DESARROLLO DE SOFTWARE PARA SISTEMAS EMPOTRADOS

Práctica 4: Caracterización de la plataforma

Héctor Pérez
Michael González

Héctor Pérez y Michael González

Objetivos

- Medidas de parámetros característicos de la plataforma
  - latencia del kernel en la respuesta a eventos
    - atención de interrupciones
    - cambios de contexto
- Desarrollo de software para sistemas empotrados basado en mecanismos de interrupción
  - introducción a los módulos del kernel de Linux
  - introducción al manejo de los GPIO
  - desarrollo de módulos sencillos que nos permitan comunicarnos con los dispositivos de forma directa

Héctor Pérez y Michael González

Introducción

- El **sistema operativo influye** en los tiempos de respuesta del código de usuario
  - gestión de eventos internos y externos
    - *tick* del sistema
    - interrupciones
  - operaciones adicionales proporcionadas por librerías
  - operaciones ejecutadas a distinta prioridad
    - por ejemplo, servicios del kernel
  - operaciones ejecutadas en exclusión mutua

Héctor Pérez y Michael González

Introducción

- Sistema operativo Linux
  - Kernel configurado como ***CONFIG_PREEMPT***
    - La mayor parte del código del kernel es expulsable
      - No se espera a que se termine de ejecutar el código del kernel para ejecutar el planificador
      - Excepciones: *secciones críticas del kernel*
    - Utilizado en sistemas con requisitos de latencia en torno a varios milisegundos
  - Todas las operaciones de configuración de los aspectos de tiempo real de una aplicación deben realizarse con privilegios de administrador
    - por ejemplo, las políticas y los parámetros de planificación

Héctor Pérez y Michael González

Latencia en el kernel

- Herramienta *cyclictest*
  - herramienta habitual en sistemas basados en Linux
  - mide la latencia de respuesta a un evento
    - tiempo entre la generación de la interrupción y la ejecución de la aplicación

```
clock_gettime((&now))
next = now + par->interval
while (!shutdown) {
        clock_nanosleep((&next))
        clock_gettime((&now))
        diff = calcdiff(now, next)
        … # update stats
        next += interval
}
```

  - disponible en el paquete ***rt-tests***

Héctor Pérez y Michael González

Latencia en el kernel

- *cyclictest --help*
  - *-n* *utiliza clock_nanosleep*
  - -p NUM prioridad del thread
  - -h genera un histograma en la salida stdout
  - -D NUM duración del test en segundos
  - -m desactiva la paginación a memoria SWAP (siempre en RAM)
  - -M sólo actualiza los resultados al obtener un peor caso

Héctor Pérez y Michael González

Estimación de la latencia en el kernel

- Crear una aplicación *carga_cpu* que simule carga en la CPU
  - sus argumentos de entrada *por el terminal* serán la prioridad, la carga y la duración
  - puede utilizarse la función *load* de la librería *ev3c-addons.a*
  - debe utilizar la política de planificación *SCHED_FIFO*
- Ejecutar la aplicación *carga_cpu* y *cyclictest* para estimar la latencia de ev3dev en los siguientes escenarios
  - razonar los resultados obtenidos

Prioridad de *cyclictest*	MAX	MAX	MAX	MAX-1	MAX	MAX
Prioridad de *carga_cpu*	MAX/2	MAX-1	MAX	MAX	MAX/2	MAX/2
Porcentaje de carga	*10*	*10*	10	*10*	50	85

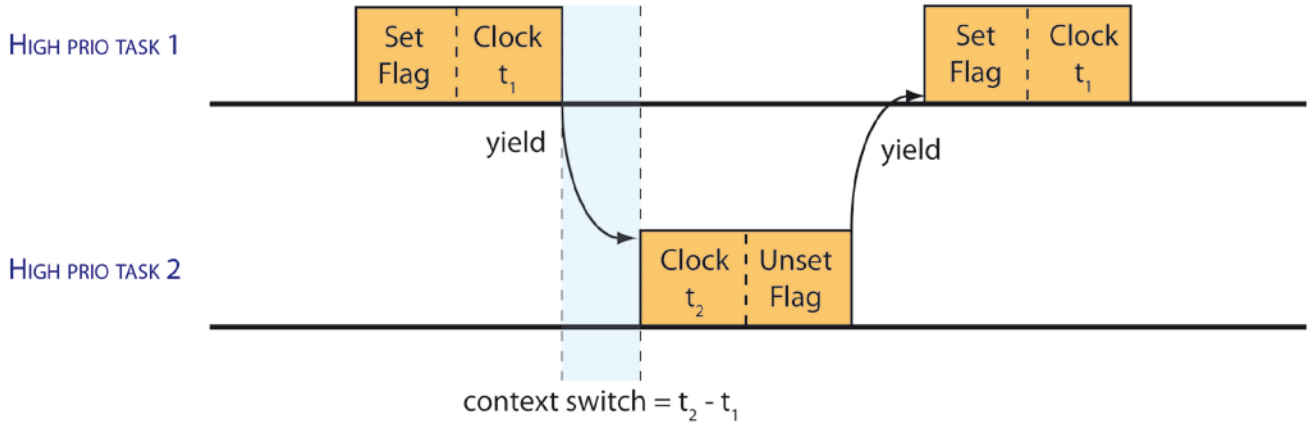
Héctor Pérez y Michael González

Estimación de la periodicidad del tick del sistema

- Compilar y ejecutar la aplicación *tick.c* proporcionada
  - Explica el algoritmo del código proporcionado
    - ilustra el algoritmo mediante una figura
  - Razona los resultados obtenidos
    - En caso de que se detecte algún *overhead adicional* en el sistema, modifica la configuración de la aplicación para estimar si se trata de una sobrecarga periódica o esporádica del sistema
  - Relaciona los resultados obtenidos con el flag de configuración del kernel *CONFIG_HZ*.

Héctor Pérez y Michael González

Estimación del cambio de contexto: algoritmo



- Las tareas se ejecutan a la misma prioridad
  - al invocar el subprograma *yield* ceden el procesador a otra tarea y se encolan de nuevo en la cola de tareas listas para ejecutar del planificador
- Puede sincronizarse la inicialización mediante un flag
  - dado que la ejecución de las tareas es controlada mediante *yield*, no se utilizarán mecanismos de acceso concurrente seguro por su alto *overhead* en relación con el coste del cambio de contexto

Héctor Pérez y Michael González

Estimación del cambio de contexto: pseudocódigo

Thread 1

Sincronización inicial;

mientras haya espacio en el buffer **loop**
    Activar el flag;
    Medir tiempo t1;
    Ceder el procesador;
end loop;

Thread 2

Sincronización inicial;

mientras haya espacio en el buffer **loop**
    si el flag está desactivado **then**
        ceder el procesador;
    end if;
    Medir tiempo t2;
    Desactivar el flag;
    Calcular el cambio de contexto;
    Almacenar medida en el buffer;
end loop;

Procesar todas las medidas;

Héctor Pérez y Michael González

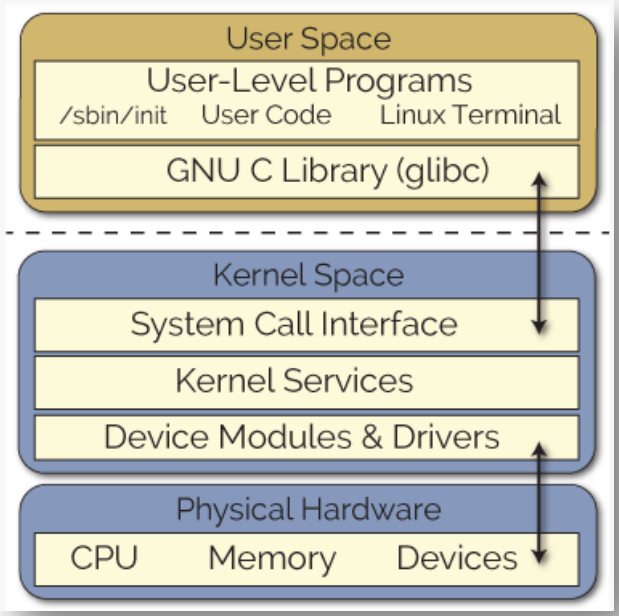
Estimación del cambio de contexto: desarrollo

- Completar el fichero *yield.c* proporcionado con el algoritmo descrito anteriormente
  - Ejecutar la aplicación y estimar los valores mínimo, máximo y promedio del coste de cambio de contexto
  - Razonar los resultados relacionándolos con los obtenidos en el apartado anterior

Héctor Pérez y Michael González

Módulos del kernel

- El kernel de Linux es **modular**
  - permite insertar/eliminar módulos de forma dinámica
  - cada módulo proporciona una serie de funcionalidades
    - por ejemplo, el manejo de un dispositivo (*driver*)
- El acceso a dispositivos es posible desde el *espacio de usuario*
  - mediante ficheros virtuales
    - *proporcionado por sysfs*
  - menor eficiencia
  - sin soporte para el manejo de interrupciones



Héctor Pérez y Michael González

Entorno de desarrollo de módulos del kernel

- Dependencias:
  - *build-essential*: proporciona la herramienta *make*
  - *linux-headers*: instalar las afines a la versión del *kernel*
- *Preparación del entorno:*
  - *Enlaces simbólicos para la compilación de módulos*

    >> sudo ln -s /usr/src/linux-headers-`uname -r` /lib/modules/ `uname –r`/build
  - *Configuración del entorno para compilar nativamente*

    >> sudo cp Makefile.Patch /usr/src/linux-headers-`uname –r`/scripts/Makefile

    >> cd /usr/src/linux-headers-`uname -r`

    >> sudo make headers_check; sudo make headers_install ; sudo make scripts

Héctor Pérez y Michael González

Desarrollo y ejecución de módulos del kernel

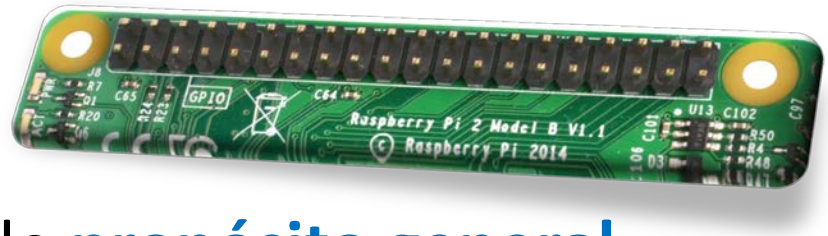
- Ejecución del ejemplo proporcionado en *moodle*
  - compila el módulo mediante **make**
  - inserta el módulo en el kernel mediante **insmod**

    >> sudo insmod modulo.ko
  - obtén información sobre el módulo
    - listado de módulos activos >> *lsmod*
    - información del módulo >> */sbin/modinfo modulo.ko*
    - salida de mensajes >> *dmesg*
  - elimina el módulo del kernel mediante **rmmod**

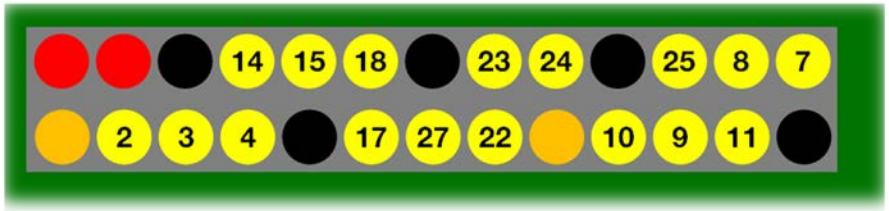
    >> sudo rmmod modulo.ko

Héctor Pérez y Michael González

GPIO



- Conjunto de pines para I/O de **propósito general**
  - permiten controlar los dispositivos
  - su funcionalidad es totalmente configurable
    - pueden configurarse como entrada o salida
    - pueden asociarse interrupciones
    - pueden utilizar distintos protocolos (SPI, I2C, UART, etc)
  - detalles en la **documentación** de cada dispositivo



Héctor Pérez y Michael González 20/04/2016

GPIO API en Linux

Basic facilities

IRQ-related facilities

```
/*
* Interface defined in linux/gpio.h
*/

int  gpio_is_valid (int number);
int  gpio_request  (unsigned gpio, const char *label);
void gpio_free     (unsigned gpio);

int gpio_direction_input  (unsigned gpio);
int gpio_direction_output (unsigned gpio, int value);

int  gpio_get_value (unsigned gpio);
void gpio_set_value (unsigned gpio, int value);

int  gpio_export  (unsigned gpio, bool direction_may_change);
void gpio_unexport(unsigned gpio);

int gpio_to_irq (unsigned gpio);
int irq_to_gpio (unsigned irq);

int gpio_set_debounce(unsigned gpio, unsigned debounce);
```

Héctor Pérez y Michael González

Uso de GPIO a través de módulos del kernel

- Descarga la aplicación ***gpio.c*** a través de moodle
  - explica el funcionamiento de la aplicación
- Realiza la configuración de la aplicación
  - debe utilizar el GPIO asociado al pin *I2C Data* del *puerto 2*
    - consulta la documentación correspondiente
  - debemos desactivar el uso del driver (*priv. administrador*)

    >> sudo su

    >> echo "raw" > /sys/class/lego-port/port1/mode
- Compila e inserta el módulo en el kernel
  - conecta previamente el sensor de contacto al puerto
  - en otro terminal, ejecuta el script de monitorización de GPIOs

    >> ./monitor_gpio.sh gpionum

Héctor Pérez y Michael González

Interrupciones

- Visión general del ***manejo de interrupciones***
  - El *procesador* detecta una señal eléctrica generada por el dispositivo y la mapea al número de interrupción (*IRQ*)
    - si la IRQ está registrada en el kernel, se ejecuta la rutina de manejo de interrupción correspondiente
    - en caso contrario, se ignora

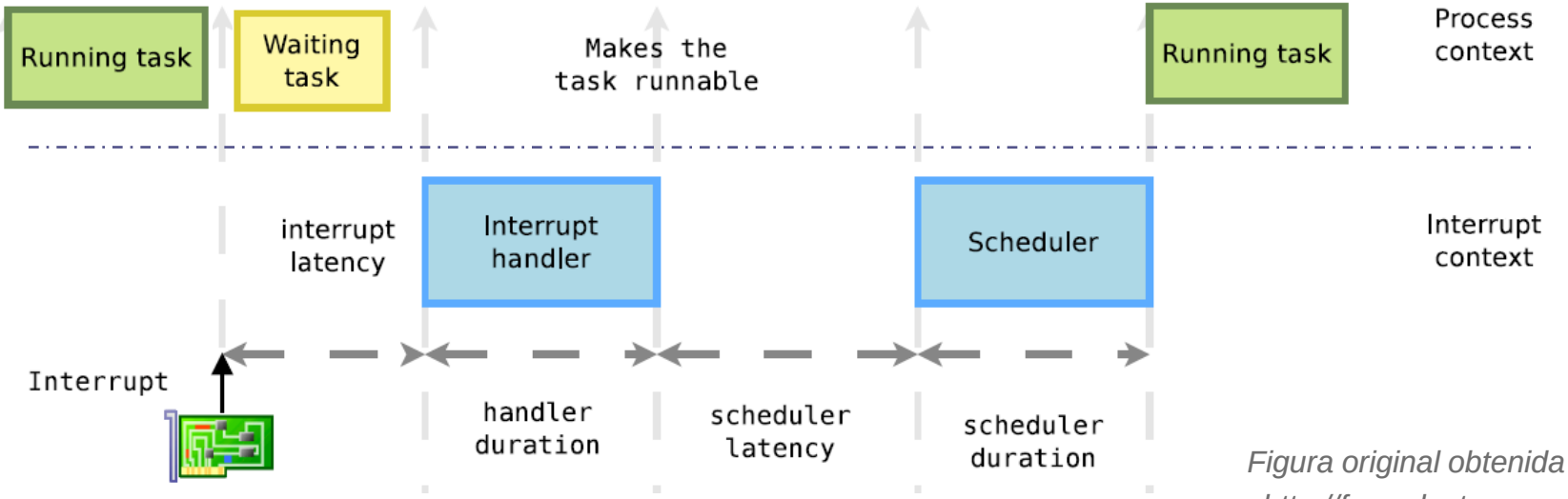


Figura original obtenida en http://free-electrons.com

Héctor Pérez y Michael González

Manejo de interrupciones en Linux

Conceptos generales

- **Se ejecuta en contexto de interrupción** *(por lo que el código debe de ser eficiente)*
- **Restrictivo en el uso de servicios del sistema operativo** *(funciones de suspensión, mecanismos de exclusión mútua, memoria dinámica, etc.)*

```
/*
* Interface defined in linux/interrupt.h
* Configuration flags defined in linux/irq.h
*/

/*
 * request_irq: register a given interrupt handler on a given interrupt line
 *  Flags:  IRQF_TRIGGER_NONE, IRQF_TRIGGER_RISING, IRQF_TRIGGER_FALLING, etc.
 */
int request_irq (unsigned int irq, irq_handler_t handler,
                 unsigned long flags, const char *name, void *dev);

/*
 * free_irq: unregister the associated interrupt handler
 */
void free_irq  (unsigned int irq, void *dev);
```

Héctor Pérez y Michael González

Implementación de un manejador de interrupción

```
/*
* Declaration of interrupt handler routine
*/
typedef irqreturn_t (*irq_handler_t) (int, void *);
```

- Parámetros de la función
  - irq — número de IRQ asociado a la interrupción
  - dev — parámetros del dispositivo
- Valor de retorno
  - IRQ_NONE — si el driver no atiende la interrupción
  - IRQ_HANDLED — si la interrupción ha sido atendida completamente

Ejemplo:

```
static irqreturn_t my_handler_interrupt (int irq, void *dev)
{
    printk(KERN_INFO "Received IRQ number %d\n", irq);
    return IRQ_HANDLED;
};
```

Héctor Pérez y Michael González

Desarrollo de un manejador de interrupciones

- Modificar la aplicación **gpio.c** para que maneje las interrupciones provenientes del sensor de contacto
  - obtener la IRQ asociada al GPIO utilizado como entrada
    - *pin I2C Data* del *puerto 2*
  - características de la rutina de interrupción
    - se invoca cuando se detecta un flanco de subida en la señal
    - escribe un mensaje en el log del sistema
- Insertar el módulo en el kernel
  - Comprobar que la rutina de interrupción se ha registrado correctamente

    >> cat /proc/interrupts
  - El log del sistema se puede consultar con **dmesg**

Héctor Pérez y Michael González

Eficiencia en el manejo de interrupciones con Linux

- Linux propone dos niveles de procesamiento
  - ***Top half***: rutina que atiende la IRQ en primera instancia
    - se ejecuta con IRQs deshabilitadas: debe proporcionar funcionalidad básica
    - fuertes restricciones en el uso de servicios del OS
  - ***Bottom half***: rutina que implementa la funcionalidad requerida
    - no se ejecuta en contexto de interrupción
      - IRQs habilitadas, uso de servicios del OS
    - diferentes estrategias (softirq, tasklet, workqueue, *kernel thread*)

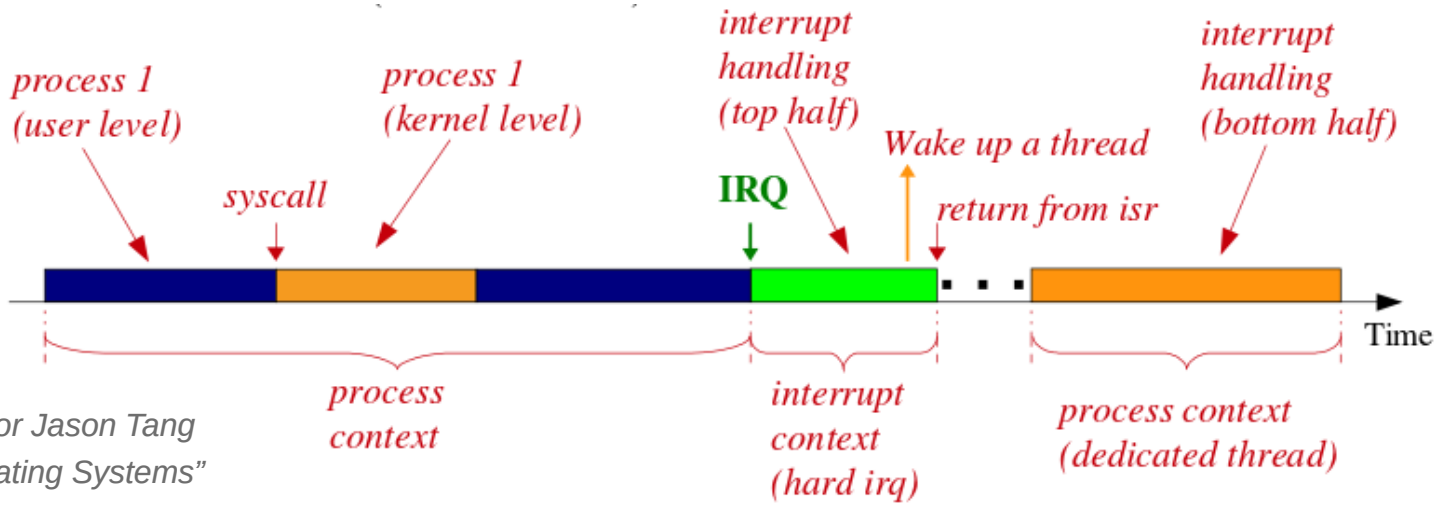


Figura realizada por Jason Tang "Principles of Operating Systems"

Héctor Pérez y Michael González

Manejo de interrupciones con threads

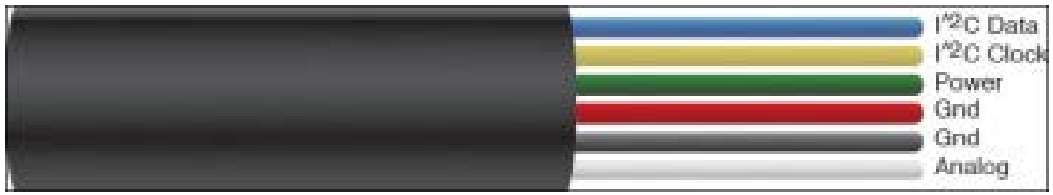
```
int request_threaded_irq (unsigned int irq, irq_handler_t handler,
                          irq_handler_t thread_fn,
                          unsigned long flags, const char *name, void *dev);
```

- El nuevo parámetro *thread_fn* representa el bottom_half
  - rutina de manejo de la interrupción mediante un thread
    - se ejecuta en el contexto de proceso
    - puede utilizar los servicios del sistema operativo
- El parámetro *handler* representa el *top_half*
  - en general, debe deshabilitar las interrupciones del dispositivo
    - el flag IRQF_ONESHOT no habilita la IRQ hasta finalizar el *bottom_half*
  - valor de retorno
    - IRQ_NONE si el driver no atiende la interrupción
    - IRQ_HANDLED si la interrupción ha sido atendida completamente
    - IRQ_WAKE_THREAD si el *bottom_half* debe terminar de procesar la IRQ

Héctor Pérez y Michael González

Estimación del tiempo de atención a interrupciones

- Modificar la aplicación ***gpio.c*** para que el manejo de la interrupción producida por el sensor de contacto se realice mediante un thread
  - indica cómo podríamos filtrar activaciones espurias de la interrupción
- Modificar la aplicación ***gpio.c*** para que pueda medirse el tiempo de atención a interrupciones con un osciloscopio
  - activar una señal en el pin 1 de salida del puerto 2
  - medir la diferencia temporal entre los flancos de subida de las señales de entrada y salida



Bibliografía

- "Linux Kernel Development", *Robert Love*
  - La última edición data de 2010, por lo que está algo desactualizado
- Curso "Principles of Operating Systems", *Jason Tang*
- Curso "Writing a Linux Kernel Module", *Derek Molloy*