Universidad del Valle de Guatemala
Sistemas Operativos
Docentes: Ing. Juan Luis Garcia Zarceño
Semestre I, 2022

Laboratorio #5

Fecha de Entrega: 30 de marzo, 2022.

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-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:

a. 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.
b. 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.
c. 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`.
d. 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` o `http://security.ubuntu.com` por `http://old-releases.ubuntu.com`.
e. 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
```

f. 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}
```

g. 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 `linux2.6.24-casio`. En adelante nos referiremos a esta carpeta donde está el *kernel* extraído como `kernel_dir`.

h. 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:

```
menu “CASIO Scheduler” config SCHED_CASIO_POLICY
     bool “CASIO scheduling policy”  default y
endmenu
```

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`.**
  La palabra reservada struct indica se est´a definiendo una estructura. El identificador ejemplo es el nombre de la estructura. Las variables declaradas dentro de las llaves de la definici´on de estructura son los miembros de la es tructura. Los miembros de la misma estructura deben tener nombres ´unicos mientras que dos estructuras diferentes pueden tener miembros con el mismo nombre. Cada definici´on de estructura debe terminar con un punto y coma. La definici´on de

struct ejemplo contiene un miembro de tipo char y otro de tipo int. Los miembros de una estructura pueden ser variables de los tipos de datos b´asicos (int, char, float,etc) o agregados como ser arreglos y otras estructuras. Una estructura no puede contener una instancia de si misma.

```
struct [structure tag] {

   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

- **Propósito y directivas del preprocesador.**
  El compilador llama primero al cpp y procesa directivas que pueden se usadas con cualquier otro tipo de archivo. Este procesador utiliza fases de traducción. Esas son el toquenizado léxico, empalmado de líneas y manejo de directivo.

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

  Las “” se usan cuando se quiere referir a un puntero del espacio de memoria y el & se usa cuando se quiere referir directamente a la dirección de memoria.

- **Propósito y modo de uso de APT y `dpkg`.**

  dpkg es el programa base para manejar paquetes Debian en el sistema. dpkg es lo que permite instalar o analizar sus contenidos. Pero este programa sólo tiene una visión parcial del universo Debian: sabe lo que está instalado en el sistema y lo que sea que se le provee en la línea de órdenes, pero no sabe nada más de otros paquetes disponibles. Como tal, fallará si no se satisface una dependencia. Por el contrario, herramientas como apt y aptitude crearán una lista de dependencias para instalar todo tan automáticamente como sea posible.

i. 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__ ...
```

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`.

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

```
...
# define SCHED_BATCH 3
#endif

#define SCHED_CASIO         6
```

- **¿Cuál es el propósito de los archivos `sched.h` modificados?**
  Sched,h define la estructura de sched param. Este contiene los parámetros requeridos para usar las políticas de programación soportadas. El struct contiene una variable que define la prioridad de calendarización llamada sched_priority.Las 3 politicas predefinidas son FIFO, Round Robin y otras.

- **¿Cuál es el propósito de la definición incluida y las definiciones existentes en el archivo?**
  Sched_batch puede tener prioridad estática o dinámica. Esta política hará que se asuma que el subproceso en intensivo en cpu.

k. 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
};
```

- **¿Qué es una *task* en Linux?**
  El término tarea se usa en el kernel de Linux para referirse a una unidad de ejecución, que puede compartir varios recursos del sistema con otras tareas en el sistema. Según el nivel de compartición, la tarea puede considerarse como un hilo o proceso convencional.

- **¿Cuál es el propósito de `task_struct` y cuál es su análogo en Windows?**
  El kernel asigna la estructura task_struct a través del asignador de losas, que puede lograr el propósito de reutilizar objetos y colorear el caché.

  El asignador de losas aquí es un método para que el núcleo asigne memoria. La asignación de la memoria del núcleo generalmente se obtiene del grupo de memoria libre. Hay dos formas principales: sistema de amigos y asignación de losas.
  La estructura task_struct está declarada en include/linux/sched.h y es actualmente de un tamaño de 1680 bytes.

  El análogo es el descriptor de proceso.

l. 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`.

- **¿Qué información contiene `sched_param`?**
  Usará la estructura sched_param cuando obtenga o establezca los parámetros de programación para un hilo o proceso.

m. Diríjase al archivo `kernel_dir/kernel/sched.c`. La política de calendarización que emplearemos es la de *earliest deadline first* (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;
}
```

- **¿Para qué sirve la función `rt_policy` y para qué sirve la llamada `unlikely` en ella?**
  La función rt_policy (definida en el archivo /kernel-source-code/kernel/sched.c) se utiliza para decidir si una determinada política de programación pertenece a la clase de tiempo real (SCHED_RR y SCHED_FIFO) o no.

- **¿Qué tipo de tareas calendariza la política EDF, en vista del método modificado?**
  El EDF calendariza tareas de la cola que están mas cercanas a su fecha limite. Por eso se llama earliest deadline first.

n. Nuestra política será implementada en un archivo llamado `sched_casio.c`. Modifique `sched.c` de la siguiente manera:

```
...
# 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.
...
```

- **Describa la precedencia de prioridades para las políticas EDF, RT y CFS, de acuerdo con los cambios realizados hasta ahora.**
  1. EDF
  2. RT
  3. CFS
  4. IDLE

o. 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
}
```

Y en la función `sched_setscheduler` realice las siguientes modificaciones:

```
... 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 ...
```

p. 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.
...
```

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`.**
  Está compuesto por el nodo del task, nodo en la cabexa de la lista de tasks, un deadline y un puntero a su respectivo task_struct.

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

```
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`.**
  El propósito es que el sistema se pueda referir a tareas calendarizadas usando la nueva política. Esta compuesta de la cabeza de la lista de casio tasks, la tarea raíz y un identificador atomic_t.

- **¿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).**
  El tipo atómico proporciona una interfaz a los medios de arquitectura atómica.
  Operaciones RMW entre CPU (las operaciones atómicas en MMIO no son compatibles y puede conducir a trampas fatales en algunas plataformas). leer-modificar-escribir es una clase de operaciones atómicas (como probar y configurar, buscar y agregar y comparar e intercambiar) que leen una ubicación de memoria y escriben un nuevo valor en ella simultáneamente, ya sea con un valor completamente nuevo o alguna función del valor anterior. el mappeo es un metodo complementario para realizar entrada/salida (e/s) en una computadora, incluida la CPU y la E/S del dispositivo.otro método es el uso de E/S dedicadas (canales) de procesadores.

r. 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);
...
```

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

s. Los diferentes calendarizadores de Linux se inicializan en la función `sched_init`. Modifique esta función de la siguiente forma:

```
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); ...
```

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

t. 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);
}
```

u. 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;
}
```

v. 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;
}
```

w. 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           = load_balance_casio,
      .move_one_task          = move_one_task_casio,
#endif

      .set_curr_task          = set_curr_task_casio,
      .task_tick              = task_tick_casio,
};
```

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

  Apunta a una lista slingly-linked en donde están los identidicadores para las siguientes tasks que debe de realizar el RTS.

x. 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?**
  se obtiene el puntero a la tarea de estructura casio almacenada en la lista vinculada de la estructura casio_rq que apunta a la tarea p. Luego, actualiza los datos e inserta casio_task en el árbol rojo-negro (insert_casio_task_rb_tree).

```
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?**
  Inicialmente, se asegura que sea sea el identificador de una sched_casio. Segundo, se verifica que haya una task con deadline cerca y que existe una lista de tasks. Si no se logra obtener un t y curr, se preemtea.

```
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)
{
}
```

y. 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
```

z. 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.

2. 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.
3. En enqueue_task_casio declare msg donde está //char msg, y donde 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.
4. 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/linux2.6.24-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.

- 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.
- 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`

- **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.**

  El aislamiento de procesos en la programación de computadoras es la segregación de diferentes procesos de software para evitar que accedan al espacio de memoria que no poseen. El concepto de aislamiento de procesos ayuda a mejorar la seguridad del sistema operativo al proporcionar diferentes niveles de privilegios a ciertos programas y restringir la memoria que esos programas pueden usar. El funcionamiento básico del aislamiento de procesos implica asignar a un proceso o programa un espacio de direcciones virtuales claramente definido. Este espacio contiene el programa y todos los datos relacionados. Si el proceso requiere más espacio, se solicita al sistema operativo y se asigna si está disponible. De esta manera, el sistema operativo puede evitar que dos procesos accedan accidental o intencionalmente a la memoria del otro.

Fuente:

- http://www.embedded.com/design/operating-systems/4204929/Real-Time-Linux-SchedulingPart-1
- https://www.embedded.com/design/operating-systems/4204971/Real-Time-Linux-SchedulingPart-2
- https://www.embedded.com/design/operating-systems/4204980/Real-Time-Linux-SchedulingPart-3