UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA

INSTITUTO DE INGENIERÍA
MAESTRÍA Y DOCTORADO EN CIENCIAS E INGENIERÍA

PLANIFICACIÓN DE TAREAS EN SISTEMAS DE TIEMPO-REAL CRÍTICOS

T E S I S

que para obtener el grado de
DOCTOR EN CIENCIAS
presenta

Vidblain Amaro Ortega

DIRECTOR
Dra. Larysa Burtseva

CO-DIRECTOR
Dr. Arnoldo Díaz Ramírez

Mexicali, B.C. ENERO, 2018

RESUMEN de la Tesis de Vidblain Amaro Ortega, presentada como requisito parcial para la obtención de grado de Doctor en Ciencias, Mexicali, Baja California, 22 de Enero de 2018.

PLANIFICACIÓN DE TAREAS EN SISTEMAS DE TIEMPO-REAL CRÍTICOS

Resumen aprobado por:

Dra. Larysa Burtseva	Dr. Arnoldo Díaz Ramírez
Directora de Tesis	Codirector de Tesis

La planificación de tareas es un tema que se ha estudiado ampliamente en el área de la computación. Sin embargo, la evolución de las tecnologías y la rapidez en que estas han emergido ha demostrado que la teoría de planificación con la que se contaba no era del todo aplicable. Así, el surgimiento de arquitecturas multinúcleo, el incremento de velocidad en las operaciones en memoria, características nuevas implementadas a nivel hardware, entre otras, han provocado que la comunidad científica retome su estudio.

En la actualidad, sistemas que se encuentran en vehículos autónomos, control de vuelos, así como dispositivos que son utilizados cotidianamente han tomado importancia. Muchos de éstos requieren de restricciones de tiempo, en el cual debe ser procesada información, ya que dependen de resultados en instantes de tiempo específicos para tomar decisiones. Dichas aplicaciones son conocidas como sistemas de tiempo-real (STR) críticos, las cuales operan en mayoría bajo hardware no tan reciente, debido a la confiabilidad que ofrecen éstos. Además, es conocido que la mayoría de los STRs se ejecutan bajo entornos basados en UNIX por su confiabilidad y soporte que se ha dado tanto por parte del área industrial como académica. Uno de los casos más comunes es el soporte en el Kernel de Linux, el cual integra diversas políticas de tiempo-real para dar soporte a los STRs. Sin embargo, existen algoritmos de planificación más eficientes que aún no son implementados. Trabajos publicados han presentado la implementación de distintos algoritmos bajo Linux, pero debido a que cada implementación requiere el uso de APIs propias, resulta en ocasiones difícil su aceptación para los grupos de desarrollo.

En esta tesis se presenta la implementación del algoritmo *Earliest Deadline First* (EDF) de tal forma que sea compatible con el estándar de POSIX, dando solución a este problema que se encuentra presente en la mayoría de las implementaciones. Así mismo, se realiza la evaluación de la eficiencia que muestran algoritmos de planificación bajo sistemas multiprocesador ofreciendo a la comunidad el conocimiento de cuáles son los más adecuados a utilizar.

Palabras clave: Sistema de tiempo-real, planificación, Linux, rate-monotonic, earliest deadline first policy, POSIX.

ABSTRACT of the thesis, presented by Vidblain Amaro Ortega, in order to obtain the DOCTOR IN SCIENCE DEGREE in COMPUTER SCIENCE. Mexicali, Baja California, México, January 22, 2018

SCHEDULING TASKS IN HARD REAL-TIME SYSTEMS

Abstract approved by:

Dra. Larysa Burtseva	Dr. Arnoldo Díaz Ramírez
Directora de Tesis	Codirector de Tesis

Task scheduling is a subject in the area of computing which has been studied extensively. However, the evolution of technologies and the rapidity in which they have emerged have shown that the available theory of scheduling is not entirely applicable. Thus, the new multi-core architectures, the increasing speed of the memory operations, new features implemented at hardware level, among others, have motivated the scientific community to resume its study.

Currently, systems are in autonomous vehicles, flight control, as well as devices that are used daily have become important. Many of these systems require time constraints in which information must be processed, because they depend on results at a specific time instant to make decisions. Such applications are known as hard real-time systems (RTS), where many of these systems operate on hardware not as recently due to the reliability offered by them. In addition, it is known that most of the RTSs are running under a UNIX-based architecture because of their reliability and support which has been given by both the industrial sector and academic environments. One of the most common cases is the support of the Linux kernel, which integrates several real-time policies to support RTSs. However, there are more efficient planning algorithms that are not yet implemented, which makes their use difficult. The published papers have presented the implementation of different algorithms under the Linux Kernel, but due to the requirement of specific APIs on each implementation, it is sometimes difficult to accept them for development groups.

This thesis presents the implementation of the Earliest Deadline First (EDF) algorithm, being compatible with the POSIX standard, providing a solution to this problem which is present in most of the implementations. Furthermore, the efficiency assessment which shows scheduling algorithms on multiprocessor systems and offers the community the knowledge suitable to be used under certain parameters.

Key words: Real-time system, Scheduling, Linux, rate-monotonic, earliest deadline first policy, POSIX.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1 Planteamiento del problema

El correcto funcionamiento de muchos sistemas computacionales y dispositivos electrónicos, depende no sólo de los efectos o los resultados que éstos produzcan, sino también del tiempo en el que son obtenidos. Las restricciones expresadas como el tiempo, representan una parte esencial de los llamados sistemas de tiempo-real (STR), los cuales son requeridos para completar tareas cuando es necesario y prestar servicios de manera correcta en plazos establecidos. Los STRs reaccionan dinámicamente de acuerdo a los cambios de su entorno, ya sea del comportamiento humano, o bien un fenómeno natural o artificial.

En la actualidad el avance de la tecnología ha permitido una miniaturización en los dispositivos electrónicos y un incremento en la capacidad de procesamiento. Estos dispositivos se encuentran en general empotrados (embebidos) dentro de aplicaciones de STRs que abarcan una amplia gama de áreas. Algunos ejemplos comunes son un sistema de activación de bolsas de aire de un automóvil, el monitoreo de los signos vitales en unidades de cuidados intensivos, un marcapasos, el procesamiento de audio y vídeo, el monitoreo de una planta nuclear, sistemas de control de vuelos, etc. (Srovnal y Kotzian, 2008; Trejo y Angulo, 2016; Younis et al., 2004). Tomando el ejemplo del marcapasos, los dispositivos de éste tipo no solamente producen impulsos eléctricos para mantener al corazón activo, sino también monitorean la actividad cardíaca de la persona, para así generar impulsos a distintas frecuencias. La importancia de un correcto funcionamiento de tal sistema, surge del hecho que si alguna de estas funciones no llegue a ofrecer sus resultados de manera correcta y en el periodo correspondiente, la persona puede sufrir un efecto drástico en su estado de salud. Tanto el tiempo como el resultado en el que se producen las señales son importantes para darle a la persona la oportunidad de realizar las actividades diarias sin ocasionar problemas en su salud.

En el contexto de aplicaciones de tiempo-real, las acciones son llamadas tareas, o procesos (Cottet et al., 2002). La organización de su ejecución en los procesadores disponibles es llamada planificación de tareas de tiempo-real. Un planificador debe satisfacer las restricciones de tiempo para la aplicación. Al conjunto de reglas que rigen

el orden de ejecución de las tareas se le denomina política de planificación (Burns, 1991).

Para garantizar un suficiente rendimiento en un STR, se requiere de la implementación de políticas de planificación eficientes, así como algoritmos precisos que permitan el uso de los recursos. Mucho se ha hecho en el estudio y desarrollo de nuevas técnicas para lograr un mejor rendimiento de los sistemas, pero éstas políticas por su propia naturaleza no ofrecen la garantía de que cualquier conjunto de tareas se ejecutará con éxito. Es por ello que surgen los estudios en entorno a las pruebas de planificabilidad (Baker, 2006; Bini y Buttazzo, 2004; Carpenter et al., 2004), los cuales representan condiciones matemáticas que permiten determinar si un conjunto de tareas al ser ejecutado por un algoritmo en un sistema es planificable o no.

Dichas pruebas se dividen en dos categorías: exactas e inexactas. A la primera de ellas se les consideran las condiciones suficientes y necesarias, ya que finalmente al aplicar tal condición matemática no hay nada por discutir en el resultado al quedar definido si un conjunto de tareas es o no lo es planificable. A la segunda categoría pertenecen las pruebas suficientes, ya que una vez aplicada una condición suficiente, si el resultado no satisface la condición, eso no implica que dicho conjunto de tareas no es planificable, sino que simplemente con la técnica de planificabilidad aplicada, no lo es, pero puede existir otro test en el que sí sea planificable.

El estudio de la teoría de la planificación para STRs inició entre los años 60’s y 70’s del siglo XX, basándose en aquella época en sistemas uniprocesador, que actualmente es visto como un tema razonablemente maduro. Sin embargo, se sabe que el avance de la tecnología ha traído el auge de los sistemas multiprocesador, para los cuales el problema de planificación de tareas es mucho más difícil, tal como lo mencionan Liu y Layland (1973).

Aunque el estudio de la planificación en STRs tiene bases sólidas bajo sistemas uniprocesador, la tecnología ha avanzado de tal manera que hoy en día es común el uso de sistemas multiprocesador o bien multinúcleo, para los cuales, el abordar la teoría de la planificación es mucho más complicado, ya que no es posible que la teoría con que se cuenta en sistemas monoprocesador migre a un sistema multiprocesador al no ser del todo aplicable.

En las arquitecturas tipo multiprocesador, usualmente se tiene más de un proceso (tarea) en ejecución en un mismo instante de tiempo, debido a que se cuenta con una mayor capacidad de procesamiento (procesadores o núcleos). Por lo tanto es necesario

llevar a cabo la elección de un algoritmo, mediante el cual un proceso debe ser ejecutado en un instante de planificación, además, decidir a qué procesador este debe ser asignado. Así mismo, el seleccionar el esquema más eficiente para lograr planificar un conjunto de tareas de una manera correcta es un problema poco estudiado, ya que se involucran factores como el tipo de aplicación, el entorno, sobre el cual estará en funcionamiento, las capacidades del hardware, entre otros.

Hay un claro avance las políticas de planificación, pero lamentablemente gran parte de los algoritmos que han sido propuestos ofrecen un bajo rendimiento con respecto al uso general de los STRs. Así también, las pruebas de planificabilidad que se han Sin embargo, cuando se habla de STRs críticos, los cuales requieren una característica particular, es decir, que sean determinísticos, representa un problema ya sea proponiendo nuevos algoritmos o mejorando estos para que manejen los recursos de una forma más eficiente.

1.2 Justificación

Debido a los avances en el hardware de los sistemas de cómputo, actualmente existe una gran disponibilidad de sistemas multiprocesadores, como por ejemplo los procesadores multinúcleo. Sin embargo, la teoría de planificación de tareas en sistemas multiprocesador de tiempo-real no ha avanzado a la misma velocidad debido a que la complejidad de implementación es mucho mayor que en sistemas monoprocesador, tal y como lo mencionan Liu y Layland (1973): "*pocos de los resultados obtenidos para sistemas monoprocesador pueden generalizarse directamente para el caso de multiprocesadores; al incrementarse el número de procesadores, se incrementa una nueva dimensión al problema de la planificación*". El simple hecho de que una tarea se usa solamente en un procesador aun cuando varios procesadores se encuentran disponibles, al mismo tiempo eleva la dificultad para la planificación bajo múltiples procesadores.

Por esta razón existe un gran interés en el estudio de esta problemática, ya que los sistemas multiprocesador se utilizan en una gran cantidad de aplicaciones de tiempo-real. Muchos de los sistemas hoy en día requieren que actúen como sistemas multiprocesadores de tiempo-real, para garantizar que los procesos se lleven a cabo de una manera correcta y precisa, y además con un menor costo. Estos sistemas demandan un gran poder de cómputo, que no está disponible en sistemas con un procesador.

Los algoritmos de planificación conocidos de la literatura, muestran un desempeño poco eficiente, y proporcionan cotas de planificabilidad poco optimistas, en muchos casos inferiores a 50% de la capacidad de procesamiento disponible. En esta tesis se propone un estudio de algoritmos de planificación para sistemas de tiempo-real que sean capaces de ofrecer mejores resultados considerando distintas circunstancias con respecto a los parámetros que mantienen las tareas, con cotas de aceptación que permiten la planificabilidad de procesos que las pruebas existentes no lo garantizan. Los algoritmos en los que se desea trabajar como primera instancia, para su estudio son aquellos que pertenecen al esquema particionado, para así pasar con aquellos del esquema global (no-particionado) y por ultimo aquellos del esquema semi-particionado (hibrido).

1.3 Objetivo

1.3.1 Objetivo general

Seleccionar los más eficientes algoritmos de planificación bajo el Kernel de Linux para los STRs críticos que se utilizan al ejecutar tareas periódicas con plazos fijos.

1.3.2 Objetivos específicos

Los objetivos específicos de la tesis son:

- Integrar el algoritmo *Earliest Deadline First* (EDF) con el sistema operativo Linux, para su evaluación en sistemas uniprocesador y multiprocesador bajo el esquema global.
- Incorporar la política de planificación EDF en el API de GLIBC de tal manera que permita ser utilizada para el desarrollo de los STRs con una librería estándar.
- Evaluar el desempeño de los algoritmos de planificación de tareas de tiempo-real para sistemas multiprocesador, considerando diferentes modelos de sistemas.

1.4 Metodología

- Revisar la bibliografía relacionada al tema.
- Analizar y estudiar los algoritmos de planificación de tareas en STRs multiprocesador.
- Analizar y estudiar las pruebas de planificabilidad de algoritmos.

- Analizar el uso de recursos compartidos bajo sistemas multiprocesador.
- Integrar al Kernel de Linux Preempt-RT la capacidad para implementar sistemas multiprocesadores de tiempo-real.
- Implementar los más importantes algoritmos de planificación de tareas en el simulador Realtss-MP para los STRs multiprocesador.

1.5 Esquema general de la tesis

Capítulo 1. En este capítulo se presenta el planteamiento y justificación del problema, el objetivo general y los objetivos específicos, y por último se detalla la metodología empleada para la solución del problema.

Capítulo 2. Se presentan definiciones básicas de los STRs.

Capítulo 3. Se presenta la implementación del algoritmo EDF bajo el sistema Linux

Capítulo 4. Se presenta los resultados de los algoritmos simulados bajo sistemas multiprocesador

Capítulo 5. Termina la investigación con conclusiones que surgieron del trabajo y plantea diversas líneas de acción futuras.

Capítulo 2

Sistemas de tiempo-real

Como ya se describió en la sección anterior, los STRs requieren conocer ampliamente el entorno en el que estos trabajan, así como considerar los aspectos básicos para lograr la factibilidad de su funcionamiento. Un trato insuficiente de estos puntos genera la mayoría de resultados no esperados, por lo que los sistemas no llegan a satisfacer las restricciones interpuestas sobre estos mismos.

2.1 Definiciones básicas

Un STR está generalmente formado por un sistema a controlar y un sistema controlador. El sistema a controlar consiste, por ejemplo, en una línea de producción automatizada con el uso de robots, mientras que el sistema controlador consiste de ordenadores y programas que administran y coordinan las actividades de los robots. El sistema a controlar se considera como un entorno, con el cual el sistema interactúa.

Un sistema controlador está compuesto por tareas, y cada tarea está sujeta a una serie de restricciones temporales. Considerando los requerimientos específicos de tiempo, los STRs se clasifican en 2 categorías (Burns, 1991):

- *STR crítico*: el incumplimiento de alguno de los requerimientos tiene efectos catastróficos sobre el sistema controlado, por lo que es imprescindible evitar esta situación. Tal sistema debe cumplir todos los requerimientos de tiempo.
- *STR no-critico*: si el sistema incumple ocasionalmente alguno de los requerimientos impuestos, produciendo una disminución en las prestaciones o calidad de la respuesta, el funcionamiento es considerado aun correcto.

Normalmente estas restricciones que se imponen a un STR se refieren al tiempo de respuesta de cada tarea. El *plazo* de una tarea es el tiempo máximo permitido para obtener una respuesta. El valor que la tarea aporta al sistema depende si termina o no dentro del plazo especificado, a partir de su instante de activación. Los plazos se clasifican de la siguiente manera:

- *Plazo estricto*: es importante que una tarea concluya su ejecución antes de su plazo. El valor que la tarea aporta al sistema es nulo si no se ejecuta antes de su plazo y el máximo si lo hace antes.

- *Plazo no estricto*: las consecuencias de que alguna tarea no termine su ejecución antes de su plazo no ponen en riesgo la integridad del sistema. El valor que la tarea aporta al sistema es máximo si esta es ejecutada antes de su plazo, disminuyendo proporcionalmente conforme al tiempo de terminación posterior al plazo.
- *Plazo firme (suave)*: si la tarea no termina su ejecución antes del plazo, la integridad del sistema no se compromete. El valor que la tarea aporta al sistema es nulo en este caso. Son similares a los plazos no estrictos, con la diferencia de que si una tarea no cumple su plazo el sistema es capaz de continuar su ejecución.

El número máximo de tareas que es posible ejecutar simultáneamente en una computadora es limitado (como máximo igual al número de procesadores), por lo tanto es necesario definir qué tareas deben ser ejecutadas en cada instante de tiempo. Estas reglas son definidas por los algoritmos, también llamados políticas de planificación, y de su elección depende en gran medida el que se satisfagan o no las restricciones temporales impuestas al sistema.

En la Figura 2.1 se generaliza la estructura de un STR.

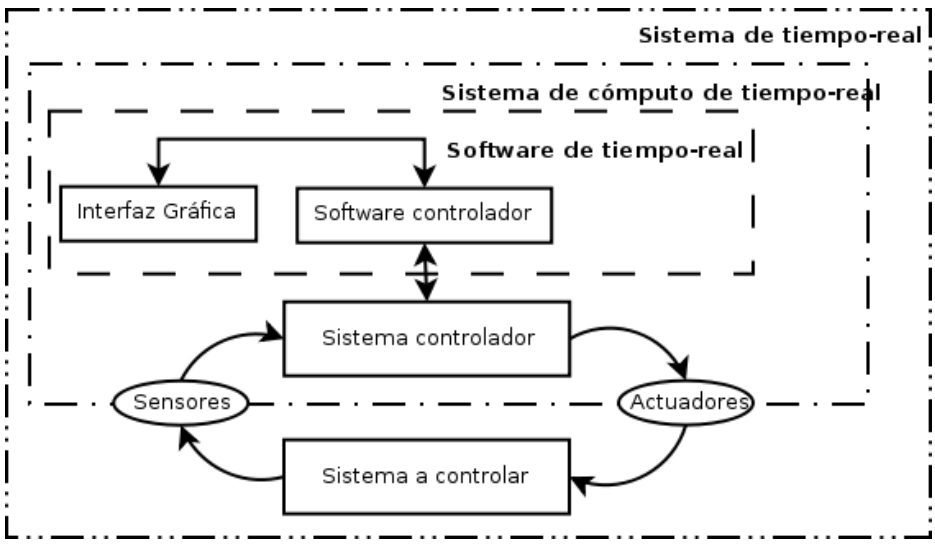

Figura 2.1 Componentes de un STR.

En una aplicación de tiempo-real, un sistema de cómputo y un entorno son dos elementos que se comportan de manera diferente debido a los dominios de tiempo (Buttazzo, 2011). El entorno se rige por la medición de la duración exacta en *tiempo cronométrico*. En cambio, el sistema de cómputo determina una secuencia de

instrucciones de máquina, las cuales se definen bajo un *tiempo cronológico*. Una aplicación de tiempo-real que está controlado por un sistema de cómputo no se preocupa por el tiempo cronométrico, o el tiempo cronológico, sino por la sincronía de ambos. A medida que el tiempo es establecido como un proceso físico, un STR tiene que adaptar la velocidad de sus acciones para cumplirse en el tiempo establecido.

Un STR está compuesto por diversas tareas denotadas por τ_i. Una tarea se define a través de varios parámetros, algunos de los cuales se muestran en la Figura 2.2, así como se menciona en (Buttazzo, 2011). La lista completa de las notaciones usadas se presenta en el Capítulo 7.

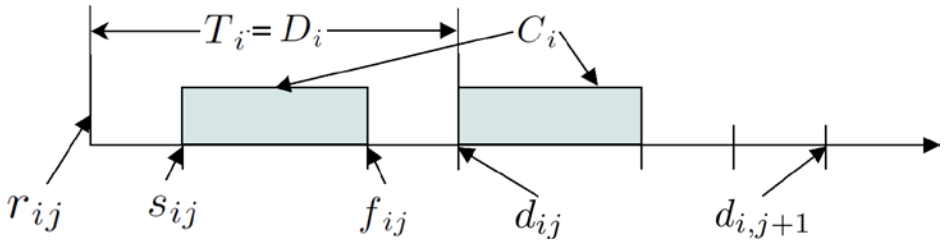

Figura 2.2 Parámetros principales de una tarea.

Una tarea que está lista para ejecutarse, se encuentra en uno de dos estados: 1) en ejecución si ha sido seleccionada por el planificador, o 2) en espera si alguna otra tarea de mayor (o de la misma) prioridad se encuentra en ejecución.

Una tarea que potencialmente está lista para ejecutarse en el procesador, o está ejecutando, es llamada *tarea activa*. Una tarea esperando por el procesador es llamada *tarea lista*, y todas las tareas esperando por el procesador son mantenidas en una cola de espera llamada *cola de tareas listas.* El planificador elige el orden de ejecución de las tareas basándose en la política establecida por el algoritmo de planificación elegido.

2.2 Restricciones a un STR

Las restricciones típicas que se especifican sobre las tareas de tiempo real son las siguientes: restricciones de tiempo, restricciones de precedencia y restricciones de exclusión mutua sobre recursos compartidos. A continuación estas restricciones se especifican más en detalle.

2.2.1 Restricciones en tiempo

Los STRs se caracterizan por actividades computacionales (tareas o procesos) con restricciones severas de tiempo que deben completarse en una orden para alcanzar un comportamiento deseado.

Una restricción en tiempo que caracteriza~~n~~ a las tareas es el plazo de respuesta. El plazo es el tiempo, antes del cual una tarea debe completar su ejecución sin causar un daño al sistema. Dependiendo de las consecuencias que causa la pérdida de los plazos, las tareas se distinguen normalmente en dos clases:

- *Duras*. Se dice que una tarea es dura si la pérdida del plazo de respuesta causa consecuencias catastróficas dentro del sistema. En este caso, cualquier instancia de las tareas debe garantizar a priori el cumplimiento del plazo. Esto se hace considerando el peor caso del tiempo de cómputo de las tareas.
- *Suaves*. Se dice que una tarea es suave si la pérdida de un plazo de respuesta no pone en peligro a personas o no causa pérdidas económicas. En los STR con las tareas suaves, la perdida de plazos produce únicamente una disminución en el desempeño del sistema.

2.2.2 Restricciones sobre recursos

Desde el punto de vista de un proceso, un recurso es cualquier dispositivo de Entrada/Salida (E/S) o estructura de software que se utiliza por el proceso. Típicamente, un recurso es una estructura de datos, un conjunto de variables, un área de memoria, un archivo, una porción de un programa o un dispositivo periférico. Un recurso que es dedicado exclusivamente a un proceso en particular se dice que es un *recurso privado*, mientras que un recurso se utiliza por dos o más tareas es llamado *recurso compartido*.

Para mantener la consistencia de datos en la mayoría de recursos compartidos, no se permite el acceso simultaneo por dos o más tareas a estos datos, sino que se requiere el cumplimiento de la condición de exclusión mutua entre las tareas que compiten por este recurso. Supongamos que *R* es un recurso exclusivo compartido por las tareas τ_1 y τ_2. Si α es la operación realizada por τ_1 sobre *R*, y β es la operación realizada sobre *R* por τ_2, entonces α y β nunca deben de ejecutarse al mismo tiempo. El conjunto de líneas de código que se ejecutan bajo restricciones de exclusión mutua se le conoce como sección

o región critica. Esto significa que cuando un recurso es asignado a alguna tarea ninguna otra podrá utilizar ese recurso hasta que sea liberado. Ejemplos de recursos son los monitores (mutex), bloqueos de lectura/escritura (locks), sockets de conexión, impresoras y servidores remotos.

2.2.3 Exclusión mutua e inversión de prioridad

La exclusión mutua conlleva en algunos casos al efecto de inversión de prioridad. Este efecto se produce cuando una tarea de prioridad alta tiene que esperar a que se libere un recurso que está siendo utilizado por otra tarea de prioridad menor. Nótese que este efecto se produce aunque la planificación del procesador sea expulsiva, lo cual viola el principio de expulsión y es una causa muy frecuente de incumplimiento de plazos. Hay muchas fuentes potenciales de inversión de prioridad, por ejemplo:

- Secciones de código no expulsable, de forma que durante su ejecución haya tareas de prioridad superior bloqueadas.
- Peticiones de uso de dispositivos que deban resolverse en orden de llegada, de forma que una tarea deba esperar a que se resuelvan las peticiones de tareas de menor prioridad efectuadas con anterioridad.

La inversión de prioridad reduce notablemente la planificabilidad de los STRs, por lo que es conveniente evitarla o, cuanto menos, reducirla. El principal esfuerzo en este sentido se ha hecho en la sincronización para recursos compartidos de exclusión mutua.

La sección de código que se ejecuta haciendo uso de un recurso compartido se conoce como sección crítica. El mecanismo normalmente utilizado para garantizar el acceso mutuamente exclusivo es el mutex, el cual protege las secciones críticas. Su funcionamiento se aprecia en la Figura 2.3 Operación de un mutex.. Este mecanismo, sin embargo, ocasiona retrasos muy grandes cuando se aplica en STRs. La inversión de prioridad no acotada se produce cuando una tarea de prioridad baja bloquea un mutex que protege a un recurso compartido con otra tarea de prioridad alta, y esta es expulsada por la ejecución de una tarea de prioridad intermedia. Esta expulsión provoca una inversión de prioridad de duración igual a la ejecución de todas las tareas de prioridad intermedia que ocasionan la expulsión de la tarea de baja prioridad. El tiempo de espera es excesivamente largo si los intervalos de tiempo que mantendrán acaparado el procesador son tardados.

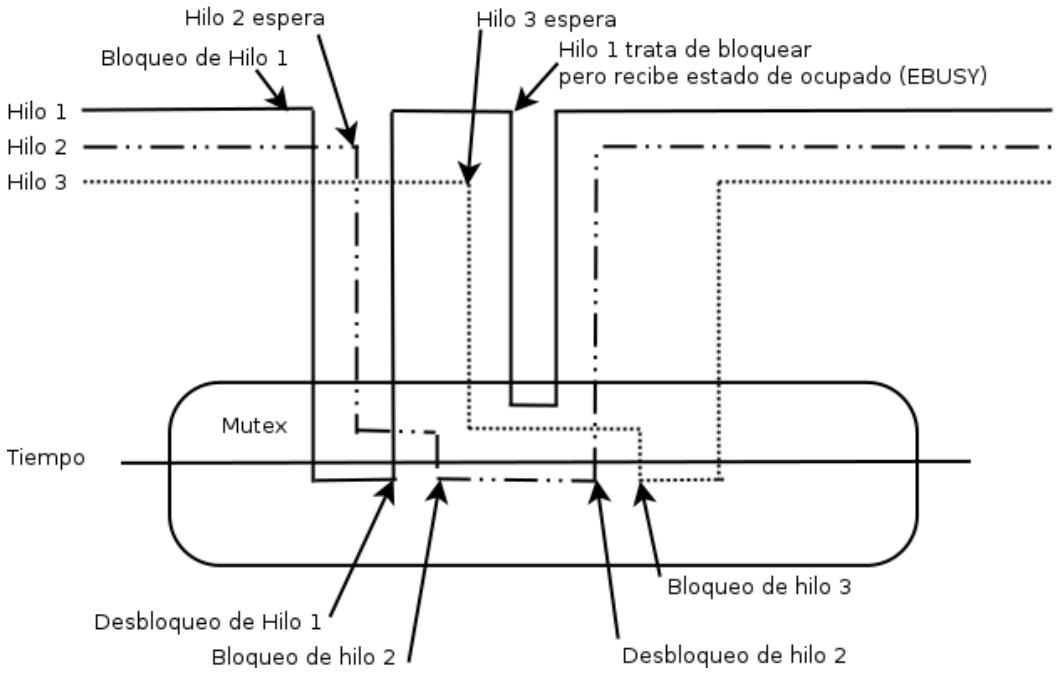

Figura 2.3 Operación de un mutex.

El uso de mutex presenta también otro problema, el bloqueo mutuo, en que dos tareas están interbloqueadas esperando ambas a que la otra libere un recurso que necesitan para ejecutar. Las consecuencias de este efecto son catastróficas, puesto que ambas tareas se quedan indefinidamente esperando y pierden sus plazos con toda seguridad.

El objetivo de los protocolos de sincronización de tiempo-real es precisamente evitar esas inversiones de prioridad no acotadas, minimizando la duración de las secciones críticas y evitar cualquier tipo de bloqueo destructivo. Los principales protocolos desarrollados para sistemas basados en prioridades fijas son (Mitra, 2001):

- *Protocolo de herencia de prioridades* (PIP): En este protocolo una sección puede ejecutarse a diferentes prioridades. La prioridad que se asigna a la sección crítica es la mayor de las prioridades de las tareas que están bloqueadas por ese recurso. Si durante la ejecución de la sección crítica una tarea de prioridad superior a la de la sección crítica intenta tomar el mutex se verá bloqueada, pero la sección crítica aumenta su prioridad automáticamente hasta esa prioridad superior. Este protocolo elimina totalmente la inversión de prioridad, aunque produce tiempos de bloqueo de peor caso mayores que los de los protocolos de protección y techo de prioridad.

- *Protocolo de techo de prioridad* (PCP): Este protocolo está introducido para evitar el problema de los bloqueos mutuos, así como los bloqueos encadenados, que presenta el protocolo de herencia de prioridades, y producidos cuando existen tareas que comparten diferentes recursos simultáneamente, de forma que pueda haber diferentes secciones críticas encadenadas en una misma tarea. Es el protocolo de herencia de prioridades más una regla acerca de la atención a la petición de bloqueo sobre un mutex libre: una tarea no bloquea un mutex libre a menos que su prioridad sea estrictamente mayor que la del techo del sistema (máximo techo de todos los mutex actualmente bloqueados por otras tareas). Este protocolo previene el bloqueo mutuo siempre y cumple la propiedad de bloquear como máximo una vez a segmentos de código contiguo. La implementación de este protocolo da lugar a una pequeña sobrecarga debido a que hay que calcular siempre el techo de prioridad del sistema.

2.3 Complejidad

Para llevar a cabo la planificación de tareas de STRs críticos es indispensable considerar la complejidad computacional. Cualificar la complejidad computacional de los problemas ofrece un panorama sobre cuales problemas son difíciles y cómo abordarlos mejor. Muchos de los problemas en la teoría de los STRs es bien sabido que son difíciles, en el sentido que estos han sido comprobados que pertenecen a alguna clase de problemas complejos ya conocidos.

Los algoritmos que demuestran una complejidad exponencial, deben ser evitados para aquellos sistemas que requieran de una planificación en línea (*online*), debido a que el tiempo que se utiliza para elegir una tarea, llega a ser extremo. Por este lado, aspectos de la planificación de tareas llegan a ser intratables e inadecuados, aun para aquellos algoritmos que realicen sus acciones fuera de línea (*offline*). Para contrarrestar este fenómeno que se presenta, es necesario considerar la computabilidad y la decidibilidad. Para planificar un conjunto de tareas arbitrario, estos términos se interpretarían de la siguiente forma:

- Computabilidad, dada una planificación de tareas, calcular si es completamente planificable.
- Decidibilidad, decidir si existe una planificación realizable.

Los problemas NP-completos presentados por (Garey y Johnson, 1979) son considerados computacionalmente intratables. Los STRs críticos recaen en algunos de ellos.

2.4 Sistemas operativos de tiempo-real

2.4.1 Principios

Un sistema operativo (SO) es aquel que es capaz de administrar el hardware de un sistema digital de cómputo, el cual está constituido por uno o más procesadores, memoria principal, teclado, interfaces de red y otros dispositivos de entrada y salida. Los SOs han evolucionado a la par con las arquitecturas de los sistemas digitales y las necesidades de las diferentes aplicaciones han causado que los desarrolladores trabajen en diversos tipos de SO para esas diversas necesidades.

El núcleo de un SO de tiempo compartido, a menudo ofrece una serie de características que para una aplicación en particular no es necesaria la mayoría de las ocasiones, pero sí están incluidas en el Kernel del SO (Silberschatz et al., 2006) haciéndolo más sofisticado, penalizando el tiempo requerido para realizar las tareas que tenga encomendadas.

Recientemente los SOs que proporcionan un Kernel mínimo (microkernel) se están haciendo muy populares, el resto de las características se añadan desde un lenguaje de alto nivel por un programador de aplicaciones. El término microkernel no quiere decir que estrictamente el tamaño que ocupa sea pequeño, sino que está diseñado de manera que realiza solamente las misiones fundamentales del sistema, dejando para otros procesos el atender a los restantes servicios del SO (Baskiyar y Meghanathan, 2004). En la Figura 2.4 es mostrado las diferencias que representan a estos tipos de sistemas operativos.

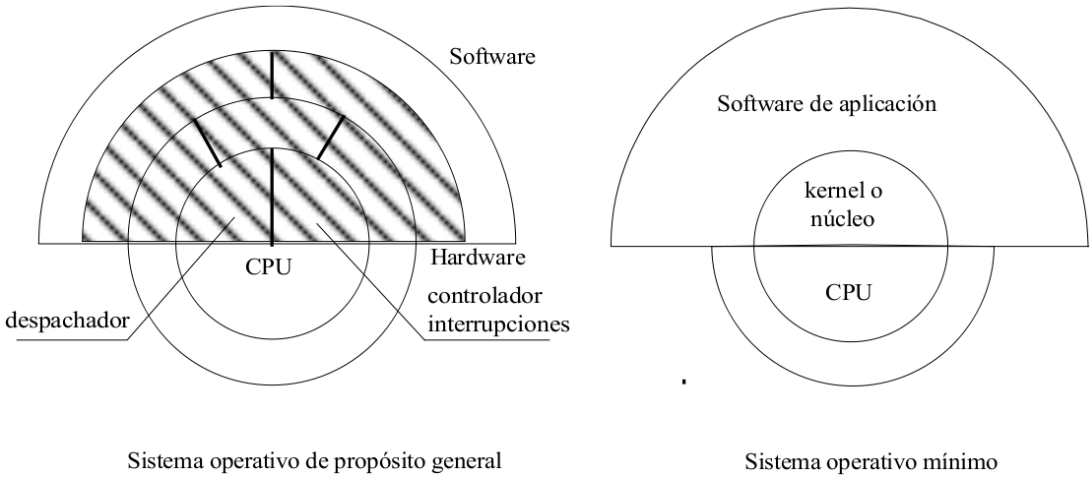

Figura 2.4 Diferencia entre un SO de propósito general y un SO mínimo.

Una de las funciones fundamentales de un SO es asignar recursos de la computadora a las diversas actividades que se deben realizar. En un Sistema Operativo de Tiempo-Real (SOTR), este proceso es complicado debido al hecho de que las tareas mantienen restricciones temporales críticas con respecto al tiempo, y unas tienen mayor prioridad que otras. El control de todo el sistema recae sobre el módulo llamado planificador de tareas responsable de la asignación de cada uno de los recursos de la CPU (Silberschatz et al., 2006).

Los SOTR se caracterizan por presentar requisitos especiales en cinco áreas generales:

1. Determinismo.
2. Sensibilidad.
3. Control del usuario.
4. Fiabilidad.
5. Tolerancia a los fallos.

Hoy en día existen una gran variedad de SOTRs, algunos de ellos se muestran en la Tabla 2.1.

Tabla 2.1 SOTRs de código abierto y comerciales.

Open Source	**Propietario**
Free RTOS	LynxOS

Preempt-RT	MERT
RTAI	OSE
RT-Linux	VxWorks
MarteOS	VRTX
Xenomai	RT-Linux
Phoenix-RTOS	QNX
	Symbian

2.4.2 Componentes

2.4.2.1 Proceso

Un proceso de una manera informal puede ser visto como un programa en ejecución, sin embargo es más que código de un programa. Un proceso se define como la unidad de trabajo en la mayoría de los SOs y se caracteriza por tener información como: el código de un programa, el contador de programa que indica la siguiente instrucción a ejecutar, una pila que contiene datos temporales, así como las direcciones de regreso, las variables temporales de datos y los parámetros del proceso. Además, los procesos tienen una sección de datos, en donde se almacenan todas las variables globales que este ocupe (Buttazzo, 2011).

Un *programa* por sí mismo no es un proceso, sino una entidad pasiva, como el contenido de un archivo o programa almacenado en disco, mientras que un *proceso* es una entidad activa que controla el sistema operativo con un contador de programa que especifica la siguiente instrucción a ejecutar y el conjunto de recursos asociados.

2.4.2.2 Estados del proceso

Durante la ejecución el estado de un proceso se cambia y por lo tanto, está definido por la actividad que esté realizando. Como se muestra en la Figura 2.5, cada proceso se encuentra en uno de los siguientes estados:

- *Nuevo*: El proceso está siendo creado. En este estado, se le asignan al proceso los recursos necesarios para inicializarse, se reserva un espacio en memoria para

cargar código, datos y pila (*stack*); por lo que se genera el bloque de control de procesos.

- *Listo*: El proceso está esperando la asignación a un procesador.
- *Espera*: El proceso está esperando por algún evento a ocurrir.
- *Ejecución*: Las instrucciones del proceso se están ejecutando.
- *Terminado*: Pasa a este estado, solamente si el proceso ha terminado su ejecución.

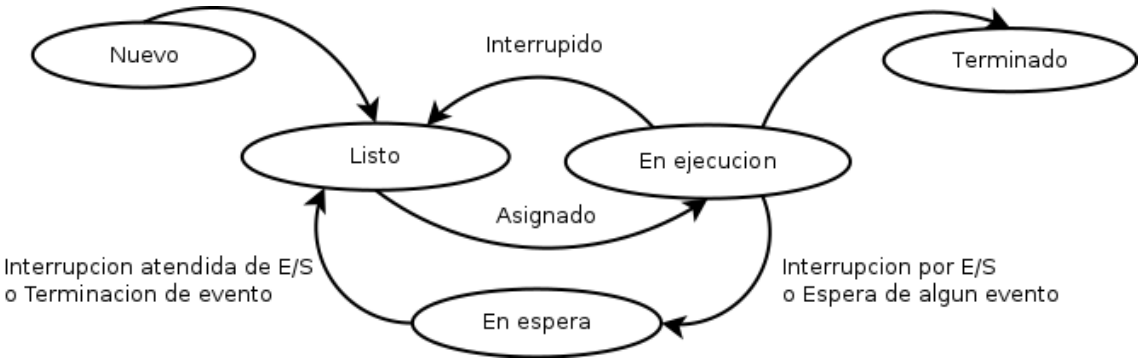

Figura 2.5 Diagrama de estados de un proceso.

En un procesador en un instante determinado sólo un proceso a la vez se estar corriendo, mientras varios procesos listos podrían estar en una cola de procesos esperando a que se les asigne el procesador. Los nombres de los estados son arbitrarios y varían entre los SOs. Los estados, en los que un proceso transita, son mencionados en todos los SOs y se presentan en la Figura 2.5.

Como también se muestra en la Figura 2.5, un proceso cambia de estado durante su ejecución, debido a alguno de los siguientes eventos:

- *Admitido*: Esta transición ocurre cuando el proceso es creado y está listo para competir por los recursos del sistema (CPU, memoria, dispositivos de E/S, entre otros).
- *Despachado*: Ocurre cuando el planificador elige al proceso de la cola de los procesos listos y le asigna el procesador.
- *Interrumpido*: Ocurre cuando el proceso cumple con su tiempo de computo (Quantum) o cuando algún proceso de mayor prioridad le quita el procesador.
- *Señal de E/S o Espera por evento*: Ocurre cuando el proceso necesita esperar por algún dispositivo E/S o por la llegada de un evento.

- *Señal de E/S o Terminación de un evento*: Es cuando un dispositivo o un evento que esperaba el proceso ya ocurrió, y por lo tanto cambia el estado a listo para su ejecución.
- *Fin*: Ocurre cuando el proceso ha terminado de realizar todas sus tareas y por lo tanto se elimina del sistema.

2.4.2.3 Interrupciones

Las dos tareas principales de una computadora son las entradas y salidas (E/S), y el procesamiento (Silberschatz et al., 2006). En muchos casos la tarea principal es la E/S, por ejemplo cuando leemos una página web o editamos un archivo, nuestro interés inmediato es leer o introducir cierta información, o bien cuando escuchamos música siempre deseamos que el audio se mantenga fluido, sin ninguna interrupción. El papel de un SO de la computadora consiste en gestionar y controlar las operaciones y dispositivos de las E/S.

El control de los dispositivos conectados a la computadora es una de las principales preocupaciones de los diseñadores de sistemas. Debido a que existe una variedad tan amplia de dispositivos de E/S, tanto en lo que respecta a sus funciones como en cuanto a su velocidad de operación. En el campo de la tecnología de dispositivos de E/S se experimentan dos tendencias que están en conflicto mutuo. De un lado, se anota una creciente estandarización de las interfaces software y hardware; esta tendencia ayuda a la compatibilidad de los dispositivos dentro de las computadoras, aun cuando las nuevas generaciones de los SOs emergen.

Por otro lado, aparece una variedad cada vez más amplia de dispositivos de E/S; algunos de estos son tan distintos de los anteriores que constituyen todo un desafío al incorporarlos en las computadoras y poner en un correcto funcionamiento bajo los SOs. Este desafío se afronta mediante una combinación de técnicas de hardware y software. Los elementos básicos del hardware de E/S, como los puertos, buses y controladores de dispositivo, permiten integrar una amplia variedad de dispositivos de E/S. Los controladores de dispositivo presentan al subsistema de E/S una interfaz uniforme de acceso a los dispositivos, de forma parecida a cómo las llamadas al sistema proporcionan una interfaz estándar entre la aplicación y el SO.

2.4.2.4 Manejador de procesos

Un proceso en el contexto considerado es un programa, cuya ejecución requiere de ciertos recursos (procesador, memoria, archivos, dispositivos E/S) para lograr su activación. El manejador de procesos proporciona servicios primordiales que todo SO debe proveer. Este incluye varias funciones de soporte de creación, terminación, planificación y despacho de procesos, así como el manejo del cambio de contexto (Buttazzo, 2011).

Un proceso durante su ejecución crea distintos subprocesos mediante llamadas al sistema a través de la rutina de creación de procesos. El proceso principal es llamado proceso *padre*, a los procesos creados a través de este son considerados *procesos hijos*. Cada proceso hijo, a su vez crea nuevos procesos formando así un árbol de procesos. Usualmente un proceso requiere de recursos, por lo que cuando se crea un subproceso, el último es capacitado para obtener los recursos directamente desde el SO obligado a tener un subconjunto de recursos desde el proceso padre.

Un proceso es terminado cuando finaliza la ejecución. En este punto el proceso solicita al SO que lo borre, regresa los datos al proceso padre y todos los recursos que ocupaba (memoria física y virtual, archivos, buffers de E/S, etc.) se devuelven al SO.

En un SO de tipo multiprogramación, se maneja el concepto de *contexto* para mantener información sobre el estado en que se encuentra cada proceso. El cambio de contexto ocurre cuando ejecución de un proceso es interrumpido para que otro proceso utilice el procesador. Esta interrupción ocurre cuando al proceso se le termina su Quantum, o cuando un proceso de mayor prioridad demanda al procesador.

Los pasos que se realizan en el cambio de contexto son los siguientes:

1. Se guarda el estado de la tarea que está en ejecución.
2. El planificador de tareas selecciona la siguiente tarea a ejecutar.
3. Se carga el estado de la siguiente tarea para que al finalizar el cambio de contexto se mantengan guardados los datos que el programa requiere para iniciar o continuar su ejecución en caso de que haya sido suspendida.

El cambio de contexto es considerado como sobrecarga, ya que durante el tiempo que este se lleva a cabo, el sistema queda incapacitado para realizar alguna operación útil. El tiempo que tarda en ejecutarse el cambio de contexto varia de una maquina a otra,

depende de la velocidad de la memoria, el número de registros que contenga y la existencia de instrucciones especiales (tales como alguna instrucción que permita cargar o almacenar todos los registros de una sola vez) (Silberschatz et al., 2006).

2.4.2.5 Manejador de memoria

Cada dato almacenado en memoria RAM cuenta con una dirección propia, siendo además un dispositivo de acceso rápido por el procesador o los dispositivos E/S. Un procesador utiliza la memoria para leer las instrucciones de un programa que se encuentran almacenadas y además sirve para escribir información temporal. Los dispositivos de E/S la utilizan para leer y escribir mediante el Acceso Directo a Memoria (*Direct Memory Access*, DMA).

Las funciones que realiza el manejador de memoria realizar son:

- Decidir qué procesos cargar en memoria cuando esta esté disponible.
- Conocer qué partes de la memoria están siendo utilizadas y por quien.
- Liberar y conceder espacio de memoria cuando se le requiera.

Los mecanismos que se utilizan para administrar la memoria, varían de acuerdo a la manera en que esta es utilizada (paginación, segmentación, reemplazo de páginas, etc.). La elección de estos mecanismos depende en gran parte del hardware, para el cual se esté diseñando un SOTR. Por ejemplo, en los sistemas empotrados la memoria es escasa y todos los procesos se encuentran residentes en memoria por lo que no es necesario tener un administrador de memoria complejo.

2.4.2.6 Manejador de reloj

El manejador de reloj realiza diferentes acciones de acuerdo a las interrupciones que genera el reloj. Entre las funciones que controla el manejador se encuentran:

- Provenir información para el calendario de procesos (planificadores).
- Mantener actualizada la fecha y la hora del día.
- Manejar las alarmas.

Como se anota, el reloj de tiempo-real permite al sistema configurar los tiempos para la planificación de las tareas. Además, mediante el manejador de reloj es posible

configurar interrupciones para controlar los dispositivos externos, recibir y sincronizar señales de comunicación y monitorear el cumplimiento de los requisitos temporales de las tareas del sistema.

2.4.2.7 Mecanismos de sincronización y comunicación

Este es otro mecanismo básico que todo Kernel debe proveer. La comunicación permite que los procesos se comuniquen y se sincronicen. Además, evita la inconsistencia de datos, la cual es un problema muy común en los sistemas concurrentes. La manera clásica para resolver la sincronización es utilizando mecanismos de exclusión mutua o semáforos. La comunicación se realiza a través de memoria compartida o por paso de mensajes mediante buzones.

2.4.3 El parche RT-Preempt

El incremento de la demanda en el mercado de sistemas caracterizados por bajas latencias, su comportamiento determinista, y el uso de hardware y software para uso industrial, ha permitido el surgimiento de nuevos SOTR, uno de ellos es RT-Preempt, desarrollado por Ingo Molnar (Rostedt y Hart, 2007), quien es desarrollador de la conocida distribución de Linux, Red Hat. RT-Preempt es un parche que es aplicado a un Kernel de Linux, el cual ofrece características adicionales, éste parche no solo ofrece beneficios a aquellas industrias que requieran de sistemas operativos de tiempo-real, sino que ofrece mejoras a la gama de núcleos de donde desciende RT-Preempt.

Este parche permite que el Kernel de Linux trabaje casi en su totalidad usando un enfoque preferente, esto es, tan luego una tarea de prioridad alta es activada, ésta adquiere el uso del procesador con una baja latencia, sin la necesidad de esperar a una tarea con menor prioridad que termine su proceso. De igual manera para limitar la impredecibilidad causada por el manejo de recursos compartidos, RT-Preempt provee protocolos de sincronización como son el de herencia de prioridades. Ofrece un completo soporte del estándar POSIX de tiempo-real, por lo que es una ventaja enorme al ofrecer protocolos ya establecidos.

Con algunas modificaciones en el Kernel, las cuales se enlistan enseguida, este parche ofrece que Linux adquiera las características que un SOTR debe poseer para

asegurar la ejecución y cumplimiento de las restricciones de los STRs, modificando principalmente las secciones que causan que Linux sea impredecible:

- Las primitivas utilizadas para bloquear el acceso a un recurso ahora son expulsables, gracias a que se emplea un mecanismo llamado *rt-mutexes* en comparación con los *spinlocks* (mecanismo de espera a través de un ciclo). Este último no permite que el Kernel sea expulsable una vez que un recurso es adquirido.
- kl protocolo de herencia de prioridad es implementado para los mecanismos de sincronización *mutexes* dentro del Kernel y no solo para procesos del usuario.
- El manejador de interrupciones ahora es considerado por el Kernel como un hilo que puede ser expulsado. En un SOPG, este tipo de procesos son ejecutados de manera inmediata, aun si otra tarea de mayor prioridad existiera en el sistema. Estos procesos están diseñados como no-expulsables.
- RT-Preempt incorpora dos *timer* de alta resolución, haciendo que el API del *timer* de Linux se divida en 2 estructuras separadas: una para proveer un *timer* a nivel Kernel y la otra para los programas de usuario, todo esto para que se ajuste al estándar de POSIX. *Timers* con alta resolución permiten que aplicaciones las aplicaciones tengan una mayor precisión en cuestiones de planificación.
- Ofrece sus funcionalidades en varias arquitecturas.

El control de las interrupciones a través de hilos permite que la latencia sea menor. Cuando una interrupción es disparada, en lugar de llamar inmediatamente al controlador, el núcleo planifica el hilo responsable de dar respuesta a ésta interrupción. Esto permite que se pueda reducir en gran parte el tiempo promedio en que una interrupción pueda ser planificada. Al igual de los demás procesos que existen en un sistema, el planificador elige de entre todos de acuerdo a su prioridad a ser ejecutados.

2.5 Conclusiones del capítulo

Para que la ejecución de un STR resulte satisfactorio, el diseñador de sistemas debe reconocer hacia qué área es aplicado el STR, para así partir y desarrollar un sistema considerando el sistema a controlar, sin pasar por alto las características del entorno en el cual este se encontrará ejecutándose, ya que aunque un SOTR ofrezca un comportamiento como tal, debe conocerse la manera en que se lleva a cabo la planificación y las

características que este ofrece. Esto para que de acuerdo al diseño el STR sea planificable y determinístico.

Capítulo 3

Algoritmos de planificación en sistemas uniprocesador

Para definir un problema de planificación en un STR crítico es necesario especificar los siguientes tres conjuntos (Stankovic et al., 1998): un conjunto Θ de N tareas, un conjunto de M procesadores y un conjunto de R recursos. Además, es necesario especificar las restricciones de tiempo y relaciones de precedencia entre las tareas. En este contexto, la planificación permite asignar los procesadores del conjunto M y los recursos del conjunto R a tareas del conjunto Θ de acuerdo a un orden que permita completar todas las tareas bajo las restricciones impuestas. Se ha demostrado que este problema es NP-completo (Baruah et al., 1990), lo que implica que su solución es difícil de obtener.

Para reducir la complejidad en la construcción de planificadores, podemos simplificar la arquitectura de la computadora, por ejemplo, restringiendo al sistema para que utilice un solo procesador, adoptar un modelo con desalojo, utilizar prioridades fijas, eliminar precedencias o restricciones de recursos, asumir una activación simultanea de tareas y suponer que en el sistema existen conjuntos de tareas homogéneos (únicamente periódicas o únicamente aperiódicas). Sin embargo, tales simplificaciones sólo resuelven casos particulares del problema.

3.1 Particularidades de algoritmos de planificación en un STR

3.1.1 Clasificación de algoritmos de planificación

De entre la gran variedad de algoritmos propuestos para la planificación de tareas en tiempo real, se identifican las siguientes clases:

- *Planificación con desalojo*. La tarea en ejecución puede ser interrumpida en cualquier momento para asignar al procesador a otra tarea, de acuerdo a una política de planificación predefinida.
- *Planificación sin desalojo*. Una tarea una vez iniciada, es ejecutada por el procesador hasta que termine su actividad. En este caso, todas las decisiones de planificación son tomadas cuando una tarea inicia o termina su ejecución.

- *Planificación estática*. En los algoritmos estáticos las decisiones de planificación están basados en parámetros fijos, los cuales son asignados a las tareas antes de su activación.
- *Planificación dinámica*. En los algoritmos dinámicos las decisiones de planificación están basadas en parámetros dinámicos que pueden cambiarse durante la ejecución del sistema.
- *Planificación fuera de línea*. Decimos que un algoritmo de planificación es usado fuera de línea si es ejecutado sobre el conjunto completo de tareas antes de la ejecución actual de las tareas. El planificador generado en esta forma, es almacenado en una tabla y después ejecutado por un despachador.
- *Planificación en línea*. Decimos que un algoritmo de planificación es usado en línea si las decisiones de planificación son tomadas a tiempo de ejecución cada vez que una nueva tarea llega o cuando una tarea termina su ejecución.
- *Planificación óptima*. Se dice que un algoritmo es óptimo si minimiza algunas funciones de costo definidas sobre el conjunto de tareas. Cuando ninguna función de costo es definida y lo único concerniente es alcanzar una planificación factible, entonces se dice que un algoritmo de planificación es óptimo si este encuentra una solución de planificación que no puede ser contradicha por ningún otro algoritmo de planificación. La solución de planificación indica si un conjunto de tareas es factible o no factible (cumple o no con sus plazos de respuesta.
- *Planificación heurística*. Un algoritmo heurístico encuentra una solución aproximada que intenta estar cerca de la solución óptima.

La clasificación se rige en trabajos de (Baruah et al., 1990; Audsley et al., 1995; Burchard et al., 1995; Anderson y Devi, 2011; Davis et al., 2016). A continuación se presenta un análisis de los tipos de planificadores tanto estáticos como dinámicos.

3.1.2 Planificadores basados en prioridades estáticas

En esta opción, las prioridades de las tareas se asignan en una forma estática, antes del inicio del sistema, y se no cambian durante su ejecución. En la planificación basada en prioridades estáticas *(off-line*), se realiza un análisis de planificabilidad de las tareas fuera de línea. Esta planificación tiene como ventajas:

- Produce sobre-cargas muy bajas, ya que la asignación de prioridades se realiza una sola vez antes de la ejecución. Las prioridades asignadas son guardadas en una tabla, la cual permite al planificador decidir el orden de ejecución de las tareas.
- Permite verificar el comportamiento temporal de las tareas a priori (predictibilidad). Es adecuada para trabajar con tareas con plazos críticos. El análisis de planificabilidad nos permite comprobar a priori si dichas tareas cumplirán sus plazos de respuesta.
- En situaciones de sobrecarga del sistema, siempre es posible predecir que las tareas de menor prioridad serán las primeras en perder sus plazos.

Sus principales desventajas son:

- Requiere un conocimiento previo de todos los parámetros de las tareas.
- Es incapaz de tratar adecuadamente las tareas aperiódicas.
- Inflexible cuando opera bajo ambientes dinámicos en donde los parámetros cambian constantemente.

3.1.3 Algoritmo Rate Monotonic

En el algoritmo Rate Monotonic (RM), la asignación de las prioridades se obtiene de acuerdo a los periodos de las tareas. A la tarea con menor periodo, se le asigna la mayor prioridad. En este algoritmo de planificación, el periodo es igual al plazo.

Fueron Liu y Layland (1973) quienes propusieron el algoritmo RM, además demostraron que:

Teorema 3.1 El algoritmo RM es óptimo dentro de los esquemas de asignación de prioridades estáticas.

Por óptimo se entiende que si se tiene una planificación factible para un conjunto dado de tareas mediante un algoritmo de asignación con prioridades estáticas, entonces ese conjunto de tareas también será planificable mediante el algoritmo RM.

En el análisis de este algoritmo (Liu y Layland, 1973), los autores proporcionan un control de admisión de tareas para RM basado en la utilización del procesador. Este control de admisión, compara la utilización U del conjunto de tareas con una cota límite que depende del número de tareas del sistema, es decir, un conjunto de n tareas no perderá

su plazo si cumple la siguiente condición:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right). \quad (3.1)$$

Teorema 3.2 La condición suficiente para que la planificación de un conjunto de *n* tareas periódicas sea factible mediante el algoritmo RM, es que su factor de utilización mínima $n\left(2^{1/n} - 1\right)$ cumpla la siguiente condición:

$$U \leq U_{min} \leq n\left(2^{1/n} - 1\right). \quad (3.2)$$

La Tabla 3.1 describe el comportamiento de la expresión anterior para distintos valores de *n*.

Tabla 3.1 Utilización mínima garantizada para *n* tareas.

$\boldsymbol{n}$	$\boldsymbol{U_{min}}$
1	1
2	0.8284
3	0.7798
4	0.7568
5	0.7433
…	…
∞	0.6931

Como se aprecia en Tabla 3.1, al aumentar el número de tareas, la utilización mínima garantizada converge en:

$$U_{min} = \ln 2 \approx 0.6931.$$

3.1.4 Planificadores basados en prioridades dinámicas

En este tipo de planificación las prioridades de las tareas son asignadas durante la ejecución del sistema, sin embargo todas las tareas y sus parámetros temporales son conocidos desde el inicio de la ejecución. En la planificación basada en prioridades dinámicas (*on-line*, en línea), se realiza un análisis de planificabilidad fuera de línea.

Sus principales ventajas son:

- Es posible aprovechar mejor el procesador. Debido a que el planificador asigna las prioridades de las tareas en una forma dinámica, el CPU puede utilizarse de forma más eficiente que en el caso de la planificación por prioridades estáticas.

Y las desventajas principales son:

- En una sobrecarga del sistema no es posible predecir que tareas pierden sus plazos de respuesta. Lo que es más grave, es que en una sobrecarga, la perdida de plazos de algunas tareas puede producir un efecto en cascada de perdida de plazos de otras tareas.
- Se incrementa la sobrecarga causada por la planificación. Esto se debe a que la planificación continuamente tiene que ordenar las tareas por orden de prioridad, lo cual introduce sobrecarga al sistema.

3.1.5 Algoritmo Earliest Deadline First

El algoritmo Earliest Deadline First (EDF) uno de los algoritmos más conocidos para el esquema de asignación dinámica es el algoritmo de planificación guiado por plazos (Deadline Driven Scheduling Algorithm, DDSA), al cual posteriormente se le denominó como el plazo más próximo primero (EDF).

En el algoritmo EDF las prioridades se asignan en una forma dinámica. La política de asignación de prioridades consiste en asignar la prioridad más alta a la tarea con plazo más cercano. Para este algoritmo la condición de planificabilidad se define a continuación.

Teorema 3.3 La condición necesaria y suficiente para que un conjunto de tareas periódicas tenga una planificación factible mediante el algoritmo EDF es:

$$U \leq 1. \tag{3.3}$$

Esta condición de planificabilidad permite que EDF consiga un factor de utilización del 100% para los conjuntos de tareas que planifica, por lo que podemos decir que EDF es óptimo globalmente. Es decir que si existe un algoritmo que proporcione una planificación factible con un determinado conjunto de tareas periódicas, entonces EDF también proporcionará una planificación factible para dicho conjunto de tareas.

En la Tabla 3.2 se muestra un conjunto de tareas, cuya utilización total es del 82% y

la Figura 3.1 muestra el comportamiento de la ejecución de las tareas bajo el algoritmo de planificación EDF. Como se observa, no hay pérdida de plazos ya que cumple con la condición del Teorema 3.3. Planificación del conjunto de tareas descritas en la Tabla 3.2 bajo algoritmo EDF se muestra en la Figura 3.1.

Tabla 3.2 Conjunto de tareas, cuya utilización es de 82%.

Tarea	T_i	C_i	**Utilización**
1	30	10	0.333
2	40	10	0.250
3	50	12	0.240
Total			0.823

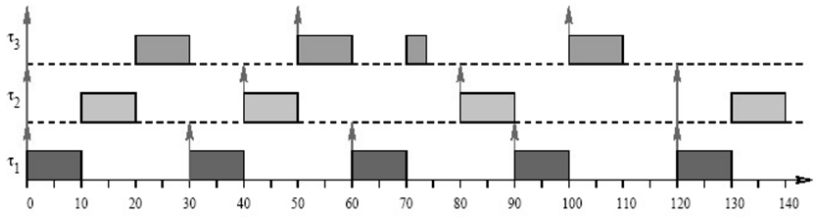

Figura 3.1 Planificación del conjunto de tareas bajo EDF.

3.2 Planificabilidad de un conjunto de tareas con un algoritmo

3.2.1 Pruebas de planificabilidad

En aplicaciones de tiempo-real críticas que requieren un comportamiento altamente predecible, el cumplimiento de plazos de respuesta de todas las tareas del sistema debe ser garantizado en una forma anticipada, es decir, antes de la ejecución de las tareas. De esta forma, si una tarea crítica no puede ser planificada dentro de su plazo de respuesta, el diseñador del sistema cambia los parámetros temporales de las tareas u optimiza el software a fin de lograr una planificación factible. Para comprobar la viabilidad de la planificación antes de la ejecución de las tareas, el sistema tiene que planear sus acciones asumiendo escenarios en el peor caso.

En sistemas de tiempo real estáticos, donde el conjunto de tareas es fijo y conocido, todas las activaciones pueden ser recalculadas fuera de línea, y la planificación completa se almacena en una tabla que contiene todas las tareas ordenadas y garantizadas. En este caso, durante la ejecución, el despachador (el cual forma parte del planificador)

simplemente sigue el orden de ejecución de las tareas definido en la tabla.

La ventaja principal del método estático es que el tiempo de ejecución del algoritmo de planificación no se toma en cuenta. Esto permite la utilización de algoritmos muy complejos para encontrar secuencias de planificación óptimas. Por otro lado, una desventaja de este tipo de planificación es que produce un sistema inflexible a cambios en el ambiente. En sistemas de tiempo real dinámicos, donde nuevas tareas se activan en tiempo de ejecución, la garantía de planificabilidad debe realizarse en línea cada vez que una tarea entra o sale del sistema.

Debido a que las pruebas de planificabilidad están basados sobre suposiciones en el peor caso, muchas tareas podrían no ejecutarse. Esto implica que la garantía de tareas duras se alcanza con un costo de reducción del desempeño promedio del sistema. Por otro lado, el beneficio de tener una prueba de planificabilidad es que las situaciones de sobrecarga pueden ser detectadas anticipadamente y para evitar efectos negativos en el sistema.

Para los STRs es crucial conocer si un conjunto de tareas es planificable o no lo es. Este problema es resuelto con una condición matemática, el cual es comúnmente llamado prueba de planificabilidad. De acuerdo a Baruah et al. (1990), este tipo de pruebas en sistemas uniprocesador pertenece es un problema co-NP-complete en el sentido fuerte para modelos no triviales. En sistemas donde las prioridades relativas de las tareas (como en el caso de RM) o trabajos (como en EDF) no varían, el análisis de planificabilidad su complejidad es menor (Burns, 1991).

Existen dos tipos de pruebas: exactos (suficiente y necesario) e inexactos (suficiente pero no necesario) Burns (1991). Si conjunto de tareas es planificado con un algoritmo y este satisface la prueba exacta, entonces se dice que todas las tareas cumplirán con sus plazos. En contraste, si un conjunto de tareas no satisface la prueba inexacta, no es posible asegurar si las tareas lograrían completar su ejecución de acuerdo a sus plazos. Un ejemplo de este tipo de pruebas fue presentado por Liu y Layland (1973), donde demostraron que el algoritmo RM es una política óptima para la asignación de prioridades estáticas. El test propuesto por ellos, mostrada en (1), establece que un conjunto de n tareas periódicas es planificable bajo RM si:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right). \qquad (3.4)$$

Los autores probaron que RM es capaz de planificar cualquier conjunto de tareas con plazos implícitos (periodos iguales a sus plazos) si la utilización total del procesador satisface que $U_T = \ln 2 \approx 0.6931$, donde UT está dado por:

$$U_\tau = \sum_{i=1}^{n} \frac{C_i}{T_i}. \tag{3.5}$$

Lehoczky et al. (1989) propusieron condición de planificabilidad suficiente y necesaria para la política RM (prueba exacta), la cual considera el factor de utilización del procesador del conjunto de tareas periódicas como una función de tiempo en un instante crítico. Sea Θ un conjunto de *N* tareas cada una con periodos $T_1 \leq T_2 \leq \cdots \leq T_n$ respectivamente, en un STR uniprocesador. La demanda acumulada en el procesador por un conjunto de tareas sobre el intervalo de tiempo [0, t] en un instante crítico es:

$$W_i(t) = \sum_{i=1}^{j} C_j \left\lceil \frac{t}{T_j} \right\rceil, \tag{3.6}$$

donde:

$$L_i(t) = \frac{W_i(t)}{t}, \tag{3.7}$$

$$L_i = \min_{\{0<t\leq T_i\}} L_i(t), \tag{3.8}$$

$$L = \max_{\{1\leq i\leq n\}} L_i, \tag{3.9}$$

$$S_i = \left\{ kT_k | k = 1, \cdots, \left\lceil T_i / T_j \right\rceil ; j = 1, \cdots, i \right\}. \tag{3.10}$$

L_i es el factor de utilización requerido para cumplir con el plazo de una tarea i, $1 \leq i \leq n$, sobre el intervalo [0,t]; W_i es la demanda acumulada en el procesador por el conjunto de tareas $\tau_1, \ldots, \tau_i$, sobre el intervalo [0,t]; S_i es el conjunto de puntos de activación de la tarea i.

Una tarea τ_i es planificable bajo RM si y solo si:

$$L_i = \min_{\{t\in S_i\}} L_i(t). \tag{3.11}$$

Además, un conjunto de n tareas es planificable bajo la política RM si y solo si:

$$L = \max_{\{1\leq i\leq n\}} L_i \leq 1. \tag{3.12}$$

Liu y Layland (1973) presentaron una prueba exacta para EDF, considerando cualquier conjunto de tareas periódicas. Un conjunto de tareas periódicas es planificable bajo EDF si y solo si:

$$U_T \leq 1 \tag{3.13}$$

En resumen, una prueba de planificabilidad asegura que una vez que una tarea es aceptada para ejecución en el sistema, esta completará su ejecución dentro del plazo de respuesta y también, su ejecución no arriesgará la factibilidad de las tareas que previamente había sido garantizada.

3.2.2 Algoritmos de mejor esfuerzo

En ciertas aplicaciones de tiempo real, las actividades computacionales tienen restricciones de tiempo suaves que deben de cumplirse siempre a fin de satisfacer los requerimientos del sistema. Sin embargo, en este caso ningún evento catastrófico ocurrirá si una o más tareas pierden sus plazos de respuesta. La única consecuencia asociada con la perdida de plazos, es la degradación en el desempeño del sistema.

Los algoritmos de planificación de mejor esfuerzo intentan alcanzar los plazos de respuesta, sin embargo no proveen garantía de planificación. Estos algoritmos obtienen un mejor desempeño que los esquemas con garantías. Debido a que las suposiciones pesimistas hechas en la planificación causan el rechazo de tareas, en algoritmos de mejor esfuerzo una tarea es abortada únicamente bajo condiciones reales de sobrecarga.

3.3 Implementación del algoritmo EDF bajo Linux

3.3.1 Trabajos relacionados

La mayor parte de los SOTRs, así como lo es Linux, ofrecen solo dos políticas de planificación: First In First-Out (FIFO) y Round Robin (RR). Y aunque éstas políticas garantizan que los STRs suaves (no-críticos) se ejecuten cumpliendo sus restricciones, no los son suficientes para aquellos STR críticos (Faggioli et al., 2009). Por otra parte, RM y EDF son políticas bien conocidas por soportar STR críticos (Buttazzo, 2011; Stankovic et al., 1998; Davis, 2014; Brun et al., 2015). El algoritmo RM, el cual asigna prioridades inversamente proporcionales al plazo de sus tareas, es una política optima que asigna

prioridades estáticas (Liu y Layland, 1973). A través de la política FIFO, el algoritmo RM es posible emularse sin requerir su implementación. En contraste, EDF asigna la mayor prioridad al trabajo con el plazo más próximo a vencer, lo cual lo hace una política óptima de asignación dinámica de prioridades. (Burns, 1991). Desafortunadamente, la política EDF no se encuentra disponible en la mayoría de los SOTRs.

Existen dos enfoques, mediante los cuales Linux es capaz de proporcionar capacidades de tiempo-real: a nivel hypervisor (RTAI, 2017; XENOMAI, 2017; Yodaiken, 1999; Koh y Choi, 2013) y a nivel del planificador del SO (Rostedt y Hart, 2007; Calandrino et al., 2006; Dellinger et al., 2011). El primero enfoque permite que tanto un SOTR como uno de propósito general coexistan en el mismo entorno. El SOTR tiene una mayor prioridad sobre el otro. En contraste, a nivel planificador, se permite lograr el comportamiento de un STR utilizando diferentes clases de planificación, las cuales tienen integradas políticas de planificación, siendo la de tiempo-real aquella clase que tiene la mayor prioridad ante las demás. Proyectos código abierto así como comerciales usan este tipo de enfoque. Uno de ellos es RT-Preempt (Rostedt y Hart, 2007), el cual se ofrece como parches de código libre que se integran fácilmente al Kernel de Linux. La elección correcta de un SOTR es un aspecto fundamental en el diseño de un STR. En este trabajo de tesis, se integró la política EDF en el Kernel de Linux, empleando el enfoque a nivel planificador y el parche RT-Preempt para lograr las capacidades para un STR. Así mismo, habilitamos el uso de la política integrada a través de la librería GBLIC (GLIBC, 2017). Asegurando de esta manera, la portabilidad de cualquier STR desarrollado haciendo uso la política que se implementó.

La mayor parte de las implementaciones de algoritmos de planificación que se han realizado, han sido para SOTR no-críticos, o definiendo llamadas al sistema propias, lo cual diverge en gran parte del estándar de POSIX (Faggioli et al., 2009; Calandrino et al., 2006; Dellinger et al., 2011; Faggioli et al., 2009a; Baltarejo et al., 2012). A continuación se describe alguno de los trabajos, así como que tipo de solución utilizan para lograr convertir un SO en un SOTR.

Distintos SOs que hacen uso del Kernel de Linux ofrecen soluciones con diferentes enfoques para proporcionar la funcionalidad de un SOTR (Rostedt y Hart, 2007; Calandrino et al., 2006; Dellinger et al., 2011), incluyendo aquellos SOs que no soportan el estándar de POSIX, empleando sus propias APIs (Faggioli et al., 2009; Dellinger et al., 2011; Faggioli et al., 2009a). Varias soluciones de proyectos de mejoras de código abierto

han sido propuestas, permitiendo que Linux ejecute tareas de tiempo-real.

SCHED_DEADLINE (Faggioli et al., 2009; Faggioli et al., 2009a; Lelli et al., 2011) es una propuesta que emplea el enfoque a nivel planificador para implementar el algoritmo EDF; sin embargo este define funciones propias, lo que difiere con el estándar de POSIX. RT-Preempt es un parche del Kernel de Linux que ofrece baja latencia (Rostedt y Hart, 2007). Además de proveer protocolos de sincronización, como lo es el protocolo de herencia de prioridades, esto para evitar la predictibilidad causada por el manejo de recursos compartidos.

El parche de RT-Preempt es compatible con el estándar de POSIX. A su caso, Litmus-RT (Calandrino et al., 2006) es un parche que permite alcanzar las capacidades, mientras de un SOTR no-critico haciendo uso del SO de Linux. Este ofrece una plataforma para la implementación de políticas de planificación y sincronización de tareas bajo arquitecturas multiprocesador, además integra una interface propia de llamadas al sistema, por lo que el API de POSIX no es compatible. Cabe destacar que este parche ofrece diversos algoritmos de planificación implementados. ChronOS (Dellinger et al., 2011) es un parche basado en el ya mencionado parche RT-Preempt, que de igual manera proporciona capacidades de un SOTR crítico. Además, ChronOS incorpora políticas de planificación a través de interfaces para proveer un conjunto de APIs que permiten el desarrollo de STRs.

El simple hecho que los SOTR propuestos por (Faggioli et al., 2009; Calandrino et al., 2006; Dellinger et al., 2011; Faggioli et al., 2009a; Baltarejo et al., 2012) especifiquen interfaces de llamadas al sistema propias para establecer una comunicación entre un SOTR y las aplicaciones, es un obstáculo que evita que los STR sean portables entre SOTRs de manera ágil. Este problema que se presenta en la mayor parte de los proyectos que se han desarrollado es atacado y propuesto una solución para asegurar la compatibilidad con POSIX. Aunque algunos trabajos como los de (Calandrino et al., 2006; Dellinger et al., 2011) permiten dicha compatibilidad, éstos no permiten el empleo de STR críticos.

3.3.2 Integración de la política EDF en el Kernel de Linux

En un SO como lo es Linux, las políticas son implementadas a través de clases dentro del planificador de tareas, entendiéndose a una clase como una estructura que agrupa

funciones y variables que las políticas de planificación hacen uso. En este caso, Linux ofrece dos tipos de clases de planificación como se observa en la Figura 3.2. Una de las clases que se encuentran integradas con el planificador de Linux es "`fair_sched_class`", la cual ofrece dos políticas (`SCHED_NORMAL` y `SCHED_BATCH`). La otra es "`rt_sched_class`" la cual integra dos políticas de prioridad estática (`SCHED_FIFO` y `SCHED_RR`) para aquellas tareas de tiempo-real, tal y como están definidas en el estándar de POSIX (The Open Group Technical Standard Base Specifications, 2008).

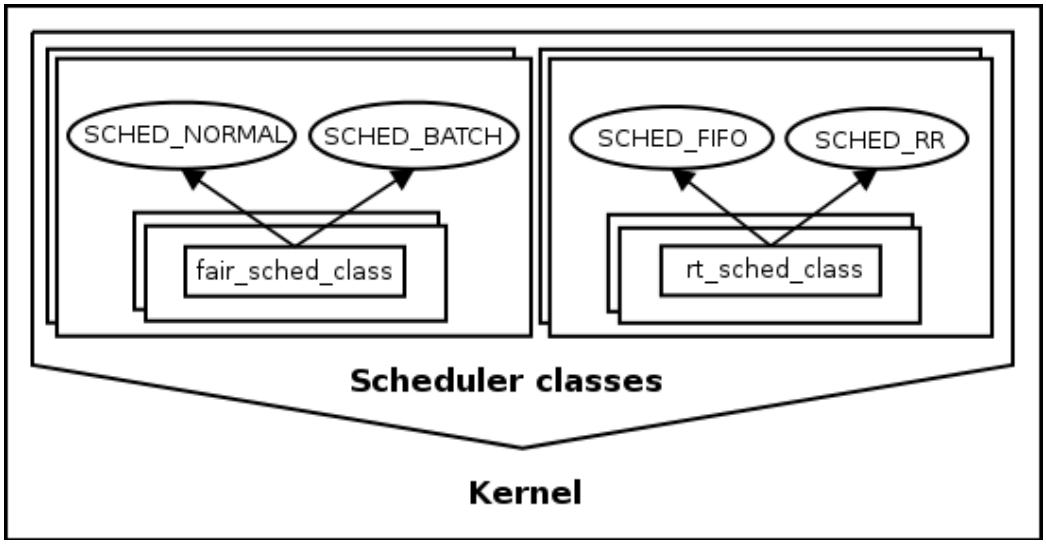

Figura 3.2 Clases definidas en el planificador así como sus respectivas políticas soportadas en el Kernel de Linux.

Además de ofrecer algunas políticas de planificación integradas en las clases, estas mismas mantienen prioridades entre los procesos asignados a estas, siendo de la prioridad 0 a la 99 para aquellas tareas que pertenecen a la clase `rt_sched_class` y de la prioridad 100 a la 139 para las tareas pertenecientes a `fair_sched_class` (Dellinger et al., 2011). Así, las tareas con valor de prioridad bajo son consideras como tareas de tiempo-real en un ambiente Linux.

El planificador de Linux mantiene el control sobre las tareas activas del sistema; temporalmente las almacena en un tipo de estructuras del tipo colas que son creadas por cada nivel de prioridad existente en Linux. El tipo de estructura depende únicamente de la clase. Las tareas activas son administradas de acuerdo a la política asociada y a su nivel

de prioridad. Así, las tareas son agrupadas en una estructura general. Una representación de esta estructura que internamente se observaría en la arquitectura del planificador seria la mostrada en la Figura 3.3.

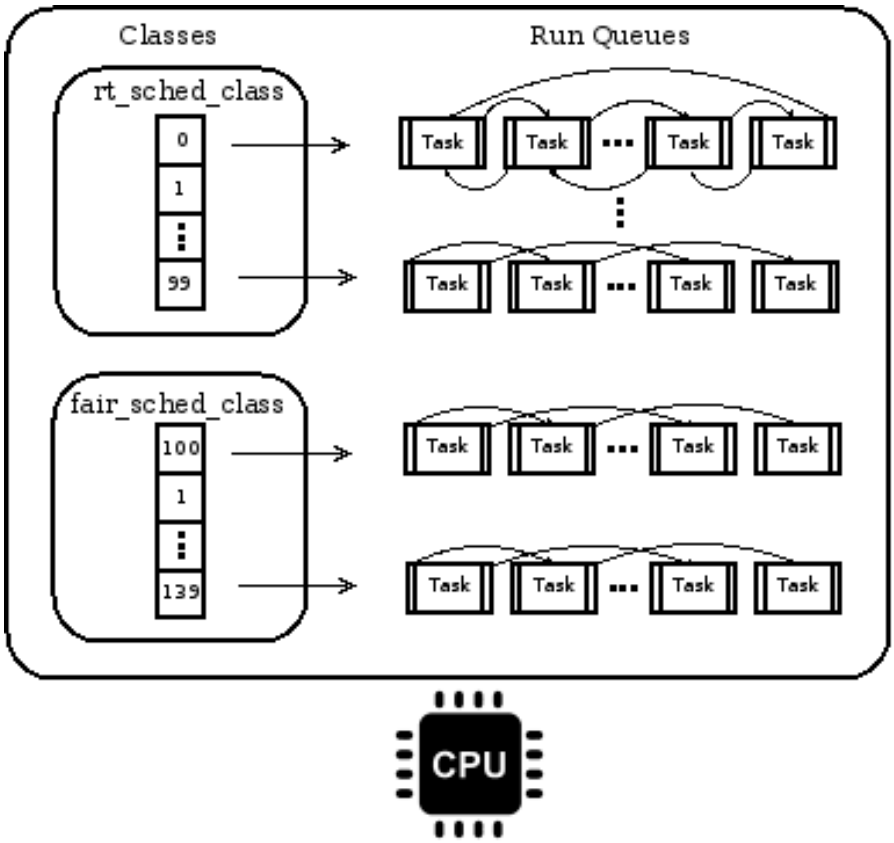

Figura 3.3 Colas de ejecución dentro del planificador de Linux.

A partir de la versión 2.6.23, el planificador de Linux ha sido diseñado como un módulo extensible, permitiendo de esta manera la implementación de distintas políticas alrededor de las clases. Sin embargo, no es por default, no es posible asegurar las restricciones de tiempo de un STR crítico, debido a que su naturaleza como SO es de un SOPG. Es decir, la expulsión de tareas no es permitida del todo completa, lo que llega a generar latencias en sus tiempos de respuesta de las tareas activas.

Los programas de usuario están constantemente restringidos ejecutarse por tiempos largos sin ninguna interrupción. La predictibilidad en el tiempo de respuesta y la latencia en un SO es una característica que debe estar presente para alcanzar una ejecución aproximada a un STR. El lograr que esta característica se cumpla, implica que el Kernel tiene que ser expulsivo en cualquier momento, es decir, si una tarea de mayor prioridad

requiere ser ejecutada, esta deberá ser ejecutada a toda consta para cumplir con el STR.

En los trabajos presentados anteriormente por (Faggioli et al., 2009; Faggioli et al., 2009a; Lelli et al., 2011), la política EDF es implementada haciendo uso de nuevas clases definidas en el planificador. En este trabajo, la propuesta por considerar la clase "`rt_sched_class`" definida en POSIX, por medio del cual la política EDF se integró dentro de dicha clase todo esto para asegurar que el estándar de POSIX seguiría cumpliéndose, ya que de otra forma, el establecimiento de nuevas llamadas al sistema provocaría que la portabilidad no se permitiera. Esto asegura que al integrar la política EDF en la clase de tiempo-real permita el uso de llamadas al sistema ya definidas en el API de Linux, asegurando la creación y la implementación de sistemas compatibles con POSIX. El desarrollar aplicaciones que sean portables entre sistemas, es una característica principal para los desarrolladores de STR. En la Tabla 3.3, se muestran de manera resumida las características principales que los proyectos de trabajos relacionados no llegan a cumplir en su totalidad.

Tabla 3.3 Características soportadas de proyectos de SOTR.

Proyecto	STRs críticos	POSIX	GLIBC	Política a nivel hilos	Referencias
SCHED_DEADLINE	+	-	-	-	Faggioli et al., 2009; Faggioli et al., 2009a; Lelli et al., 2011
Litmus-RT	-	-	+	+	Calandrino et al., 2006
ChronOS	-	+	+	+	Dellinger et al., 2011
Xenomai	+	+	+	-	XENOMAI, 2017; Koh y Choi, 2013
RTAI	+	+	-	-	RTAI, 2017; Koh y Choi, 2013
RT-Linux	+	+	-	-	Yodaiken, 1999
SCHED_EDF	+	+	+	+	

La política EDF es implementada dentro de la clase "`sched_setscheduler`" como se muestra en la Figura 3.4. La implementación no se interfiere con el código ya

establecido dentro de dicha clase del Kernel, lo que permite seguir manteniendo el uso de las capacidades ofrecidas por la clase de tiempo-real.

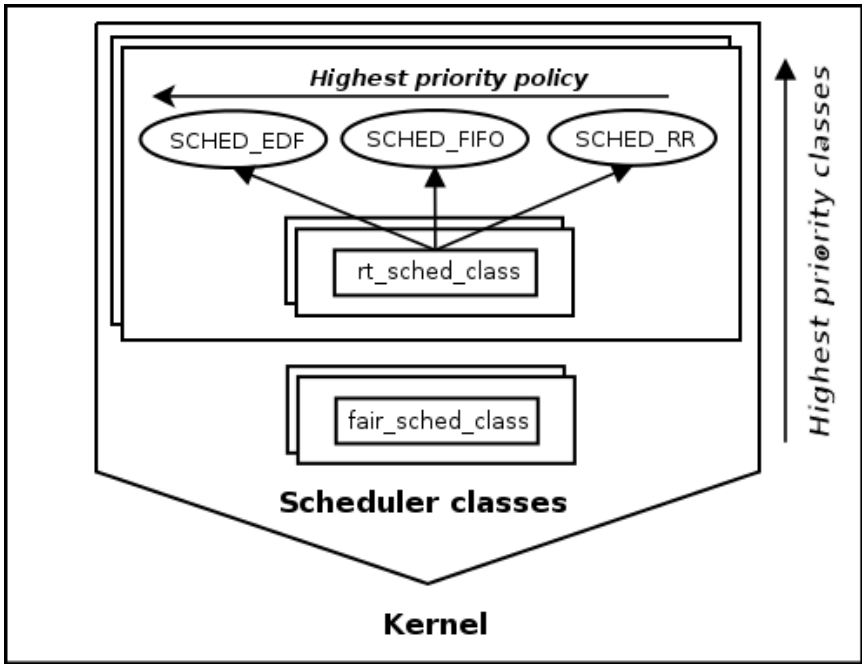

Figura 3.4 Incorporación de la política EDF dentro de la clase "`rt_sched_class`" en el Kernel.

Como se muestra en la Figura 3.4, una clase misma clase mantiene aquellas tareas de tiempo-real. Las tares son planificadas bajo diferentes políticas: `SCHED_FIFO`, `SCHED_RR` o `SCHED_EDF`. La agrupación presentada en dicha figura, garantiza que cualquier tarea asociada a la política EDF sea seleccionada como primer instancia aun si otras tareas de diferentes políticas de la misma clase soliciten el uso del procesador. Por lo que, las tareas asociadas a EDF tendrán una mayor prioridad ante cualquier otra tarea con distinta política.

La integración del algoritmo EDF requiere además dos parámetros indispensables al planificarse un conjunto de tareas bajo dicha política, el plazo relativo y el absoluto. Estos son agregados como atributos a la estructura general llamada `task_struct`, por medio del cual, Linux administra cada tarea y mantiene el control sobre cada tarea activa del sistema.

```
struct task_struct {
```

```
   ...
   struct sched_rt_entity rt;
   #ifdef CONFIG_SCHED_EDF_POLICY
   struct timespec rel_deadline;
   struct timespec abs_deadline;
   #endif
   ...
};
```

Las variables `rel_deadline` y `abs_deadline`, almacenan los valores de los plazos relativos y absolutos respectivamente. Estas se encuentran definidas de tipo `timespec`, el cual acepta valores con una resolución mayor a nanosegundos. Aunque un desarrollador de STRs especifica únicamente el plazo relativo de una tarea, el plazo absoluto es calculado y mapeado internamente en cada activación de una tarea periódica. Estos pasos son completamente trasparentes para el usuario. Así la variable `sched_rel_deadline` se encuentra integrada en la estructura `sched_param` que permitirá a los usuarios definir el plazo de una tarea:

```
struct sched_param {
   int sched_priority;
   #ifdef CONFIG_SCHED_EDF_POLICY
   struct timespec sched_rel_deadline;
   #endif
};
```

Un desarrollador es capaz de planificar un proceso (tarea) utilizando la política `SCHED_EDF` y definir los parámetros del proceso a través de la llamada al sistema `sched_setscheduler`. Esto garantiza que el estándar de POSIX siga siendo compatible.

Para considerar a la política EDF como la más prioritaria ante todas las demás, el planificador de Linux debió ser modificado de tal forma que se incorporar a la clase de tiempo-real esta nueva política, así como actualizar los plazos absolutos en cada instante de planificación de una tarea del tipo EDF. Primeramente, cuando una tarea se activa en el sistema, existe una función llamada `enqueue_task_rt()`, la cual realiza la tarea de inserción en su cola específica en el nivel de prioridad indicado. Otra de las funciones

que se modifico fue `check_preempt_curr_rt()`, la cual es invocada cuando una tarea es activada después de estar en espera y es necesario conocer si aún su prioridad es la misma ante las demás tareas. La función `pick_next_rt_entity()`, selecciona la tarea siguiente que debe ser ejecutada, de acuerdo a las políticas definidas en la clase de tiempo-real, considerándose que si las tareas con política EDF asignadas, tendrán mayor prioridad que las demás. En esta función se verifica la política de planificación asignada a cada tarea.

Usualmente las tareas son inicializadas con ciertos parámetros, como la política y la prioridad, entre otros. Los parámetros de las tareas son modificados en tiempo de ejecución mientras se invocan distintas funciones como `switched_to_rt()` o `prio_changed_rt()`. La mayor parte de los cambios realizados en el Kernel se muestra en la Figura 3.5, como un diagrama de paquetes para su mayor comprensión.

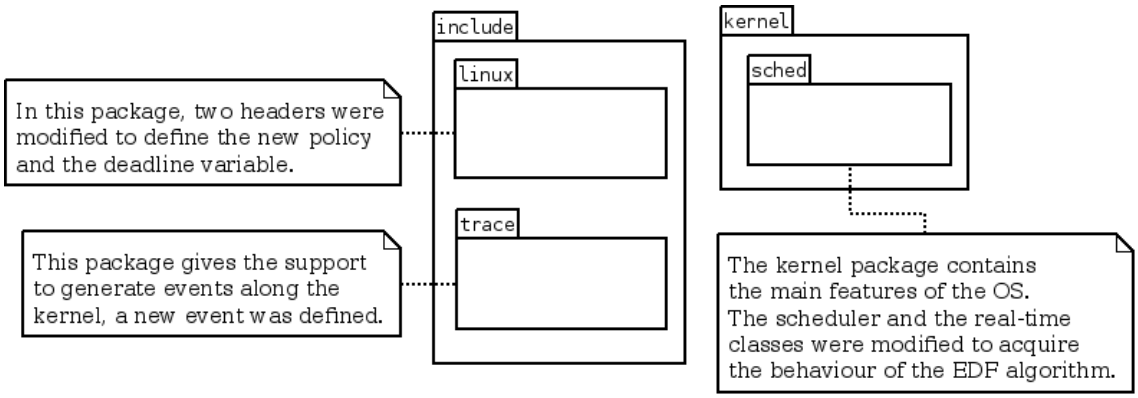

Figura 3.5 Representación de los mayores cambios realizados en el condigo y archivos de cabecera dentro del Kernel de Linux.

3.3.3 Soporte de GLIBC

Las llamadas al sistema se encuentran disponibles a través de un API, el cual permite que se lleva a cabo la comunicación entre los programas de usuario y las funciones propias del SO, de una manera flexible y transparente para el usuario. Esto hace que los desarrolladores se enfocan a la programación de las aplicaciones, sin necesidad de controlar la arquitectura y/o funcionamiento de las funciones del SO. La librería GLIBC (GLIBC, 2017) como se muestra en la Figura 3.6, es uno de estos APIs que permiten realizar dicha comunicación entre los programas y el Kernel, y que a su vez es compatible con el estándar de POSIX bajo ambientes de Linux. Esto hace que sea elegido para

permitir a los programadores utilizar la nueva política presentada (`SCHED_EDF`).

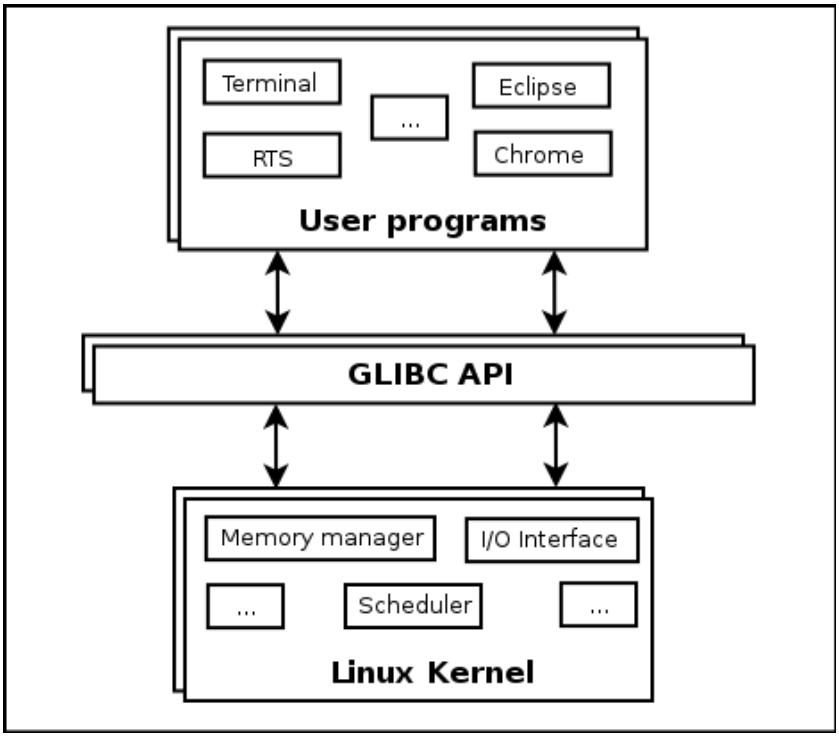

Figura 3.6 Comunicación entre los programas de usuario y el Kernel de Linux a través del API de GLIBC.

Fue necesario añadir algunas secciones de código dentro de la librería para permitir a los programadores de STRs emplear la política `SCHED_EDF` a través de hilos. Sin embargo, la funcionalidad de GLIBC no fue modificada. La Figura 3.7 muestra en forma de diagrama de paquetes los cambios realizados para reconocer la nueva política de planificación `SCHED_EDF` y el nuevo parámetro `sched_rel_deadline`, el cual es integrado en la estructura de `sched_param`:

```
struct  sched_param {
   int  sched_priority;
   struct timespec  sched_rel_deadline;
};
```

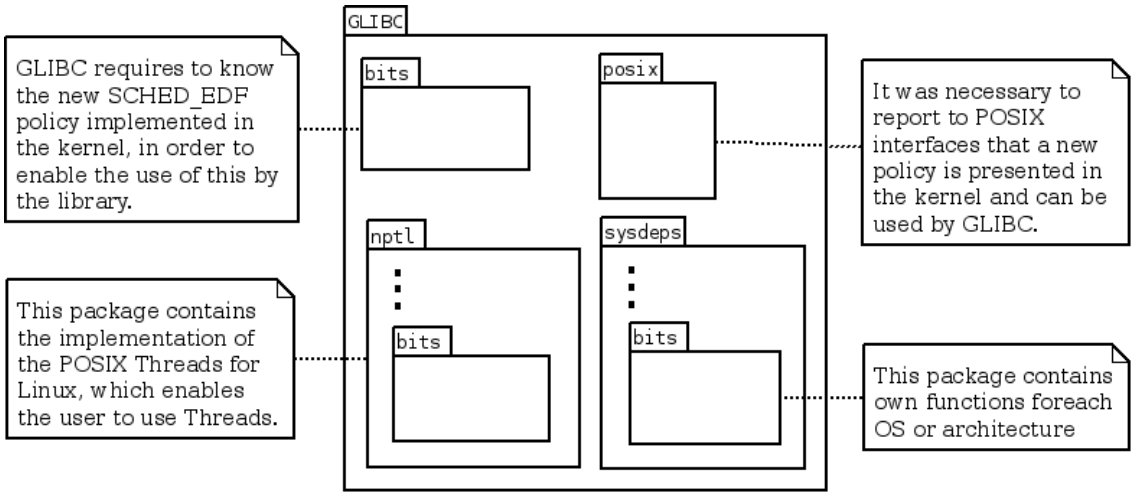

Figura 3.7 Diagrama de paquetes con de las carpetas que contienen el código modificado dentro de la librería GLIBC.

3.3.4 Tracer: FTRACE

La aplicación FTRACE (FTRACE, 2017) es una herramienta que permite generar y almacenar los eventos que ocurren dentro del planificador de tareas de Linux, para posteriormente ser analizados. Esta opción es de gran ayuda observar y verificar el correcto funcionamiento de un STR, todo esto para evaluar el rendimiento, así como para depurar las aplicaciones.

Los datos generados por cada evento que ocurre son almacenados temporalmente en una estructura de datos temporal del tipo *ring buffer*. Para evaluar el rendimiento de la política de planificación, un nuevo evento fue integrado en FTRACE, esto para almacenar información acerca de cuándo una tarea es activada y su plazo absoluto se actualiza. Lo anterior se realiza en una función llamada `trace_sched_edf_abs_deadline`. Su objetivo es guardar el evento en la estructura de datos temporal. Esta función es implementada en FTRACE como un macro, llamados prototipo, de tal manera que su funcionalidad se lleva a cabo en tiempo de ejecución. Una vez almacenado el evento, este es posible exportarse a cualquier formato definido por el usuario.

Así, el prototipo es integrado en la case de tiempo-real, específicamente en aquellas secciones de código donde el plazo absoluto de una tarea se actualiza. El empleo de la herramienta FTRACE resulta en la mayoría de los casos algo complejo, debido a que esta poco documentado, por lo que requiera que el usuario tenga conocimientos de script. Una herramienta llamada TRACE-CMD, facilita el proceso de trazado de eventos haciendo

de intermediario entre el usuario y FTRACE, aligerando de esta manera el problema (FTRACE, 2017).

TRACE-CMD ejecuta los programas de usuario por lo que cualquier evento generado por estos es capturado. La información que se genera es exportada hacia un formato más amigable. En la integración que se llevó a cabo, la información es convertida al formato KIWI. La aplicación de KIWI (KIWI, 2017) permite visualizar de manera gráfica cada uno de los eventos generados por FTRACE. El procedimiento de generación y conversión de datos se aprecia mejor en la Figura 3.8.

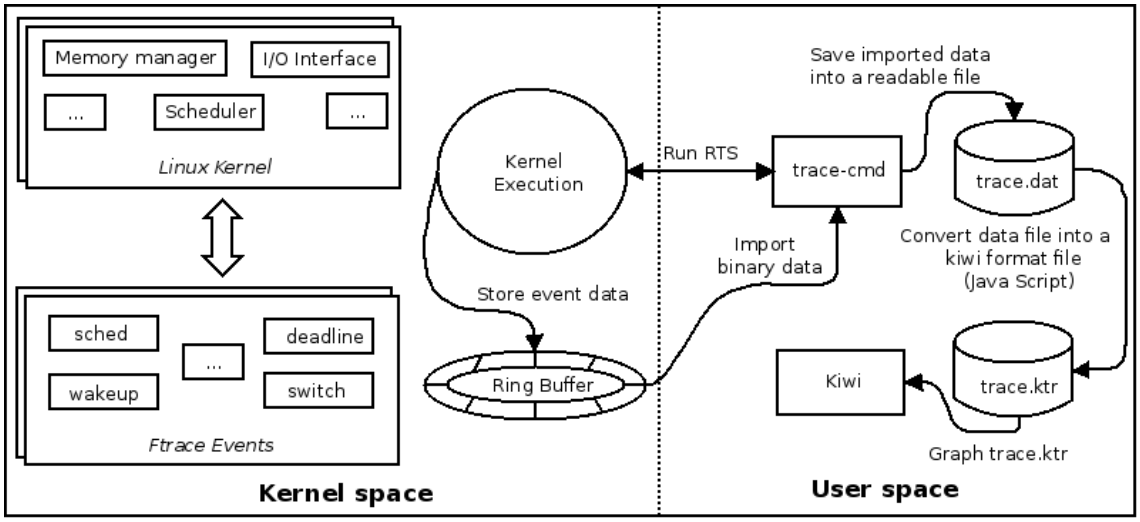

Figura 3.8 Colección de la información para el usuario final.

3.3.5 Evaluación del planificador EDF

Un experimento fue llevado a cabo en una maquina con procesador Intel Pentium IV de 2.8 GHz de velocidad, así como con 1.5 GB de memoria RAM que trabaja a 400 MHz, 20 GB de disco duro a 150 Mbps. Esta máquina solo poseía un solo núcleo para obtención de un comportamiento deseable de un sistema uniprocesador. La prueba de las políticas de EDF y RM se efectuó usando dos aplicaciones desarrolladas con el propósito de simular escenarios realistas. Tres hilos independientes eran creados con diferentes plazos y tiempos de ejecución (computo). Los parámetros por cada uno de los hilos se presentan en la Tabla 3.4, donde el tiempo está dado en milisegundos.

Tabla 3.4 Parámetros del conjunto de tareas.

Hilo	C_i	D_i	T_i
1	290	700	700

2	50	600	600
3	190	400	400

Llevando a cabo el cálculo del factor de utilización total del sistema se obtuvo lo siguiente:

$$U_\tau = \frac{290}{700} + \frac{50}{600} + \frac{190}{400},$$
$$U_\tau = 0.972.$$

La prueba de planificabilidad introducida por Lehoczky et al. (1989) fue empleada para verificar que el conjunto de tareas es planificable utilizando bajo la política RM:

S3 = {400, 600, 700},
W1 = C1 + C2 + C3 = 530 ≤ 400,
W2 = C1 + C2 + 2C3 = 720 ≤ 600,
W3 = C1 + 2C2 + 2C3 = 770 ≤ 700.

La prueba demostró que el hilo 1 no es capaz de ejecutarse completamente antes de que su plazo llegue durante la primera activación. Los hilos 2 y 3 muestran el mismo comportamiento.

En la Figura 3.9 se aprecia que las tareas no son planificables bajo RM. Esto quiere decir que algunos plazos no llegan a cumplirse, debido a que el primer trabajo del hilo 1 pierde su plazo al mantener la prioridad más baja de las tareas. Si los parámetros no se modifican, las tares no serán ejecutadas correctamente bajo ningún otra política con RT-Preempt. Sin embargo, dado que $U_T \leq 1$, el conjunto de tareas es absolutamente planificable utilizando la política EDF.

En la Figura 3.10, se observa la ejecución de las tareas utilizando EDF, donde se muestra la ejecución de dos hiperperiodos completos. Las flechas en dirección hacia arriba representan las activaciones de cada trabajo correspondiente a cada tarea, así como también representan los plazos de cada tarea en cada activación. Esto permite observar lo siguiente:

- Todas las tareas cumplen con sus plazos respectivos.
- Tareas que no son de tiempo-real (agrupadas como Idle) son procesadas solo cuando tareas de tiempo-real no requieren ser ejecutadas.

- El planificador es similar en ambos hiperperiodos, y es posible notarlo en la figura a través de una línea roja vertical.

En la Tabla 3.5, se presentan algunas métricas que fueron recabadas durante el primer hiperperiodo. Se demuestra que el tiempo requerido en microsegundos para realizar un cambio de contexto es mayor con respecto a RM. Sin embargo, se presentó el doble de instantes de expulsión utilizando la política RM. Además, cada vez que una tarea es activada, EDF usa una fracción de tiempo mayor que RM. Sin embargo, el número de expulsiones dependen del algoritmo empleado para planificar las tareas. Esto está demostrado por Buttazzo (2011) que el RM genera más instantes de expulsión que el EDF. Así pues, aun si SCHED_FIFO requiere menos tiempo para llevar a cabo los cambios de contexto, los tiempos de SCHED_EDF son minimizados durante la ejecución debido a que este no produce tantos puntos de expulsión.

Tabla 3.5 Tiempos de respuesta de las tareas de tiempo-real bajo las políticas RM y EDF.

Políticas	**Cambios de contexto**	**Activación de una tarea**	**Puntos de expulsión**
SCHED_FIFO (RM)	11.077	2.310	14
SCHED_EDF	12.344	3.364	7

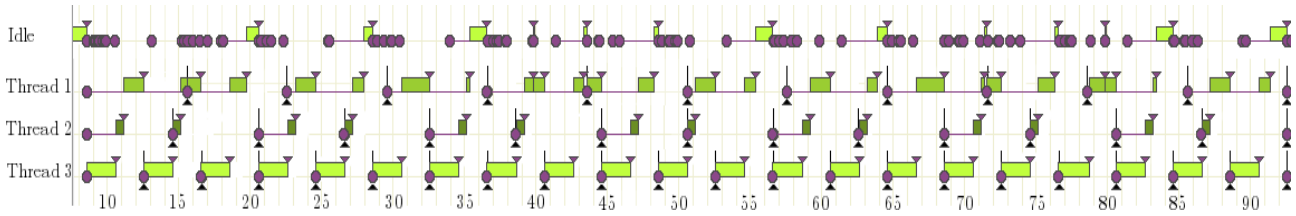

Figura 3.9 Conjunto de tareas planificadas utilizando RM.

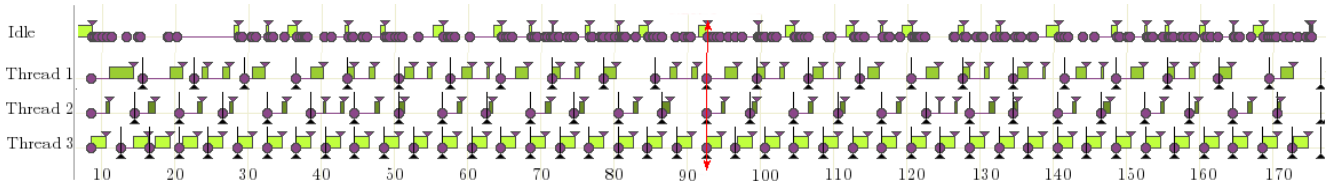

Figura 3.10 Conjunto de tareas planificado utilizando EDF.

3.4 Conclusiones del capítulo

La integración de la política EDF dentro de la case de tiempo-real para STRs uniprocesador es un aspecto que se resalta, de tal manera que la política EDF hace posible una mayor utilización del procesador que con RM, permitiendo que STRs que no son planificables bajo las políticas existentes, lo sean haciendo uso de esta implementación. En el experimento que se llevó a cabo, EDF demuestra un mejor rendimiento que RM. Se incluyó la generación de trazas dentro de la clase del planificador para una fácil depuración de aplicaciones y de este modo proveer compatibilidad con la herramienta de FTRACE.

Otra contribución en esta implementación es integrar una nueva política para los desarrolladores de STRs sin definir una nueva clase en el Kernel, adquiriendo las características que proporciona la clase de tiempo-real así como sus beneficios, con el fin que cumpliera con el estándar de POSIX Thread para alcanzar la portabilidad de las aplicaciones. La meta de este trabajo se logró habilitando la política `SCHED_EDF` a través de la librería GLIBC. Un estado del arte demostró que no sólo proyectos ya presentados previamente cumplieran con las características de un STRs crítico, sino que además es compatible con el estándar de POSIX sin adaptaciones o creación de nuevas interfaces al hacer uso de hilos.

Capítulo 4

Algoritmos de planificación en sistemas multiprocesador

Dado el auge de dispositivos con arquitecturas multiprocesador o multinúcleo, el desarrollo de aplicaciones sobre estos es cada vez mayor, ya que se vuelve casi inevitable hoy en día desarrollar aplicaciones donde solo actúen sobre una única arquitectura. Debido a esto, es de gran interés, encontrar nuevas técnicas de planificación que permitan a STRs llevarse a cabo utilizando de manera eficiente los recursos.

4.1 Clasificación de algoritmos de planificación en sistemas multiprocesadores

Los sistemas multiprocesadores suelen clasificar en 3 categorías desde la perspectiva de la planificación:

1. *Homogéneos*: los procesadores son idénticos, por lo que la velocidad de ejecución de todas las tareas es igual en cualquier procesador.
2. *Uniformes*: la velocidad de ejecución de una tarea depende sólo de la velocidad del procesador.
3. *Heterogéneos*: los procesadores son diferentes, por lo que la velocidad de ejecución de una tarea depende tanto del procesador como de la tarea, además hay restricciones en la asignación de las tareas a los procesadores.

Así mismo las tareas se dividen en dos modelos básicos: tareas periódicas y las esporádicas. En ambos modelos, las tareas se hacen referencia en una secuencia infinita llamadas activaciones (*jobs*) (Carpenter et al., 2004). En el modelo de tareas periódicas, el arribo de los jobs de una tarea es estrictamente periódico, separado por un intervalo de tiempo fijo, llamado *período* (Davis y Burns, 2011). En el modelo de tareas esporádicas cada job de una tarea puede arribar en cualquier momento una vez que haya transcurrido un intervalo mínimo de tiempo desde que un job anterior de la misma tarea haya ocurrido.

Los algoritmos de planificación o bien políticas de planificación también, son clasificados por los cambios de prioridad, así como por la manera en la que se asignan las tareas a los procesadores (Carpenter et al., 2004).

- *Asignación*

    - No migración: cada tarea es asignada a un solo procesador y no se permite migrar a cualquier otro procesador.
    - Migración a nivel de tarea: los jobs de una tarea pueden ser ejecutados en diferentes procesadores, desde luego cada job solamente se ejecuta en un solo procesador.
    - Migración a nivel de job: un simple job puede migrar y ejecutarse en diferentes procesadores; desde luego la ejecución en paralelo no es permitida.
- *Prioridad*
    - Prioridad fija de la tarea: cada tarea posee una prioridad fija, y de igual manera sus jobs poseen la misma prioridad.
    - Prioridad fija del job: cada job de una tarea puede poseer diferente prioridad, pero estas prioridades son estáticas. Un ejemplo de estos algoritmos es el EDF.
    - Prioridad dinámica: una misma tarea posee diferentes prioridades durante el transcurso del tiempo.

Así mismo, los algoritmos de planificación, donde la migración no es permitida, son llamados particionados (Baruah, 2007). Aquellos, donde la migración es permitida, son llamados globales (Baruah y Fisher, 2006), la Figura 4.1 ofrece un mejor panorama sobre ésta clasificación. La mayor parte de las investigaciones se han basado sobre algoritmos globales de planificación. Para que una política de planificación sea considerada capaz de planificar cualquier conjunto de tareas bajo un STR, debe pasar por ciertos análisis teóricos, además de proporcionar métricas de rendimiento lo suficientemente optimistas. Los algoritmos de planificación que hoy en día se conocen para STRs bajo sistemas multiprocesador, ofrecen un bajo rendimiento con respecto al uso de la capacidad de procesamiento, desperdiciando así mucho del potencial que los sistemas de cómputo ofrecen.

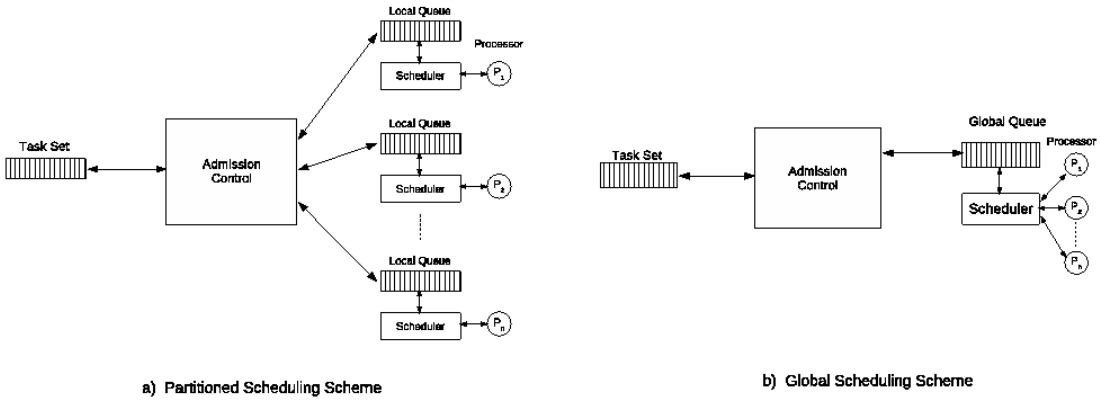

Figura 4.1 Clasificación de los algoritmos de planificación bajo sistemas multiprocesador.

Dichos algoritmos son clasificados tal como se observa en la Tabla 4.1, donde los algoritmos que se encuentran en el área de la práctica son del segundo tipo, puesto que los algoritmos del primer tipo también conocidos como *pfair*, son teóricamente posibles, pero al llevarse a la práctica tienen una sobrecarga enorme por lo que no son factibles (Carpenter et al., 2004).

Tabla 4.1 Clasificación de los algoritmos de planificación bajo sistemas multiprocesador.

Tipo	**Máximo factor de utilización**
Global (migración a nivel de las activaciones) Prioridad dinámica	m
Otros tipos	$(m+1)/2$

Como se ha visto en sección, el estudio de la planificación de tareas de tiempo-real bajo sistemas multiprocesador conlleva a varios puntos de vista que se deben tomar en cuenta y que implican distintos problemas, para los cuales no en muchos se han logrado buenos resultados, por lo que se ve necesario e interesante lograr tanto teóricamente como prácticamente la solución u optimización de problemas aún abiertos para su investigación.

4.2 Modelo genérico de sistemas

El primer modelo de tareas fue propuesto por (Liu y Layland 1973) en un estudio de algoritmos para STRs críticos monoprocesador. Es conocido como el modelo de tareas

periódicas y considera las siguientes restricciones: las tareas son periódicas e independientes, los plazos son iguales a los periodos, las tareas pueden ser expulsadas del procesador en cualquier instante y ninguna tarea se suspende voluntariamente. Si alguna tarea hace uso de recursos compartidos, debe considerarse el tiempo máximo que el recurso se mantiene ocupado.

Nuevos retos en los avanzados STRs, han requerido de modelos que permitan representar toda variedad de arquitecturas avanzadas. Uno de los modelos iniciales propuestos para sistemas multiprocesador, desarrollado por Lipari (1998), considera tareas periódicas, así como independientes, y multiprocesadores homogéneos. Para aquel tiempo, el uso de STRs con tareas independientes era algo tan básico, que generalmente no cumplía con los requerimientos de la industria.

En la Figura 4.1 se presenta un entorno genérico de un STR multiprocesador con recursos compartidos, que sistematiza restricciones de modelación que se encuentran en sistemas actuales. Este entorno es utilizado también para atribuir los algoritmos a evaluar. Los tres esquemas de asignación de tareas se consideran. El esquema global asigna a cada tarea en un procesador disponible cada instante de planificación. En el esquema particionado cada procesador mantiene una cola de tareas por ejecutar, no se permite que las tareas migren hacia otras colas. El esquema hibrido permite agrupar procesadores, asignando tareas de tal forma que estas puedan migrar en el grupo establecido.

En base al tipo de algoritmo, el planificador adquiere un comportamiento expulsivo o no-expulsivo. Para cualquier esquema, las tareas en un STR se comportan en distintas formas, como periódicas, esporádicas o aperiódicas, con prioridad estática a nivel de tarea o dinámica sea a nivel de tarea o a nivel de trabajo. Así, los plazos de las tareas están definidos como implícitos, arbitrarios o limitados.

En STRs con recursos compartidos, el acceso a cada uno de éstos sólo es posible mediante dos formas: sin bloqueo o con bloqueo de las tareas. De esta última se desprenden dos más, sin espera o con espera del recurso si está siendo utilizado. Por otro lado, el acceso a dos o más recursos en una misma sección crítica se encuentra ligado al desarrollo del sistema.

La evaluación de algoritmos en el presente trabajo, se basa en la Figura 4.2, considerando las siguientes suposiciones:

- Un esquema de planificación es fija en cada algoritmo.

- Las tareas son periódicas.
- Las prioridades son dinámicas a nivel tarea o a nivel trabajo.
- Los plazos son implícitos.
- El acceso a los recursos es con bloqueo sin espera.
- El tipo de protección es de recursos no-anidados.

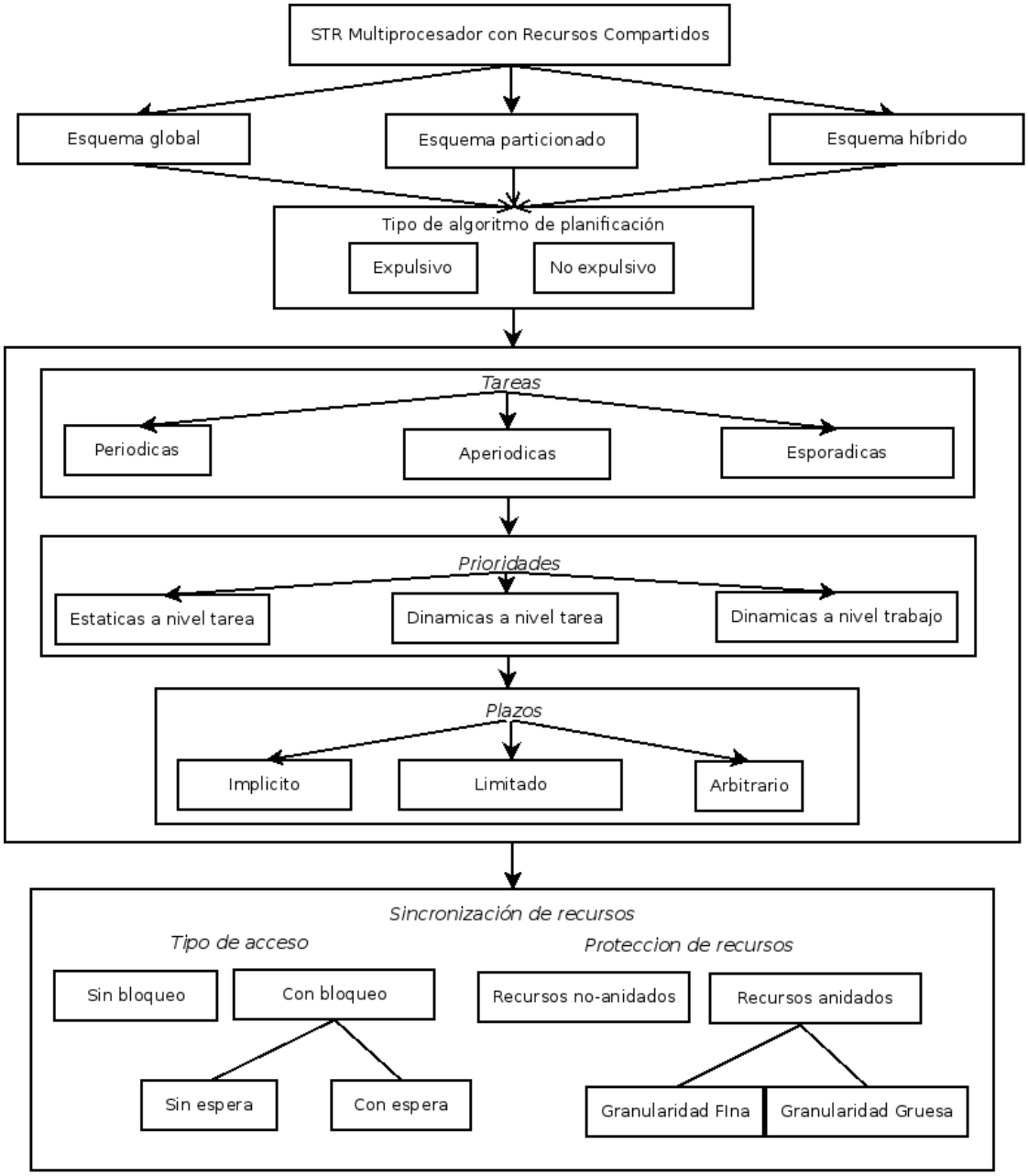

Figura 4.2 Entorno genérico de un STR multiprocesador con recursos compartidos.

4.3 Esquemas de planificación

Existen tres estrategias para la planificación de tareas en STRs con multiprocesadores. En un esquema global (llamado también no-particionado) cada tarea de tiempo-real se ejecuta en un procesador diferente en cada activación o reanudación. En contraste, en un esquema particionado, todas las instancias de una tarea son ejecutadas en un sólo procesador. Por otro lado, existe el esquema hibrido (semi-particionado), el cual permite a las tareas migrar entre procesadores de acuerdo a un arreglo establecido. El esquema particionado tiene algunas ventajas sobre el esquema global. En primer lugar, es menos complejo ya que 1) no introduce sobrecarga al planificador y 2) las tareas son asignadas al procesador una sola vez al principio de la ejecución. En segundo lugar, un algoritmo para un sistema uniprocesador con prioridad fija, como por ejemplo RM, es aplicable bajo un esquema particionado debido a que las tareas no migran entre procesadores. Sin embargo, el problema de encontrar una asignación óptima en un sistema multiprocesador es NP-hard estricto (Leung y Whitehead, 1982).

El esquema particionado es en donde se ha concentrado la mayor investigación, principalmente porque es más fácil su implementación para garantizar la planificabilidad en tiempo de ejecución (Burke, 2016). Muchos algoritmos heurísticos para éste método se han propuesto. Por ejemplo, Liu y Layland (1973), presentaron dos métodos de asignación, el Rate Monotonic Next-Fit (RMNF) y el Rate Monotonic First-Fit (RMFF). Burchard et al. (1995) propusieron otros dos algoritmos: el Rate Monotonic Small Tasks (RMST), el cual particiona cada tarea en un número pequeño de sub-tareas. Otro algoritmo es el Rate Monotonic General Tasks (RMGT), el cual primero particiona todas las tareas periódicas dentro de dos sub-conjuntos acorde a sus utilizaciones. Las tareas, cuya utilización sea igual o menor a 1/3, se agrupan en un sub-conjunto. Esas tareas son primero asignadas a los procesadores acorde al algoritmo RMST. Después, las tareas más largas, cuya utilización sea más grande de 1/3, son asignadas por First-Fit (FF) a los procesadores, en los cuales se tiene al menos una tarea asignada por el algoritmo RMST.

El esquema no-particionado no ha recibido tanta atención, debido a varias limitaciones. Así, actualmente no existe una prueba eficiente de planificabilidad bajo este esquema. Se conoce una prueba necesaria y suficiente de planificabilidad con una complejidad de tiempo exponencial (Stankovic et al., 1998). La complejidad fue reducida a polinomial con una prueba de planificabilidad suficiente (Calandrino et al., 2006; Dellinger et al., 2011) y con una complejidad de tiempo pseudo-polinomial. Pero en

(Ramesh y Ramachandraiah, 2015), la prueba llega a ser pesimista cuando el número de las tareas crece y en (Bastoni et al. 2010; Ismail et al., 2015) llega a ser pesimista cuando el número de procesadores se incrementa. Segundo, no se ha encontrado en literatura un esquema óptimo para la asignación de prioridades.

Un esquema hibrido de planificación permite migrar a las tareas hacia los procesadores del sistema pero de una manera restringida. Varios autores como (Burke, 2015; Ismail et al., 2015), hacen mención a esta restricción como una agrupación tipo clúster, donde se asemeja a una estructura compuesta por computadoras que forman parte de un servidor de alto rendimiento.

En la Tabla 4.2 se presenta una clasificación de algoritmos de planificación de los cuales algunos de estos son considerados para la comparación. Existen diversos criterios para la comparación de algoritmos, como la utilización de la CPU, tasa de procesamiento, tasa de tareas con alta demanda, tiempo de ejecución, tiempo de espera, tiempo de respuesta. Las evaluaciones realizadas en (Calandrino et al., 2006; Ramesh y Ramachandraiah, 2015; Bastoni et al., 2010), donde se compraran algoritmos pertenecientes a un mismo esquema, consideran estos criterios de acuerdo a su tasa de: planificabilidad, cambios de contexto realizado por cada algoritmo, adquisición de recursos, recursos anidados soportados y sobrecarga acumulativa.

Tabla 4.2 Clasificación de algoritmos de planificación.

Particionado - Expulsable			Semi - Particionado Expulsable			Global - Expulsable			
TLS	JLS	JLD	TLS	JLS	JLD	TLS	JLS	JLD	TLS
RM-NF RM-FF	EDF-BF EDF-WF			RNLP OMIP		RM	RNLP OMIP		
RM-BF	RBOUND-MP					fpEDF	adaptive TkC		
RM-WF	EDF-NF		M-BROE	M-BROE		M-BROE	M-BROE		
FMLP FMLP+ MPCP DPCP	FLMP FMLP+ DPCP		FMLP+	FMLP+	FMLP+	FLMP FMLP+ MPCP MSRP	FLMP FMLP+ DPCP MSRP	MSRP	
P-CP STM-HRT	P-CP STM-HRT						edf-US[2m-1]		P-CP
	EDF-FF								

4.4 Evaluación de algoritmos de planificación bajo sistemas multiprocesador

Para llevar a cabo la evaluación de los algoritmos considerados dentro del estado del arte de sistemas multiprocesador, se estableció utilizar una muestra de *S* conjuntos de tareas utilizando diferentes parámetros. Se define la razón de planificabilidad de un algoritmo multiprocesador *A* para una muestra *S*, como la proporción del número de tareas planificables por el algoritmo *A* y el número total de conjuntos de tareas:

$$\rho s(A) = \frac{numero\ de\ tareas\ aceptadas\ por\ A}{numero\ total\ de\ conjuntos\ de\ tareas} * 100 \qquad (4.1)$$

Con el fin de obtener resultados que nos permitan evaluar el rendimiento de los algoritmos de planificación bajo distintas características de los conjuntos de tareas, se llevó a cabo experimentos utilizando diferentes formas de generación de tareas en *S*.

4.4.1 Rendimiento como función de la utilización del sistema multiprocesador

El propósito de este experimento es evaluar el rendimiento de los algoritmos de planificación en función de su factor de utilización del sistema. Se consideró un sistema multiprocesador conformado por ocho procesadores, donde se generó varias muestras de conjuntos de tareas identificado como S_{U_S}, donde U_S es el factor de utilización del sistema con valores de factor de utilización de entre [0.3, 1]. Así cada muestra S_{U_S} está conformada por 1000 conjuntos de *n* tareas. El número de tareas se encuentran en el rango de entre [9, 40]. Los periodos de las tareas se encuentran uniformemente distribuidas en el rango de entre [100, 500]. Se llevó a cabo dos experimentos. El primero de ellos considero solo tareas con poca demanda del procesador, donde la utilización máxima de cada tarea se encuentra denotada por $\propto$, siguiendo una distribución uniforme en el rango de [0.01, 0.3]. El segundo experimento incluyo tareas con alta y baja demanda de procesador, donde $\propto$ mantiene una distribución uniforme en el rango de [0.01, 0.7]. Los tiempos de ejecución de cada tarea son generados con valores de $1 \leq C_i \leq \propto T_i$.

4.4.2 Algoritmos Particionados

Como se ha discutido previamente, el esquema particionado se encuentra definido como la estrategia para asignar tareas hacia un procesador, y por una condición de

planificabilidad, el cual es utilizado para verificar si el conjunto de tareas asignado a un procesador es planificable, de acuerdo a una política de planificación (ej. RM) Debido a las diferentes entre los enfoques utilizados por las condiciones de planificabilidad que se han propuesto para RM, es posible clasificar estas en condiciones basadas en la utilización del sistema (*LL*, *OP*, y *UO*), y condiciones basadas en su periodo. (*PO*, *R-Bound*). A continuación se analiza el rendimiento de RM bajo el esquema particionado utilizando esta clasificación. Ya que las condiciones basadas en los periodos utilizan más información acerca del conjunto de tareas (por ejemplo, los valores de los periodos) para establecer sus cotas de planificabilidad, esperamos que los algoritmos basados en este tipo de condiciones ofrezcan un mejor rendimiento.

En las Figuras 4.3, 4.4 y 4.5 se observa la tasa de tareas aceptadas por los algoritmos RM-NF, RM-FF, RM-BF, y RM-WF utilizando las condiciones de planificabilidad (*LL*, *IP* y *UO*), donde $\propto = 0.3$. Para las condiciones *IP* y *UO* se ha se ha incluido a los algoritmos RMNF-DU, RMFF-DU, RMBF-DU, y RMWF-DU, los cuales ordenan a las tareas de acuerdo a su factor de utilización no-creciente, antes de asignar a estas a algún procesador disponible. Se observa que todos los algoritmos mantienen un porcentaje de tareas planificables igual a 100 para aquellas donde $U_s < 0.6$.

En la Figura 4.3 se muestra el porcentaje de aceptación de los algoritmos que utilizaron la condición de planificabilidad de *LL*. Para $0.6 \leq U_s < 0.75$, el porcentaje de tareas planificables mantiene la relación $\rho_U(RM - BF) > \rho_U(RM - FF) > \rho_U(RM - WF) > \rho_U(RM - NF)$. La aceptación de taras para estos algoritmos es cero para $U_s \geq 0.75$.

En la Figura 4.4 se muestra el porcentaje de aceptación de los algoritmos que utilizan la condición *IP*. Donde los algoritmos RMFF-DU, RMBF-DU, y RMWF-DU presentan una aceptación del 100 por ciento para $0.6 \leq U_s < 0.75$. Mientras que el porcentaje de tareas aceptadas de los demás algoritmos decrece rápidamente. Sin embargo, ningún algoritmo con los que se experimenta es capaz de planificar cualquier conjunto de tareas cuando $U_s \geq 0.75$.

En la Figura 4.5 se observa el comportamiento de aceptación de los algoritmos utilizando la condición *UO*. Aquí es posible observar que RMBF-DU y RMWF-DU son algoritmos con un mejor rendimiento cuando $0.6 \leq U_s < 0.9$. Claro está que estos algoritmos tiene un porcentaje de aceptación del 100 para $U_s \leq 0.8$, pero cuando $U_s \geq$

0.9 el número de conjuntos de tareas planificables es cero.

Una comparación de aquellos algoritmos que mostraron un mejor rendimiento en las Figuras 4.3, 4.4, y 4.5, los cuales emplean condiciones basadas en la utilización, se muestra en la Figura 4.6. Aquí se observa que aquellos algoritmos que utilizan la política de utilización decreciente (*DU*) tiene un mayor porcentaje de aceptación. RMBF-DU y RMWF-DU registraron un mejor rendimiento sobre los demás algoritmos para $0.75 \leq U_s < 0.9$.

En la Figura 4.7 se observa el porcentaje de aceptación de aquellos algoritmos basados en una condición basada en sus periodos, donde se incluye además a RM-NF-LL y RMBF-DU-UO. Se aprecia que con excepción de RM-NF-LL, todos los algoritmos muestran un porcentaje de aceptación similar, que en la mayoría de los casos es igual a 100 por ciento. Sin embargo, los algoritmos RM-ST y RM-GT muestran un ligero mejor rendimiento sobre los demás para $U_s < 0.9$.

La Figura 4.8 muestra el rendimiento obtenido al experimentarse con los algoritmos basados en EDF. Se observa que todos ellos tienen una aceptación del 100 por ciento para $U_s \leq 0.8$, y que para los algoritmos EDF-BF-DU y EDF-WF-DU llega a ser del 100 por ciento para $U_s \leq 0.95$.

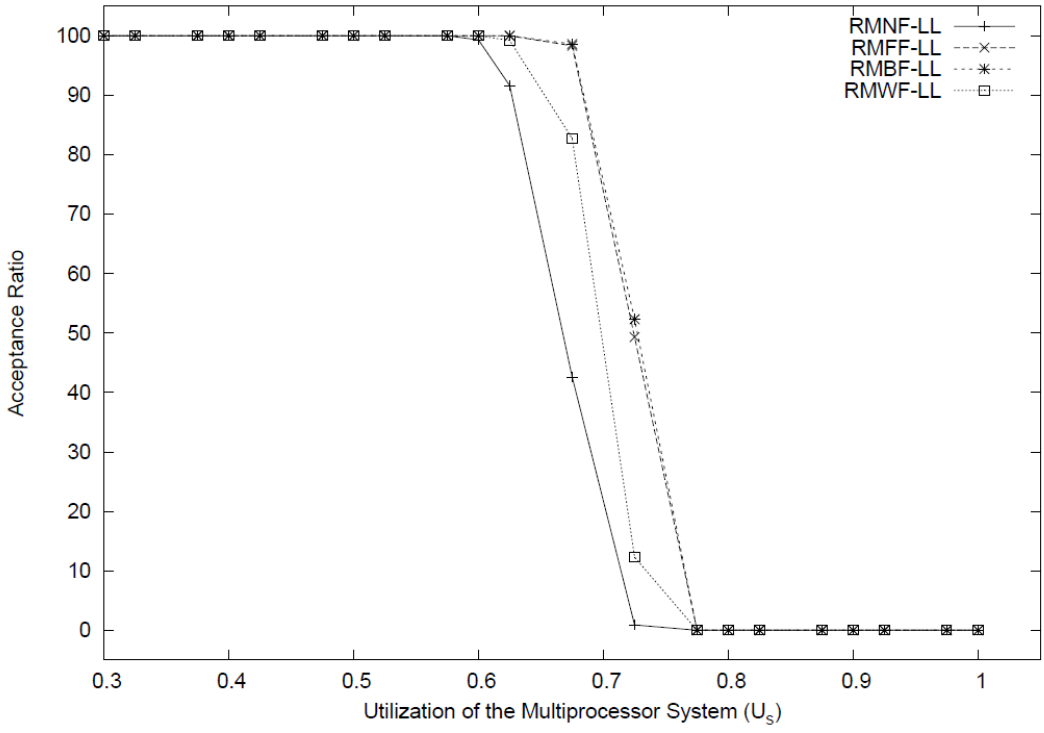

Figura 4.3 Rendimiento en función de U_s utilizando $\propto = 0.3$ y la condición *LL*.

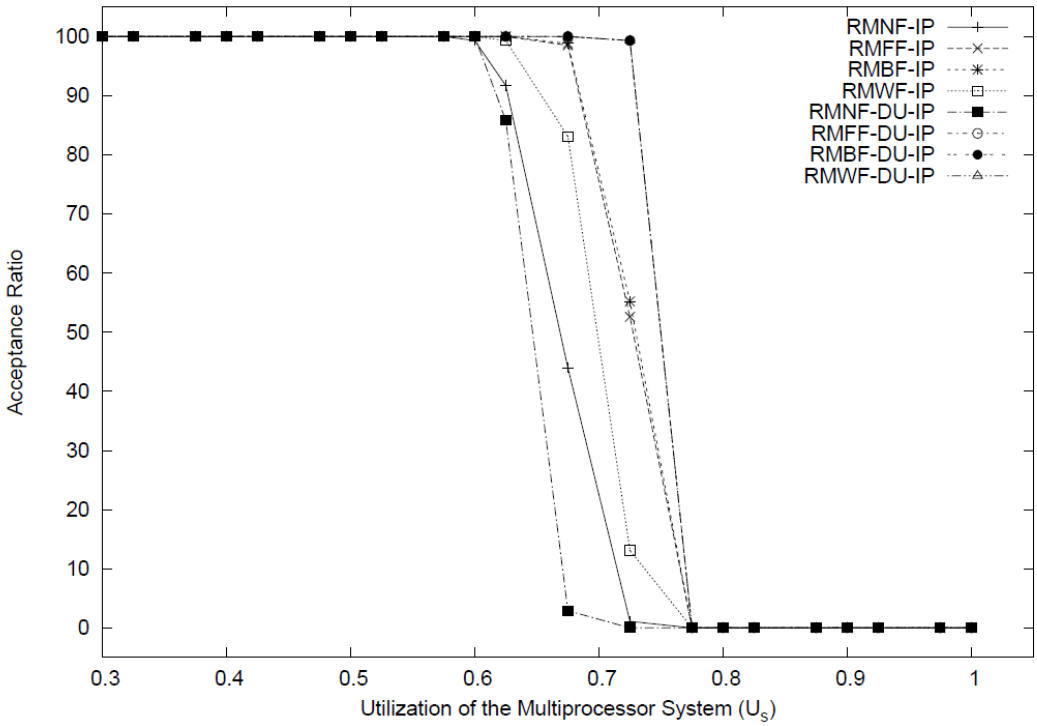

Figura 4.4 Rendimiento en función de U_s utilizando $\propto = 0.3$ y la condición *IP*.

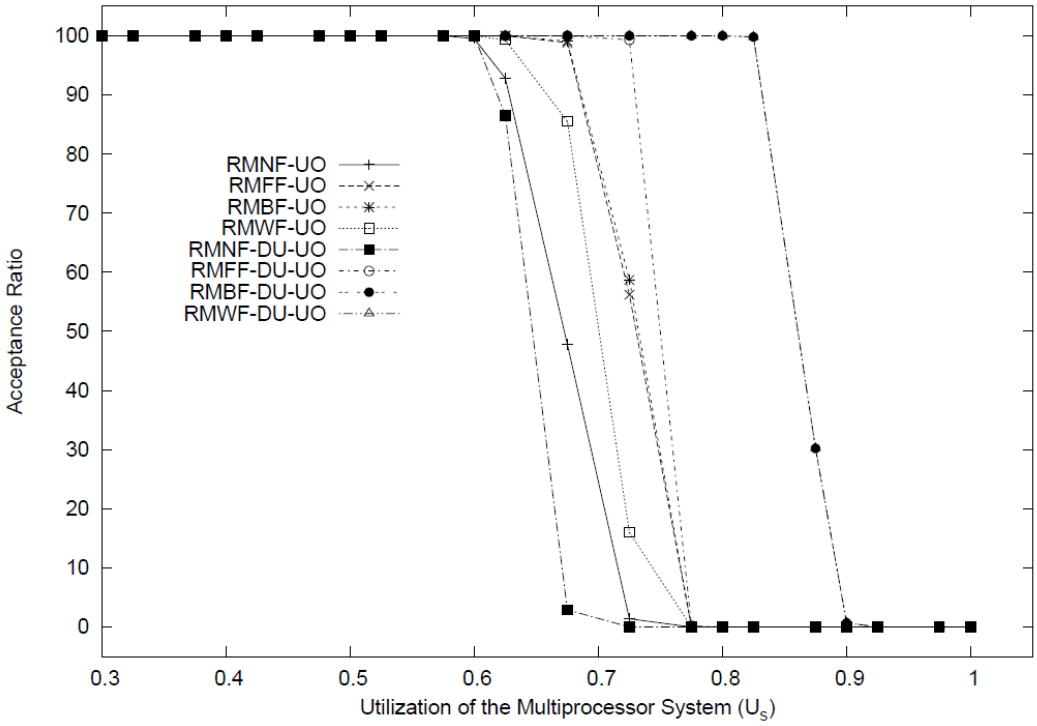

Figura 4.5 Rendimiento en función de U_s utilizando $\propto = 0.3$ y la condición *UO*.

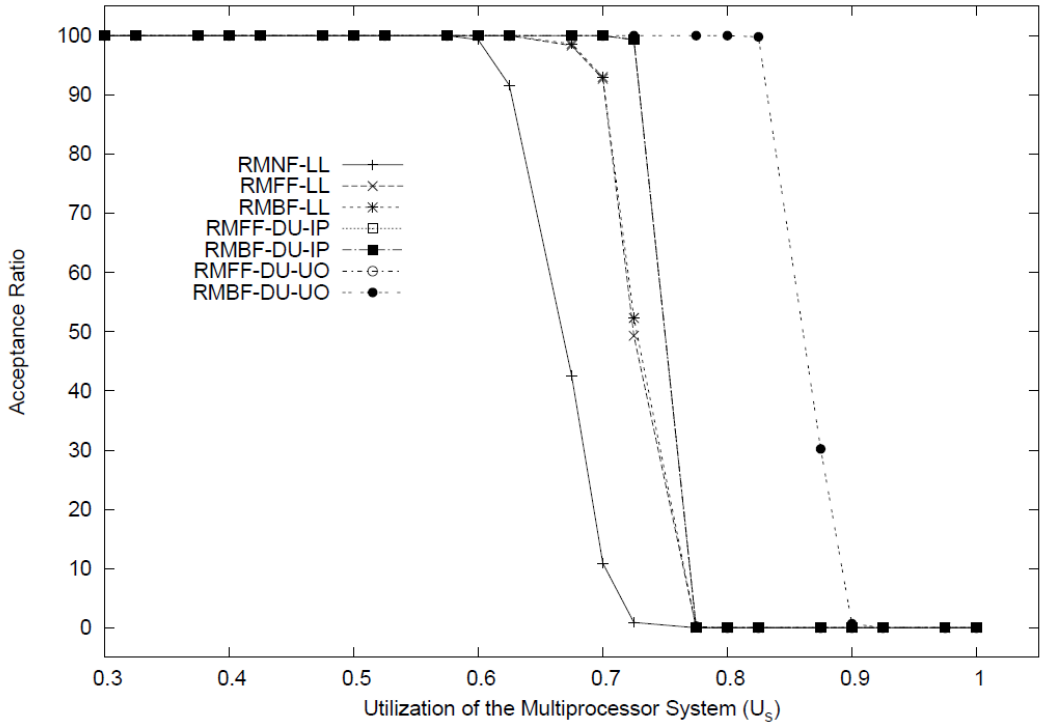

Figura 4.6 Rendimiento en función de U_s utilizando $\propto = 0.3$ y las condiciones *LL*, *IP* y *UO*.

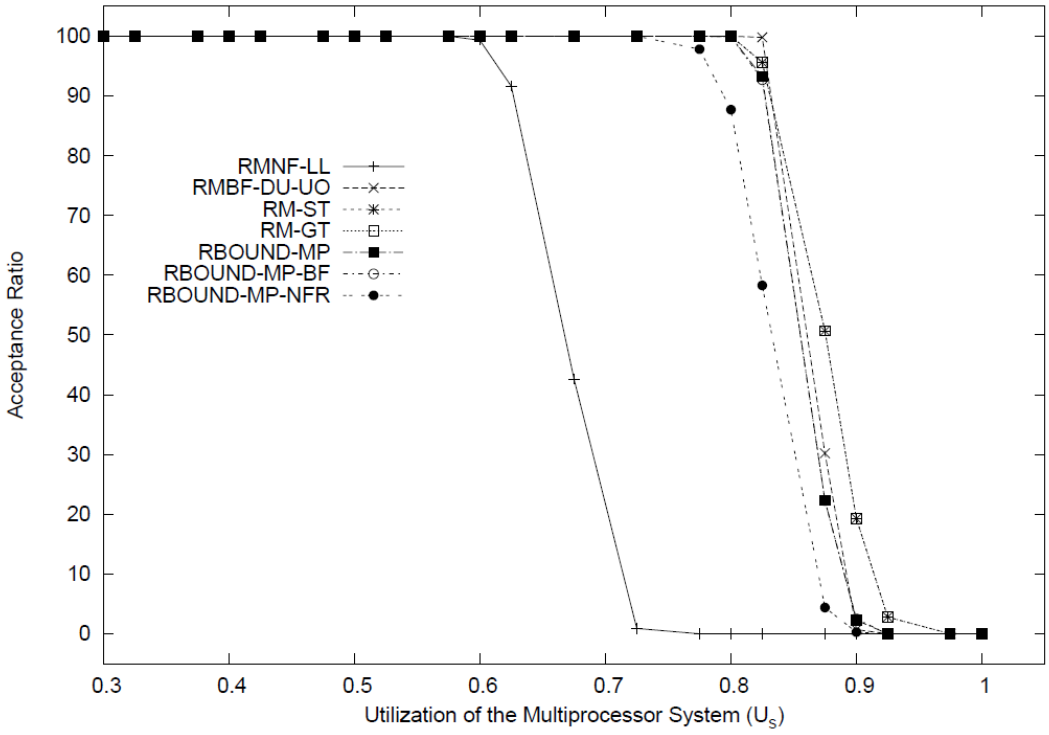

Figura 4.7 Rendimiento en función de U_s, utilizando $\propto = 0.3$, de aquellos algoritmos basados en *RM*.

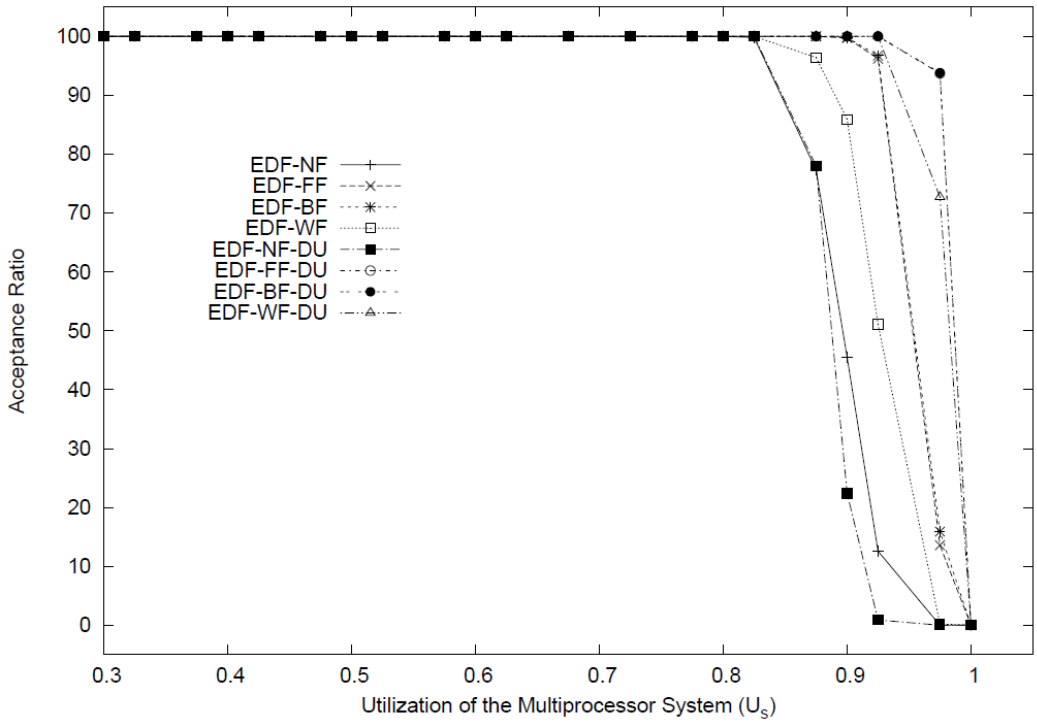

Figura 4.8 Rendimiento en función de U_s, utilizando $\propto = 0.3$, de aquellos algoritmos basados en *EDF*.

En la Figura 4.9 se realiza una comparación de aquellos algoritmos que obtuvieron un mejor rendimiento en el experimento. Se aprecia que en sistemas que están compuestos por tareas con poca demanda del procesador, los algoritmos basados en EDF tienen un mejor rendimiento que los basados en RM. Además, los algoritmos EDF-FF-DU y EDF-BF-DU muestran un rendimiento excelente.

Por otra parte, cuando el sistema está compuesto por tareas con poca y alta demanda del procesador, esto es, para $\propto = 0.7$, los algoritmos RM-NF, RM-FF, RM-BF, y RM-WF, los cuales utilizan una condición de planificabilidad basada en la utilización, muestran una tasa de aceptación similar a aquellas que tienen $\propto = 0.3$. Sin embargo, estos algoritmos emplean la política de utilización decreciente. En la Figura 4.10 se observa el porcentaje de aceptación para algunos algoritmos que utilizan condiciones de planificabilidad basadas en la utilización para $\propto = 0.7$. Se aprecia que el algoritmo RMBF-DU-UO tiene un mejor rendimiento a partir de que $U_s > 0.8$.

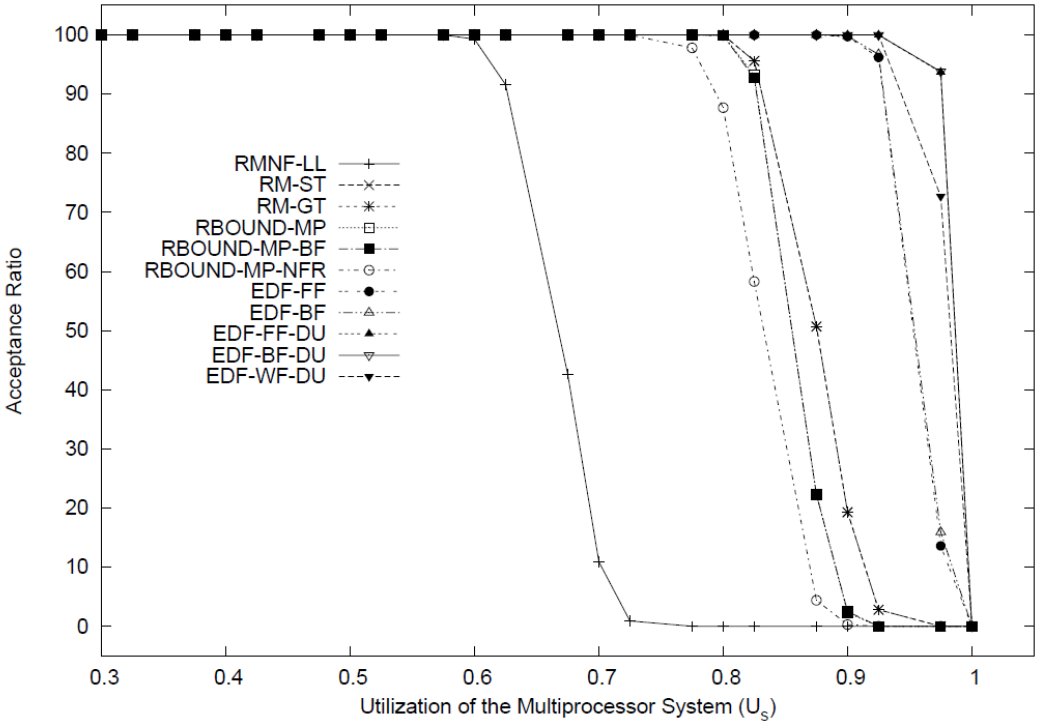

Figura 4.9 Rendimiento en función de U_s, utilizando $\propto = 0.3$, de aquellos algoritmos con el mejor rendimiento.

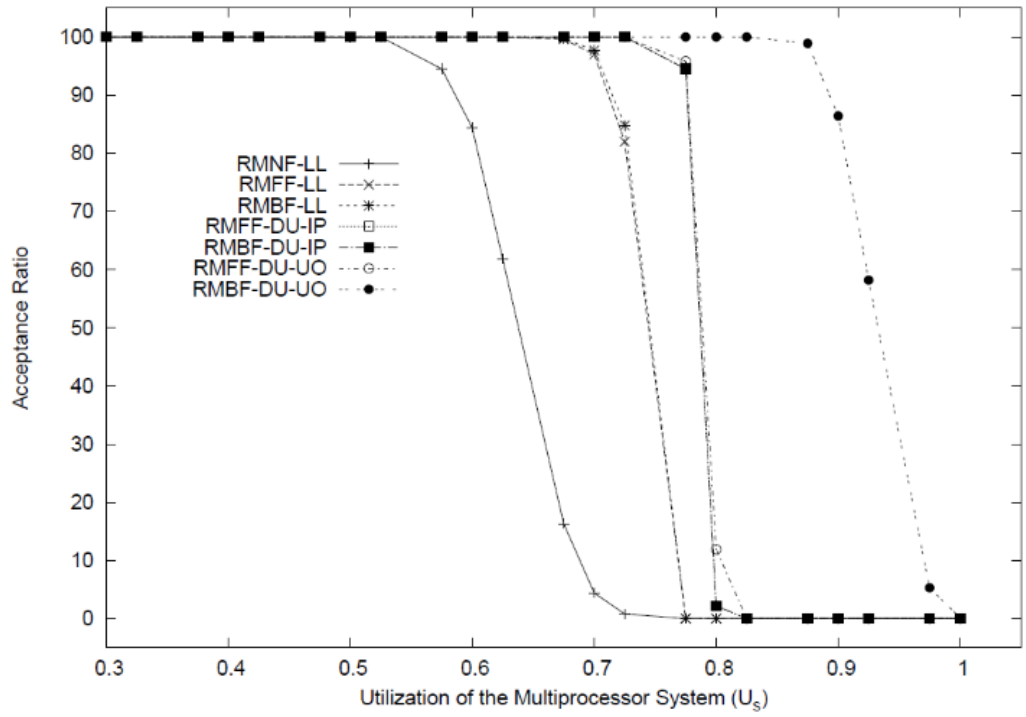

Figura 4.10 Rendimiento en función de U_s utilizando $\propto = 0.7$, para algoritmos basados en *RM* y condiciones *LL*, *IP* y *UO*.

En la Figura 4.11 se observa la tasa de aceptación obtenida por los algoritmos que utilizan condiciones basadas en sus periodos, donde también se incluyen a los algoritmos RMNF-LL y RMBF-DU-UO, para $\propto = 0.7$. Para $U_s \geq 0.5$, su tasa de aceptación satisface la relación: $\rho_U(RMBF - DU - UO) > \rho_U(RBound - MP - BF) > \rho_U(RBound - MP) > \rho_U(RM - ST) > \rho_U(RBound - MP - NFR) > \rho_U(RM - GT) > \rho_U(RMNF - LL)$.

Es claro que la política RMBF-DU-UO es la que mejor se desempeña de todos los algoritmos basados en RM con $U_s > 0.5$. Debido a que la generación de tareas se llevó de manera sintética para este experimento, no existió una relación entre los periodos de las tareas para que al emplearse y evaluarse algoritmos utilizando condiciones de planificabilidad basados en sus periodos no tomaran una ventaja sobre los demás algoritmos. Además, dado que el número de tareas fue el mismo para ambos valores de $\propto$(0.3 y 0.7), y a que la utilización de cada tarea estuvo uniformemente distribuida en el rango de [0.01, 0.3] y [0.01, 0.7] con $\propto = 0.7$, respectivamente, hubo más tareas con poca demanda que aquellas que con $\propto = 0.3$.

La Figura 4.12 muestra la tasa de aceptación para algoritmos basados en EDF para $\propto = 0.7$. La mayoría ofrece un bien rendimiento y así como para $\propto = 0.3$, los algoritmos EDF-FF-DU y EDF-BF-DU ofrecen el mejor rendimiento cuando $U_s \geq 0.85$. En la Figura 4.13 se presenta el porcentaje de aceptación de aquellos algoritmos particionados con el mejor rendimiento en el experimento. Se hace notar que cuando el sistema mantiene tareas con poca y alta demanda (con $\propto$ uniformemente distribuida), los algoritmos particionados basados en EDF ofrecen un mejor rendimiento que los que se basan en RM.

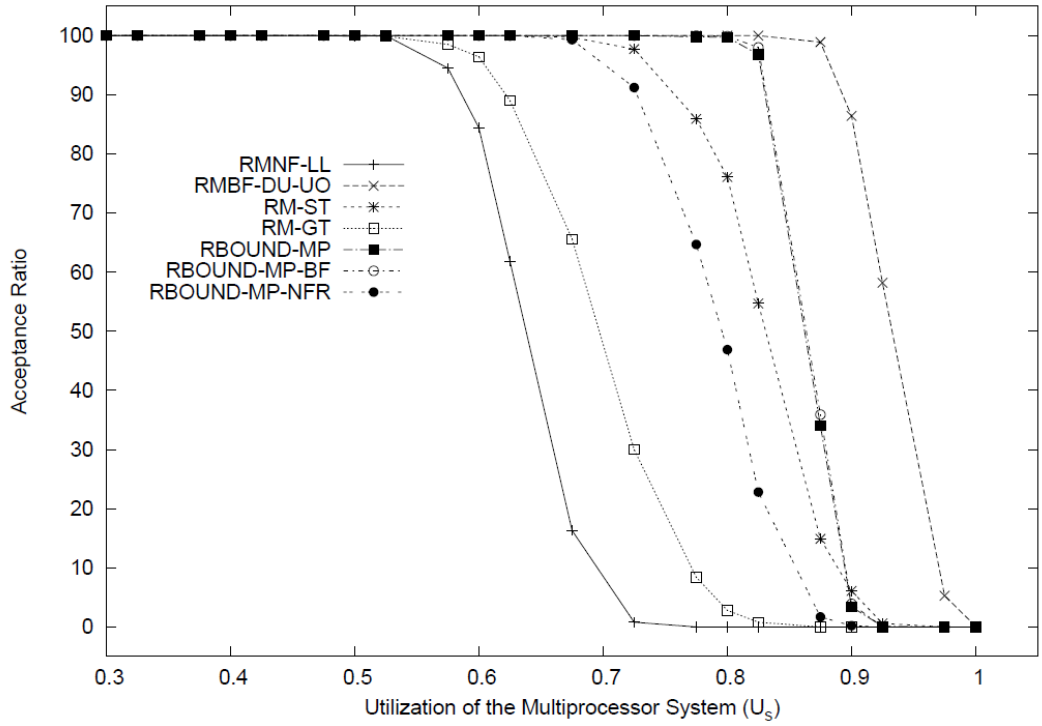

Figura 4.11 Rendimiento en función de U_s utilizando $\propto = 0.7$, y algoritmos basados en *RM*.

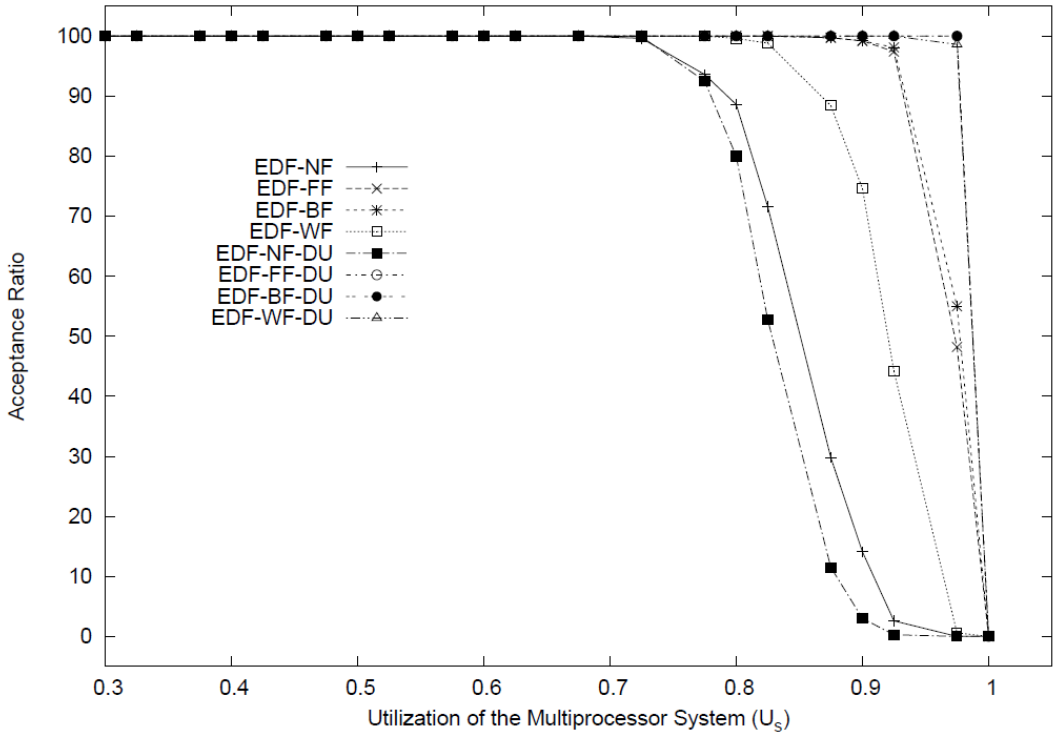

Figura 4.12 Rendimiento en función de U_s utilizando $\propto = 0.7$, y algoritmos basados en *EDF*.

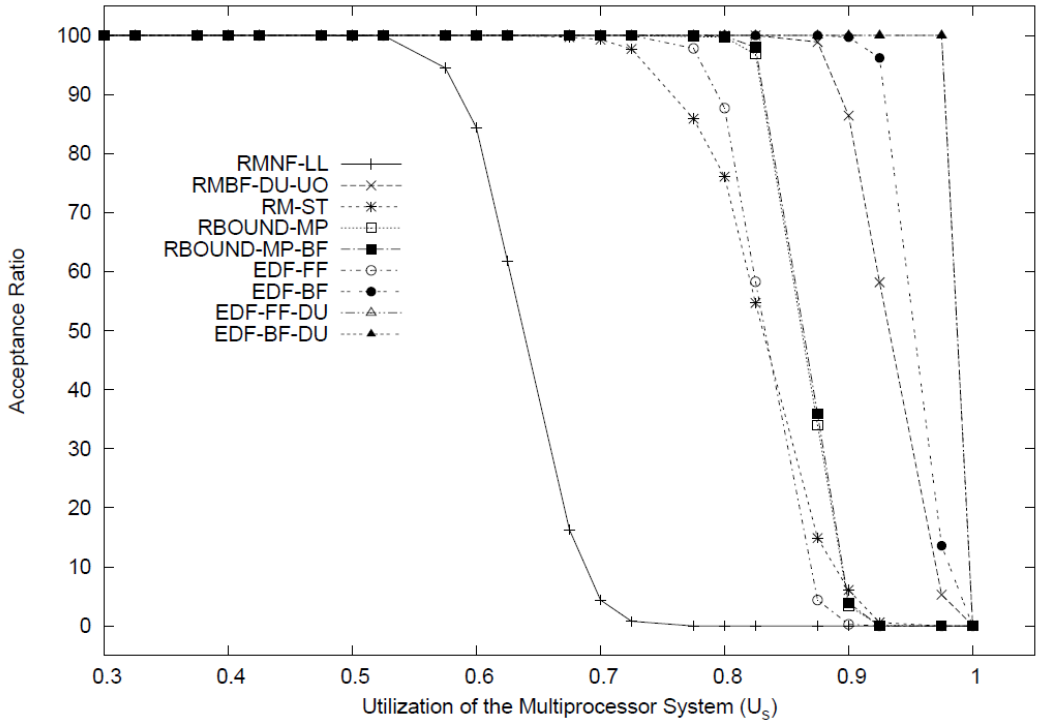

Figura 4.13 Rendimiento en función de U_s utilizando $\propto = 0.7$, de aquellos algoritmos con el mejor rendimiento.

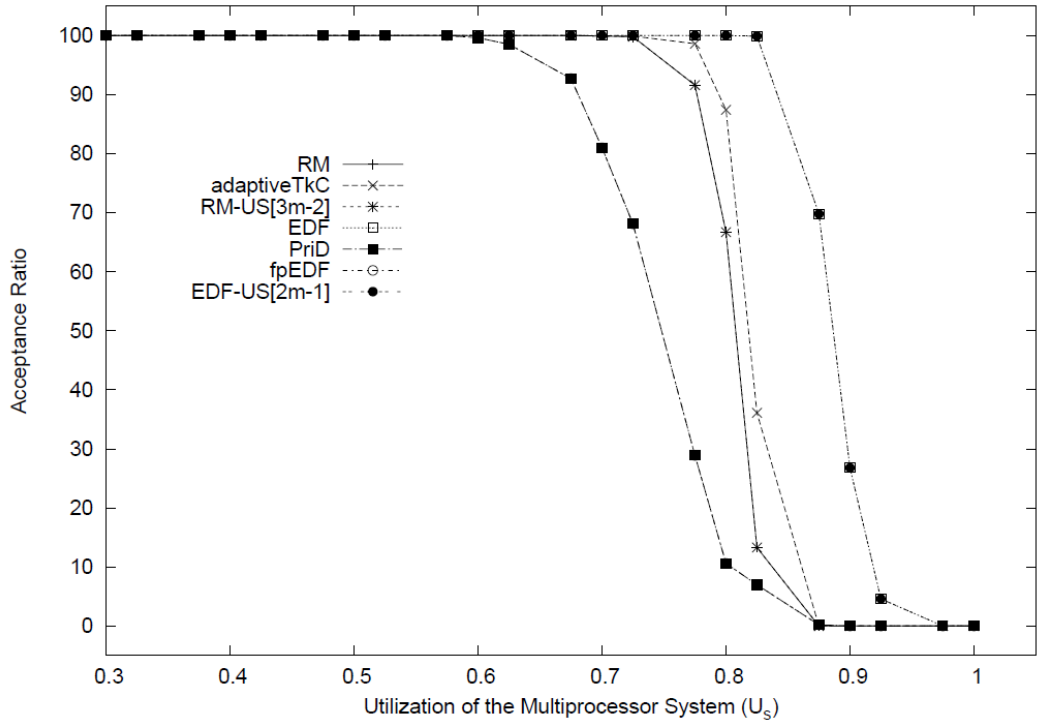

Figura 4.14 Rendimiento en función de U_s utilizando $\propto = 0.3$, para algoritmos globales.

4.4.3 Algoritmos globales

Se realizó la evaluación así mismo para aquellos algoritmos del esquema global, en la Figura 4.14 se observa el rendimiento ofrecido para $\propto = 0.3$. La tasa de aceptación de estos algoritmos presentados satisface la relación siguiente:

$\rho_U(EDF) = \rho_U(EDF - US[2m-1]) > \rho_U(fpEDF) = \rho_U(PriD) > \rho_U(adaptiveTkC) > \rho_U(RM - US[3m-2]) = \rho_U(RM)$.

Los algoritmos basados en EDF muestran un mejor rendimiento que aquellos en RM. Además, los algoritmos EDF y EDF-US[2m-1] su tasa de aceptación es mayor que los demás para $U_s \geq 0.875$. Por otro lado, el algoritmo adaptiveTkC mostro el mejor rendimiento sobre aquellos algoritmos basados en RM.

En la Figura 4.15 se muestra la misma comparación solo que utilizando $\propto = 0.7$. Se aprecia que la relación entre la tasa de aceptación en estas condiciones es similar a la que se presenta utilizando $\propto = 0.3$, esto es, $\rho_U(EDF) > \rho_U(EDF - US[2m-1]) > \rho_U(fpEDF) > \rho_U(PriD) > \rho_U(adaptiveTkC) > \rho_U(RM - US[3m-2]) > \rho_U(RM)$.

Sin embargo, el rendimiento de todos los algoritmos es decreciente, que es mayor a los basados en RM.

4.4.4 Rendimiento en función de tareas con alta demanda de procesador

El objetivo de este experimento es evaluar el rendimiento de los algoritmos de planificación en función del porcentaje de tareas de alta demanda de procesador en el conjunto de tareas. Se considera un sistema multiprocesador compuesto por 8 procesadores. Se generan diversas muestras de conjuntos de tareas definidas por S_{ht}, donde *ht* es el porcentaje de aquellas tareas con alta demanda en el conjunto de tareas, con valores de 0, 0.1, 0.2, 0.3, 0.4 y 0.5. Se define como una tarea de baja demanda como aquella que tiene un máximo factor de utilización uniformemente distribuido en el rango de $\left(1/3, 0.9\right]$. Los periodos de las tareas se encuentran además uniformemente distribuidos en el rango de [100, 500]. Los tiempos de ejecución de las tareas son generadas con valores de $1 \leq C_i \leq \propto T_i$. Cada muestra S_{ht} está compuesta por 1000 conjuntos de *n* tareas. El número de tareas va un rango de [9, 40]. El factor de utilización total del sistema de cada conjunto de tareas U_s esta uniformemente distribuido en el rango

de [0.3, 1].

En la Figura 4.16 se aprecia la tasa de aceptación obtenida por algoritmos basados en RM que utilizan condiciones de planificabilidad basadas en la utilización.

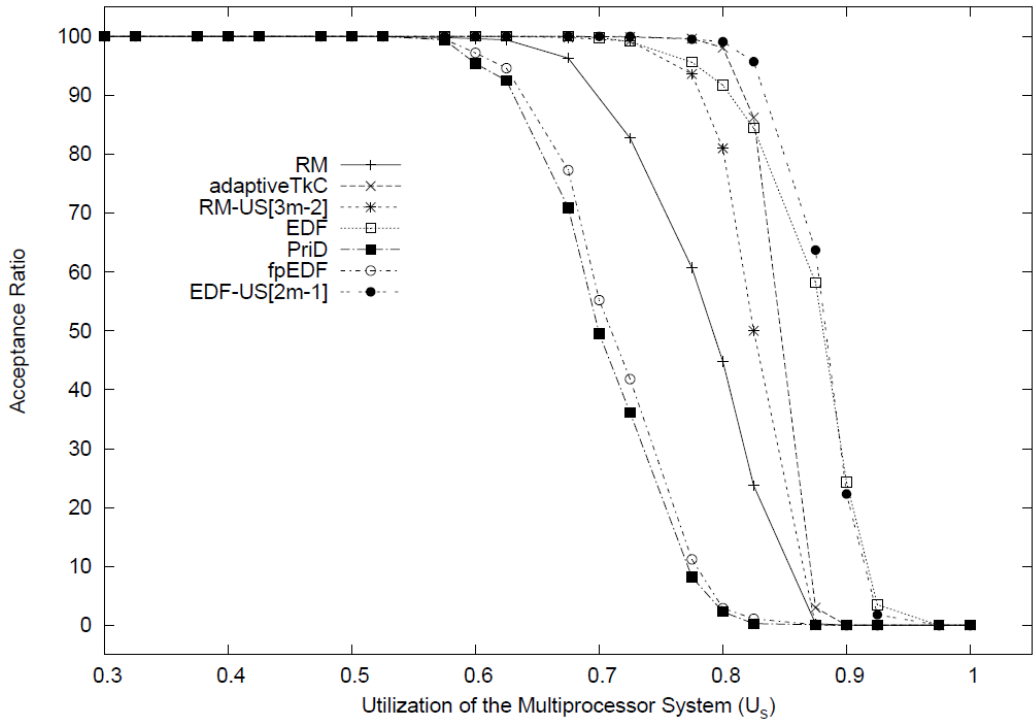

Figura 4.15 Rendimiento en función de U_s utilizando $\propto = 0.7$, para algoritmos globales.

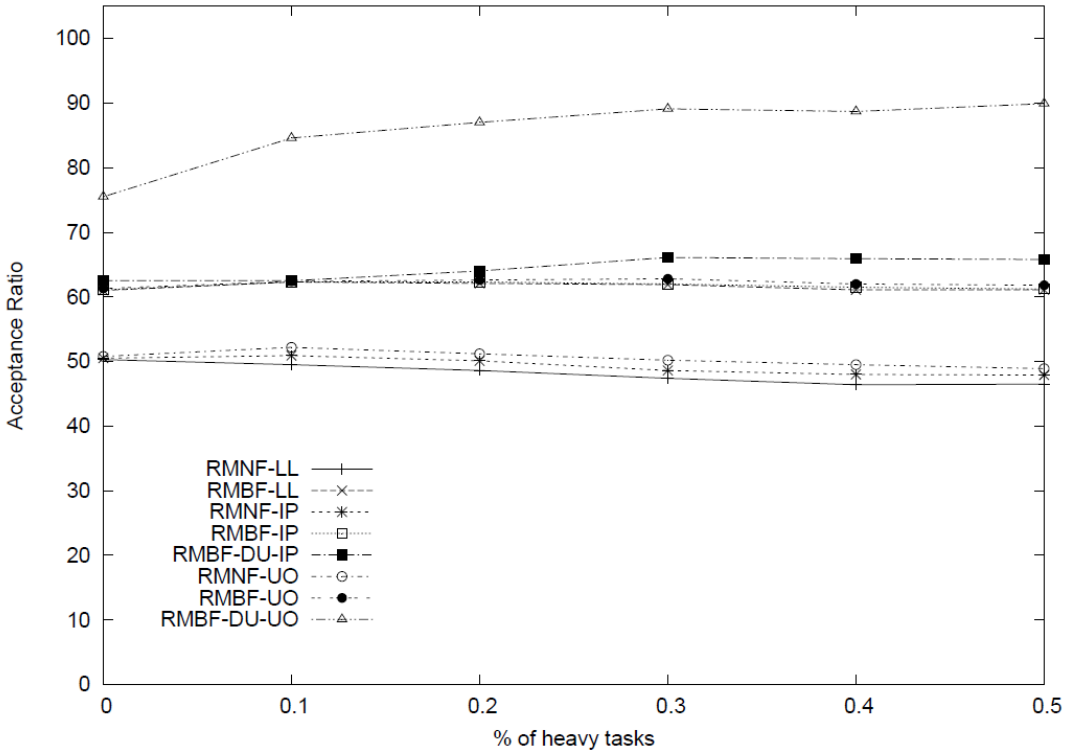

Figura 4.16 Rendimiento en función del % de tareas con alta demanda utilizando $\propto = 0.7$, para algoritmos globales.

4.5 Conclusiones del Capitulo

Las evaluaciones realizadas a los algoritmos presentados anteriormente nos demuestran aspectos importantes a considerar durante la elección de algún algoritmo para su uso bajo un STR. Todo esto para lograr cumplir con las restricciones que cada sistema requiere. Se observa que los mejores algoritmos que presentan una mejor eficiencia hablando en términos generales son los algoritmos particionados, aun cuando su capacidad tiende ser utilizada al 100%. Además su implementación resulta ser idónea para llevarlos a cabo en entornos reales ya que no generan numerosos cambios de contexto, caso contrario a lo que se presenta en la práctica con los algoritmos globales. Aunque por otro lado en el caso de globales, el número de tareas que cumplen sus restricciones es mayor. Es claro, que de acuerdo a lo descrito por diversos autores de STRs la eficiencia de algoritmos globales es mucho mayor, aunque la dificultad de implementarlos bajo sistemas reales es mucho más complejo, lo que limita la experimentación de estos.

Capítulo 5

Conclusiones

A razón que la tecnología avanza y ofrece nuevas funcionalidades a los usuarios, los entornos de los sistemas que se ejecutan bajo STRs, tienden a adentrarse a nuevas áreas. Inicialmente los STRs estaban destinados para aplicarse en áreas militares, aeroespaciales, medicina, nuclear, aquellas disciplinas que presentan alto riesgo si algo no llega a suceder a tiempo de acuerdo a lo planeado. Recientemente, numerosos aplicaciones de tiempo-real han logrado la incursión en sistemas ciber-fisicos. Especialistas sugieren que estas se integran cada vez lo más profundo al diseño del producto, por lo que es de gran importancia asegurar su comportamiento correcto en las situaciones tanto regulares como críticas. Un automóvil autónomo es ejemplo claro de estos.

En el desarrollo de esta tesis, se trabajó sobre el estudio de STRs, específicamente sobre SOTRs que permiten soportar STRs críticos. Esto llevó a la implementación del algoritmo EDF en un Kernel *open source* (Linux), que anteriormente solo en algunos SOTRs era soportado, o bien se ofrece a costa de que se pierde la generalidad del entorno del SOTR. En este trabajo, se presenta dicha implementación. Con esta característica se permite continuar con el soporte de POSIX ofreciendo la portabilidad de los sistemas. Gracias a esto, es posible utilizar el parche sobre cualquier distribución basada en Linux. Además, este desarrollo se ofreció para *The Real Time Linux collaborative project*[1] buscando una mejora continua por parte del público que está interesado en colaborar con el proyecto y continuar con el soporte y extensión.

El segundo enfoque de esta tesis, fue la evaluación de algoritmos en sistemas multiprocesador, lo cual se realizó a través del simulador Realtss-MP. Para un algoritmo se mostraron los entornos y características que le permiten llegar a tener un mejor resultado considerando distintas métricas. A partir de esta selección se permite diseñar un STR que elija entre uno y otro algoritmo, y considere él que mejor se adapte de acuerdo a las características en los que se lleve a cabo. Tal selección repercute en el rendimiento del sistema.

Durante el desarrollo de esta tesis, distintos autores han propuesto diversos

[1] https://wiki.linuxfoundation.org/realtime/rtl/start

algoritmos, muchos de los cuales presentan características que los hacen eficientes, pero sólo considerando ciertos modelos. Sin embargo, gran parte de éstos presentan un problema grave que es la sobrecarga, resultando imposible su uso en entornos reales.

Como trabajo futuro, se busca llevar a cabo un estudio de sistemas en un nuevo nicho de tecnología como lo es el IoT, donde hasta el momento no se ha presentado trabajos de alto impacto. Es un área que en recientes años ha tenido un crecimiento hablando en términos de utilización de estos dispositivos que adoptan dicha tecnología, desde dispositivos en el hogar, como dispositivos en el trabajo o bien en nuestros traslados de un lugar hacia otro. Además, es esencial considerarse hoy en día el uso eficiente de la energía, debido al cuidado del ambiente y evitarse desperdiciar electricidad que comúnmente es utilizada a través de baterías y que por ende no se tiene una disposición correcta de este tipo de materiales.

Referencias

Anderson, J. H., Devi, U. (2011). Multiprocessor real-time scheduling. *Journal of Systems Architecture*, 57(5), 485-486.

Anderson, J.H., Srinivasan A. (2001). Mixed pfair/erfair scheduling of asynchronous periodic tasks. In Proceedings of the 13th Euromicro Conference on Real-Time Systems, 76-85.

Anderson, J., Srinivasan, A. (2000) Pfair scheduling: beyond periodic task systems. Proceedings Seventh International Conference on Real-Time Computing Systems and Applications, 297-306.

Audsley, N., Burns, A., Richardson, M., Tindell, K., Wellings, A. (1993). Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, **8**(5), 284-292.

Audsley, N., Burns, A., Davis, R., Tindell, K., Wellings, A. (1995). Fixed priority pre-emptive scheduling: An historical perspective. Real-Time Systems, **8**(2-3), 173-198.

Back, H., Chwa, H. S., Shin, I. (2012). Schedulability Analysis and Priority Assignment for Global Job-Level Fixed-Priority Multiprocessor Scheduling. In IEEE 18th Real-Time and Embedded Technology and Applications Symposium (RTAS), 297-306.

Baltarejo, P., Pereira, N., Tovar, E (2012). Enhancing the real-time capabilities of the Linux Kernel. *Special Interest Group on Embedded Systems Review* (*SIGBED Review*), **9**(4) 45–48.

Baker, T. (2006). An analysis of fixed-priority schedulability on a multiprocessor. Real-Time Systems, **32**(1-2), 49-71.

Baruah, S. K., Rosier, L.E., Howell, R. R. (1990). Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, **2**(4) 301–324.

Baruah, S. K., Goossens, J. (2003). *Rate-monotonic scheduling on uniform multiprocessors*. In: Proceedings of the 23rd International Conference on Distributed Computing Systems, Providence, RI, USA, **52**(7) 966–970.

Baruah, S. K. (2000). Scheduling periodic tasks on uniform multiprocessors. Proceedings 12th Euromicro Conference on Real-Time Systems, 7-13.

Baruah, S., Fisher, N. (2006). The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. IEEE Transactions on Computers, **55**(7), 918-923.

Baskiyar, S., Meghanathan, N. (2004). A Survey of Contemporary Real-time Operating Systems. *Informatica*, 29(2), 233-240.

Baruah, S. (2007). Techniques for multiprocessor global schedulability analysis. In Proceedings of the 28th IEEE International Real-Time Systems Symposium, 119-128.

Bertogna, M., Cirinei, M., Lipari, G. (2005). Improved schedulability analysis of EDF

on multiprocessor platforms. In 17th Euromicro Conference on Real-Time Systems, 209-218.

Bini, E., Buttazzo, G. (2004). Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, **53**(11), 1462-1473.

Blazewicz, J., Ecker, K., Pesch, E., Schmidt, G., Weglarz, J. (2007). Handbook on Scheduling: From Theory to Applications. Springer Publisher.

Burns, A. (1991). Scheduling hard real-time systems: a review. *Software Engineering Journal* **6**(3) 16–28.

Brun, A., Guo, C., Ren, S. (2015). A Note on the EDF Preemption Behavior in “Rate Monotonic Versus EDF: Judgment Day”. *Embedded Systems Letters*, **7**(3) 89-91.

Burchard, A., Liebeherr, J., Oh, Y., Son, S. H. (1995). New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, **44**(12), 1429-1442.

Buttazzo, G. C. (2011). Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Springer Publisher.

Buttazzo, G. C. (2005). Rate Monotonic vs. EDF: Judgment day. *Real-Time Systems*, **29**(1) 5–26.

Calandrino, J. M., Leontyev, H., Block, A., Devi, U. C., et al. (2006). *LITMUS-RT: A testbed for empirically comparing real-time multiprocessor schedulers*. In: Proceedings of the 27th Symposium on Real-Time Systems, Rio de Janeiro, Brazil, 11–26.

Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J., Baruah, S. (2004). A categorization of real-time multiprocessor scheduling problems and algorithms. Handbook on Scheduling Algorithms, Methods, and Models. Chapman Hall/CRC Publisher.

Cottet, F., Delacroix, J., Kaiser, C., Mammeri, Z. (2002). Scheduling in Real-Time Systems. Wiley Publisher.

Cucu-Grosjean, L., Goossens, J. (2011). Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms. *Journal of Systems Architecture*, **57**(5), 567-569.

Davis, R. I., Burns, A. (2011). A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 43(4) 35:1--35:44.

Davis, R. I. (2014). A Review of Fixed Priority and EDF Scheduling for Hard Real-time Uniprocessor Systems. *Real-Time Systems*, **11**(1) 8-19.

Davis, R. I., Cucu-Grosjean, L., Bertgona, M., Burns, A. (2016). A review of priority assignment in real-time systems. *Journal of Systems Architecture*, **65**(C) 64-82.

Dellinger, M., Garyali, P., Ravindran, B. (2011). *CHRONOS Linux: A best-effort real-time multiprocessor Linux kernel*. In: 48th Design Automation Conference (DAC), NY, USA, 474–479.

Dertouzos, M., Mok, A. (1989). Multiprocessor online scheduling of hard real-time

tasks. *IEEE Transactions Software Enginerring*. 15 (12), 1497-1506.

Dongwook, K., Woojoong, L, Park, C. (2007). Kernel thread scheduling in real-time Linux for wearable computer. *ETRI Journal*, **29**(3), 270-280.

Diaz-Ramirez, A., Orduno, D., Mejia-Alvarez, P. (2012). A multiprocessor real-time scheduling simulation tool. In 22nd International Conference on Electrical Communications and Computers (CONIELECOMP), 157-161.

Faggioli, D., Checconi, F., Trimarchi, M., Scordino, C. (2009). *An EDF scheduling class for the Linux Kernel*. In: Proceedings of the 11th Real-Time Linux Workshop, Dresden, Germany.

Faggioli, D., Trimarchi, M., Checconi, F., Bertogna, M., et al. (2009a). *An implementation of the earliest deadline first algorithm in L inux*. In: Proceedings of the ACM symposium on Applied Computing, Honolulu, USA, 1984–1989.

FTRCE home page. https://www.kernel.org/doc/Documentation/trace/ftrace.txt. (Consultado el 20 de Noviembre de 2017).

GLIBC home page, https://www.gnu.org/software/libc/. (Consultado el 20 de Noviembre de 2017).

Funaoka, K., Kato, S., Yamasaki, N. (2008). New Abstraction for Optimal Real-Time Scheduling on Multiprocessors. In 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 357-364.

Garey, M. R., Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co.

INTEGRITY home page. http://www.ghs.com/products/rtos/integrity.html. (Consultado el 20 de Noviembre de 2017).

KIWI home page. http://www.gti-ia.upv.es/sma/tools/kiwi/index.php. (Consultado el 20 de Noviembre de 2017).

Koh, J. H., Choi, B. W. (2013). Real-time Performance of Real-time Mechanisms for RTAI and Xenomai in Various Running Conditions. *International Journal of Control and Automation*, **6**(1) 235-246.

Lauzac, S., Melhem, R., Mosse, D. (1998). Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. Proceeding. 10th EUROMICRO Workshop on Real-Time Systems, 188-195

Lehoczky, J., Sha, L., Ding, Y. (1989). *The Rate Monotonic scheduling algorithm: exact characterization and average case behavior*. In: Proceedings of the Symposium on Real Time Systems, Santa Monica, CA, USA, 166–171.

Lelli, J., Faggioli, D., Cucinotta, T. (2011). *An efficient and scalable implementation of global EDF in Linux*. In: Proceedings of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Porto, Portugal, 6–15.

Liu, C. L., Layland, J. W. (1973). Scheduling algorithms for multi-programming in a hard-real-time environment. *Journal o f t h e ACM*, **20**(1) 46–61.

Love, R. (2010). Linux Kernel Development. Addison-Wesley Professional.

Min-Allah, N., Khan, S. U., Ghani, N., Li, J., Wang, L., Bouvry, P. (2011). A comparative study of rate monotonic schedulability tests. *The Journal of Supercomputing*, **59**(3). 1419-1430.

QNX Neutrino home. http://www.qnx.com/products/neutrino-rtos/neutrino- rtos.html. (Consultado el 20 de Noviembre de 2017).

Park, M., Han, S., Kim, H., Cho, S., Cho, Y. (2005). Comparison of deadline-based scheduling algorithms for periodic real-time tasks on multiprocessor. *IEICE Transactions on Information and Systems*, **E88-D**(3), 658-661.

Rajkumar, R., Lee, I., Sha, L., Stankovic, J. (2010). *Cyber-physical systems: The next computing revolution*. In: Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC), Anaheim, CA, USA, 731-736.

Rostedt, S., Hart, D. V. (2007). *Internals of the RT patch*. In: Proceedings of the Linux Symposium, Ottawa, Canada, 161–172.

RTAI home page, https://www.rtai.org/. (Consultado el 20 de Noviembre de 2017).

Mitra, S. K. (2001). Digital Signal Processing: A Computer-Based approach. McGraw-HIll.

Schneider, R., Goswami, D., Masrur, A., Becker, M., et al. (2013). Multi-layered Scheduling of Mixed-criticality Cyber-physical Systems. Journal of System Architecture, **59**(10) 1215-1230.

Sha, L., Abdelzaher, T., Årzén, K. E., Cervin, A., et al. (2004). Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, **28**(2-3) 101-155.

Sha, L., Rajkumar, R. Lehoczky, J.P. (1990). Priority inheritance protocols: an approach to real-time synchronization. IEEE Transactions on Computers, **39** (9), 1175-1185.

Silberschatz, A., Baer, P. G., Greg G. (2006). Fundamentos de Sistemas Operativos. McGraw-Hill.

Srovnal, V., Kotzian, J. (2008). *Development of a flight control system for an ultralight airplane*. In: Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT 2008), Wisła, Poland, IEEE Computer Society.

Stankovic, J. A., Ramamritham, K., Spuri, M. (1998). Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms. Kluwer Academic Publishers.

Stankovic, J. (1988). Misconceptions about real-time computing: a serious problem for next-generation systems. Computer, **21** (1), 10-19.

Stankovic, J., Spuri, M., Di Natale, M., Buttazzo, G. (1995). Implications of classical scheduling results for real-time systems. Computer, **28**(6), 16-25.

The Open Group Technical Standard Base Specifications. (2008) In: Edition IEEE Standard 1003.1, Institute of Electrical and Electronic Engineers and the Open Group, 7.

Thekkilakattil, A., Dobrin, R., Punnekkat, S. (2015). The limited-preemptive feasibility of real-time tasks on uniprocessors. *Real-Time Systems*, **51**(3), 247-273.

TRACE-CMD home page. https://git.kernel.org/cgit/linux/kernel/git/rostedt/trace-cmd.git. (Consultado el 20 de Noviembre de 2017).

Trejo, K., Angulo, C. (2016). Single-Camera Automatic Landmarking for People Recognition with an Ensemble of Regression Trees. *Computación y Sistemas*, **20**(1) 19-28.

Vázquez-Santacruz, E., Cruz-Santos, W., Gamboa-Zúñiga, M. (2015). Design and Implementation of an Intelligent System for Controlling a Robotic Hospital Bed for Patient Care Assistance. *Computación y Sistemas*, 19(3) 467-474.

VxWORKS home page. http://windriver.com/products/vxworks/. (Consultado el 20 de Noviembre de 2017).

XENOMAI home page. http://www.xenomai.org/. (Consultado el 20 de Noviembre de 2017).

Yodaiken, V. (1999). *The RTLinux manifesto*. In: Proceedings of the 5th Linux Conference, Raleigh, North Carolina, USA.

Younis, M .F., Aboutabl, M., Kim, D. (2004). Software Environment for Integrating Critical Real-time Control Systems. Journal of Systems Architecture, **50**(11) 649-674.

Nomenclatura

τ_i	Tarea, $i = 1,\ldots, N$.
m	Procesador, $m = 1,\ldots, M$.
j	Índice de la activación de una tarea (trabajo).
Θ	Conjunto de tareas, $\Theta = \{\tau_1,\ldots, \tau_N\}$, $\tau_i = \{J_{i1},\ldots, J_{ij},\ldots\}$.
R_i	Recurso i.
φ_i	Fase de tarea i. Es un instante de liberación de la primera activación de la tarea i, con respecto al instante 0.
$r_{i,j}$	Tiempo de liberación. Es el instante de tiempo, en el cual la tarea i está lista para ser ejecutada en la activación j.
D_i	Plazo relativo. Es el máximo tiempo de respuesta permitido para la tarea i.
$d_{i,j}$	Plazo absoluto. Es el tiempo, en el cual el trabajo J_{ij} debe ejecutarse, $d_{ij} = r_{i,j} + D_i$.
T_i	Periodo. Es el tiempo fijo entre dos activaciones sucesivas de tarea i.
C_i	Tiempo de ejecución en el peor caso (WCET). Es el máximo tiempo de ejecución de la tarea i.
U_i	Factor de utilización de la tarea i.
U_T	Factor de utilización total del sistema.

Abreviaciones

BF	Best Fit
DDSA	Deadline Driven Scheduling Algorithm
DMA	Direct Memory Access
DU	Utilización decreciente
E/S	Entrada/Salida
EDF	Earliest Deadline First
FF	First First
FIFO	First-In First-Out
FMLP	Flexible Multiprocessor Locking Protocol
MPCP	Multiprocessor PCP
MSRP	Multiprocessor SRP
NF	Near First
PCP	Protocolo de techo de prioridad
PIP	Protocolo de herencia de prioridades
RM	Rate Monotonic
RMGT	Rate Monotonic General Tasks
RMNF	Rate Monotonic Next-Fit
RMNF	Rate Monotonic First-Fit
RMST	Rate Monotonic Small Tasks
RMWF	Rate Monotonic Worst First
RR	Round Robin
SO	Sistema Operativo
STR	Sistema de Tiempo-Real
SOTR	Sistema Operativo de Tiempo-Real

Anexos

9.1 Parche SSELinux-EDF

```
diff --git a/Makefile b/Makefile
index bc4dd5b..6d82e30 100644
--- a/Makefile
+++ b/Makefile
@@ -1,7 +1,9 @@
 VERSION = 3
 PATCHLEVEL = 4
 SUBLEVEL = 61
-EXTRAVERSION =
+# EDF_implementation++
+EXTRAVERSION = edfV2
+# EDF_implementation--
 NAME = Saber-toothed Squirrel

 # *DOCUMENTATION*
diff --git a/arch/microblaze/boot/dts/system.dts
b/arch/microblaze/boot/dts/system.dts
deleted file mode 120000
index 7cb6578..0000000
--- a/arch/microblaze/boot/dts/system.dts
+++ /dev/null
@@ -1 +0,0 @@
-../../platform/generic/system.dts
\ No newline at end of file
diff --git a/arch/microblaze/boot/dts/system.dts
b/arch/microblaze/boot/dts/system.dts
new file mode 100644
index 0000000..7cb6578
--- /dev/null
+++ b/arch/microblaze/boot/dts/system.dts
@@ -0,0 +1 @@
+../../platform/generic/system.dts
\ No newline at end of file
diff --git a/arch/x86/Kconfig b/arch/x86/Kconfig
index 1c03dfc..1c4d65b 100644
--- a/arch/x86/Kconfig
+++ b/arch/x86/Kconfig
@@ -2211,6 +2211,22 @@ config HAVE_TEXT_POKE_SMP
      bool
      select STOP_MACHINE if SMP

+# EDF_implementation++
+# We define the new schedule policy in the menu
+menu "EDF SCHEDULER"
+
+config SCHED_EDF_POLICY
+     bool "EDF scheduling policy"
```

```
+	default y
+
+# Define the tracer var
+config TRACE_SCHED_EDF
+	bool "Trace scheduler transitions"
+	default y
+
+endmenu
+# EDF_implementation--
+
 source "net/Kconfig"

 source "drivers/Kconfig"
diff --git a/include/linux/init_task.h b/include/linux/init_task.h
index 07d1b36..e5dbad6 100644
--- a/include/linux/init_task.h
+++ b/include/linux/init_task.h
@@ -153,6 +153,8 @@ extern struct task_group root_task_group;
  *  INIT_TASK is used to set up the first task table, touch at
  * your own risk!. Base=0, limit=0x1fffff (=2MB)
  */
+/* EDF_implementation++ */
+#ifdef CONFIG_SCHED_EDF_POLICY
 #define INIT_TASK(tsk)	\
 {                                                                       \
 	.state		= 0,                                            \
@@ -174,6 +176,8 @@ extern struct task_group root_task_group;
 		.time_slice	= RR_TIMESLICE,                         \
 		.nr_cpus_allowed = NR_CPUS,                             \
 	},                                                              \
+	.rel_deadline	= {0,0},                                \
+	.abs_deadline	= {0,0},                                \
 	.tasks		= LIST_HEAD_INIT(tsk.tasks),                    \
 	INIT_PUSHABLE_TASKS(tsk)                                \
 	INIT_CGROUP_SCHED(tsk)                                          \
@@ -218,6 +222,74 @@ extern struct task_group root_task_group;
 	INIT_TASK_RCU_PREEMPT(tsk)                                      \
 	INIT_CPUSET_SEQ                                                 \
 }
+#else
+#define INIT_TASK(tsk)	\
+{                                                                       \
+	.state		= 0,                                            \
+	.stack		= &init_thread_info,                            \
+	.usage		= ATOMIC_INIT(2),                               \
+	.flags		= PF_KTHREAD,                                   \
+	.prio		= MAX_PRIO-20,                                  \
+	.static_prio	= MAX_PRIO-20,                                  \
+	.normal_prio	= MAX_PRIO-20,                                  \
+	.policy		= SCHED_NORMAL,                                 \
+	.cpus_allowed	= CPU_MASK_ALL,                                 \
+	.mm		= NULL,                                         \
```

```
+     .active_mm      = &init_mm,                                          \
+     .se             = {                                                  \
+             .group_node     = LIST_HEAD_INIT(tsk.se.group_node),         \
+     },                                                                   \
+     .rt             = {                                                  \
+             .run_list       = LIST_HEAD_INIT(tsk.rt.run_list),           \
+             .time_slice     = RR_TIMESLICE,                              \
+             .nr_cpus_allowed = NR_CPUS,                                  \
+     },                                                                   \
+     .tasks          = LIST_HEAD_INIT(tsk.tasks),                         \
+     INIT_PUSHABLE_TASKS(tsk)                                      \
+     INIT_CGROUP_SCHED(tsk)                                               \
+     .ptraced= LIST_HEAD_INIT(tsk.ptraced),                        \
+     .ptrace_entry   = LIST_HEAD_INIT(tsk.ptrace_entry),                  \
+     .real_parent    = &tsk,                                              \
+     .parent         = &tsk,                                              \
+     .children       = LIST_HEAD_INIT(tsk.children),                      \
+     .sibling= LIST_HEAD_INIT(tsk.sibling),                        \
+     .group_leader   = &tsk,                                              \
+     RCU_INIT_POINTER(.real_cred, &init_cred),                            \
+     RCU_INIT_POINTER(.cred, &init_cred),                                 \
+     .comm           = INIT_TASK_COMM,                                    \
+     .thread         = INIT_THREAD,                                       \
+     .fs             = &init_fs,                                          \
+     .files          = &init_files,                                       \
+     .signal         = &init_signals,                              \
+     .sighand= &init_sighand,                          \
+     .nsproxy= &init_nsproxy,                          \
+     .pending= {                                                   \
+             .list = LIST_HEAD_INIT(tsk.pending.list),                    \
+             .signal = {{0}}},                                            \
+     .blocked= {{0}},                                  \
+     .alloc_lock     = __SPIN_LOCK_UNLOCKED(tsk.alloc_lock),              \
+     .journal_info   = NULL,                                              \
+     .cpu_timers     = INIT_CPU_TIMERS(tsk.cpu_timers),                   \
+     .pi_lock= __RAW_SPIN_LOCK_UNLOCKED(tsk.pi_lock),\
+     .timer_slack_ns = 50000, /* 50 usec default slack */                 \
+     INIT_TIMER_LIST                                                      \
+     .pids = {                                                            \
+             [PIDTYPE_PID]  = INIT_PID_LINK(PIDTYPE_PID),                 \
+             [PIDTYPE_PGID] = INIT_PID_LINK(PIDTYPE_PGID),                \
+             [PIDTYPE_SID]  = INIT_PID_LINK(PIDTYPE_SID),                 \
+     },                                                                   \
+     .thread_group   = LIST_HEAD_INIT(tsk.thread_group),                  \
+     INIT_IDS                                                      \
+     INIT_PERF_EVENTS(tsk)                                                \
+     INIT_TRACE_IRQFLAGS                                                  \
+     INIT_LOCKDEP                                                         \
+     INIT_FTRACE_GRAPH                                                    \
+     INIT_TRACE_RECURSION                                                 \
+     INIT_TASK_RCU_PREEMPT(tsk)                                           \
+     INIT_CPUSET_SEQ                                                      \
```

```
+}
+#endif
+/* EDF_implementation-- */


 #define INIT_CPU_TIMERS(cpu_timers)                                    \
diff --git a/include/linux/sched.h b/include/linux/sched.h
index 7a750f5..143ebb6 100644
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -39,14 +39,26 @@
 #define SCHED_BATCH         3
 /* SCHED_ISO: reserved but not implemented yet */
 #define SCHED_IDLE          5
+/* EDF_implementation++ */
+#ifdef CONFIG_SCHED_EDF_POLICY
+#define SCHED_EDF           6
+#endif
+/* EDF_implementation-- */
+
 /* Can be ORed in to make sure the process is reverted back to
SCHED_NORMAL on fork */
 #define SCHED_RESET_ON_FORK     0x40000000

 #ifdef __KERNEL__

+/* EDF_implementation++ */
+#ifdef CONFIG_SCHED_EDF_POLICY
+struct sched_param;
+#else
 struct sched_param {
     int sched_priority;
 };
+#endif
+/* EDF_implementation-- */

 #include <asm/param.h>      /* for HZ */

@@ -95,6 +107,15 @@ struct sched_param {

 #include <asm/processor.h>

+/* EDF_implementation++ */
+#ifdef CONFIG_SCHED_EDF_POLICY
+struct sched_param {
+    int sched_priority;
+    struct timespec sched_rel_deadline;
+};
+#endif
+/* EDF_implementation-- */
+
 struct exec_domain;
```

```
 struct futex_pi_state;
 struct robust_list_head;
@@ -1283,6 +1304,14 @@ struct task_struct {
      const struct sched_class *sched_class;
      struct sched_entity se;
      struct sched_rt_entity rt;
+
+/* EDF_implementation++ */
+#ifdef CONFIG_SCHED_EDF_POLICY
+     struct timespec rel_deadline;
+     struct timespec abs_deadline;
+#endif
+/* EDF_implementation-- */
+
 #ifdef CONFIG_CGROUP_SCHED
      struct task_group *sched_task_group;
 #endif
diff --git a/include/trace/events/sched.h
b/include/trace/events/sched.h
index ea7a203..903fce3 100644
--- a/include/trace/events/sched.h
+++ b/include/trace/events/sched.h
@@ -426,6 +426,40 @@ TRACE_EVENT(sched_pi_setprio,
                      __entry->oldprio, __entry->newprio)
 );

+/* EDF_implementation++ */
+#ifdef       CONFIG_SCHED_EDF_POLICY
+/*
+ * Tracepoint for a task assigned / updated abs_deadline:
+ */
+TRACE_EVENT(sched_edf_abs_deadline,
+
+     TP_PROTO(struct task_struct *p),
+
+     TP_ARGS(p),
+
+     TP_STRUCT__entry(
+             __array(char,   comm,   TASK_COMM_LEN   )
+             __field(pid_t,  pid                     )
+             __field(int,    prio                    )
+             __field(long,   abs_deadline_tv_sec             )
+             __field(long,   abs_deadline_tv_nsec            )
+     ),
+
+     TP_fast_assign(
+             memcpy(__entry->comm, p->comm, TASK_COMM_LEN);
+             __entry->pid            = p->pid;
+             __entry->prio           = p->prio;
+             __entry->abs_deadline_tv_sec            = p-
>abs_deadline.tv_sec;
+             __entry->abs_deadline_tv_nsec           = p-
```

```
>abs_deadline.tv_nsec;
+     ),
+
+     TP_printk("comm=%s pid=%d prio=%d abs_deadline=%ld.%ld ",
+               __entry->comm, __entry->pid, __entry->prio,
+               __entry->abs_deadline_tv_sec, __entry-
>abs_deadline_tv_nsec)
+);
+#endif
+/* EDF_implementation-- */
+
 #endif /* _TRACE_SCHED_H */

 /* This part must be outside protection */
diff --git a/kernel/sched/core.c b/kernel/sched/core.c
index 8674878..468d368 100644
--- a/kernel/sched/core.c
+++ b/kernel/sched/core.c
@@ -1726,6 +1726,13 @@ static void __sched_fork(struct task_struct
*p)
      memset(&p->se.statistics, 0, sizeof(p->se.statistics));
 #endif

+/* EDF_implementation++ */
+#ifdef       CONFIG_SCHED_EDF_POLICY
+     p->abs_deadline.tv_sec = p->rel_deadline.tv_sec = 0;
+     p->abs_deadline.tv_nsec = p->rel_deadline.tv_nsec = 0;
+#endif
+/* EDF_implementation-- */
+
      INIT_LIST_HEAD(&p->rt.run_list);

 #ifdef CONFIG_PREEMPT_NOTIFIERS
@@ -3434,12 +3441,13 @@ pick_next_task(struct rq *rq)
       * Optimization: we know that if all tasks are in
       * the fair class we can call that function directly:
       */
-     if (likely(rq->nr_running == rq->cfs.h_nr_running)) {
-             p = fair_sched_class.pick_next_task(rq);
-             if (likely(p))
-                     return p;
-     }
-
+/* EDF_implementation++ */
+//   if (likely(rq->nr_running == rq->cfs.h_nr_running)) {
+//           p = fair_sched_class.pick_next_task(rq);
+//           if (likely(p))
+//                   return p;
+//   }
+/* EDF_implementation-- */
      for_each_class(class) {
              p = class->pick_next_task(rq);
              if (p)
```

```
@@ -4312,19 +4320,54 @@ static struct task_struct
*find_process_by_pid(pid_t pid)
      return pid ? find_task_by_vpid(pid) : current;
 }

+/* EDF_implementation++ */
+#ifdef       CONFIG_SCHED_EDF_POLICY
+static void __setscheduler_params(struct task_struct *p, int policy,
int prio, const struct timespec *rel_deadline)
+#else
 static void __setscheduler_params(struct task_struct *p, int policy,
int prio)
+#endif
+/* EDF_implementation-- */
 {
      p->policy = policy;
+/* EDF_implementation++ */
+#ifdef       CONFIG_SCHED_EDF_POLICY
+     if(p->policy == SCHED_EDF){
+             p->rel_deadline = *rel_deadline;
+             /*
+              * CHECK the next code line (Does we must change the
+              * rt_priority to 0, or we have to leave the priority
+              * that the user SET via sched_params ?)
+              */
+             p->rt_priority =99;
+             printk(KERN_DEBUG "EDF - core.c - __setscheduler_params
"
+                               "- setting params to edf task!");
+     } else {
+             p->rel_deadline.tv_sec = 0;
+             p->rel_deadline.tv_nsec = 0;
+             p->rt_priority = prio;
+     }
+#else
      p->rt_priority = prio;
+#endif
+/* EDF_implementation-- */
      p->normal_prio = normal_prio(p);
      set_load_weight(p);
 }

 /* Actually do priority change: must hold rq lock. */
+/* EDF_implementation++ */
+#ifdef       CONFIG_SCHED_EDF_POLICY
+static void
+__setscheduler(struct rq *rq, struct task_struct *p, int policy, int
prio, const struct timespec *rel_deadline)
+{
+     __setscheduler_params(p, policy, prio,rel_deadline);
+#else
 static void
```

```
 __setscheduler(struct rq *rq, struct task_struct *p, int policy, int
prio)
 {
      __setscheduler_params(p, policy, prio);
+#endif
+/* EDF_implementation-- */
      /* we are holding p->pi_lock already */
      p->prio = rt_mutex_getprio(p);
      if (rt_prio(p->prio))
@@ -4373,10 +4416,22 @@ recheck:
              reset_on_fork = !!(policy & SCHED_RESET_ON_FORK);
              policy &= ~SCHED_RESET_ON_FORK;

+/* EDF_implementation++ */
+#ifdef CONFIG_SCHED_EDF_POLICY
+             if (policy != SCHED_FIFO && policy != SCHED_RR &&
+                             policy != SCHED_EDF &&
+                             policy != SCHED_NORMAL && policy !=
SCHED_BATCH &&
+                             policy != SCHED_IDLE) {
+                     /* sched policy is not valid */
+                     return -EINVAL;
+             }
+#else
              if (policy != SCHED_FIFO && policy != SCHED_RR &&
                              policy != SCHED_NORMAL && policy !=
SCHED_BATCH &&
                              policy != SCHED_IDLE)
                      return -EINVAL;
+#endif
+/* EDF_implementation-- */
      }

      /*
@@ -4384,12 +4439,46 @@ recheck:
       * 1..MAX_USER_RT_PRIO-1, valid priority for SCHED_NORMAL,
       * SCHED_BATCH and SCHED_IDLE is 0.
       */
+/* EDF_implementation++ */
+#ifdef CONFIG_SCHED_EDF_POLICY
+     if (param->sched_priority < 0 ||
+                 (p->mm && param->sched_priority > MAX_USER_RT_PRIO-
1) ||
+                 (!p->mm && param->sched_priority > MAX_RT_PRIO-1)) {
+             /* sched priority is not valid */
+             return -EINVAL;
+     }
+
+     if (rt_policy(policy) != (param->sched_priority != 0)) {
+             /* sched deadline is not valid | deadline | prioity */
+             return -EINVAL;
+     }
```

```
+    if (policy == SCHED_EDF && (param->sched_rel_deadline.tv_sec <=
0 &&
+                                    param-
>sched_rel_deadline.tv_nsec <= 0 )) {
+            /* sched deadline is not valid | deadline | prioity */
+            return -EINVAL;
+    }
+
+//  if((rt_policy(policy) &&
+//                  !(param->sched_rel_deadline.tv_sec <= 0 &&
+//                                    param-
>sched_rel_deadline.tv_nsec <= 0 )) ||
+//                  (rt_policy(policy) != (param->sched_priority !=
0))) {
+//          printk(KERN_DEBUG "EDF - core.c - __sched_setscheduler "
+//                              "- sched deadline is not valid |
deadline [%ld : %ld] "
+//                              "| prioity [%d]",
+//                              param->sched_rel_deadline.tv_sec,
+//                              param->sched_rel_deadline.tv_nsec,
+//                              param->sched_priority);
+//          return -EINVAL;
+//  }
+#else
     if (param->sched_priority < 0 ||
-        (p->mm && param->sched_priority > MAX_USER_RT_PRIO-1) ||
-        (!p->mm && param->sched_priority > MAX_RT_PRIO-1))
+                (p->mm && param->sched_priority > MAX_USER_RT_PRIO-
1) ||
+                (!p->mm && param->sched_priority > MAX_RT_PRIO-1))
             return -EINVAL;
     if (rt_policy(policy) != (param->sched_priority != 0))
             return -EINVAL;
+#endif
+/* EDF_implementation-- */

     /*
      * Allow unprivileged RT tasks to decrease priority:
@@ -4498,7 +4587,14 @@ recheck:
      * itself.
      */
     if (rt_mutex_check_prio(p, newprio)) {
+/* EDF_implementation++ */
+#ifdef      CONFIG_SCHED_EDF_POLICY
+            __setscheduler_params(p, policy, param-
>sched_priority,&param->sched_rel_deadline);
+            /* the method __setscheduler_params was called again */
+#else
             __setscheduler_params(p, policy, param->sched_priority);
+#endif
+/* EDF_implementation-- */
             task_rq_unlock(rq, p, &flags);
```

```
              return 0;
      }
@@ -4511,7 +4607,13 @@ recheck:
              p->sched_class->put_prev_task(rq, p);

      prev_class = p->sched_class;
+/* EDF_implementation++ */
+#ifdef       CONFIG_SCHED_EDF_POLICY
+     __setscheduler(rq, p, policy, param->sched_priority,&param-
>sched_rel_deadline);
+#else
      __setscheduler(rq, p, policy, param->sched_priority);
+#endif
+/* EDF_implementation-- */

      if (running)
              p->sched_class->set_curr_task(rq);
@@ -4569,8 +4671,18 @@ do_sched_setscheduler(pid_t pid, int policy,
struct sched_param __user *param)
      struct task_struct *p;
      int retval;

+/* EDF_implementation++ */
+#ifdef       CONFIG_SCHED_EDF_POLICY
+     if (!param || pid < 0) {
+             printk(KERN_DEBUG "EDF - core.c - do_sched_setscheduler
"
+                                "- The params passed are wrong!");
+             return -EINVAL;
+     }
+#else
      if (!param || pid < 0)
              return -EINVAL;
+#endif
+/* EDF_implementation-- */
      if (copy_from_user(&lparam, param, sizeof(struct sched_param)))
              return -EFAULT;

@@ -4659,7 +4771,15 @@ SYSCALL_DEFINE2(sched_getparam, pid_t, pid,
struct sched_param __user *, param)
      if (retval)
              goto out_unlock;

+/* EDF_implementation++ */
+#ifdef       CONFIG_SCHED_EDF_POLICY
+     if (p->policy == SCHED_EDF)
+             lp.sched_rel_deadline = p->rel_deadline;
+     lp.sched_priority = p->rt_priority;
+#else
      lp.sched_priority = p->rt_priority;
+#endif
+/* EDF_implementation-- */
```

```
      rcu_read_unlock();

      /*
@@ -5080,6 +5200,11 @@ SYSCALL_DEFINE1(sched_get_priority_max, int,
policy)
      switch (policy) {
      case SCHED_FIFO:
      case SCHED_RR:
+/* EDF_implementation++ */
+#ifdef       CONFIG_SCHED_EDF_POLICY
+     case SCHED_EDF:
+#endif
+/* EDF_implementation-- */
              ret = MAX_USER_RT_PRIO-1;
              break;
      case SCHED_NORMAL:
@@ -5105,6 +5230,11 @@ SYSCALL_DEFINE1(sched_get_priority_min, int,
policy)
      switch (policy) {
      case SCHED_FIFO:
      case SCHED_RR:
+/* EDF_implementation++ */
+#ifdef       CONFIG_SCHED_EDF_POLICY
+     case SCHED_EDF:
+#endif
+/* EDF_implementation-- */
              ret = 1;
              break;
      case SCHED_NORMAL:
@@ -7573,13 +7703,24 @@ EXPORT_SYMBOL(__might_sleep);
 static void normalize_task(struct rq *rq, struct task_struct *p)
 {
      const struct sched_class *prev_class = p->sched_class;
+/* EDF_implementation++ */
+#ifdef       CONFIG_SCHED_EDF_POLICY
+     const struct timespec empty = {0,0};
+#endif
+/* EDF_implementation-- */
      int old_prio = p->prio;
      int on_rq;

      on_rq = p->on_rq;
      if (on_rq)
              dequeue_task(rq, p, 0);
+/* EDF_implementation++ */
+#ifdef       CONFIG_SCHED_EDF_POLICY
+     __setscheduler(rq, p, SCHED_NORMAL, 0,&empty);
+#else
      __setscheduler(rq, p, SCHED_NORMAL, 0);
+#endif
+/* EDF_implementation-- */
      if (on_rq) {
```

```
              enqueue_task(rq, p, 0);
              resched_task(rq->curr);


GLIBC EDF PATCH

diff --git a/bits/sched.h b/bits/sched.h
index e79db04..c943330 100644
--- a/bits/sched.h
+++ b/bits/sched.h
@@ -29,10 +29,17 @@
 #define SCHED_FIFO   1
 #define SCHED_RR     2

+/* EDF_implementation++ */
+#define SCHED_EDF            6
+/* EDF_implementation-- */
+
 /* Data structure to describe a process' schedulability.  */
 struct sched_param
 {
   int __sched_priority;
+  /* EDF_implementation++ */
+  struct timespec __sched_rel_deadline;
+  /* EDF_implementation-- */
 };

 #endif       /* need schedparam */
@@ -44,6 +51,9 @@ struct sched_param
 struct __sched_param
   {
     int __sched_priority;
+    /* EDF_implementation++ */
+     struct timespec __sched_rel_deadline;
+    /* EDF_implementation-- */
   };
 # undef __need_schedparam
 #endif
diff --git a/nptl/pthreadP.h b/nptl/pthreadP.h
index 197401a..59d69bb 100644
--- a/nptl/pthreadP.h
+++ b/nptl/pthreadP.h
@@ -609,7 +609,9 @@ extern void __wait_lookup_done (void)
attribute_hidden;
 static inline int
 check_sched_policy_attr (int pol)
 {
-  if (pol == SCHED_OTHER || pol == SCHED_FIFO || pol == SCHED_RR)
+    /* EDF_implementation++ */
+  if (pol == SCHED_OTHER || pol == SCHED_FIFO || pol == SCHED_RR ||
pol == SCHED_EDF)
+      /* EDF_implementation-- */
```

```
     return 0;

   return EINVAL;
diff --git a/nptl/pthread_create.c b/nptl/pthread_create.c
index 9d7f52f..4ef3a06 100644
--- a/nptl/pthread_create.c
+++ b/nptl/pthread_create.c
@@ -230,6 +230,9 @@ __free_tcb (struct pthread *pd)
 static int
 start_thread (void *arg)
 {
+     /* EDF_implementation++ */
+//   printf("PTHREAD - pthread_create.c - start_thread - init\n");
+     /* EDF_implementation-- */
   struct pthread *pd = (struct pthread *) arg;

 #if HP_TIMING_AVAIL
@@ -274,7 +277,9 @@ start_thread (void *arg)
       (void) INTERNAL_SYSCALL (rt_sigprocmask, err, 4, SIG_UNBLOCK,
&mask,
                                NULL, _NSIG / 8);
     }
-
+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - start_thread - line
278\n");
+  /* EDF_implementation-- */
   /* This is where the try/finally block should be created.  For
      compilers without that support we do use setjmp.  */
   struct pthread_unwind_buf unwind_buf;
@@ -285,51 +290,97 @@ start_thread (void *arg)

   int not_first_call;
   not_first_call = setjmp ((struct __jmp_buf_tag *)
unwind_buf.cancel_jmp_buf);
+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - start_thread - line
289\n");
+  /* EDF_implementation-- */
   if (__builtin_expect (! not_first_call, 1))
     {
+       /* EDF_implementation++ */
+//     printf("PTHREAD - pthread_create.c - start_thread - line
292\n");
+       /* EDF_implementation-- */
       /* Store the new cleanup handler info.  */
       THREAD_SETMEM (pd, cleanup_jmp_buf, &unwind_buf);
-
+      /* EDF_implementation++ */
+//      printf("PTHREAD - pthread_create.c - start_thread - line
295\n");
+      /* EDF_implementation-- */
```

```
       if (__builtin_expect (pd->stopped_start, 0))
     {
+       /* EDF_implementation++ */
+//             printf("PTHREAD - pthread_create.c - start_thread -
line 298\n");
+       /* EDF_implementation-- */
        int oldtype = CANCEL_ASYNC ();
-
+       /* EDF_implementation++ */
+//     printf("PTHREAD - pthread_create.c - start_thread - line
300\n");
+       /* EDF_implementation-- */
        /* Get the lock the parent locked to force synchronization.
*/
        lll_lock (pd->lock, LLL_PRIVATE);
+       /* EDF_implementation++ */
+//     printf("PTHREAD - pthread_create.c - start_thread - line
303\n");
+       /* EDF_implementation-- */
        /* And give it up right away.  */
        lll_unlock (pd->lock, LLL_PRIVATE);
-
+       /* EDF_implementation++ */
+//     printf("PTHREAD - pthread_create.c - start_thread - line
306\n");
+       /* EDF_implementation-- */
        CANCEL_RESET (oldtype);
+       /* EDF_implementation++ */
+//     printf("PTHREAD - pthread_create.c - start_thread - line
308\n");
+       /* EDF_implementation-- */
     }
-
+      /* EDF_implementation++ */
+//      printf("PTHREAD - pthread_create.c - start_thread - line
310\n");
+      /* EDF_implementation-- */
       LIBC_PROBE (pthread_start, 3, (pthread_t) pd, pd-
>start_routine, pd->arg);
-
+      /* EDF_implementation++ */
+//      printf("PTHREAD - pthread_create.c - start_thread - line
312\n");
+      /* EDF_implementation-- */
       /* Run the code the user provided.  */
 #ifdef CALL_THREAD_FCT
       THREAD_SETMEM (pd, result, CALL_THREAD_FCT (pd));
 #else
       THREAD_SETMEM (pd, result, pd->start_routine (pd->arg));
+      /* EDF_implementation++ */
+//      printf("PTHREAD - pthread_create.c - start_thread - finish
if section line 314\n");
```

```
+      /* EDF_implementation-- */
 #endif
     }
-
+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - start_thread - line
317\n");
+  /* EDF_implementation-- */
   /* Call destructors for the thread_local TLS variables.  */
 #ifndef SHARED
   if (&__call_tls_dtors != NULL)
 #endif
     __call_tls_dtors ();

+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - start_thread - line
324\n");
+  /* EDF_implementation-- */
+
   /* Run the destructor for the thread-local data.  */
   __nptl_deallocate_tsd ();

+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - start_thread - line
329\n");
+  /* EDF_implementation-- */
+
   /* Clean up any state libc stored in thread-local variables.  */
   __libc_thread_freeres ();

+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - start_thread - line
334\n");
+  /* EDF_implementation-- */
+
   /* If this is the last thread we terminate the process now.  We
      do not notify the debugger, it might just irritate it if there
      is no thread left.  */
-  if (__builtin_expect (atomic_decrement_and_test
(&__nptl_nthreads), 0))
-    /* This was the last thread.  */
+  if (__builtin_expect (atomic_decrement_and_test
(&__nptl_nthreads), 0)) {
+      /* EDF_implementation++ */
+//    printf("PTHREAD - pthread_create.c - start_thread - line 340
exit\n");
+      /* EDF_implementation-- */
+      /* This was the last thread.  */
     exit (0);
+  }

   /* Report the death of the thread if this is wanted.  */
```

```
   if (__builtin_expect (pd->report_events, 0))
@@ -354,11 +405,20 @@ start_thread (void *arg)
                                                                  pd, pd-
>nextevent));
          }

+       /* EDF_implementation++ */
+//     printf("PTHREAD - pthread_create.c - start_thread - thread
death line 368\n");
+       /* EDF_implementation-- */
        /* Now call the function to signal the event.  */
        __nptl_death_event ();
+       /* EDF_implementation++ */
+//     printf("PTHREAD - pthread_create.c - start_thread - line
371\n");
+       /* EDF_implementation-- */
      }
     }

+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - start_thread - line
375\n");
+  /* EDF_implementation// */
   /* The thread is exiting now.  Don't set this bit until after
we've hit
      the event-reporting breakpoint, so that td_thr_get_info on us
while at
      the breakpoint reports TD_THR_RUN state rather than
TD_THR_ZOMBIE.  */
@@ -433,7 +493,9 @@ start_thread (void *arg)
      The exit code is zero since in case all threads exit by calling
      'pthread_exit' the exit status must be 0 (zero).  */
   __exit_thread_inline (0);
-
+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - start_thread - finished
start_thread\n");
+  /* EDF_implementation-- */
   /* NOTREACHED */
   return 0;
 }
@@ -453,6 +515,9 @@ __pthread_create_2_1 (newthread, attr,
start_routine, arg)
   bool free_cpuset = false;
   if (iattr == NULL)
     {
+       /* EDF_implementation++ */
+//     printf("PTHREAD - pthread_create.c - __pthread_create_2_1 -
line 474\n");
+       /* EDF_implementation-- */
       lll_lock (__default_pthread_attr_lock, LLL_PRIVATE);
       default_attr = __default_pthread_attr;
```

```
       size_t cpusetsize = default_attr.cpusetsize;
@@ -476,22 +541,32 @@ __pthread_create_2_1 (newthread, attr,
start_routine, arg)
      }
       lll_unlock (__default_pthread_attr_lock, LLL_PRIVATE);
       iattr = &default_attr;
+      /* EDF_implementation++ */
+//      printf("PTHREAD - pthread_create.c - __pthread_create_2_1 -
line 498\n");
+      /* EDF_implementation-- */
     }

   struct pthread *pd = NULL;
   int err = ALLOCATE_STACK (iattr, &pd);
   int retval = 0;

+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - __pthread_create_2_1 - line
503\n");
+  /* EDF_implementation-- */
   if (__builtin_expect (err != 0, 0))
     /* Something went wrong.  Maybe a parameter of the attributes is
        invalid or we could not allocate memory.  Note we have to
        translate error codes.  */
     {
+       /* EDF_implementation++ */
+//     printf("PTHREAD - pthread_create.c - __pthread_create_2_1 -
line 509\n");
+       /* EDF_implementation-- */
       retval = err == ENOMEM ? EAGAIN : err;
       goto out;
     }
-
-
+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - __pthread_create_2_1 - line
513\n");
+  /* EDF_implementation-- */
   /* Initialize the TCB.  All initializations with zero should be
      performed in 'get_cached_stack'.  This way we avoid doing this
if
      the stack freshly allocated with 'mmap'.  */
@@ -528,16 +603,21 @@ __pthread_create_2_1 (newthread, attr,
start_routine, arg)
      is valid and what is not.  */
   pd->schedpolicy = self->schedpolicy;
   pd->schedparam = self->schedparam;
-
   /* Copy the stack guard canary.  */
 #ifdef THREAD_COPY_STACK_GUARD
   THREAD_COPY_STACK_GUARD (pd);
 #endif
```

```
+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - __pthread_create_2_1 - line
555\n");
+  /* EDF_implementation-- */
   /* Copy the pointer guard value.  */
 #ifdef THREAD_COPY_POINTER_GUARD
   THREAD_COPY_POINTER_GUARD (pd);
 #endif
+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - __pthread_create_2_1 - line
560\n");
+  /* EDF_implementation-- */

   /* Determine scheduling parameters for the thread.  */
   if (__builtin_expect ((iattr->flags & ATTR_FLAG_NOTINHERITSCHED)
!= 0, 0)
@@ -546,23 +626,37 @@ __pthread_create_2_1 (newthread, attr,
start_routine, arg)
       INTERNAL_SYSCALL_DECL (scerr);

       /* Use the scheduling parameters the user provided.  */
-      if (iattr->flags & ATTR_FLAG_POLICY_SET)
-    pd->schedpolicy = iattr->schedpolicy;
-      else if ((pd->flags & ATTR_FLAG_POLICY_SET) == 0)
+      if (iattr->flags & ATTR_FLAG_POLICY_SET) {
+       /* EDF_implementation++ */
+//             printf("PTHREAD - pthread_create.c -
__pthread_create_2_1 - line 570\n");
+       /* EDF_implementation-- */
+       pd->schedpolicy = iattr->schedpolicy;
+      } else if ((pd->flags & ATTR_FLAG_POLICY_SET) == 0)
     {
+       /* EDF_implementation++ */
+//             printf("PTHREAD - pthread_create.c -
__pthread_create_2_1 - line 574\n");
+       /* EDF_implementation-- */
       pd->schedpolicy = INTERNAL_SYSCALL (sched_getscheduler, scerr,
1, 0);
       pd->flags |= ATTR_FLAG_POLICY_SET;
     }

-      if (iattr->flags & ATTR_FLAG_SCHED_SET)
-    memcpy (&pd->schedparam, &iattr->schedparam,
-            sizeof (struct sched_param));
-      else if ((pd->flags & ATTR_FLAG_SCHED_SET) == 0)
+      if (iattr->flags & ATTR_FLAG_SCHED_SET){
+       /* EDF_implementation++ */
+//             printf("PTHREAD - pthread_create.c -
__pthread_create_2_1 - line 580\n");
+       /* EDF_implementation-- */
+       memcpy (&pd->schedparam, &iattr->schedparam,
```

```
+                     sizeof (struct sched_param));
+      } else if ((pd->flags & ATTR_FLAG_SCHED_SET) == 0)
      {
+       /* EDF_implementation++ */
+//             printf("PTHREAD - pthread_create.c -
__pthread_create_2_1 - line 585\n");
+       /* EDF_implementation-- */
        INTERNAL_SYSCALL (sched_getparam, scerr, 2, 0, &pd-
>schedparam);
        pd->flags |= ATTR_FLAG_SCHED_SET;
      }
-
+      /* EDF_implementation++ */
+//      printf("PTHREAD - pthread_create.c - __pthread_create_2_1 -
line 589\n");
+      /* EDF_implementation-- */
       /* Check for valid priorities.  */
       int minprio = INTERNAL_SYSCALL (sched_get_priority_min, scerr,
1,
                                       iattr->schedpolicy);
@@ -571,6 +665,9 @@ __pthread_create_2_1 (newthread, attr,
start_routine, arg)
       if (pd->schedparam.sched_priority < minprio
        || pd->schedparam.sched_priority > maxprio)
      {
+       /* EDF_implementation++ */
+//             printf("PTHREAD - pthread_create.c -
__pthread_create_2_1 - line 598\n");
+       /* EDF_implementation-- */
        /* Perhaps a thread wants to change the IDs and if waiting
           for this stillborn thread.  */
        if (__builtin_expect (atomic_exchange_acq (&pd->setxid_futex,
0)
@@ -580,22 +677,31 @@ __pthread_create_2_1 (newthread, attr,
start_routine, arg)
        __deallocate_stack (pd);

        retval = EINVAL;
+       /* EDF_implementation++ */
+//     printf("PTHREAD - pthread_create.c - __pthread_create_2_1 -
ERROR ON line 608\n");
+       /* EDF_implementation-- */
        goto out;
      }
     }
-
   /* Pass the descriptor to the caller.  */
   *newthread = (pthread_t) pd;

   LIBC_PROBE (pthread_create, 4, newthread, attr, start_routine,
arg);
-
```

```
+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - __pthread_create_2_1 - line
616\n");
+  /* EDF_implementation-- */
   /* Start the thread.  */
   retval = create_thread (pd, iattr, STACK_VARIABLES_ARGS);
+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - __pthread_create_2_1 - line
619\n");
+  /* EDF_implementation-- */

  out:
   if (__glibc_unlikely (free_cpuset))
     free (default_attr.cpuset);
-
+  /* EDF_implementation++ */
+//  printf("PTHREAD - pthread_create.c - __pthread_create_2_1 -
Successful thread creation\n");
+  /* EDF_implementation-- */
   return retval;
 }
 versioned_symbol (libpthread, __pthread_create_2_1, pthread_create,
GLIBC_2_1);
diff --git a/nptl/sysdeps/pthread/createthread.c
b/nptl/sysdeps/pthread/createthread.c
index 2a9a723..af6bd2a 100644
--- a/nptl/sysdeps/pthread/createthread.c
+++ b/nptl/sysdeps/pthread/createthread.c
@@ -128,7 +128,9 @@ do_clone (struct pthread *pd, const struct
pthread_attr *attr,
      {
        res = INTERNAL_SYSCALL (sched_setscheduler, err, 3, pd->tid,
                                pd->schedpolicy, &pd->schedparam);
-
+      /* EDF_implementation++ */
+//    printf("PTHREAD - createthread.c - do_clone - setting sched
params\n");
+      /* EDF_implementation-- */
        if (__builtin_expect (INTERNAL_SYSCALL_ERROR_P (res, err), 0))
          goto err_out;
      }
@@ -138,7 +140,9 @@ do_clone (struct pthread *pd, const struct
pthread_attr *attr,
      not yet have the flag set.  No need to set the global variable
      again if this is what we use.  */
   THREAD_SETMEM (THREAD_SELF, header.multiple_threads, 1);
-
+  /* EDF_implementation++ */
+//  printf("PTHREAD - createthread.c - do_clone - finished
do_clone\n");
+  /* EDF_implementation-- */
   return 0;
```

```
 }

@@ -248,6 +252,8 @@ create_thread (struct pthread *pd, const struct
pthread_attr *attr,
   if (res == 0 && stopped)
     /* And finally restart the new thread.  */
     lll_unlock (pd->lock, LLL_PRIVATE);
-
+  /* EDF_implementation++ */
+  printf("PTHREAD - createthread.c - finished create_thread\n");
+  /* EDF_implementation-- */
   return res;
 }
diff --git a/nptl/sysdeps/unix/sysv/linux/x86/bits/pthreadtypes.h
b/nptl/sysdeps/unix/sysv/linux/x86/bits/pthreadtypes.h
index 28e5144..b01a9de 100644
--- a/nptl/sysdeps/unix/sysv/linux/x86/bits/pthreadtypes.h
+++ b/nptl/sysdeps/unix/sysv/linux/x86/bits/pthreadtypes.h
@@ -43,7 +43,12 @@
 #  define __SIZEOF_PTHREAD_BARRIERATTR_T 4
 # endif
 #else
-# define __SIZEOF_PTHREAD_ATTR_T 36
+/* EDF_implementation++ */
+/* We change the size of pthread_attr_t from 36
+ * to 40  because we add the rel_deadline parameter*/
+# define __SIZEOF_PTHREAD_ATTR_T 40
+/* # define __SIZEOF_PTHREAD_ATTR_T 36 */
+/* EDF_implementation++ */
 # define __SIZEOF_PTHREAD_MUTEX_T 24
 # define __SIZEOF_PTHREAD_MUTEXATTR_T 4
 # define __SIZEOF_PTHREAD_COND_T 48
diff --git a/posix/sched.h b/posix/sched.h
index f7da255..3d0a551 100644
--- a/posix/sched.h
+++ b/posix/sched.h
@@ -41,7 +41,9 @@ typedef __pid_t pid_t;
 #include <bits/sched.h>
 /* Define the real names for the elements of `struct sched_param'.
*/
 #define sched_priority      __sched_priority
-
+/* EDF_implementation++ */
+#define sched_rel_deadline   __sched_rel_deadline
+/* EDF_implementation-- */

 __BEGIN_DECLS

diff --git a/sysdeps/unix/sysv/linux/bits/sched.h
b/sysdeps/unix/sysv/linux/bits/sched.h
index 555529d..2415d26 100644
--- a/sysdeps/unix/sysv/linux/bits/sched.h
```

```
+++ b/sysdeps/unix/sysv/linux/bits/sched.h
@@ -28,6 +28,9 @@
 #define SCHED_OTHER          0
 #define SCHED_FIFO           1
 #define SCHED_RR             2
+/* EDF_implementation++ */
+#define SCHED_EDF            6
+/* EDF_implementation-- */
 #ifdef __USE_GNU
 # define SCHED_BATCH         3
 # define SCHED_IDLE          5
@@ -72,6 +75,9 @@
 struct sched_param
   {
     int __sched_priority;
+    /* EDF_implementation++ */
+    struct timespec __sched_rel_deadline;
+    /* EDF_implementation-- */
   };

 __BEGIN_DECLS
@@ -103,6 +109,9 @@ __END_DECLS
 struct __sched_param
   {
     int __sched_priority;
+    /* EDF_implementation++ */
+     struct timespec __sched_rel_deadline;
+    /* EDF_implementation-- */
   };
 # undef __need_schedparam
 #endif
```

9.2 check_preempt_curr_rt

```
check_preempt_curr_rt(...) {
#ifdef CONFIG_SCHED_EDF_POLICY
if the task policy is SCHED_EDF
if the absolute deadline of the task woken is less than the running
task one
invoke the function resched_task() for reschedule tasks
else
if the priority of the task woken is higher than the running task one
invoke the function resched_task() for reschedule tasks
#else
if the priority of the woken task is higher than the running task one
invoke the function resched_task() for reschedule tasks
#endif
}
```

9.3 enqueue_task_rt

```
enqueue_task_rt(…) {
...
#ifdef CONFIG_SCHED_EDF_POLICY
if the task policy is SCHED_EDF
get current time
update task absolute deadline according to the current time and its
relative deadline
generate a trace of the absolute deadline calculated
#endif
...
}
```

9.4 pick_next_rt_entity

```
pick_next_rt_entity(...) {
get the rt priority levels
get the first priority level index containing rt tasks
#ifdef CONFIG_SCHED_EDF_POLICY
loop through each run queue
loop through each task in the rt array of each priority level
if index is not a RT priority level
exit loop
loop through each task on the priority level queue
if task policy is not SCHED_EDF
exit loop
if task relative deadline is less or equal to 0
exit loop
if there is a next task in the current queue
if absolute deadline of current task in the loop is less than the
next one
next task to be executed is current
else
next task to be executed is current
if the current priority level rt array is empty
find the index of the next priority level rt array
go to the beginning of the loop
if there is not selected a next task to be executed
find the index of the next priority level rt array in the current run
queue
select the next rt array of current run queue
#else
select the rt array according to the index
next task to be executed is the first stored in the array
```

```
#endif
return the next task to be executed
}
```

9.5 prio_changed_rt

```
prio_changed_rt(...) {
...
if task is the current running one
...
else
#ifdef CONFIG_SCHED_EDF_POLICY
if task policy is SCHED_EDF
if task absolute deadline is less than running task one
need to re-schedule running task
exit
else
#endif
if task priority is less than running task one
need to re-schedule running task
}
```

9.6 set_curr_task_rt

```
set_curr_task_rt() {
select the running task
...
#ifdef CONFIG_SCHED_EDF_POLICY
if the task policy is SCHED_EDF
get current time
update task absolute deadline according to the current time and its
relative deadline
generates a trace of the absolute deadline calculated
#endif
...
}
```

9.7 switched_to_rt

```
switched_to_rt(...) {
...
if task is in the active run queue and is not the same as the current
running one
```

```
...
else
#ifdef CONFIG_SCHED_EDF_POLICY
...
if the switched task policy is SCHED_EDF
if task absolute deadline is less than running task
need to re-schedule running task
#endif
if task priority is higher than the running task one
need to re-schedule running task
}
```

9.8 sched_edf_abs_deadline-prototype

```
#ifdef CONFIG_SCHED_EDF_POLICY
TRACE_EVENT(sched_edf_abs_deadline,
TP_PROTO(struct task_struct *p),
TP_ARGS(p),
TP_STRUCT__entry(
__array( char, comm, TASK_COMM_LEN)
__field( pid_t, pid)
__field( int, prio)
__field( long, abs_deadline_tv_sec)
__field( long, abs_deadline_tv_nsec)),
TP_fast_assign(
memcpy(__entry->comm, p->comm, TASK_COMM_LEN);
__entry->pid = p->pid;
__entry->prio = p->prio;
__entry->abs_deadline_tv_sec = p->abs_deadline.tv_sec;
__entry->abs_deadline_tv_nsec = p->abs_deadline.tv_nsec;),
TP_printk("comm=%s pid=%d prio=%d abs_deadline=%ld.
%ld", entry->comm, __entry->pid, __entry->prio,,
__entry->abs_deadline_tv_sec,__entry->abs_deadline_tv_nsec));
#endif
```

9.9 edf3Threads.c

```
/*
* main.c
*
*
* gcc monotonicThreads.c -o monotonicThreads.o -lrt -lpthread
* ./monotonicThreads.o
*/
#define _GNU_SOURCE
#include <stdio.h>
```

```
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <sys/mman.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include "times.h"
#define MAX_SAFE_STACK (8*1024) /* The maximum stack size which is
guranteed safe to access without
faulting */
struct timespec initialTime;
void setaffinity(int cpu);
void * inThread(void *args);
void * prThread(void *args);
void * ouThread(void *args);
void stack_prefault(void);
void timespec_add_us(struct timespec *t, long us);
int timespec_cmp(struct timespec *a, struct timespec *b);
int main(int argc, char **argv) {
/*Declare variables*/
pthread_t in, pr, out;
pthread_attr_t inAttr, prAttr, ouAttr;
int fifo_min_priority, fifo_max_priority;
struct sched_param param1, param2, param3;
int status;
struct timespec t = {0,0};
struct timespec clock_resolution = {-1,-1};
clock_gettime(CLOCK_MONOTONIC,&clock_resolution);
printf("Clock resolution is %ld seconds, %06ld
nanoseconds\n",clock_resolution.tv_sec, clock_resolution.tv_nsec);
/* Lock memory */
if(mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
perror("mlockall failed");
exit(-2);
}
/* Pre-fault our stack */
stack_prefault();
status = pthread_attr_init(&inAttr);
if (status != 0) {
perror("Init attr1");
exit(2);
}
status =
pthread_attr_setinheritsched(&inAttr,PTHREAD_EXPLICIT_SCHED);
if (status != 0) {
perror("EXPLICIT attr1");
exit(2);
}
status = pthread_attr_init(&prAttr);
if (status != 0) {
perror("Init attr2");
```

```
exit(2);
}
status =
pthread_attr_setinheritsched(&prAttr,PTHREAD_EXPLICIT_SCHED);
if (status != 0) {
perror("EXPLICIT attr2");
exit(2);
}
status = pthread_attr_init(&ouAttr);
if (status != 0) {
perror("Init attr3");
exit(2);
}
status =
pthread_attr_setinheritsched(&ouAttr,PTHREAD_EXPLICIT_SCHED);
if (status != 0) {
perror("EXPLICIT attr3");
exit(2);
}
#if defined (_POSIX_THREAD_PRIORITY_SCHEDULING) && !defined (sun)
status = pthread_attr_setschedpolicy(&inAttr, SCHED_EDF);
if (status != 0) {
perror("Unable to set SCHED_FIFO policy to Thread 1.\n");
exit(1);
}
status = pthread_attr_setschedpolicy(&prAttr, SCHED_EDF);
if (status != 0) {
perror("Unable to set SCHED_FIFO policy to Thread 2.\n");
exit(1);
}
status = pthread_attr_setschedpolicy(&ouAttr, SCHED_EDF);
if (status != 0) {
perror("Unable to set SCHED_FIFO policy to Thread 3.\n");
exit(1);
} else {
fifo_min_priority = sched_get_priority_min(SCHED_EDF);
if (fifo_min_priority == -1) {
perror("Get SCHED_FIFO min priority");
exit(1);
}
fifo_max_priority = sched_get_priority_max(SCHED_EDF);
if (fifo_max_priority == -1) {
perror("Get SCHED_FIFO max priority");
exit(1);
}
printf("MAX: %d\nMIN: %d\n",fifo_max_priority,fifo_min_priority);
param1.sched_priority = fifo_max_priority;
param2.sched_priority = fifo_max_priority;
param3.sched_priority = fifo_max_priority;
t.tv_nsec = 700000000;
param1.sched_rel_deadline = t;
status = pthread_attr_setschedparam(&inAttr, &param1);
```

```
if (status != 0) {
perror("Unable to set params to Thread 1.\n");
}
printf("SCHED_PARAM 1\n");
t.tv_nsec = 600000000;
param2.sched_rel_deadline = t;
status = pthread_attr_setschedparam(&prAttr, &param2);
if (status != 0) {
perror("Unable to set params to Thread 2.\n");
}
printf("SCHED_PARAM 2\n");
t.tv_nsec = 400000000;
param3.sched_rel_deadline = t;
status = pthread_attr_setschedparam(&ouAttr, &param3);
if (status != 0) {
perror("Unable to set params to Thread 3.\n");
}
printf("SCHED_PARAM 3\n");
}
#else
printf("Priority scheduling not supported\n");
#endif
printf("--------------------------\n");
//setaffinity(0);
initialTime = clock_resolution;
initialTime.tv_sec = initialTime.tv_sec + 2;
/* Create threads*/
printf("THREAD CREATE BEFORE\n");
pthread_create(&in, &inAttr, (void *) &inThread, NULL);
printf("THREAD CREATE 1\n");
pthread_create(&pr, &prAttr, (void *) &prThread, NULL);
printf("THREAD CREATE 2\n");
pthread_create(&out, &ouAttr, (void *) &ouThread, NULL);
printf("THREAD CREATE 3\n");
pthread_join(in, NULL);
pthread_join(pr, NULL);
pthread_join(out, NULL);
printf("MAIN THREAD FINISHED 2\n");
pthread_exit(NULL);
}
void stack_prefault(void) {
unsigned char dummy[MAX_SAFE_STACK];
memset(&dummy, 0, MAX_SAFE_STACK);
return;
}
void setaffinity(int cpu) {
struct sched_param sched_param;
//struct timespec rel={0,0};
cpu_set_t mask;
CPU_ZERO(&mask);
CPU_SET(cpu, &mask);
printf("AFFINITY BEFORE\n");
```

```
if (sched_setaffinity(0, sizeof(mask), &mask) == -1) {
perror("Coult not set affinity");
}
printf("AFFINITY LATER\n");
}
void timespec_add_us(struct timespec *t, long us) {
t->tv_nsec += us*1000;
if (t->tv_nsec > 1000000000) {
t->tv_nsec = t->tv_nsec - 1000000000;// + ms*1000000;
t->tv_sec += 1;
}
}
int timespec_cmp(struct timespec *a, struct timespec *b) {
if (a->tv_sec > b->tv_sec) return 1;
else if (a->tv_sec < b->tv_sec) return -1;
else if (a->tv_sec == b->tv_sec) {
if (a->tv_nsec > b->tv_nsec) return 1;
else if (a->tv_nsec == b->tv_nsec) return 0;
else return -1;
}
}
void * inThread(void *args) {
/* Entry for inThread */
/* Create variables */
struct timespec periodActivation, nextActivation, now;
int period_time,i,j,k,x;
int count=0;
double y;
period_time=700000000;
periodActivation.tv_sec = 0;
periodActivation.tv_nsec = period_time;
nextActivation = initialTime;
clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &nextActivation,
NULL);
for (i=0; i<=700; i++)
for (j=i; j<=550; j++)
for (k=1; k<=500; k++)
x=5000+700+600;
count++;
while (count<24) {
clock_gettime(CLOCK_MONOTONIC, &now);
timespec_add(&nextActivation, &periodActivation);
clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &nextActivation,
NULL);
for (i=0; i<=700; i++)
for (j=i; j<=550; j++)
for (k=1; k<=500; k++)
x=5000+700+600;
count++;
}
pthread_exit(NULL);
}
```

```
void * prThread(void *args) {
/* Entry for prThread */
/* Create variables */
struct timespec periodActivation, nextActivation, now;
int period_time,i,j,k,x;
int count=0;
period_time=600000000;
periodActivation.tv_sec = 0;
periodActivation.tv_nsec = period_time;
nextActivation = initialTime;
clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &nextActivation,
NULL);
for (i=0; i<=500; i++)
for (j=i; j<=400; j++)
for (k=1; k<=300; k++)
x=5000+700+600;
count++;
while (count<28) {
clock_gettime(CLOCK_MONOTONIC, &now);
timespec_add(&nextActivation, &periodActivation);
clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &nextActivation,
NULL);
for (i=0; i<=500; i++)
for (j=i; j<=400; j++)
for (k=1; k<=300; k++)
x=5000+700+600;
count++;
}
pthread_exit(NULL);
}
void * ouThread(void *args) {
/* Entry for ouThread */
/* Create variables */
struct timespec periodActivation, nextActivation, now;
int period_time,i,j,k,x;
int count=0;
period_time=400000000;
periodActivation.tv_sec = 0;
periodActivation.tv_nsec = period_time;
nextActivation = initialTime;
clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &initialTime, NULL);
for (i=0; i<=700; i++)
for (j=i; j<=600; j++)
for (k=1; k<=500; k++)
x=5000+700+600;
count++;
while (count<42) {
clock_gettime(CLOCK_MONOTONIC, &now);
timespec_add(&nextActivation, &periodActivation);
clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &nextActivation,
NULL);
for (i=0; i<=700; i++)
```

```
for (j=i; j<=600; j++)
for (k=1; k<=500; k++)
x=5000+700+600;
count++; }
pthread_exit(NULL);
}
```

9.10 rm3Threads.c

```
/*
* main.c
*
*
* gcc monotonicThreads.c -o monotonicThreads.o -lrt -lpthread
* ./monotonicThreads.o
*/
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <sys/mman.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include "times.h"
#define MAX_SAFE_STACK (8*1024) /* The maximum stack size which is
guranteed safe to access without
faulting */
struct timespec initialTime;
void setaffinity(int cpu);
void * inThread(void *args);
void * prThread(void *args);
void * ouThread(void *args);
void stack_prefault(void);
void timespec_add_us(struct timespec *t, long us);
int timespec_cmp(struct timespec *a, struct timespec *b);
int main(int argc, char **argv) {
/*Declare variables*/
pthread_t in, pr, out;
pthread_attr_t inAttr, prAttr, ouAttr;
int fifo_min_priority, fifo_max_priority;
struct sched_param param1, param2, param3;
int status;
struct timespec t = {0,0};
struct timespec clock_resolution = {-1,-1};
clock_gettime(CLOCK_MONOTONIC,&clock_resolution);
printf("Clock resolution is %ld seconds, %06ld
nanoseconds\n",clock_resolution.tv_sec, clock_resolution.tv_nsec);
```

```
/* Lock memory */
if(mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
perror("mlockall failed");
exit(-2);
}
/* Pre-fault our stack */
stack_prefault();
status = pthread_attr_init(&inAttr);
if (status != 0) {
perror("Init attr1");
exit(2);
}
status =
pthread_attr_setinheritsched(&inAttr,PTHREAD_EXPLICIT_SCHED);
if (status != 0) {
perror("EXPLICIT attr1");
exit(2);
}
status = pthread_attr_init(&prAttr);
if (status != 0) {
perror("Init attr2");
exit(2);
}
status =
pthread_attr_setinheritsched(&prAttr,PTHREAD_EXPLICIT_SCHED);
if (status != 0) {
perror("EXPLICIT attr2");
exit(2);
}
status = pthread_attr_init(&ouAttr);
if (status != 0) {
perror("Init attr3");
exit(2);
}
status =
pthread_attr_setinheritsched(&ouAttr,PTHREAD_EXPLICIT_SCHED);
if (status != 0) {
perror("EXPLICIT attr3");
exit(2);
}
#if defined (_POSIX_THREAD_PRIORITY_SCHEDULING) && !defined (sun)
status = pthread_attr_setschedpolicy(&inAttr, SCHED_FIFO);
if (status != 0) {
perror("Unable to set SCHED_FIFO policy to Thread 1.\n");
exit(1);
}
status = pthread_attr_setschedpolicy(&prAttr, SCHED_FIFO);
if (status != 0) {
perror("Unable to set SCHED_FIFO policy to Thread 2.\n");
exit(1);
}
status = pthread_attr_setschedpolicy(&ouAttr, SCHED_FIFO);
```

```
if (status != 0) {
perror("Unable to set SCHED_FIFO policy to Thread 3.\n");
exit(1);
} else {
fifo_min_priority = sched_get_priority_min(SCHED_EDF);
if (fifo_min_priority == -1) {
perror("Get SCHED_FIFO min priority");
exit(1);
}
fifo_max_priority = sched_get_priority_max(SCHED_EDF);
if (fifo_max_priority == -1) {
perror("Get SCHED_FIFO max priority");
exit(1);
}
printf("MAX: %d\nMIN: %d\n",fifo_max_priority,fifo_min_priority);
param1.sched_priority = 97;
param2.sched_priority = 98;
param3.sched_priority = 99;
status = pthread_attr_setschedparam(&inAttr, &param1);
if (status != 0) {
perror("Unable to set params to Thread 1.\n");
}
printf("SCHED_PARAM 1\n");
status = pthread_attr_setschedparam(&prAttr, &param2);
if (status != 0) {
perror("Unable to set params to Thread 2.\n");
}
printf("SCHED_PARAM 2\n");
status = pthread_attr_setschedparam(&ouAttr, &param3);
if (status != 0) {
perror("Unable to set params to Thread 3.\n");
}
printf("SCHED_PARAM 3\n");
}
#else
printf("Priority scheduling not supported\n");
#endif
printf("--------------------------\n");
//setaffinity(0);
initialTime = clock_resolution;
initialTime.tv_sec = initialTime.tv_sec + 2;
/* Create threads*/
printf("THREAD CREATE BEFORE\n");
pthread_create(&in, &inAttr, (void *) &inThread, NULL);
printf("THREAD CREATE 1\n");
pthread_create(&pr, &prAttr, (void *) &prThread, NULL);
printf("THREAD CREATE 2\n");
pthread_create(&out, &ouAttr, (void *) &ouThread, NULL);
printf("THREAD CREATE 3\n");
pthread_join(in, NULL);
pthread_join(pr, NULL);
pthread_join(out, NULL);
```

```
printf("MAIN THREAD FINISHED 2\n");
pthread_exit(NULL);
}
void stack_prefault(void) {
unsigned char dummy[MAX_SAFE_STACK];
memset(&dummy, 0, MAX_SAFE_STACK);
return;
}
void setaffinity(int cpu) {
struct sched_param sched_param;
//struct timespec rel={0,0};
cpu_set_t mask;
CPU_ZERO(&mask);
CPU_SET(cpu, &mask);
printf("AFFINITY BEFORE\n");
if (sched_setaffinity(0, sizeof(mask), &mask) == -1) {
perror("Coult not set affinity");
}
printf("AFFINITY LATER\n");
}
void timespec_add_us(struct timespec *t, long us) {
t->tv_nsec += us*1000;
if (t->tv_nsec > 1000000000) {
t->tv_nsec = t->tv_nsec - 1000000000;// + ms*1000000;
t->tv_sec += 1;
}
}
int timespec_cmp(struct timespec *a, struct timespec *b) {
if (a->tv_sec > b->tv_sec) return 1;
else if (a->tv_sec < b->tv_sec) return -1;
else if (a->tv_sec == b->tv_sec) {
if (a->tv_nsec > b->tv_nsec) return 1;
else if (a->tv_nsec == b->tv_nsec) return 0;
else return -1;
}
}
void * inThread(void *args) {
   /* Entry for inThread */
   /* Create variables */
   struct timespec periodActivation, nextActivation, now;
   int period_time,i,j,k,x;
   int count=0;
   double y;
   period_time=700000000;
   periodActivation.tv_sec = 0;
   periodActivation.tv_nsec = period_time;
   nextActivation = initialTime;
   clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &nextActivation,
NULL);
   for (i=0; i<=700; i++)
      for (j=i; j<=550; j++)
         for (k=1; k<=500; k++)
```

```
          x=5000+700+600;
   count++;
   while (count<24) {
      clock_gettime(CLOCK_MONOTONIC, &now);
      timespec_add(&nextActivation, &periodActivation);
      clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &nextActivation,
NULL);
      for (i=0; i<=700; i++)
         for (j=i; j<=550; j++)
            for (k=1; k<=500; k++)
               x=5000+700+600;
      count++;
   }
   pthread_exit(NULL);
}
void * prThread(void *args) {
   /* Entry for prThread */
   /* Create variables */
   struct timespec periodActivation, nextActivation, now;
   int period_time,i,j,k,x;
   int count=0;
   period_time=600000000;
   periodActivation.tv_sec = 0;
   periodActivation.tv_nsec = period_time;
   nextActivation = initialTime;
   clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &nextActivation,
NULL);
      for (i=0; i<=500; i++)
         for (j=i; j<=400; j++)
            for (k=1; k<=300; k++)
               x=5000+700+600;
   count++;
   while (count<28) {
      clock_gettime(CLOCK_MONOTONIC, &now);
      timespec_add(&nextActivation, &periodActivation);
      clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &nextActivation,
NULL);
      for (i=0; i<=500; i++)
         for (j=i; j<=400; j++)
            for (k=1; k<=300; k++)
               x=5000+700+600;
      count++;
   }
   pthread_exit(NULL);
}
void * ouThread(void *args) {
   /* Entry for inThread */
   /* Create variables */
   struct timespec periodActivation, nextActivation, now;
   int period_time,i,j,k,x;
   int count=0;
   period_time=400000000;
```

```
   periodActivation.tv_sec = 0;
   periodActivation.tv_nsec = period_time;
   nextActivation = initialTime;
   clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &initialTime,
NULL);
   for (i=0; i<=700; i++)
      for (j=i; j<=600; j++)
         for (k=1; k<=500; k++)
            x=5000+700+600;
   count++;
   while (count<42) {
      clock_gettime(CLOCK_MONOTONIC, &now);
      timespec_add(&nextActivation, &periodActivation);
      clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &nextActivation,
NULL);
      for (i=0; i<=700; i++)
         for (j=i; j<=600; j++)
            for (k=1; k<=500; k++)
               x=5000+700+600;
      count++; }
   pthread_exit(NULL);
}
```

9.11 times.h

```
/*
* times.h
*/
#ifndef TIME_H_
#define TIME_H_
#define NSEC_PER_SEC 1000000000L
#define timespec_normalize(t){\
   if((t)->tv_nsec >= NSEC_PER_SEC){\
      (t)->tv_nsec -= NSEC_PER_SEC;\
      (t)->tv_sec++;\
   } else if((t)->tv_nsec < 0){\
      (t)->tv_nsec += NSEC_PER_SEC;\
      (t)->tv_sec--;\
   }\
}
#define timespec_add(t1, t2) do{\
   (t1)->tv_nsec += (t2)->tv_nsec;\
   (t1)->tv_sec += (t2)->tv_sec;\
   timespec_normalize(t1);\
   }while (0);
   #define timespec_sub(t1, t2){\
   (t1)->tv_nsec -= (t2)->tv_nsec;\
   (t1)->tv_sec -= (t2)->tv_sec;\
   timespec_normalize(t1);\
```

```
}while (0);
#endif /* TIME_H_ */
```