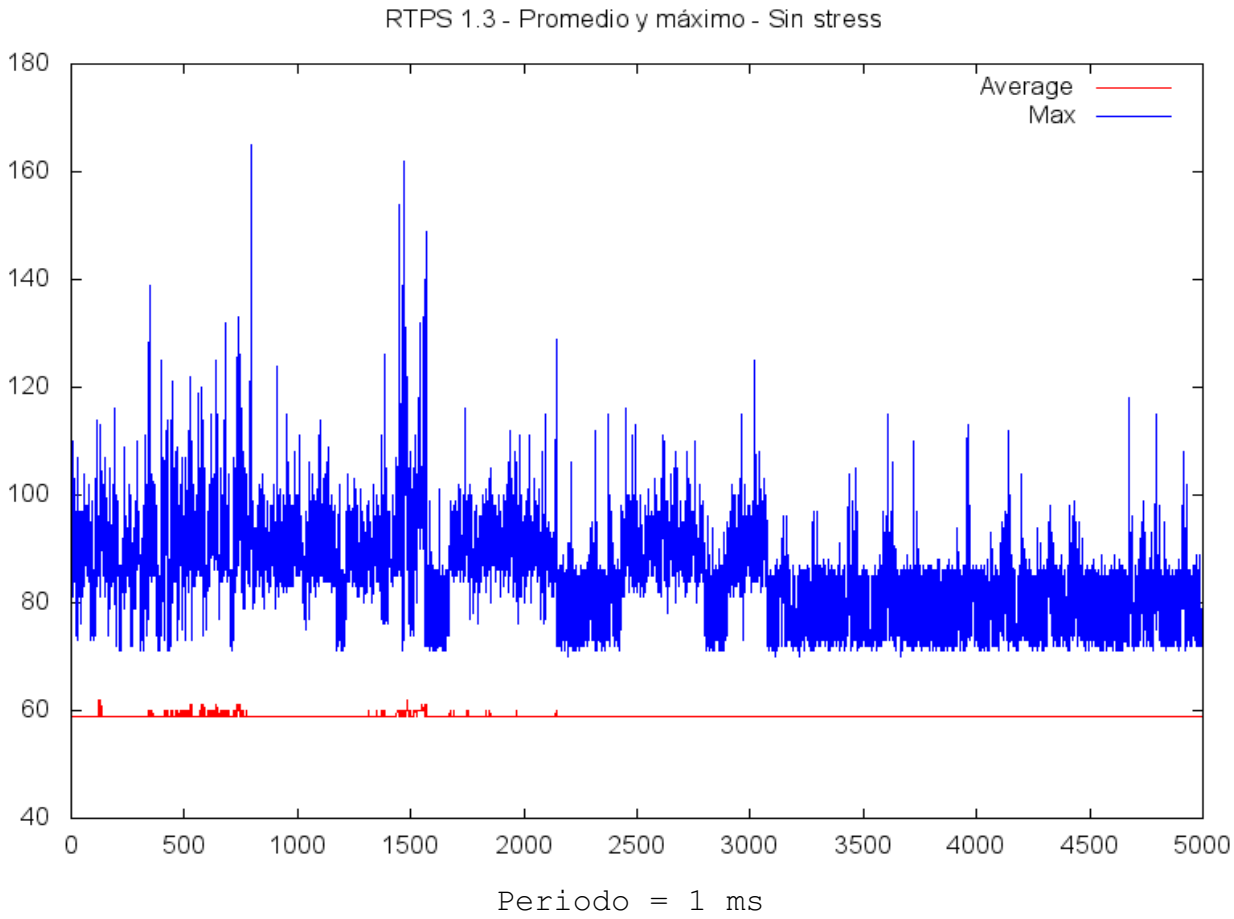


Periodo = 200 us

En éste gráfico, el promedio máximo no excede los 80 microsegundos, el cual es mucho menor al que excede el gráfico anterior, el cual era aproximadamente 105 microsegundos. Veamos finalmente el gráfico correspondiente a la prueba con período igual a 500 y 1000 microsegundos:



Periodo = 500 us



En estos dos últimos gráficos el máximo se mantiene en promedio no menor a 70 microsegundos. El último gráfico (período de 1 milisegundo) muestra mayor variabilidad que todos los anteriores. Todos los gráficos muestran picos altos de latencias que no exceden los 200 microsegundos, por lo cual éste tiempo podría llegar a ser un período mínimo a seleccionar.

Sin stress con recolector de basura dinámico

Se mostrarán las pruebas hechas con rtps configurado para utilizar el recolector de basura dinámico, también optimizado. Se hicieron pruebas con un período inicial de 120 microsegundos (como en la anterior prueba), y se fue incrementando su período hasta llegar a 500. No hubo una sola prueba que haya durado 24 horas sin incumplir un plazo.

A continuación el detalle de las pruebas:

```
120 us: MISS -- Activaciones: 3398     Latencia máxima: 582240 ns
150 us: MISS -- Activaciones: 1824757  Latencia máxima: 557077 ns
```

```
200 us: MISS -- Activaciones: 1367292  Latencia máxima: 575649 ns
300 us: MISS -- Activaciones: 53175558 Latencia máxima: 682095 ns
400 us: MISS -- Activaciones: 4371971  Latencia máxima: 968012 ns
500 us: MISS -- Activaciones: 16667048 Latencia máxima: 1 ms y 149672 ns
```

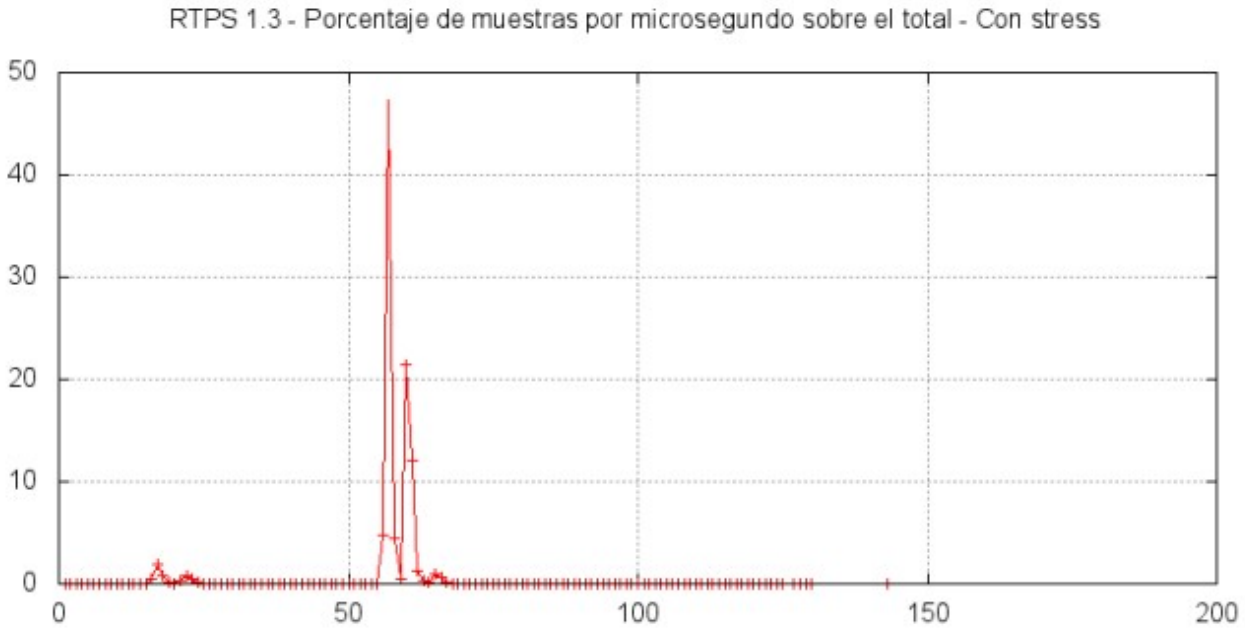
Dadas estas pruebas, se concluye que el recolector de basura dinámico no es adecuado para utilizar con JamaicaVM. Las pruebas restantes se harán con el recolector de basura estático, como se hizo en la prueba anterior.

Con stress

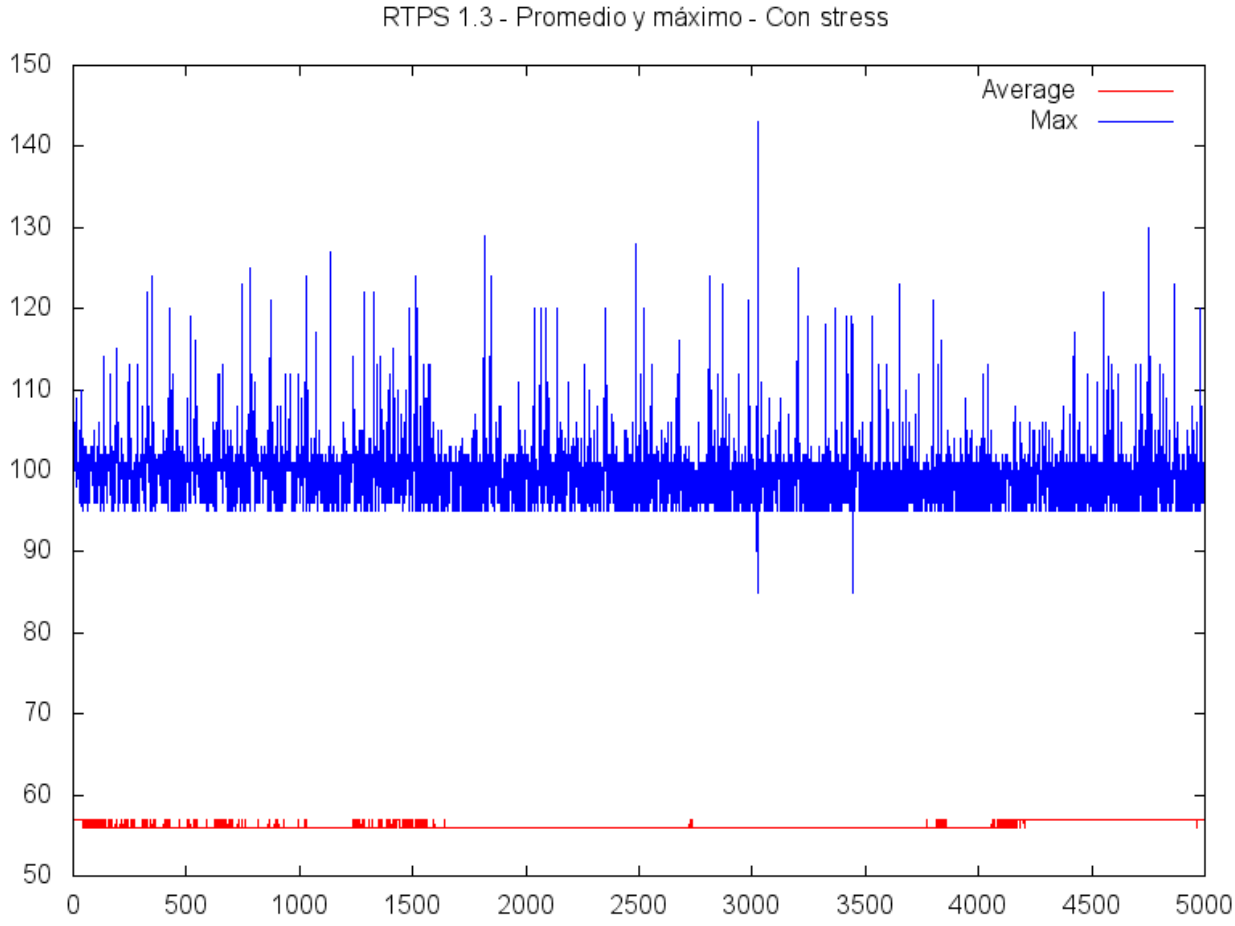
De forma similar a las pruebas anteriores, se ejecutan varios tests comenzando por un período de 120 us y luego se va incrementando hasta encontrar un período que no falle durante 24 hs, pero esta vez con stress.

```
120 us: MISS -- Activaciones: 1720204   Latencia máxima: 409518 ns
130 us: MISS -- Activaciones: 19356322  Latencia máxima: 615942 ns
140 us: MISS -- Activaciones: 39887996  Latencia máxima: 551718 ns
150 us: OK   -- Activaciones: 575993287 Latencia máxima: 143574 ns
200 us: OK   -- Activaciones: 431994965 Latencia máxima: 133758 ns
300 us: OK   -- Activaciones: 287996642 Latencia máxima: 125218 ns
500 us: OK   -- Activaciones: 172797989 Latencia máxima: 142469 ns
  1 ms: OK   -- Activaciones: 86398995  Latencia máxima: 159448 ns
```

Los resultados son muy similares en general a las pruebas sin stress. En particular se observa que tuvo éxito un test con período menor al equivalente a la prueba anterior, de 150 microsegundos contra 160 microsegundos. A continuación los gráficos del test con período igual a 150 microsegundos:



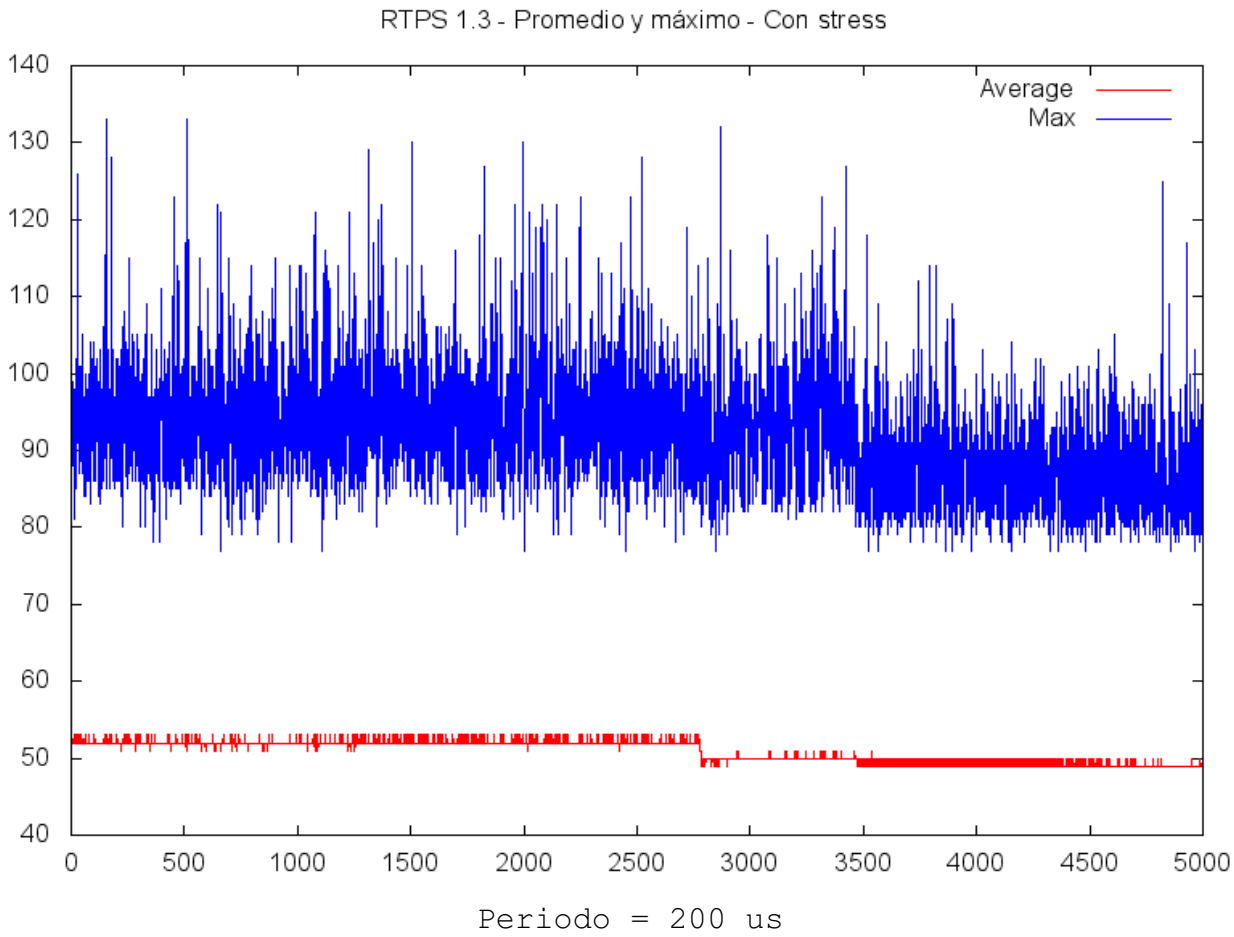
Periodo = 150 us

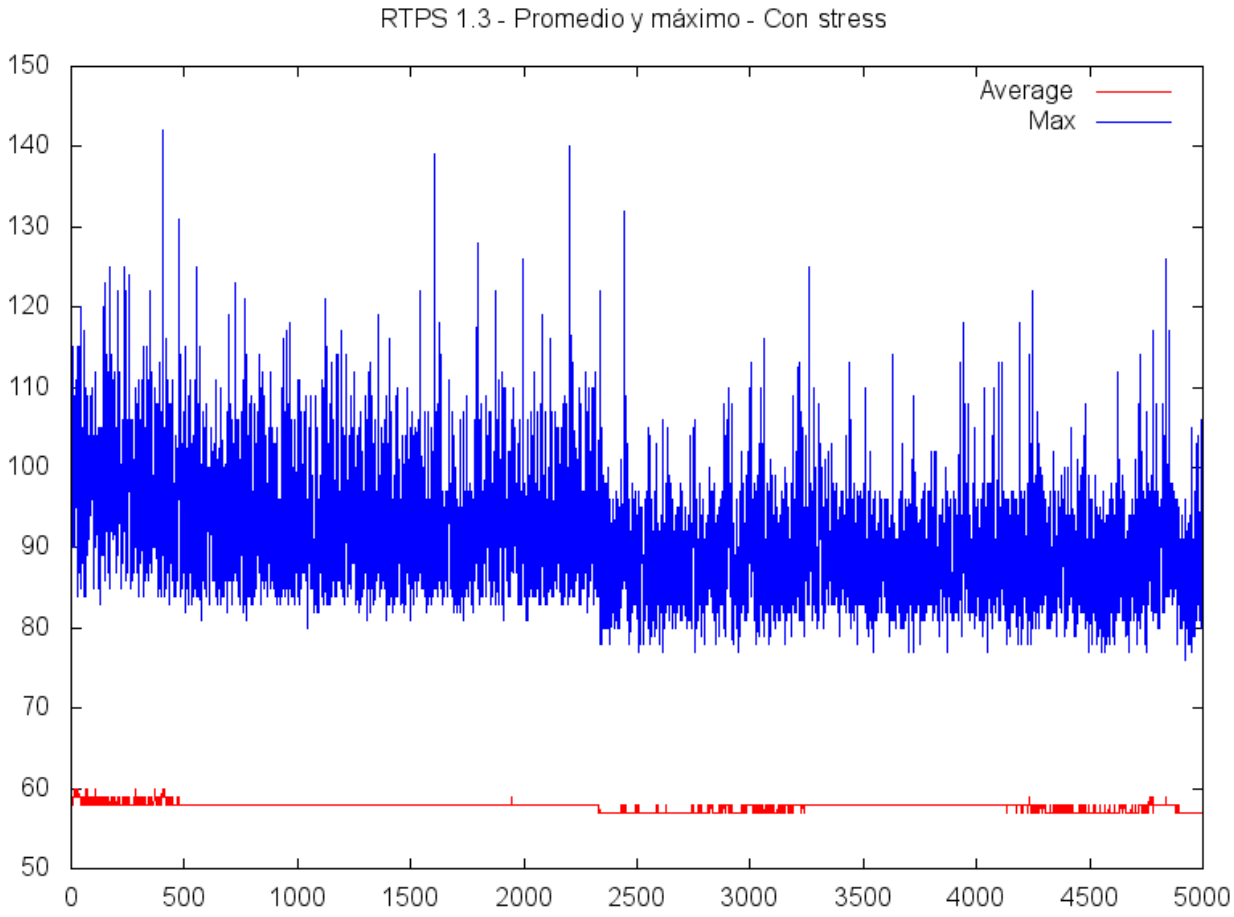


Periodo = 150 us

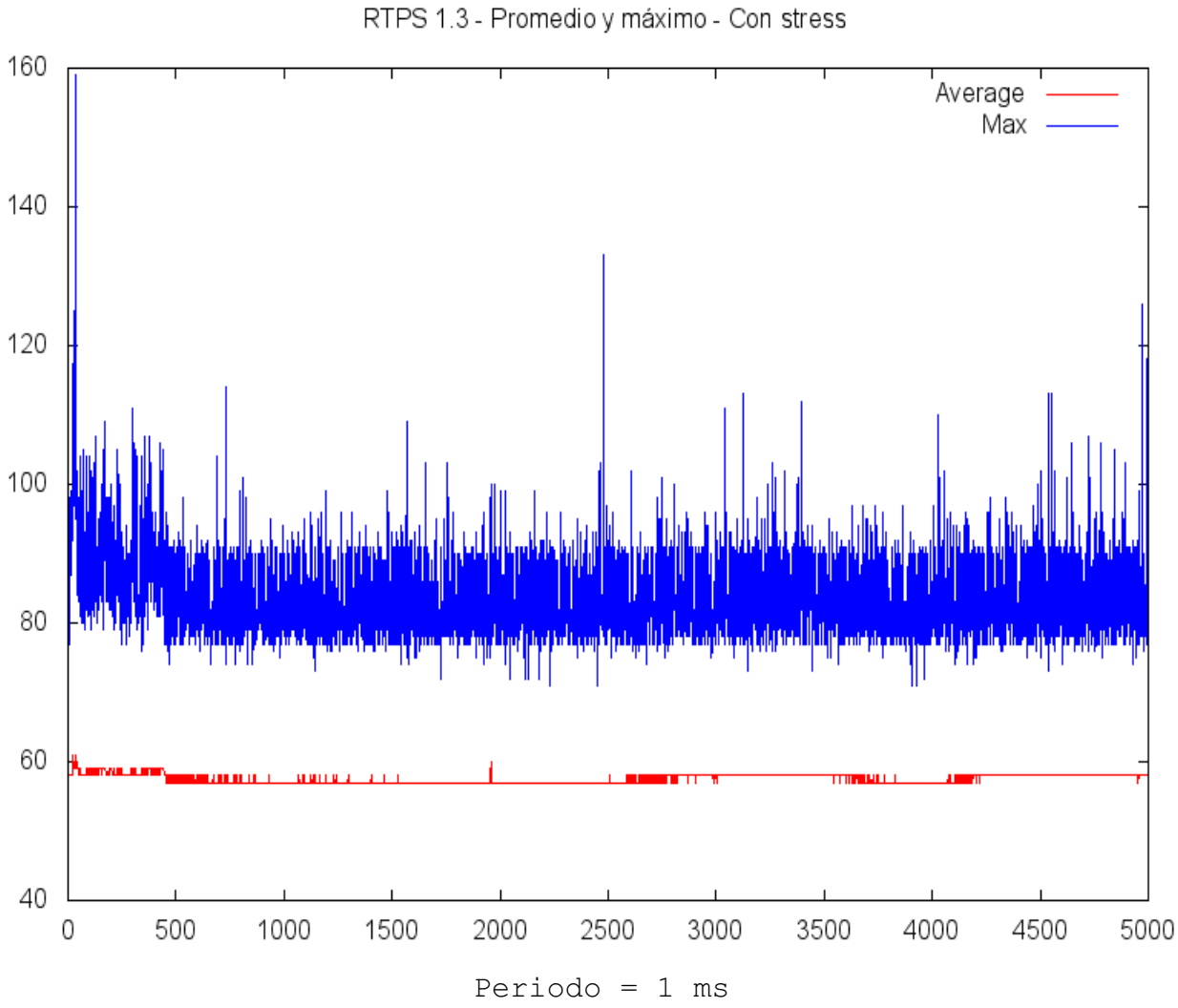
Se observan gráficos muy similares a la prueba anterior sin stress. La diferencia más importante se encuentra en el segundo gráfico, es que en general los máximos (salvo dos picos que apuntan hacia abajo) no son inferiores a 95 microsegundos, y en el gráfico anterior este valor estaba cerca de 105 microsegundos. También tener en cuenta que la presente prueba tuvo éxito con un período menor.

Finalmente se muestran los gráficos temporales de las pruebas exitosas (200, 500 y 1000 us). Los gráficos son muy similares a los gráficos de las pruebas anteriores, y también cumplen el hecho de que los máximos no exceden los 200 microsegundos.





Periodo = 500 us


RTPS 1.3 - Promedio y máximo - Con stress
Average
Max
160
140
120
100
80
60
40
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Periodo = 1 ms

8.12 Resumen de los resultados obtenidos

La siguiente tabla resume los tiempos obtenidos en las pruebas de cyclictest y rtps de 24 horas. Para rtps, se seleccionó el período de 200 microsegundos como un período adecuado para poder utilizar JamaicaVM sin incumplimiento de plazos para desarrollar aplicaciones de tiempo real duro.

Se puede observar que la latencia obtenida utilizando JamaicaVM es casi cuatro veces mayor que la latencia que se puede obtener con un programa en C.

Latencias	cyclictest	rtps
Sin stress	**38 us** (intervalo = 100 us)	143 us (período = 150 us) **133 us** (período = 200 us)
Con stress	**46 us** (intervalo = 100 us)	140 us (período = 160 us) **134 us** (período = 200 us)

9 Conclusiones y trabajos futuros

Se demostró en la presente investigación que se puede utilizar Java para programar aplicaciones de tiempo real hard, con tiempos razonablemente pequeños.

En el sistema utilizado, las latencias obtenidas en Java son casi cuatro veces mayores a las obtenidas en C. Se concluye que C es más rápido y eficiente, con menores latencias que Java, pero al mismo tiempo es un lenguaje que dificulta el desarrollo por la falta de claridad en el código y la utilización de sintaxis complicadas y oscuras como lo son el manejo de punteros.

A pesar de tener latencias más grandes en Java, éste ofrece un lenguaje más limpio y moderno orientado a objetos, incorpora una gran cantidad de herramientas que facilitan el desarrollo de aplicaciones de tiempo real: planificador FPS, detección de incumplimiento de plazos, manejo de excepciones asincrónicas, distintos tipos de memorias, definición de planificables periódicos, aperiódicos y esporádicos. En particular JamaicaVM ofrece un recolector de basura de tiempo real que soluciona algunos problemas no resueltos en la especificación RTSJ actual.

Se propone como trabajos futuros medir las latencias de JamaicaVM utilizando memorias acotadas para el manejo de la memoria de los threads de tiempo real y compararlos con los resultados obtenidos aquí. También sería interesante probar una ejecución sobre un sistema embebido con interfaces con el mundo exterior, como lo puede ser un robot, y posteriormente probar comunicación entre dispositivos embebidos.

Bibliografía

[1] Aicas. Aicas JamaicaVM. https://www.aicas.com/cms/en/JamaicaVM

[2] The Real Time for Java Expert Group, Real Time Specification for Java Version 1.0.2, 2006

[3] Alam Burns, Andy Wellings, Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX, 4. Addison-Wesley, 2009.

[4] Giorgio C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 3. Springer, 2011.

[5] Andy Wellings, Concurrent and Real-Time Programming in Java, 1. John Wiley & Sons, 2004.

[6] Hermann Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, 2. Springer, 2011.

[7] Qing Li, Caroline Yao, Real-Time Concepts for Embedded Systems, 1. CRC Press, 2003.

[8] Michael González Harbour. Programming real-time systems with C/C++ and POSIX.

[9] Bruce Eckel, Thinking in Java, 4. Pearson, 2006.

[10] Herbert Schildt, The Java Complete Reference, 9. Oracle Press, 2014.

[11] Fridtjof Siebert, Hard Real-Time Garbage Collection in Modern Object Oriented Programming Languages, 1. Aicas, 2002.

[12] Scott Oaks, Java Performance: The Definitive Guide, 1. O'Reilly, 2014.

[13] Aicas, JamaicaVM 6.3 — User Manual: Java Technology for Critical EmbeddedSystems, 2014

[14] Luotao Fu, Robert Schwebel. rt-preempt.
https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO