# Laboratorio #5

Fecha de Entrega: 4 de mayo, 2021.

Descripción: en este laboratorio se programará y agregará una política de calendarización de CPU a un sistema Linux, con lo que se visitarán los componentes de su *kernel* involucrados en la calendarización de procesos, profundizando el ejemplo de clase sobre calendarización en Linux. Deberá entregar un archivo con respuestas a las preguntas planteadas en este documento, así como los archivos requeridos en ciertos incisos. Este laboratorio se basa en un tutorial en tres partes por Paulo Baltarejo Sousa y Luis Lino Ferreira (ver fuente al final).

Materiales: necesitará Ubuntu 8.04 *a.k.a. Hardy Heron* ([http://old-](http://old-/) releases.ubuntu.com/releases/hardy/ubuntu-8.04.4-desktop-i386.iso); y el *kernel* 2.6.24 de Linux (se descargará durante el laboratorio). Descargue también el material del tutorial de Sousa y Ferreira, en https://sourceforge.net/projects/linuxedfschedul/files/?source=navbar.

Contenido:

1. Descargue Ubuntu *Hardy Heron* a su computadora y cree una máquina virtual con este sistema operativo. Cuide que la plataforma sea de 32 bits y asígnele memoria RAM y espacio en disco para que el sistema funcione cómodamente.
2. Inicie su sistema Ubuntu. En las fuentes descargables del tutorial indicadas al principio de este documento se encuentra un conjunto de archivos de código llamado tasks.tar.bz2. Extraiga el contenido de este archivo en su máquina virtual y ejecute, mediante una terminal ubicada en el directorio de la extracción, el comando:

sudo ./casio\_system system > pre\_casio.txt.

Esto almacenará los resultados de la ejecución en un archivo de texto.

1. Necesitamos descargar e instalar unos paquetes, pero, por ser ésta una versión bastante vieja de Ubuntu, debemos redirigir el manejador de paquetes apt-get a los repositorios de versiones antiguas. Diríjase al directorio /etc/apt/ y cree una copia de *backup* del archivo sources.list.
2. Abra una terminal y ejecute el siguiente comando:

sudo sed –i –re ‘s/([a-z]{2}\.)?archive.ubuntu.com| \ security.ubuntu.com/old-releases.ubuntu.com/g’ /etc/apt/sources.list

Como alternativa, abra el archivo /etc/apt/sources.list. En él reemplace el inicio de las direcciones que comienzan con [http://archive.ubuntu.com](http://archive.ubuntu.com/) o [http://security.ubuntu.com](http://security.ubuntu.com/) por [http://old-releases.ubuntu.com.](http://old-releases.ubuntu.com/)

1. A continuación, actualizaremos APT y prepararemos el ambiente para compilar el *kernel* más adelante. Ejecute los siguientes comandos:

sudo apt-get update

sudo apt-get install build-essential sudo apt-get install libncurses5-dev sudo apt-get install kernel-package

1. Luego, estableceremos nuestra área de trabajo. Cree dos carpetas en el directorio /home llamadas scheduler\_dev y scheduler. Asegúrese de que su usuario es dueño de ambas carpetas (con ls -Al) o ejecute la siguiente instrucción para asignarlo como tal:

sudo chown –R su\_usuario\_aquí scheduler{,\_dev}

1. Ingrese a scheduler\_dev y descargue el *kernel* 2.6.24 de Linux con el siguiente comando:

sudo wget https://mirrors.edge.kernel.org/pub/linux/kernel/v2.6/linux- 2.6.24.tar.bz2 --no-check-certificate

Extraiga el contenido de este paquete usando el comando tar -xvjf seguido del nombre del archivo descargado. Cámbiele el nombre a la carpeta que se produce con la extracción a linux- 2.6.24-casio. En adelante nos referiremos a esta carpeta donde está el *kernel* extraído como kernel\_dir.

1. Para agregar una política de calendarización a Linux primero será necesario registrarla como una opción en el menú de configuración del *kernel*. Cree un *backup* de, y abra para modificación, el archivo kernel\_dir/arch/x86/Kconfig, y agregue en alguna ubicación fácil de hallar las siguientes instrucciones:

**Text

Description automatically generated**

menu “CASIO Scheduler” config SCHED\_CASIO\_POLICY

bool “CASIO scheduling policy” default y

endmenu

**Figura 1.** Evidencia de elaboración del laboratorio.

**Nota**: CASIO son las siglas para el nombre del curso que desarrolló esta política de calendarización, correspondientes a *Conceitos Avançados de Sistemas Operativos*.

Aunque ya ha visto bastante código en C y probablemente tenga experiencia previa con el lenguaje, es importante conocer algunas de sus características para que el código que se provea en este laboratorio no sea copiado ciegamente, sino entendido en el proceso. Por ello, investigue y resuma:

* **Funcionamiento y sintaxis de uso de structs.**

En C o C++ *struct* es una palabra clave que permite definir tipos de dato personalizados, mediante la agrupación de variables relacionadas de diferentes tipos en un solo tipo. (GeeksforGeeks, 2022)

Para crear una estructura basta con utilizar la palabra struct darle un nombre a dicha estructura y definir sus miembros en dentro de corchetes. Algo que no se había mencionado antes es que cada una de las variables en la estructura es conocido como un miembro de la estructura. (W3 Schools, s. f.)

Map

Description automatically generated

**Figura 2.**Ejemplo de creación de una estructura en C. (GeeksforGeeks, 2022)

Graphical user interface, text, application, chat or text message

Description automatically generated

**Figura 3.** Ejemplo práctico de uso de estructura. (GeeksforGeeks, 2022)

* **Propósito y directivas del preprocesador.**

Las directivas son instrucciones que se le dan al compilador para que tome medidas específicas, estas generalmente se utilizan para manejar fácilmente los códigos origen, es decir, que sean fáciles de editar y compilar en cualquier entorno. (Corob, 2022)

Según Corob, en *Directivas de preprocesador* las directivas reconocidas por el preprocesador son:

* + - #define
    - #elif
    - #else
    - #endif
    - #error
    - #if
    - #ifdef
    - #ifndef
    - #import
    - #include
    - #line
    - #pragma
    - #undef
    - #using

Como se puede notar, todas las directivas están precedidas por un símbolo #, estás líneas se incluyen dentro del código. Cuando el preprocesador hace la revisión antes de la compilación, si encuentra directivas las resuelve antes de la compilación y generación de código.(*Preprocessor directives - C++ Tutorials*, s. f.)

* **Diferencia entre \* y & en el manejo de referencias a memoria (punteros).**

Para establecer la diferencia entre el operador \* y &, primero es necesario establecer la definición de un puntero. Un puntero no es más que una variable que apunta a una dirección de memoria en la cual se encuentra el operando al cual se desea acceder. (González, 2013)

El operador **&** proporciona la dirección de memoria de un objeto/variable, mientras que el operador \*, también conocido como operador indirección, devuelve el contenido de una posición de memoria. (González, 2013)

**Text

Description automatically generated with medium confidence**

**Figura 4.** Definición de un puntero. (González, 2013)

Text

Description automatically generated

**Figura 5.** Diferencia práctica entre \* y &. (González, 2013)

* **Propósito y modo de uso de APT y dpkg**

APT hace referencia a *Advance Packaging*  Tool. Es un manejador de paquetes que gestiona automáticamente las dependencias y otras operaciones en paquetes ya existentes en el sistema para poder instalar nuevos paquetes. (*APT*, s. f.)

Background pattern

Description automatically generated with low confidence

**Figura 6.** Algunos modos de uso de APT. (*APT*, s. f.)

Al igual que APT dpkg es un gestor de paquetes, si embargo de un tipo de paquetes en específico, estos son los paquetes debian. En el manual de Ubuntu, se define como una “herramienta para instalar, compilar, eliminar y manipular los paquetes Debian”. Es un comando, que con una variedad de opciones permite desde instalar un fichero o paquete hasta remover arquitecturas. (Canonical, s. f.)

Text

Description automatically generated with medium confidence

**Figura 7.** Uso de dpkg para instalación de un paquete debian.

1. A continuación, cree un *backup* de, y abra, el archivo

kernel\_dir/include/linux/sched.h. Modifíquelo de la siguiente manera:

|  |  |  |
| --- | --- | --- |
| ...  #define | SCHED\_NORMAL | 0 |
| #define | SCHED\_FIFO | 1 |
| #define | SCHED\_RR | 2 |
| #define | SCHED\_BATCH | 3 |
| /\* SCHED\_ISO: reserved but not implemented yet \*/ #define SCHED\_IDLE 5  #ifdef CONFIG\_SCHED\_CASIO\_POLICY  #define SCHED\_CASIO 6  #endif  #ifdef KERNEL  ... | | |

Text

Description automatically generated

**Figura 8.** Evidencia de elaboración del laboratorio.

Note que lo que este extracto de código le indica con los colores es que agregue la parte de

#ifdef luego de #define SCHED\_IDLE 5.

1. En el archivo /usr/include/bits/sched.h realice la siguiente modificación:

...

# define SCHED\_BATCH 3 #endif

#define SCHED\_CASIO

6

Text

Description automatically generated

**Figura 9.** Evidencia de elaboración del laboratorio.

* + **¿Cuál es el propósito de los archivos sched.h modificados?**

Los archivos sched.h contienen los parámetros de calendarización requeridos para la implementación de cada política de calendarización implementada en el sistema, por lo tal, al modificarlos, esto da paso a poder agregar nuevas políticas de calendarización al definir sus parámetros en dicho documento. (The Single UNIX, s. f.)

* + **¿Cuál es el propósito de la definición incluida y las definiciones existentes en el archivo?**

EL propósito de las definiciones existentes y la definición que se agrega, tal como lo indica el comentario a son los algoritmos de calendarización, básicamente es la definición de cada uno de los algoritmos de calendarización existentes.

Tomemos en cuenta que la directiva #define crea una macro, la cual es la asociación de un identificador con un token (Corob, 2021). Por ejemplo, para el caso del identificador SCHED\_OTHER el valor asociado es 0, esto quiere decir que cada vez que aparezca SCHED\_OTHER en el código será reemplazado por 0. Esto se puede ver de forma análoga como la definición de variables en un código en Python.

1. En kernel\_dir/include/linux/sched.h busque la definición de la estructura task\_struct (debería estar en la línea 921). Se agregarán a ella los parámetros con los que se relacionará una *task* general con una *task* calendarizada por nuestra nueva política. Para ello su modificación al archivo debe ser la siguiente:

struct task\_struct {

...

#endif

struct prop\_local\_single dirties; #ifdef CONFIG\_SCHED\_CASIO\_POLICY

unsigned int casio\_id; unsigned long long deadline;

#endif

};

Text

Description automatically generated

**Figura 10.** Evidencia de elaboración de laboratorio.

* + **¿Qué es una *task* en Linux?**

En Linux, una tarea se define como una unidad en ejecución que puede compartir recursos con otras tareas, y dependiendo del nivel de intercambio de dichos recursos las tareas pueden considerarse un hilo o proceso. (2021)

* + **¿Cuál es el propósito de task\_struct y cuál es su análogo en Windows?**

En Linux, una task\_struct es una estructura de datos del kernel que almacena toda la información que el procesador tiene y necesita acerca de un proceso (*Task\_struct en Linux Kernel - programador clic*, s. f.). Algunos de los miembros de esta estructura son:

* + - Indicador: es el identificador del proceso
    - Estado
    - Prioridad
    - Contador del programa: dirección de la siguiente instrucción a ejecutar

En Windows el análogo a la estructura task\_struct puede ser EPROCESS, ya que almacena información correspondiente al proceso, como el ID del proceso, su estado de salida y entre otros. (Aelterman, 2019)

Diagram

Description automatically generated

**Figura 11.** Ejemplo de una task\_struct en Linux. (*Task\_struct en Linux Kernel - programador clic*, s. f.)

Diagram, table

Description automatically generated

**Figura 12.** Estructura de un  *executive process block -EPROCESS.* (Aelterman, 2019)

1. En este mismo archivo busque también la estructura sched\_param (línea 47) y agréguele los mismos parámetros al final (siempre dentro de un bloque #ifdef). En

/usr/include/bits/sched.h hay dos definiciones de sched\_param (en realidad, una es para sched\_param). Incluya estos cambios en ellas también, pero sin encerrarlos en un bloque #ifdef. Grabe y cierre sched.h.

Text

Description automatically generated

**Figura 13.** Evidencia de elaboración de laboratorio.

* + **¿Qué información contiene sched\_param?**

La estructura sched\_param define los parámetros de calendarización, en este caso se agregan aquellos parámetros necesario s para una tarea CASIO, en este caso se define el id de la tarea, como casio\_id y también se define el *deadline* relativo de la tarea. (Staff, 2022)

1. Diríjase al archivo kernel\_dir/kernel/sched.c. La política de calendarización que emplearemos es la de *earliest deadline first*
2. (EDF), por lo que debemos indicar al sistema operativo que nuestra política pertenece a esta clase. Busque la función rt\_policy y modifíquela de la siguiente manera (sin olvidar crear una copia de *backup* del archivo):

static inline int rt\_policy(int policy)

{

if (unlikely(policy == SCHED\_FIFO) || unlikely(policy == SCHED\_RR) #ifdef CONFIG\_SCHED\_CASIO\_POLICY

|| unlikely(policy == SCHED\_CASIO) #endif

){

return 1;

}

return 0;

}

Text

Description automatically generated

**Figura 14.** Evidencia de elaboración de laboratorio.

* + **¿Para qué sirve la función rt\_policy y para qué sirve la llamada unlikely en ella?**

La función rt\_policy (real-time policy) se usa para determinar si una política de calendarización pertenece a la clase de tiempo real o no. (Staff, 2022)

La función unlikely, es una \_\_builtin\_expect() function, y lo que hace es darle “pistas” al compilador para optimizar la rama de ejecución, ordenando los eventos por “probabilidad” para así optimizar el uso del *pipeline* del procesador. Básicamente lo que hace es indicarle cuál es el evento más “probable” o en este caso, improbable. (*FAQ/LikelyUnlikely - Linux Kernel Newbies*, 2017)

En la **figura 6** al usar unlikely lo que se dice es que no se espera retornar 1 porque se espera que la comparación de todas las políticas retorne 0. (Lynxbeedev, 2021)

* + **¿Qué tipo de tareas calendariza la política EDF, en vista del método modificado?**

Como se mencionó anteriormente la función rt\_policy, hace referencia a la clase de tiempo real, por lo tal, la política EDF calendariza tareas en tiempo real, ya que como bien se sabe, en un sistema de tiempo real, el algoritmo de calendarización se encarga de que todas las tareas finalicen antes de su respectivo *deadline*, justo como trabaja EDF. (Staff, 2010)

1. Nuestra política será implementada en un archivo llamado sched\_casio.c. Modifique

sched.c de la siguiente manera:Text

Description automatically generated

...

# include “sched\_debug.c” #endif

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

#include “sched\_casio.c” #endif

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

#define sched\_class\_highest (&casio\_sched\_class) #else

#define sched\_class\_highest (&rt\_sched\_class) #endif

/\*

\* Update delta\_exec, delta\_fail fields for rq.

...

**Figura 15.** Evidencia de elaboración de código.

* + **Describa la precedencia de prioridades para las políticas EDF, RT y CFS, de acuerdo con los cambios realizados hasta ahora.**

**EDF** (*Earlier Deadline First*) es un calendarizador dinámico, que tal y como su nombre lo indica asigna las prioridades de acorde al valor de *deadline* de las tareas, es decir que otorga la prioridad más alta al valor más bajo de *deadline*. (Staff, 2010)

**CFS** (Completely Fair Scheduling) es utilizada para la calendarización de tareas “normales” o aquellas no se consideran *real-time*. Su priorización de tareas es con base a *virtual runtime* el cual indica la cantidad de tiempo que ha pasado en el procesador. Como bien lo indica su nombre, está política de calendarización busca ser “justa” entonces por lo tal, para la calendarización, utiliza un *red-black tree* colocando del lado izquierdo aquellos que tienen menor *vruntime* (*virtual runtime*)ya que estas tareas con las que más “derecho” a procesador tienen, para así ser justo en el uso de procesador de las tareas. (Devored, 2013)

**RT** (*Real-Time*) esta es una política de calendarización estática, es decir el kernel no calcula dinámicamente la prioridad de las tareas. En este caso, las prioridades a las tareas se asignan con forme la tarea pasa al estado *runnable*, básicamente implementa un algoritmo de calendarizaión FIFO. (*Real-Time | The Linux Process Scheduler | InformIT*, 2013)

1. Para que los procesos puedan calendarizarse con nuestra política deben cambiar su calendarizador con llamadas a sistema durante su ejecución. En la función setscheduler realice la siguiente modificación:

...

p->policy = policy; switch(p->policy){

...

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

case SCHED\_CASIO:

p->sched\_class = &casio\_sched\_class; break;

#endif

}

Text

Description automatically generated

**Figura 16.** Evidencia de elaboración de código.

Y en la funcición sched\_setscheduler realice las siguientes

...

if (policy < 0)

policy = oldpolicy = p->policy;

else if (policy != SCHED\_FIFO && policy != SCHED\_RR &&

policy != SCHED\_NORMAL && policy != SCHED\_BATCH && policy != SCHED\_IDLE

/\*)\*/

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

&& policy != SCHED\_CASIO

#endif

)

return –EINVAL;

...

/\* can’t change other user’s priorities \*/ if ((current->euid != p->euid) &&

(current->euid != p->uid)) return –EPERM;

}

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

if (policy == SCHED\_CASIO){

p->deadline = param->deadline; p->casio\_id = param->casio\_id;

}

#endif

...

modificaciones:

Text

Description automatically generatedText

Description automatically generated

**Figura 17.** Evidencia de elaboración de código.

1. Ahora definiremos las *tasks* que son calendarizables con nuestra política, y su *ready queue*. Recuerde que el calendarizador CFS para tareas normales en Linux usa un árbol *red-black* para organizar sus procesos por prioridad. En nuestra política haremos lo mismo, pero, por ser una implementación de EDF, las etiquetas de los nodos en el árbol serán las *deadlines* de las tareas. Siempre en sched.c aplique la siguiente modificación:

...

struct rt\_rq{

...

};

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

struct casio\_task{

struct rb\_node casio\_rb\_node; unsigned long long absolute\_deadline; struct list\_head casio\_list\_node; struct task\_struct\* task;

};

struct casio\_rq{

struct rb\_root casio\_rb\_root; struct list\_head casio\_list\_head; atomic\_t nr\_running;

};

#endif

/\*

\* This is the main, per-CPU runqueue data structure.

...

Text

Description automatically generated

**Figura 18.** Evidencia de elaboración de código.

Note que nuestra política se apoya en el uso de estructuras de datos provistas por el *kernel* en

<linux/list.h> y <linux/rbtree.h>. Un árbol *red-black* mantendrá nuestra *ready queue*.

* + **Explique el contenido de la estructura casio\_task.**

La estructura casio\_task define la información que se almacena de cada tarea. Como se puede observar en la **figura 18** en la estructura casio\_task se tienen los siguientes miembros:

* + - casio\_rb\_node: se utiliza para organizar las tareas en un árbol *red-black*
    - absolute\_deadline: tal como lo indica el nombre, almacena el *deadline* de la tarea
    - casio\_list\_node: esta lista permita organizar todas las tareas CASIO en una lista doblemente enlazada (*double linked list*)
    - task: es un punter al descriptor del proceso.

(Staff, 2010)

1. Para que el sistema pueda referirse a las tareas calendarizadas de acuerdo con nuestra política, debemos aplicar la siguiente modificación en sched.c:

Text

Description automatically generated**Figura 19.** Evidencia de elaboración de código.

struct rq {

...

struct rt\_rq rt;

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

struct casio\_rq casio\_rq; #endif

...

* + **Explique el propósito y contenido de la estructura casio\_rq.**

La estructura casio\_rq hace referencia a la *run-queue* del procesado, la cual almacena todos los procesos en estado *runnable* asignados al mismo. Algo interesante es que la política de calendarización utiliza este tipo de *queue* para seleccionar el siguiente proceso a ser ejecutado.

La estructura casio\_rq contiene lo siguiente:

* + - casio\_rb\_root: define la raíz del árbol red-black
    - casio\_list\_head: almacena todas las tareas CASIO asignadas a un procesador
    - nr\_running: define el número de tareas CASIO en la *queue*

(Staff, 2010)

* + **¿Qué es y para qué sirve el tipo atomic\_t? Describa brevemente los conceptos de operaciones RMW (*read-modify-write*) y *mappeo* de dispositivos en memoria (MMIO).**

Atomic\_t es un tipo de datos especial con un conjunto de operaciones *read-modify-write atómicas* que garantizan que una variable será atómica sin la necesidad de realizar un bloqueo o algún método de sincronización como mutex. (Corbet, 2004)

Las operaciones RMW, como bien se indica en el nombre, son operaciones de lectura, modificación y/o escritura, y estás pueden verse de la siguiente forma:

* + - Operaciones simples, sin valor de retorno
    - Operaciones que retornan el valor modifcado
    - Operaciones que retornan el valor original

(Kernel, s. f.)

*Memory mapped I/O* es una manera en la cual se asigna el bus del sistema de manera que el CPU pueda comunicarse tanto como con la memoria como con los dispositivos de entrada y salida. El kernel utiliza una parte del espacio de memoria de la RAM para mapear los registros de los dispositivos de E/S, entonces, se puede manipular los dispositivos de memoria igual que la memoria y los dos cuentan con el mismo espacio de memoria. (GeeksforGeeks, 2019)

A screenshot of a computer

Description automatically generated with medium confidence

**Figura 20.** Ejemplificación de MMIO

1. Cuando un proceso cambie su política de calendarización, para usar nuestra política debe ser agregado a la lista. Modifique nuevamente la función sched\_setscheduler para que refleje los siguientes cambios:

...

if (unlikely(oldpolicy != -1 && oldpolicy != p->policy)){ policy = oldpolicy = -1;

task\_rq\_unlock(rq); spin\_unlock\_irqrestore(&p->pi\_lock, flags); goto recheck;

}

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

if (policy == SCHED\_CASIO){ add\_casio\_task\_2\_list(&rq->casio\_rq, p);

}

#endif

update\_rq\_clock(rq);

...

Text

Description automatically generated

**Figura 21.** Evidencia de elaboración de código.

Note que esta modificación emplea un método que todavía no hemos definido.

1. Los diferentes calendarizadores de Linux se inicializan en la función sched\_init. Modifique esta función de la siguiente forma:

Text

Description automatically generated

void init sched\_init(void){

...

rq->nr\_running = 0;

rq->clock = 1;

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

init\_casio\_rq(&rq->casio\_rq); #endif

init\_cfs\_rq(&rq->cfs, rq);

...

**Figura 22.** Evidencia de elaboración de código.

Note, de nuevo, que la función llamada no ha sido definida todavía.

1. Todo lo que hemos hecho hasta ahora ha servido para configurar el uso de la política de calendarización EDF en el sistema. Ahora implementaremos la política como tal. Cree el archivo kernel\_dir/kernel/sched\_casio.c y programe la función de inicialización de la *ready queue* para nuestras *tasks*:

void init\_casio\_rq(struct casio\_rq\* casio\_rq){ casio\_rq->casio\_rb\_root=RB\_ROOT; INIT\_LIST\_HEAD(&casio\_rq->casio\_list\_head); atomic\_set(&casio\_rq->nr\_running, 0);

}

1. Luego programe las funciones para el manejo de la lista de casio\_tasks:

void add\_casio\_task\_2\_list(struct casio\_rq\* rq, struct task\_struct\* p){ struct list\_head\* ptr = NULL;

struct casio\_task\* new = NULL; struct casio\_task\* casio\_task = NULL;

//char msg

if (rq && p){

new = (struct casio\_task\*)kzalloc(sizeof(struct casio\_task), GFP\_KERNEL); if (new){

casio\_task = NULL; new->task = p;

new->absolute\_deadline = 0; list\_for\_each(ptr, &rq->casio\_list\_head){

casio\_task = list\_entry(ptr, struct casio\_task, casio\_list\_node); if (casio\_task){

if (new->task->casio\_id < casio\_task->task->casio\_id){ list\_add(&new->casio\_list\_node, ptr);

return;

}

}

}

list\_add(&new->casio\_list\_node, &rq->casio\_list\_head);

//logs

} else {

printk(KERN\_ALERT “add\_casio\_task\_2\_list: kzalloc\n”);

}

} else {

printk(KERN\_ALERT “add\_casio\_task\_2\_list: null pointers\n”);

}

}

void rem\_casio\_task\_list(struct casio\_rq\* rq, struct task\_struct\* p){ struct list\_head\* ptr = NULL;

struct list\_head\* next = NULL;

struct casio\_task\* casio\_task = NULL;

//char msg

if (rq && p){

list\_for\_each\_safe(ptr, next, &rq->casio\_list\_head){

casio\_task = list\_entry(ptr, struct casio\_task, casio\_list\_node); if (casio\_task){

if (casio\_task->task->casio\_id == p->casio\_id){ list\_del(ptr);

//logs kfree(casio\_task); return;

}

}

}

}

}

struct casio\_task\* find\_casio\_task\_list(struct casio\_rq\* rq, struct task\_struct\* p){ struct list\_head\* ptr = NULL;

struct casio\_task\* casio\_task = NULL; if (rq && p){

list\_for\_each(ptr, &rq->casio\_list\_head){

casio\_task = list\_entry(ptr, struct casio\_task, casio\_list\_node); if (casio\_task){

if (casio\_task->task->casio\_id == p->casio\_id){ return casio\_task;

}

}

}

}

return NULL;

}

1. Ahora programe las funciones para el manejo del *red-black tree* de casio\_tasks:

void insert\_casio\_task\_rb\_tree(struct casio\_rq\* rq, struct casio\_task\* p){ struct rb\_node\*\* node = NULL;

struct rb\_node\* parent = NULL; struct casio\_task\* entry = NULL; node = &rq->casio\_rb\_root.rb\_node; while(\*node != NULL){

parent = \*node;

entry = rb\_entry(parent, struct casio\_task, casio\_rb\_node); if (entry){

if (p->absolute\_deadline < entry->absolute\_deadline){ node = &parent->rb\_left;

} else {

node = &parent->rb\_right;

}

}

}

rb\_link\_node(&p->casio\_rb\_node, parent, node); rb\_insert\_color(&p->casio\_rb\_node, &rq->casio\_rb\_root);

}

void remove\_casio\_task\_rb\_tree(struct casio\_rq\* rq, struct casio\_task\* p){ rb\_erase(&(p->casio\_rb\_node), &(rq->casio\_rb\_root));

p->casio\_rb\_node.rb\_left = p->casio\_rb\_node.rb\_right = NULL;

}

struct casio\_task\* earliest\_deadline\_casio\_task\_rb\_tree(struct casio\_rq\* rq){ struct rb\_node\* node = NULL;

struct casio\_task\* p = NULL;

node = rq->casio\_rb\_root.rb\_node; if (node == NULL)

return NULL;

while (node->rb\_left != NULL){ node = node->rb\_left;

}

p = rb\_entry(node, struct casio\_task, casio\_rb\_node); return p;

}

1. Las funciones que recién definimos son como el *backend* de nuestra política de calendarización. Recordemos que en setscheduler agregamos una condicional para que se tomara &casio\_sched\_class como política de calendarización del sistema. Ahora definiremos esta clase, pero nótese que no hablamos de una clase del paradigma de orientación a objetos sino de una clase de calendarización. Esta clase es en realidad la declaración de una constante de tipo struct sched\_class, que requiere la definición de ciertos valores para funcionar como una calendarización en el sistema (similar a una interfaz en Java). Incluya el siguiente código en sched\_casio.c:

const struct sched\_class casio\_sched\_class = {

.next = &rt\_sched\_class,

.enqueue\_task = enqueue\_task\_casio,

.dequeue\_task = dequeue\_task\_casio,

.check\_preempt\_curr = check\_preempt\_curr\_casio,

.pick\_next\_task = pick\_next\_task\_casio,

.put\_prev\_task = put\_prev\_task\_casio,

#ifdef CONFIG\_SMP

.load\_balance

.move\_one\_task #endif

= load\_balance\_casio,

= move\_one\_task\_casio,

.set\_curr\_task

.task\_tick

= set\_curr\_task\_casio,

= task\_tick\_casio,

};

* + **¿Qué indica el campo .next de esta estructura?**

Este campo contiene un puntero a sched\_class que se usa para organizar los módulos del calendarizador por prioridad en una lista vinculada. Debido a que casio\_sched\_class es el módulo con mayor prioridad, entonces se apunta al siguiente módulo, el cual es rt\_sched\_class el cual implementa el módulo Real Time. (Staff, 2022)

1. Ahora definiremos las funciones que conforman nuestra clase de calendarización. Asegúrese de incluir este código ANTES de la declaración de casio\_sched\_class:

static void enqueue\_task\_casio(struct rq\* rq, struct task\_struct\* p, int wakeup)

{

struct casio\_task\* t = NULL;

//char msg if (p){

t = find\_casio\_task\_list(&rq->casio\_rq, p); if (t){

t->absolute\_deadline = sched\_clock() + p->deadline; insert\_casio\_task\_rb\_tree(&rq->casio\_rq, t); atomic\_inc(&rq->casio\_rq.nr\_running);

//logs

} else {

printk(KERN\_ALERT "enqueue\_task\_casio\n");

}

}

}

static void dequeue\_task\_casio(struct rq\* rq, struct task\_struct\* p, int sleep)

{

struct casio\_task\* t = NULL;

//char msg if(p){

t = find\_casio\_task\_list(&rq->casio\_rq,p); if (t){

//logs

remove\_casio\_task\_rb\_tree(&rq->casio\_rq, t); atomic\_dec(&rq->casio\_rq.nr\_running);

if(t->task->state == TASK\_DEAD || t->task->state == EXIT\_DEAD

|| t->task->state==EXIT\_ZOMBIE){ rem\_casio\_task\_list(&rq->casio\_rq, t->task);

}

} else {

printk(KERN\_ALERT "dequeue\_task\_casio\n");

}

}

}

* + **Tomando en cuenta las funciones para manejo de lista y *red-black tree* de casio\_tasks, explique el ciclo de vida de una casio\_task desde el momento en el que se le asigna esta clase de calendarización mediante sched\_setscheduler. El objetivo es que indique el orden y los escenarios en los que se ejecutan estas funciones, así como las estructuras de datos por las que pasa. ¿Por qué se guardan las casio\_tasks en un *red-black tree* y en una lista encadenada?**

1. Luego de que a la tarea se le asigna la clase de calendarización de CASIO esta se llama la función *add\_casio\_task\_2\_list* la cual se encarga de añadir dicha tarea a la *linked list* utilizada para el manejo de todas las tareas.
2. Cuando la tarea pasa a un estado ejecutable (*runnable*) se llama a la función *enqueue\_task\_casio* la cual se encarga de insertar la tarea en el árbol *red-black* y de actualizar el valor del *absolute deadline* de dicha tarea. Por otro lado, cuando una tarea ya no está en estado ejecutable se llama a la función *dequeue\_task\_casio* la cual elimina la tarea del árbol red-black. Algo importante en esto es que *dequeue\_task\_casio* se llama solo si la tarea se ha ejecutado, por otor lado, si la tarea no se ejecuta entonces se elimina de la *linked list* mediante la función rem\_casio\_task\_list.
3. Ya sea en el proceso de agregar una tarea al árbol o de quitar una tarea del árbol, en ambos casos siempre se llama a la función *check\_preempt\_curr\_*casio esto para validar si la tarea actual debería de ser “sustituida”.
4. Luego se escoge la tarea que será ejecutada por el procesador mediante la función *pick\_next\_task\_casio,* esta función se llama cuando la tarea actual debe de ser “sustituida”. Básicamente lo que hace es que mediante la función *earliest\_deadline\_casio\_task\_rb\_tree* se retona la tarea con el menor *deadline*, y es esta la que se escoge para ser ejecutada.

Las CASIO tasks se guardan en un red-black tree debido a que así es mucho más eficiente obtener aquella que contiene el *dedline* menor, ya que se van ordenando de izquierda a derecha, es decir, solo necesita buscar en la rama izquierda del árbol para encontrar a la tarea con el *deadline* más corto.

Por otro lado, se utiliza una lista encadenada para almacenar todas las tareas en general, debido a que estás además de almacenar un dato en particular, también almacenan un puntero al siguiente elemento en la lista. Por lo que así es fácilmente modificable cuál es la tarea que sigue en la lista, ya que solo será necesario cambiar el puntero al cual apunta.

static void check\_preempt\_curr\_casio(struct rq\* rq, struct task\_struct\* p)

{

struct casio\_task\* t = NULL; struct casio\_task\* curr = NULL;

if (rq->curr->policy != SCHED\_CASIO){ resched\_task(rq->curr);

} else {

t = earliest\_deadline\_casio\_task\_rb\_tree(&rq->casio\_rq); if (t){

curr = find\_casio\_task\_list(&rq->casio\_rq, rq->curr); if (curr){

if (t->absolute\_deadline < curr->absolute\_deadline) resched\_task(rq->curr);

} else {

printk(KERN\_ALERT "check\_preempt\_curr\_casio\n");

}

}

}

}

* + **¿Cuándo *preemptea* una casio\_task a la *task* actualmente en ejecución?**

Una casio\_task puede “reemplazar” a la función actualmente en ejecución en dos escenarios:

1. Si la actual tarea en ejecución no es una casio\_task, y existe al menos una casio\_task en el run-queue; o
2. La tarea actualmente en ejecución sí es una casio task, pero existe otra casio\_task en el árbol con un *deadline* menor que el de la actual. (Staff, 2022)

static struct task\_struct\* pick\_next\_task\_casio(struct rq\* rq)

{

struct casio\_task\* t = NULL;

t = earliest\_deadline\_casio\_task\_rb\_tree(&rq->casio\_rq); if (t){

return t->task;

}

return NULL;

}

static void put\_prev\_task\_casio(struct rq\* rq, struct task\_struct\* prev)

{

}

#ifdef CONFIG\_SMP

static unsigned long load\_balance\_casio(struct rq\* this\_rq, int this\_cpu, struct rq\* busiest,

unsigned long max\_load\_move,

struct sched\_domain\* sd, enum cpu\_idle\_type idle, int\* all\_pinned, int\* this\_best\_prio)

{

return 0;

}

static int move\_one\_task\_casio(struct rq\* this\_rq, int this\_cpu, struct rq\* busiest,

struct sched\_domain\* sd, enum cpu\_idle\_type idle)

{

return 0;

}

#endif

static void set\_curr\_task\_casio(struct rq\* rq)

{

}

static void task\_tick\_casio(struct rq\* rq, struct task\_struct\* p)

{

}

1. Habiendo llegado a este punto ya tenemos lista nuestra política de calendarización, pero vamos a agregar elementos que nos permitan llevar registro de los eventos que suceden durante la calendarización. Comenzaremos por ir a kernel\_dir/include/linux/sched.h y aplicar la siguiente modificación:

...

#endif /\* KERNEL \*/

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

#define CASIO\_MSG\_SIZE 400

#define CASIO\_MAX\_EVENT\_LINES 10000

#define CASIO\_ENQUEUE 1

#define CASIO\_DEQUEUE 2

#define CASIO\_CONTEXT\_SWITCH 3

#define CASIO\_MSG 4

struct casio\_event{ int action;

unsigned long long timestamp; char msg[CASIO\_MSG\_SIZE];

};

struct casio\_event\_log{

struct casio\_event casio\_event[CASIO\_MAX\_EVENT\_LINES]; unsigned long lines;

unsigned long cursor;

};

void init\_casio\_event\_log();

struct casio\_event\_log\* get\_casio\_event\_log();

void register\_casio\_event(unsigned long long t, char\* m, int a); #endif

#endif

1. Ahora definiremos estas funciones en kernel\_dir/kernel/sched\_casio.c. Agregue al inicio de este archivo lo siguiente:

struct casio\_event\_log casio\_event\_log;

struct casio\_event\_log\* get\_casio\_event\_log(){ return &casio\_event\_log;

}

void register\_casio\_event(unsigned long long t, char\* m, int a){ if (casio\_event\_log.lines < CASIO\_MAX\_EVENT\_LINES){

casio\_event\_log.casio\_event[casio\_event\_log.lines].action = a; casio\_event\_log.casio\_event[casio\_event\_log.lines].timestamp = t; strncpy(casio\_event\_log.casio\_event[casio\_event\_log.lines].msg, m,

CASIO\_MSG\_SIZE – 1);

casio\_event\_log.lines++;

} else {

printk(KERN\_ALERT “register\_casio\_event: full\n”);

}

}

void init\_casio\_event\_log(){ char msg[CASIO\_MSG\_SIZE];

casio\_event\_log.lines = casio\_event\_log.cursor = 0;

snprintf(msg, CASIO\_MSG\_SIZE, “init\_casio\_event\_log: (%lu:%lu)”, casio\_event\_log.lines, casio\_event\_log.cursor);

register\_casio\_event(sched\_clock(), msg, CASIO\_MSG);

}

Note que algunas líneas se hicieron demasiado largas y no cupieron en los márgenes de este documento. Puesto que en el próximo inciso vamos a repetir estas instrucciones convendremos en lo siguiente: cuando se diga ‘registre un evento con el mensaje “mensaje %d %lu” con valores valor1 y valor2; y con bandera CASIO\_MSG” se estará indicando que, en el código, se incluya lo siguiente:

snprintf(msg, CASIO\_MSG\_SIZE, “mensaje %d %lu”, valor1, valor2); register\_casio\_event(sched\_clock(), msg, CASIO\_MSG) ;

donde “mensaje %d %lu” es un *string* con especificadores de formato cuyos valores corresponden a valor1 y valor2. Puesto que estas instrucciones requieren la variable msg, se incluirá el recordatorio ‘declare msg’ para que, donde se le indique, incluya el código

char msg[CASIO\_MSG\_SIZE];

aa. Vamos a registrar algunos eventos:

1. En add\_casio\_task\_2\_list declare msg en donde está el comentario //char msg, y donde está el comentario //logs registre un evento con el mensaje “add\_casio\_task\_2\_list: %d:%d:%llu” con valores new->task-

>casio\_id, new->task->pid, new->absolute\_deadline; y con bandera

CASIO\_MSG.

1. En rem\_casio\_task\_list declare msg donde está //char msg, y donde está //logs registre un evento con el mensaje “rem\_casio\_task\_list: %d:%d:%llu”, con valores casio\_task->task->casio\_id, casio\_task->task->pid, casio\_task->absolute\_deadline; y con bandera CASIO\_MSG.
2. En enqueue\_task\_casio declare msg donde está //char msg, y donde de está//logs registre un evento con el mensaje “(%d:%d:%llu)”, con valores p->casio\_id, p>pid, t->absolute\_deadline; y con bandera CASIO\_ENQUEUE.
3. Finalmente, en dequeue\_task\_casio declare msg donde está //char msg, y donde está //logs registre un evento con el mensaje “(%d:%d:%llu)”, con valores t->task->casio\_id, t->task->pid, t->absolute\_deadline; y con bandera CASIO\_DEQUEUE.

bb. Un evento que debemos registrar pero que no controlamos desde sched\_casio.c es el cambio de contexto que involucra una o dos *casio tasks*. Para ello debemos dirigirnos a kernel\_dir/kernel/sched.c y aplicar la siguiente modificación:

...

prev->sched\_class->put\_prev\_task(rq, prev); next = pick\_next\_task(rq, prev);

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

char msg[CASIO\_MSG\_SIZE];

if (prev->policy == SCHED\_CASIO || next->policy == SCHED\_CASIO){

if (prev->policy == SCHED\_CASIO && next->policy == SCHED\_CASIO){

//logs1

} else {

if (prev->policy == SCHED\_CASIO){

//logs2

} else {

//logs3

}

}

register\_casio\_event(sched\_clock(), msg, CASIO\_CONTEXT\_SWITCH);

}

#endif

sched\_info\_switch(prev, next);

...

Reemplazando //logs1, //logs2 y //logs3 por llamadas a snprintf cuyos primeros dos argumentos sean msg y CASIO\_MSG\_SIZE; y cuyos últimos argumentos sean, respectivamente:

1. “prev->(%d:%d), next->(%d:%d)”, prev->casio\_id, prev->pid, next->casio\_id, next->pid
2. “prev->(%d:%d), next->(-1:%d)”, prev->casio\_id, prev->pid, next->pid
3. “prev->(-1:%d), next->(%d:%d)”, prev->pid, next->casio\_id, next->pid

cc. Finalmente, modificaremos kernel\_dir/fs/proc/proc\_misc.c para que nuestra bitácora se almacene en un archivo que como usuarios podamos abrir y leer (recordemos que nuestro *log* y todo lo que éste almacena están en *kernel space*).

...

#undef K

}

#ifdef CONFIG\_SCHED\_CASIO\_POLICY #define CASIO\_MAX\_CURSOR\_LINES\_EVENTS 1

static int casio\_open(struct inode\* inode, struct file\* file){ return 0;

}

static int casio\_release(struct inode\* inode, struct file\* file){ return 0;

}

static int casio\_read(char\* filp, char\* buf, size\_t count, loff\_t\* f\_pos){ char buffer[CASIO\_MSG\_SIZE];

unsigned int len = 0, k, i;

struct casio\_event\_log\* log = NULL; buffer[0] = ‘\0’;

log = get\_casio\_event\_log(); if (log){

if (log->cursor < log->lines){

k = (log->lines > (log->cursor

+

CASIO\_MAX\_CURSOR\_LINES\_EVENTS)) ? (log->cursor + CASIO\_MAX\_CURSOR\_LINES\_EVENTS)

: (log->lines);

for (i = log->cursor; i < k; i++){

len = snprintf(buffer, count, “%d, %llu, %s\n”, buffer,

log->casio\_event[i].action, log->casio\_event[i].timestamp, log->casio\_event[i].msg);

}

log->cursor = k;

}

if (len)

copy\_to\_user(buf, buffer, len);

}

return len;

}

static const struct file\_operations proc\_casio\_operations = {

.open = casio\_open,

.read = casio\_read,

.release = casio\_release,

};

#endif

extern struct seq\_operations fragmentation\_op;

...

...

entry->proc\_fops = &proc\_sysrq\_trigger\_operations;

}

#endif

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

{

struct proc\_dir\_entry\* casio\_entry;

casio\_entry = create\_proc\_entry(“casio\_event”, 0666, &proc\_root); if (casio\_entry){

casio\_entry->proc\_fops = &proc\_casio\_operations; casio\_entry->data = NULL;

}

}

#endif

}

Con esto terminamos las modificaciones al sistema que implementan la nueva política de calendarización. Antes de compilar el *kernel* acceda al *Makefile* en kernel\_dir y asigne a la variable EXTRAVERSION el valor –casio. Además, copie el archivo de configuración del *kernel* actual a esta carpeta con el siguiente comando:

sudo cp /boot/config-2.6.24-26-generic .config

Note el espacio antes de .config. Ahora copie todo el contenido de scheduler\_dev/linux- 2.6.24-casio a scheduler (use la opción –a del comando cp). Se recomienda crear una *snapshot* (al menos) en este punto. Diríjase a scheduler/linux-2.6.24-casio y ejecute lo siguiente:

sudo make oldconfig

Este proceso de compilación toma un archivo de configuración existente y crea uno nuevo, pidiendo *input* al usuario sobre las características nuevas o desconocidas que tenga el *kernel* a compilarse. Para cada pregunta que se le realice habrá un valor entre corchetes y, en caso de ser una pregunta con respuesta “sí” o “no”, se señalará con una letra mayúscula la opción por defecto. Asegúrese de que CASIO Scheduler sea configurada con ‘y’ y todas las demás opciones con su valor por defecto. Al terminar, compile el *kernel* con el siguiente comando:

sudo make-kpkg --initrd kernel\_image 2>../errors

Cualquier error detectado durante la compilación se almacenará en el archivo errors, en el directorio scheduler. Una vez termine la compilación, instale el *kernel* con el siguiente comando:

sudo dpkg –i linux-image-...deb

Al terminar este proceso, reinicie su máquina. Si todo salió bien, al iniciar el sistema podrá presionar una tecla para acceder al menú de GRUB, desde donde podrá entrar a su nuevo sistema.

Text

Description automatically generated

**Figura 23.** Evidencia de correcto funcionamiento de la instalación del kernel.

* **Ejecute nuevamente el archivo casio\_system tal como se hizo al inicio del laboratorio, pero guardando los resultados en un archivo diferente. Adjunte ambos archivos de resultados de casio\_system a su entrega, comentando sobre sus diferencias.**

La primera diferencia que se pudo observar, no tanto a nivel de resultados sino más bien a nivel de calendarizador es el tiempo que toma hacer dicha calendarización y la ejecución de las tareas. Por ejemplo, en el caso del calendarizador inicial, este tomó a lo mucho 40 segundos es crear el archivo con resultados, mientras que por otro lado, con el calendarizador CASIO se tomó aproximadamente 10min.

Text

Description automatically generatedText

Description automatically generated

**Figura 24.** Comparación de resultados con diferentes calendarizadores.

Al comparar los 2 archivos, fácil diferencia que se puede encontrar es que ahora luego de establecer el calendarizador la prioridad cambia de 0 a 6. Esto pasa, debido a que en instrucciones anteriores se indicó que si se utilizaba el calendarizador CASIO este tenía un valor de 6, a diferencia del calendarizador definido como normal, el cual tiene un identificador de 0.

Por otro lado, también se puede observar en ambos archivos que el orden de impresión de las tareas es distinto, si bien puede que esto no tenga nada que ver con el calendarizador, sin embargo, si se nota en la imagen de *post\_casio.txt* en este caso las tareas se encuentran ordenadas de acorde al *deadline* de cada una, y mostrando así que ejecuta de último la que tiene un *deadline* más grande. A diferencia de la imagen *pre\_casio.txt* en la cual se ve que las tareas se ejecutan en orden de como llegan.

* Ubique el archivo de *log* de eventos registrados por la calendarización implementada. Adjunte este archivo con su entrega.
* **Agregue comentarios explicativos a los archivos casio\_task.c y casio\_system.c que permitan entender el propósito y funcionamiento de este código. Asegúrese de aclarar el uso de instrucciones y estructuras que no conozca (como, por ejemplo, los *timers* y la estructura itimerval). ¿Qué información contiene el archivo system que se especifica como argumento en la ejecución de casio\_system?**

El archivo system contiene toda la información de configuración de las tareas. En la primera línea define la duración de cada tarea, y las siguientes líneas, cada una representan cada una de las tareas. En la **figura 23** se puede observar cómo cada una de las columnas representa un valor de configuración de la tarea, por ejemplo, en el caso de la columna 1 esta indica el id de la tarea, y así sucesivamente hasta la columna 7 la cual indica el max-offset.

Text

Description automatically generated

**Figura 25.** Representación en cuanto a información de configuración de la tarea de las columnas del archivo sytem.

* **Investigue el concepto de aislamiento temporal en relación a procesos. Explique cómo el calendarizador SCHED\_DEADLINE, introducido en la versión 3.14 del *kernel* de Linux, añade al algoritmo EDF para lograr aislamiento temporal.**

Aislamiento temporal de procesos o *process isolation* como comúnmente se le conoce, es la segregación de procesos o *threads* para evitar, entre ellos, la interrupción en la ejecución de otros procesos o *threads*. El proceso de aislamiento puede ir desde limitar el uso de memoria de cierto *thread* hasta indicar en qué cores de CPU específicos se puede ejecutar dicho *thread*. De manera general, el aislamiento consiste en asegurar que un proceso A no utilice el espacio de memoria designado para un proceso B. (NI, 2021)

El calendarizador SCHED\_DEADLINE para asilar tareas entre ellas utiliza un mecanismo llamado *Constant Bandwith Server* (CBS). CBS es un algoritmo para proveer aislamiento temporal de tareas (Kernel, s. f.-b). Este algoritmo, en la implementación de SCHED\_DEADLINE asigna los *deadlines* de calendarización de la siguiente manera:

* Cada tarea SCHED\_DEADLINE se compone de un *runtime, deadline* y un periodo. Tanto el *scheduling deadline* y *reamining* runtime son inicializados en 0. Cuando una tarea se “despierta” el calendarizador verifica:

A picture containing text

Description automatically generated

**Figura 26.** Modificación de parámetros en el proceso de aislamiento. (Kernel, s. f.-b)

Si la condición descrita en la **figura 23** se cumple, entonces el valor de *scheduling deadline* y *reamining* runtime son actualizados de lo contrario se quedan como están hasta el momento. (Kernel, s. f.-b)

* Cuando el tiempo de ejecución restante es menor o igual a 0, se dice que la tarea está limitada, y no puede ser calendarizada hasta su *scheduling deadline*. El tiempo de reabastecimiento para la tarea se modifca con el valor actual del *scheduling deadline*. (Kernel, s. f.-b)
* Cuando el tiempo actual es igual al restablecimiento entonces se actualiza el *remaining time* y *scheduling deadline* de la siguiente forma:

Text

Description automatically generated with medium confidence

**Figura 27.** Modificación de parámetros dentro del proceso de aislamiento. (Kernel, s. f.-b)

**Fuente:**

* + <http://www.embedded.com/design/operating-systems/4204929/Real-Time-Linux-Scheduling-> Part-1
  + [https://w](http://www.embedded.com/design/operating-systems/4204971/Real-Time-Linux-Scheduling-)ww.embe[dded.com/desi](http://www.embedded.com/design/operating-systems/4204971/Real-Time-Linux-Scheduling-)g[n/operating-systems/4204971/Real-Time-Linux-Scheduling-](http://www.embedded.com/design/operating-systems/4204971/Real-Time-Linux-Scheduling-) Part-2
  + [https://w](http://www.embedded.com/design/operating-systems/4204980/Real-Time-Linux-Scheduling-)ww.embe[dded.com/desi](http://www.embedded.com/design/operating-systems/4204980/Real-Time-Linux-Scheduling-)g[n/operating-systems/4204980/Real-Time-Linux-Scheduling-](http://www.embedded.com/design/operating-systems/4204980/Real-Time-Linux-Scheduling-) Part-3

**Bibliografía**

A. (2021, 2 noviembre). *¿Qué es una tarea en Linux?* CompuHoy.com. https://www.compuhoy.com/que-es-una-tarea-en-linux/

Aelterman, S. (2019, 17 junio). *Processes, Threads, and Jobs in the Windows Operating System | Microsoft Press Store*. Microsoftpressstore.Com. https://www.microsoftpressstore.com/articles/article.aspx?p=2233328

*APT*. (s. f.). help.ubuntu.com. https://help.ubuntu.com/kubuntu/desktopguide/es/apt-get.html

Canonical, C. (s. f.). *Ubuntu Manpage: dpkg - Gestor de paquetes de Debian*. Manpages.Ubuntu.Com. http://manpages.ubuntu.com/manpages/trusty/es/man1/dpkg.1.html

Corbet. (2004, 18 febrero). *No more 24-bit atomic\_t*. Lwn.Net. https://lwn.net/Articles/71732/

Corob, C.-M. (2021, 3 agosto). *#define directive (C/C++)*. Microsoft Docs. https://docs.microsoft.com/en-us/cpp/preprocessor/hash-define-directive-c-cpp?view=msvc-170

Corob, C.-M. (2022, 2 mayo). *Directivas de preprocesador*. Microsoft Docs. https://docs.microsoft.com/es-es/cpp/preprocessor/preprocessor-directives?view=msvc-170

Devored, E. (2013, 4 octubre). *What is the concept of vruntime in CFS*. Stack Overflow. https://stackoverflow.com/questions/19181834/what-is-the-concept-of-vruntime-in-cfs

*FAQ/LikelyUnlikely - Linux Kernel Newbies*. (2017, 30 diciembre). kernelnewbies.org. https://kernelnewbies.org/FAQ/LikelyUnlikely

GeeksforGeeks. (2019, 21 noviembre). *Memory mapped I/O and Isolated I/O*. https://www.geeksforgeeks.org/memory-mapped-i-o-and-isolated-i-o/

GeeksforGeeks. (2022, 13 febrero). *Structures in C*. https://www.geeksforgeeks.org/structures-c/

González, P. (2013, 20 febrero). *Punteros en C*. lsi.vc.ehu.eus. Recuperado 6 de mayo de 2022, de https://lsi.vc.ehu.eus/pablogn/docencia/manuales/C/Punteros\_en\_C.pdf

Kernel. (s. f.-a). *Atomic\_t*. kernel.org. https://www.kernel.org/doc/Documentation/atomic\_t.txt

Kernel. (s. f.-b). *Deadline Task Scheduling — The Linux Kernel documentation*. Kernel.Org. https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html#scheduling-algorithm

Lynxbeedev, L. (2021, 16 octubre). *Understanding Linux kernel likely and unlikely macros*. Lynxbee. https://lynxbee.com/understanding-linux-kernel-likely-and-unlikely-macros/#.Ync-NlTMJEY

NI. (2021, 15 enero). *Process Isolation on the NI Linux Real-Time OS*. https://www.ni.com/es-cr/support/documentation/supplemental/16/process-isolation-on-the-ni-linux-real-time-os.html

*Preprocessor directives - C++ Tutorials*. (s. f.). cplusplus.com. https://www.cplusplus.com/doc/tutorial/preprocessor/

*Real-Time | The Linux Process Scheduler | InformIT*. (2013, 13 noviembre). Informit. https://www.informit.com/articles/article.aspx?p=101760&seqNum=4

Staff, E. (2010, 26 julio). *Implementing a new real-time scheduling policy for Linux: Part 1*. Embedded.Com. https://www.embedded.com/implementing-a-new-real-time-scheduling-policy-for-linux-part-1/

Staff, E. (2022, 2 febrero). *Implementing a new real-time scheduling policy for Linux: Part 2*. Embedded.Com. https://www.embedded.com/implementing-a-new-real-time-scheduling-policy-for-linux-part-2/

*Task\_struct en Linux Kernel - programador clic*. (s. f.). programmerclick.com. https://programmerclick.com/article/99732488565/

The Single UNIX. (s. f.). *sched.h - execution scheduling (REALTIME)*. pubs.opengroup.org. https://pubs.opengroup.org/onlinepubs/7908799/xsh/sched.h.html

W3 Schools. (s. f.). *C Structures (structs)*. W3schools.Com. https://www.w3schools.com/c/c\_structs.php