

Resumen Articulos y Respuestas: *Linux Interrupts: The Basic Concepts*

Cristian Olarte, Ana Vargas y Andres Badillo

17/04/2025

Resumen del artículo: *Linux Interrupts: The Basic Concepts*

Este artículo aborda los conceptos fundamentales de las interrupciones en el sistema operativo Linux, explicando su implementación, manejo y utilidad en el kernel. Las interrupciones permiten responder a eventos externos o internos que requieren atención inmediata del procesador, interrumpiendo el flujo normal de ejecución de los procesos.

En el sistema Linux, una CPU puede estar ejecutando un proceso en modo usuario, atendiendo una interrupción de hardware, o ejecutando tareas aplazadas como *softirqs* o *tasklets*. Estas actividades tienen una jerarquía de prioridades bien definida.

Las interrupciones se clasifican como enmascarables, no enmascarables, generadas por software y excepciones. Linux utiliza la **Interrupt Descriptor Table (IDT)** para asociar vectores de interrupción (0 a 255) con sus manejadores correspondientes. Los **exception handlers** permiten responder a errores del sistema, enviando señales al proceso implicado. Además, el sistema incorpora estructuras de datos como `irq_desc[]` y `irqaction` para gestionar el estado y las acciones asociadas a cada IRQ (línea de solicitud de interrupción).

Finalmente, se detallan mecanismos para sistemas multiprocesador como el uso de **APICs** y estructuras para optimizar el rendimiento, garantizando una gestión eficiente y segura de las interrupciones en entornos concurrentes.

Respuestas a las preguntas

1. ¿Qué es una interrupción?, ¿Por qué son necesarias?, ¿Qué tipos de interrupciones existen?, ¿Cuáles son sus características?

Una interrupción es una señal que pausa la ejecución actual del procesador para atender una tarea urgente. Son necesarias para optimizar el uso del procesador, permitir multitarea y responder rápidamente a eventos externos o internos.

Tipos:

- **Enmascarables:** Se pueden bloquear temporalmente. Son típicas de dispositivos de E/S.
- **No enmascarables:** No pueden ser bloqueadas. Se usan para eventos críticos.
- **Por software:** Generadas por instrucciones como `INT n`.
- **Excepciones:** Eventos sincrónicos como división por cero o fallos de página.

2. ¿Qué son los “exception handler”?

Son funciones definidas por el sistema para responder a excepciones detectadas por el CPU. Tienen la tarea de diagnosticar y manejar errores, enviando señales al proceso causante, o finalizándolo si no es recuperable. Pueden ejecutar funciones en C, guardar el estado del sistema y restaurar el flujo al terminar.

3. ¿Qué son las interrupciones generadas por software?, ¿Para qué sirven?, ¿Qué algoritmos/diagramas de flujo se usan para el manejo?

Son interrupciones iniciadas por instrucciones en código de usuario, como `INT 0x80`, utilizadas para acceder a servicios del sistema (llamadas al sistema).

Algoritmos usados:

- **Softirqs, tasklets, bottom halves:** permiten aplazar parte del trabajo del manejador.
- `do_IRQ()`, `handle_IRQ_event()`: funciones del kernel que gestionan las interrupciones.
- Estructura IDT: contiene los vectores asociados a las interrupciones.

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

Las IRQ (*Interrupt Request Lines*) son canales físicos o virtuales que utilizan los dispositivos para solicitar atención del procesador.

Estructuras clave en Linux:

- `irq_desc[]`: almacena el estado, el tipo y el controlador asociado a cada IRQ.
- `irqaction`: define el manejador, nombre del dispositivo y banderas.
- `hw_interrupt_type`: describe las funciones de bajo nivel del controlador de interrupciones.

Estas estructuras permiten compartir IRQs entre dispositivos, controlar prioridades y garantizar la seguridad del sistema ante múltiples interrupciones simultáneas.

Resumen y abstracción del artículo: *BOOTKITS: PAST, PRESENT & FUTURE*

Este artículo técnico explora la evolución, mecanismos e implicaciones de los **bootkits**, un tipo de malware que se instala en las etapas más tempranas del arranque del sistema operativo. Su ejecución anticipada les permite evadir software de seguridad tradicional y establecer una presencia persistente y sigilosa en el sistema.

1. Definición y propósito

Los bootkits son una evolución de los rootkits tradicionales. Inyectan el sistema antes de que el sistema operativo cargue, logrando ejecutar código malicioso en modo kernel. Modifican el MBR (Master Boot Record), VBR (Volume Boot Record) o componentes UEFI para controlar el flujo de arranque.

2. Evolución histórica

- Década de 1980: surgimiento de virus de sector de arranque como *Brain* y *Elk Cloner*.
- 2005: primeros *proof of concept* (BootRoot, Vbootkit).
- 2007: aparición de bootkits reales como *Mebroot*.
- 2010–2014: evolución hacia técnicas más sigilosas y plataformas modernas como UEFI y Android.

3. Técnicas de infección y evasión

- Modificación de sectores de arranque (MBR o VBR).
- Reemplazo del cargador de arranque EFI legítimo.
- Cifrado, hooking, manipulación de registros de interrupción y almacenamiento oculto.
- Uso de su propio stack de red (caso *Gapz*) para evitar firewalls.

4. Clasificación de bootkits

- **MBR-based:** modifican el código del sector MBR (ej. *TDL4*, *Mebroot*).
- **VBR-based:** alteran el VBR o campos como los *hidden sectors* (ej. *Rovnix*, *Gapz*).
- **UEFI-based:** manipulan cargadores EFI, drivers DXE o ROMs (ej. *Dreamboot*).

5. Casos destacados

Gapz: infecta el VBR modificando únicamente campos críticos para redirigir el flujo hacia su código malicioso. Está compuesto por bloques modulares encargados de funciones como cifrado, comunicación en red y hooking.

Dreamboot: bootkit UEFI que reemplaza el bootloader y manipula funciones internas como `OsIArchTransferToKernel` para desactivar medidas de seguridad (ej. PatchGuard).

6. Herramientas de análisis y defensa

- **CHIPSEC:** framework de Intel para análisis de BIOS/UEFI y detección de bootkits.
- **Hidden File System Reader:** permite acceder a almacenamiento oculto usado por malware avanzado.

7. Conclusiones clave

- Aunque tecnologías como **Secure Boot** dificultan los ataques, los bootkits siguen siendo viables.
- La diversidad de firmware y la falta de actualizaciones representan un riesgo para muchos sistemas.
- El malware orientado a UEFI y firmware plantea nuevos retos en ciberseguridad, especialmente en ataques dirigidos.

Explicación paso a paso del funcionamiento de la simulación del brazo robótico en PyBullet

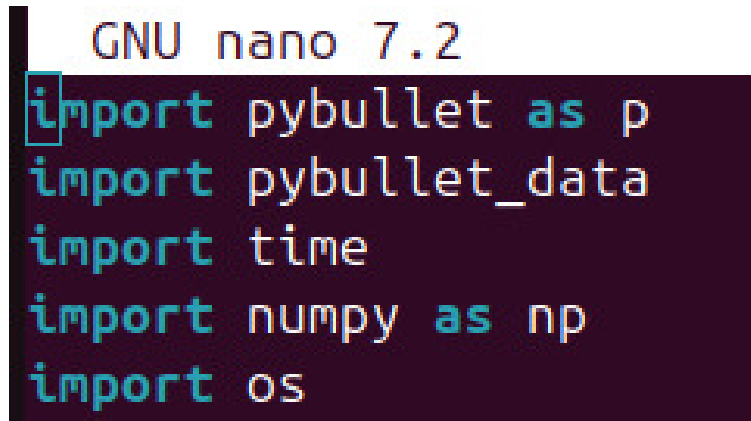
A continuación se detalla el proceso técnico que se llevó a cabo para la construcción y ejecución de una simulación de un brazo robótico con dos articulaciones, utilizando la librería PyBullet en Python. Se describen los componentes clave del código, su propósito, y se incluyen espacios para insertar las evidencias visuales correspondientes.

1. Importación de librerías

Se utilizan las siguientes librerías:

- `pybullet` para controlar la física y simulación 3D.
- `pybullet_data` para acceder a modelos de entorno predeterminados.
- `numpy` para el manejo de rangos de los sliders.
- `os` para verificar si el archivo URDF del robot existe en la carpeta.

Imagen: Captura de pantalla del código de importación.



```
GNU nano 7.2
import pybullet as p
import pybullet_data
import time
import numpy as np
import os
```

2. Verificación del archivo URDF

Antes de ejecutar la simulación, el script valida que el archivo `two_joint_robot_custom.urdf` exista en el mismo directorio. Si no se encuentra, el programa muestra un mensaje de error y se detiene.

Imagen: Error mostrado si el archivo URDF no existe.



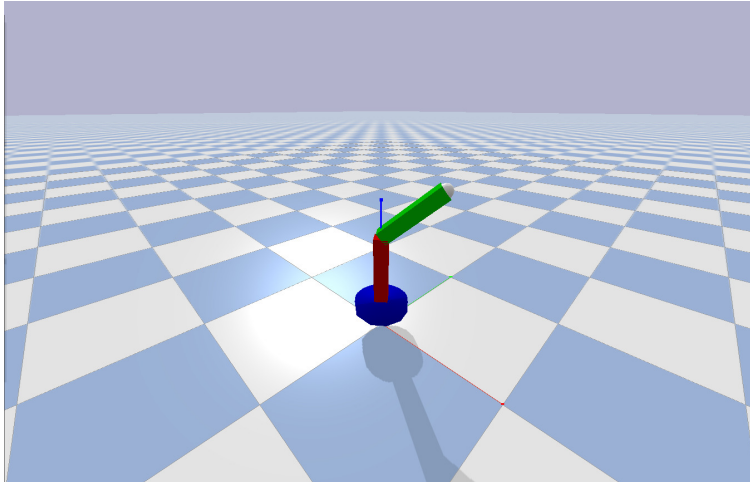
```
ERROR: No se encuentra el archivo two_joint_robot_custom.urdf
Asegúrate de guardar el archivo URDF en la misma carpeta que este script
Presiona Enter para salir...
```

3. Inicialización de PyBullet y configuración del entorno

Se inicia PyBullet en modo gráfico con `p.connect(p.GUI)` y se ajusta la visualización:

- Se establece la gravedad con `p.setGravity(0, 0, -9.8)`.
- Se carga un plano con `plane.urdf`.
- Se ajusta la cámara para enfocar el centro del robot.

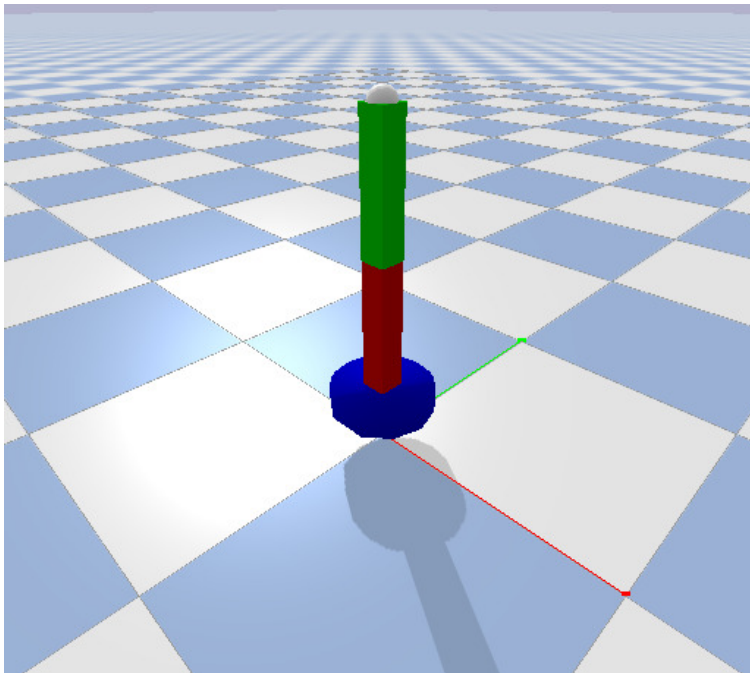
Imagen: Simulador con el plano cargado y el espacio preparado.



4. Carga del robot

Se carga el modelo del robot utilizando el archivo URDF personalizado y se fija su base al suelo con `useFixedBase=True`. La posición inicial evita que el robot se hunda en el plano.

Imagen: Vista del robot cargado en la escena con los colores definidos.



5. Detección e impresión de articulaciones

El programa identifica cuántas articulaciones (joints) posee el modelo mediante `p.getNumJoints()` y muestra sus nombres usando `p.getJointInfo()`. Esto permite confirmar que se están controlando las articulaciones correctas.

Imagen: Resultado en consola mostrando los nombres e índices de los joints.

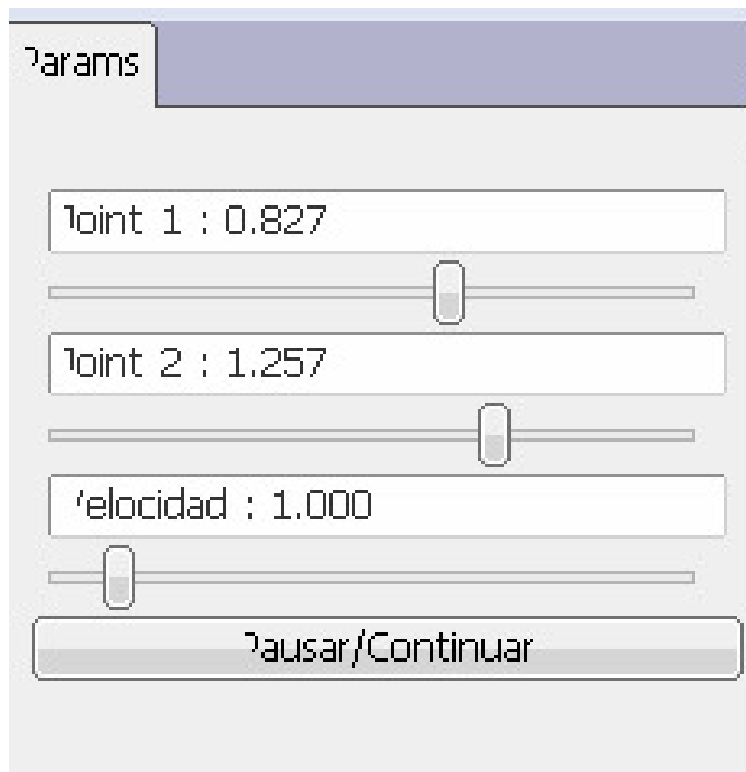
```
Número de joints detectados: 3
Joint 0: joint1
Joint 1: joint2
Joint 2: joint_end_effector
```

6. Creación de sliders interactivos

Se crean sliders con `p.addUserDebugParameter()` para:

- Controlar el ángulo de las dos articulaciones.
- Modificar la velocidad de la simulación.
- Activar y desactivar una pausa manual.

Imagen: Panel de sliders activos dentro del entorno gráfico de PyBullet.



7. Bucle de simulación

En el ciclo `while True`, el script:

- Lee los valores de los sliders.

- Controla las articulaciones usando `POSITION_CONTROL`.
- Aplica los cambios a través de `p.stepSimulation()`.
- Ajusta la velocidad con `time.sleep()`.

Imagen: Movimiento del brazo robótico mientras se ajustan los sliders.

```
while True:
    current_pause_value = p.readUserDebugParameter(pause_button)
    if current_pause_value != last_pause_value:
        print("Simulación pausada/continuada")
        last_pause_value = current_pause_value

    # Leer sliders
    joint1_value = p.readUserDebugParameter(joint1_slider)
    joint2_value = p.readUserDebugParameter(joint2_slider)
    speed = p.readUserDebugParameter(speed_slider)

    # Controlar articulaciones (revisamos los joints válidos)
    p.setJointMotorControl2(robotId, 0, p.POSITION_CONTROL, targetPosition=joint1_value, force=100)
    p.setJointMotorControl2(robotId, 1, p.POSITION_CONTROL, targetPosition=joint2_value, force=100)

    p.stepSimulation()
    time.sleep(0.01 / speed)
```

8. Finalización segura

Cuando se presiona `Ctrl + C`, el script muestra un mensaje de cierre y solicita al usuario presionar `Enter` antes de desconectarse de `PyBullet`.

Imagen: Mensaje de finalización mostrado en consola.

```
XIO: fatal IO error 62 (Timer expired) on X server ":1"
after 39893 requests (39893 known processed) with 0 events remaining.
```

Despliegue de la simulación con Docker

Para garantizar la portabilidad de la aplicación y evitar la necesidad de instalar dependencias manualmente, se realizó el despliegue del simulador del brazo robótico utilizando contenedores Docker. A continuación se detalla el proceso.

1. Estructura del proyecto

Se organizaron los siguientes archivos en una misma carpeta:

- `BrazoRobotico.py`: script principal del simulador.
- `two_joint_robot_custom.urdf`: archivo de descripción del modelo robótico.
- `Dockerfile`: instrucciones para construir la imagen del contenedor.

Imagen: Vista de la estructura de archivos del proyecto.

```
(myenv) cristianolarte@cristianolarte:~/Documentos/Tarea4$ ls
BrazoRobotico.py  Dockerfile  myenv  two_joint_robot_custom.urdf
```


2. Creación del Dockerfile

Se utilizó una imagen base de Python 3.10 en su versión slim. Además, se instalaron las dependencias necesarias para el entorno gráfico de PyBullet. El archivo `Dockerfile` se configuró de la siguiente forma:

```
FROM python:3.10-slim

RUN apt update && apt install -y \
    libgl1-mesa-glx \
    libglu2.0-0 \
    libsm6 \
    libxext6 \
    libxrender1 \
    && rm -rf /var/lib/apt/lists/*

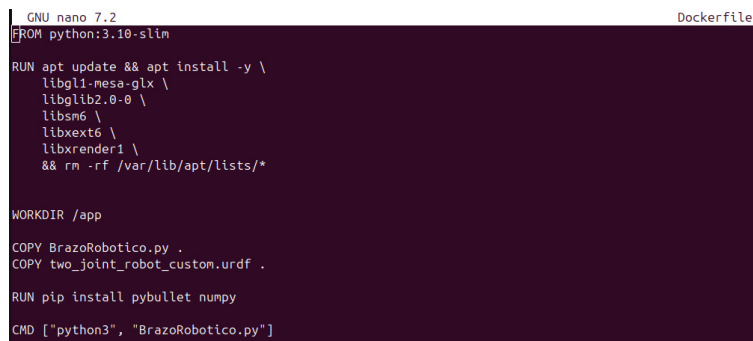
WORKDIR /app

COPY BrazoRobotico.py .
COPY two_joint_robot_custom.urdf .

RUN pip install pybullet numpy

CMD ["python3", "BrazoRobotico.py"]
```

Imagen: Dockerfile abierto en el editor de texto.



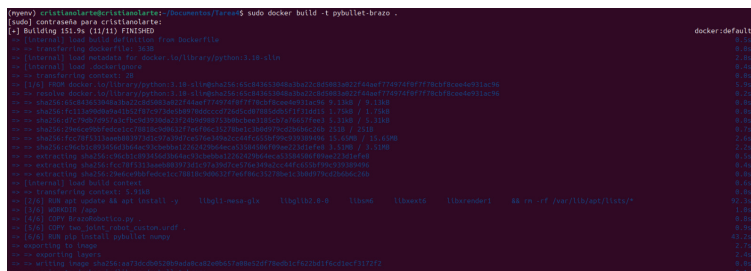
3. Construcción de la imagen Docker

Una vez ubicado en la carpeta que contiene el `Dockerfile`, se ejecutó el siguiente comando para construir la imagen:

```
docker build -t pybullet-brazo .
```

Esto generó una imagen etiquetada como `pybullet-brazo`.

Imagen: Consola mostrando la construcción exitosa.

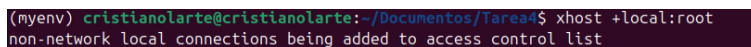


4. Configuración del entorno gráfico

Para que el contenedor pueda mostrar la ventana gráfica del simulador, se dio permiso a Docker para acceder al servidor X11 local mediante el siguiente comando:

```
xhost +local:root
```

Imagen: Permisos otorgados a Docker para mostrar GUI.



5. Ejecución del contenedor

Finalmente, se ejecutó el contenedor con la configuración adecuada para mostrar la interfaz gráfica:

```
docker run --rm \
    -e DISPLAY=$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix \
    pybullet-brazo
```

Este comando lanza la simulación del brazo robótico directamente desde el contenedor.

Imagen: Simulación ejecutándose correctamente desde Docker.

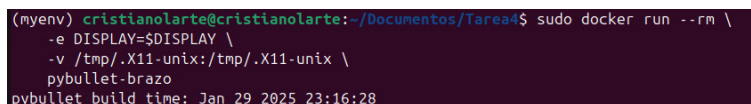


Imagen: Entorno de Docker.

