"Linux Interrupts: The Basic Concepts"

Miguel Angel Jimenez Morales
Dikersson Alexis Cañon Vanegas
Sistemas Operativos

El documento explica conceptos fundamentales de interrupciones en Linux (kernel 2.4.18-10). Las interrupciones son eventos asíncronos generados por dispositivos de hardware, mientras que las excepciones son eventos síncronos generados durante la ejecución de instrucciones. Las interrupciones se clasifican en enmascarables y no enmascarables, mientras que las excepciones incluyen fallos, trampas y abortos.

Linux maneja interrupciones dividiendo su tratamiento en dos partes: un manejador rápido para hardware (Top Half) y una segunda parte aplazada para manejo posterior (Bottom Half, softirqs y tasklets). También describe brevemente el manejo de interrupciones en sistemas multiprocesador (SMP), usando controladores avanzados (APIC) e interrupciones interprocesador (IPIs).

1. ¿Qué es una interrupción? ¿Por qué son necesarias las interrupciones? ¿Qué tipos existen y cuáles son sus características?

- Interrupción: Es un evento asíncrono generado típicamente por dispositivos de hardware que detiene temporalmente la ejecución actual del procesador para atender tareas prioritarias.

- Necesarias: Permiten al procesador responder rápidamente a eventos externos críticos, mejorar la eficiencia y no desperdiciar tiempo esperando a que ocurran ciertos eventos.

- Tipos:

- Enmascarables: Se pueden bloquear temporalmente usando un indicador IF.

- No enmascarables: No se pueden bloquear y siempre tienen prioridad máxima.

- Características: Prioridad, rapidez, tratamiento en dos fases (inmediata y aplazada).

2. ¿Qué son los "exception handler"?

- Son funciones específicas que maneja el kernel al ocurrir excepciones.
- Cada excepción tiene un manejador que determina cómo actuar frente al error (corregir, informar, terminar el proceso).
- Normalmente emiten señales UNIX al proceso involucrado.

3. ¿Qué son las interrupciones generadas por software? ¿Para qué sirven y qué algoritmos o diagramas de flujo son usados para el manejo de interrupciones?

- Generadas por software: Son interrupciones activadas explícitamente por instrucciones (INT n), no por dispositivos externos.

- Sirven: Principalmente para realizar llamadas al sistema (syscalls) o para implementar trampas para depuración.

- Algoritmos/diagramas usados:

    - Guardar estado del registro.

    - Ejecutar manejador específico (ISR).

    - Restaurar estado tras finalizar.

    - Uso de funciones comunes como do_IRQ() o do_softirq() para tratamiento eficiente.

4. ¿Qué son las IRQ y las estructuras de datos en relación con las interrupciones?

- IRQ (Interrupt Request): Son solicitudes de interrupción generadas por hardware. Se asocian a vectores específicos y pueden compartirse entre varios dispositivos.

- Estructuras de datos: Incluyen descriptores de interrupción (irq_desc_t), listas de acciones (irqaction), descriptores de tipo de interrupción (hw_interrupt_type), y la tabla de descriptores de interrupción (IDT). Estas estructuras permiten al kernel gestionar eficientemente las interrupciones y asociarlos manejadores específicos.

"BOOTKITS: PAST, PRESENT & FUTURE":

El artículo examina la evolución de los bootkits, un tipo de malware que infecta los sectores de arranque para lograr persistencia y evadir detección. Aborda su desarrollo desde los primeros virus de sector de arranque en los 80 hasta los bootkits modernos que atacan sistemas con UEFI (Unified Extensible Firmware Interface). También explica cómo han eludido medidas de seguridad como la firma obligatoria de controladores en Windows de 64 bits.

Puntos clave analizados

1. Origen y evolución histórica
    - Los primeros bootkits surgieron a partir de virus de sector de arranque en sistemas como Apple II y MS-DOS (ej: Elk Cloner, Brain).
    - Evolucionaron para sortear restricciones modernas, como la firma digital de controladores en Windows de 64 bits.
2. Clasificación de bootkits
    - MBR bootkits: Infectan el Master Boot Record (ej: Mebroot, TDL4).

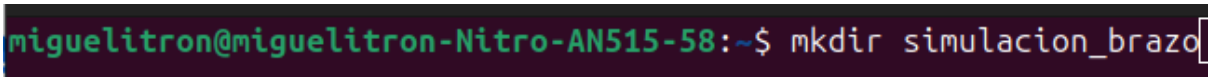
    - VBR bootkits: Corrompen el Volume Boot Record o el Initial Program Loader (ej: Rovnix, Gapz).
3. Casos destacados
    - TDL4: Sobrescribe el MBR para cargar drivers maliciosos antes del arranque del sistema.
    - Rovnix: Primer bootkit basado en VBR con técnicas avanzadas de hooking.
    - Gapz: Bootkit sigiloso que apenas modifica el VBR y usa cifrado y comunicación en red.
    - DreamBoot: Primer bootkit de prueba para UEFI en Windows 8.
4. Seguridad en UEFI
    - UEFI reemplaza al BIOS tradicional y usa particiones GPT y cargadores firmados.
    - Sin embargo, ya existen bootkits que infectan UEFI (ej: DreamBoot).
    - Principales vectores de ataque:
        - Sustitución del bootloader.
        - Explotación de drivers DXE.
        - Manipulación de Option ROMs.
5. Herramientas defensivas y forenses
    - CHIPSEC: Framework de Intel para auditar BIOS/UEFI y análisis forense.
    - Hidden File System Reader: Herramienta para extraer almacenamiento oculto usado por bootkits como TDL4, Rovnix y Flame.
6. Futuro de las amenazas
    - Secure Boot mitiga algunos ataques, pero los atacantes adaptan sus métodos.
    - Equipos antiguos sin Secure Boot siguen en riesgo.
    - UEFI representa una nueva superficie de ataque debido a la falta de actualizaciones generalizadas.

Simulación y despliegue de un brazo robótico con PyBullet:

1. Generar el corrimiento de la aplicación: Se desarrolló una simulación básica de un brazo robótico utilizando la librería PyBullet, que permite crear entornos de física realista en 3D. El archivo principal, brazo_robotico.py, incluye la carga de un plano base y un robot de dos articulaciones modelado mediante un archivo .urdf personalizado (two_joint_robot_custom.urdf).

La simulación fue ejecutada localmente utilizando un entorno virtual de Python, con los siguientes pasos:

1) Se crea una carpeta para el proyecto nuevo:



2) Se ingresa a la carpeta:

```
miguelitron@miguelitron-Nitro-AN515-58:~$ cd brazo_robotico
```

3) Se crea un entorno Virtual de Python

```
miguelitron@miguelitron-Nitro-AN515-58:~$ python3 -m venv venv
```

4) Activar el entorno virtual

```
miguelitron@miguelitron-Nitro-AN515-58:~$ source venv/bin/activate
```

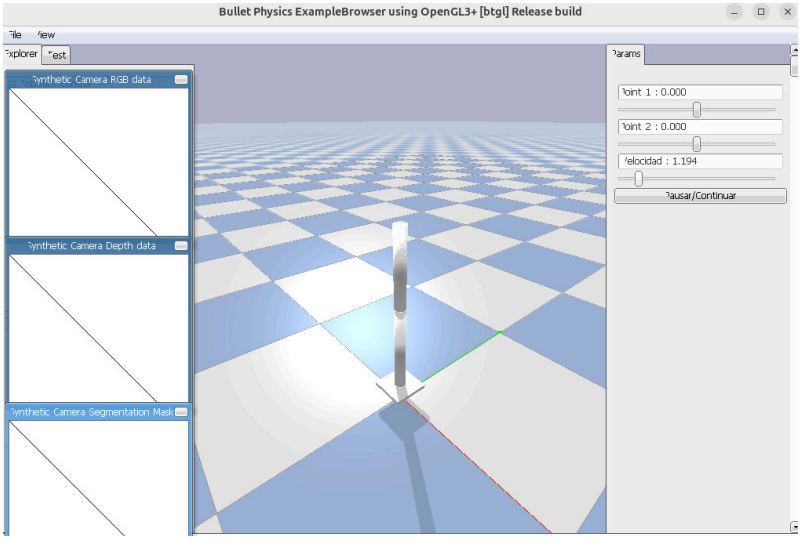
5) Instalar la librería PyBullet

```
miguelitron@miguelitron-Nitro-AN515-58:~$ pip install pybullet
```

6) Se ejecuta la aplicación:

```
miguelitron@miguelitron-Nitro-AN515-58:~$ python brazo_robotico.py
```

La simulación muestra la interacción de un brazo robótico sobre un plano con control básico de sus articulaciones.



Código para el movimiento del brazo robótico:

```
import pybullet as p

import pybullet_data

import time

import random


def run_random_sequence(total_moves=8, transition_steps=500, hold_steps=120,
sleep=1./240.):

    """

    Simula un brazo robótico de 2 articulaciones que realiza movimientos aleatorios
suaves.


    Parámetros:

    - total_moves: Cantidad de movimientos a ejecutar

    - transition_steps: Pasos de simulación para transición entre poses

    - hold_steps: Pasos para mantener cada pose

    - sleep: Tiempo entre pasos de simulación (controla velocidad)

    """


    # Conexión al simulador PyBullet en modo visualización (GUI)

    p.connect(p.GUI)


    # Configuración física del entorno

    p.setGravity(0,0,-9.8)  # Gravedad en dirección -Z (9.8 m/s²)


    # Habilita acceso a los archivos de datos de PyBullet

    p.setAdditionalSearchPath(pybullet_data.getDataPath())
```

```
# Posiciona la cámara (distancia, azimuth, elevación, target)
p.resetDebugVisualizerCamera(1.5, 50, -35, [0,0,0.5])

# Carga un plano como superficie de referencia
p.loadURDF("plane.urdf")

# Carga el modelo del robot desde archivo URDF
# [0,0,0.1] = posición inicial ligeramente elevada
# useFixedBase=True = el robot no se caerá
robot = p.loadURDF("two_joint_robot_custom.urdf", [0,0,0.1], useFixedBase=True)

# Estado inicial de las articulaciones (0,0)
prev = (0.0, 0.0)

# Bucle principal de movimientos
for move in range(total_moves):
# Genera ángulos aleatorios para ambas articulaciones (en radianes)
target = (random.uniform(-3.14,3.14), random.uniform(-3.14,3.14))
print(f"Movimiento {move+1}/{total_moves}: {target}")

# Interpolación suave entre la posición anterior y la nueva
for i in range(transition_steps):
# Factor de interpolación (0→1)
α = i/transition_steps

# Interpolación lineal para cada articulación
```

```
            j1 = prev[0]*(1-α) + target[0]*α  # Articulación 1
            j2 = prev[1]*(1-α) + target[1]*α  # Articulación 2

            # Control de posición para ambas articulaciones
            p.setJointMotorControl2(robot, 0, p.POSITION_CONTROL, j1, force=500)
            p.setJointMotorControl2(robot, 1, p.POSITION_CONTROL, j2, force=500)

            # Avanza la simulación y pausa breve
            p.stepSimulation()
            # Actualiza la posición anterior para el próximo movimiento
            time.sleep(sleep)

            # Mantener la posición final por un tiempo
            for _ in range(hold_steps):
            p.stepSimulation()
            time.sleep(sleep)


            prev = target

            # Espera entrada del usuario antes de cerrar
            input("¡Secuencia completada! Presiona Enter para salir...")
            p.disconnect()

if __name__=="__main__":
            # Ejecución con parámetros personalizados:
```

```
    # 10 movimientos, transiciones más lentas (800 pasos), pausas más largas (240 pasos)
    run_random_sequence(total_moves=10, transition_steps=800, hold_steps=240)
```

Descripción general del código: Movimiento aleatorio de un brazo robótico con Py Bullet

Simulación de un Brazo Robótico con Py Bullet

El código implementa un brazo robótico de dos grados de libertad en un entorno 3D simulado con Py Bullet, ejecutando movimientos aleatorios y fluidos para emular un comportamiento realista.

Configuración Inicial

- Se establece el entorno de simulación con gravedad y un plano de referencia.
- El modelo del brazo robótico se carga desde un archivo URDF, definiendo su estructura física.
- La cámara se ajusta para proporcionar una vista óptima de la simulación.

Secuencia de Movimientos

El brazo realiza una serie de movimientos predefinidos (total_moves), donde en cada iteración:

1. Generación de posiciones aleatorias:
    - Cada articulación recibe un ángulo objetivo aleatorio dentro del rango [-π, π].
2. Transición suave entre posiciones:
    - Se aplica interpolación para mover gradualmente las articulaciones desde su estado actual hasta la nueva posición, evitando cambios bruscos.
3. Mantenimiento de la pose:
    - El brazo se detiene brevemente en cada posición antes de continuar al siguiente movimiento.

Este enfoque permite simular un comportamiento robótico natural, útil para estudiar dinámicas, optimizar trayectorias o validar diseños mecánicos en un entorno virtual.

⏹️ Finalización

Al finalizar la secuencia, la simulación se pausa esperando que el usuario presione Enter, y luego se cierra correctamente el entorno Py Bullet.

3. Desarrollar el despliegue en Docker de la misma aplicación

Se construyó una imagen Docker para contener la aplicación y poder ejecutarla sin necesidad de instalar dependencias manualmente en el sistema anfitrión.

Contenido del archivo Dockerfile:

```
FROM python:3.10-slim

RUN apt update && apt install -y \
        libgl1-mesa-glx \
        python3-dev \
        build-essential \
        && rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY . /app

RUN pip install --no-cache-dir numpy pybullet

CMD ["python", "brazo_robotico.py"]
```

Después de que tenemos nuestro Docker, usamos los comandos para construit y ejecutar la aplicación:

```
miguelitron@miguelitron-Nitro-AN515-58:~$ sudo docker build -t pybullet-brazo
```

Para correr sin interfaz gráfica (Modo headless):

```
miguelitron@miguelitron-Nitro-AN515-58:~$ sudo docker run --rm pybullet-brazo
```

Para correr con GUI (modo visual):

```
miguelitron@miguelitron-Nitro-AN515-58:~$ xhost +local:docker
sudo docker run --rm \
  -e DISPLAY=$DISPLAY \
  -v /tmp/.X11-unix:/tmp/.X11-unix \
  pybullet-brazo
```

Damos entrey se ejecuta el programa con normalidad.

Para finalizar estos son algunos archivos incluidos en el proyecto:

- brazo_robotico.py: Script principal de simulación.

- two_joint_robot_custom.urdf: Modelo del brazo robótico en formato URDF.
- Dockerfile: Archivo para despliegue automatizado en contenedor.
- Aplicación.mkv, Docker.mkv: Evidencias en video del funcionamiento local y en Docker.
- .gitignore (opcional): Para evitar subir carpetas innecesarias como venv/.