



Universidad de Guadalajara
Centro Universitario de Ciencias Exactas e Ingenierías
División de Tecnologías para la Integración Ciber-Humana
Departamento de Ciencias Computacionales
Ingeniería en Computación



Ejercicio 06.

Programa que usa *Checkpointing*.

Alumno

Hernández Cortez Kevin Uriel.

Materia

Computación Tolerante a Fallas.

I7036, D06, 2024A

Profesor

Lopez Franco Michel Emanuel.

INTRODUCCIÓN

El checkpointing es una técnica esencial en la programación para garantizar la tolerancia a fallos computacionales. Nos puede dar la capacidad de guardar y restaurar el estado de ejecución de una aplicación, permitiendo la recuperación eficiente en caso de fallos inesperados. En entornos críticos, como los servidores, procesos largos o sistemas distribuidos, implementar la funcionalidad de checkpointing puede ser crucial para minimizar la pérdida de datos y asegurar la continuidad de cualquier servicio. En este trabajo, el uso de *pickle* en Python se presenta como una herramienta valiosa para la serialización de objetos y la creación de “instantáneas” del estado de la aplicación.

DESARROLLO

Código

```
import pickle
import os

class EstadoEjecucion:
    def __init__(self):
        self.Nombre = None
        self.Fecha = None
def guardar_estado(estado, archivo):
    with open(archivo, 'wb') as f:
        pickle.dump(estado, f)
    print("Datos guardados correctamente.")
def cargar_estado(archivo):
    if os.path.exists(archivo):
        with open(archivo, 'rb') as f:
            estado = pickle.load(f)
        print("Datos cargados correctamente.")
        return estado
    else:
        print("No hay datos previos en el archivo.")
        return None

#main
if __name__ == "__main__":
    #verificamos si hay datos guardados en el archivo
    estado_previo = cargar_estado("checkpoint.pkl")
    if estado_previo: #si hay datos, imprimirlos
        print("Datos encontrados:", estado_previo.__dict__)
    else: #si no hay datos, capturarlos
        estado_actual = EstadoEjecucion()
        estado_actual.Nombre = input("Ingrese su nombre: ")
        estado_actual.Fecha = int(input("Ingrese su fecha de cumpleaños
(dia): "))
        #guardar datos en un archivo de nombre "checkpoint"
        guardar_estado(estado_actual, "checkpoint.pkl")
        print("Datos guardados:", estado_actual.__dict__)
```

Explicación

Primeramente, se crea una clase llamada *EstadoEjecucion*, tiene dos variables llamadas *Nombre* y *Fecha*, ambas inicializadas con "None"

```
class EstadoEjecucion:
    def __init__(self):
        self.Nombre = None
        self.Fecha = None
```

Luego, tenemos dos funciones: *guardar_estado* y *cargar_estado*. La función *guardar_estado* toma un objeto estado y un nombre de archivo *archivo*. Utiliza `pickle.dump` para serializar y guardar el objeto estado en un archivo binario ('wb' indica escritura binaria). Luego imprime un mensaje indicando que el estado se guardó correctamente. La función *cargar_estado* toma un nombre de archivo "*archivo*". Utiliza `os.path.exists` para verificar si el archivo existe antes de intentar cargar los datos. Si el archivo existe, utiliza `pickle.load` para deserializar y cargar el objeto estado. Luego, imprime un mensaje indicando que el estado se cargó correctamente y devuelve el objeto estado. Si el archivo no existe, imprime un mensaje indicando que no hay datos previos y devuelve *None*.

```
def guardar_estado(estados, archivo):
    with open(archivo, 'wb') as f:
        pickle.dump(estados, f)
    print("Datos guardados correctamente.")
def cargar_estado(archivo):
    if os.path.exists(archivo):
        with open(archivo, 'rb') as f:
            estado = pickle.load(f)
        print("Datos cargados correctamente.")
        return estado
    else:
        print("No hay datos previos en el archivo.")
        return None
```

Finalmente tenemos la función *main*. Aquí, se llama a la función *cargar_estado* para verificar si hay datos previos en el archivo "checkpoint.pkl". Si hay datos previos, imprime los datos. Si no hay datos previos, crea una nueva instancia de *EstadoEjecucion*, solicita al usuario ingresar valores para *Nombre* y *Fecha*, guarda el nuevo estado en el archivo y lo imprime. Este enfoque refleja un escenario de checkpointing donde se verifica y recupera datos previos si existen, o se capturan y guardan nuevos datos si no hay datos previos.

```
#main
if __name__ == "__main__":
    #verificamos si hay datos guardados en el archivo
    estado_previo = cargar_estado("checkpoint.pkl")
    if estado_previo: #sí hay datos, imprimirlos
        print("Datos encontrados:", estado_previo.__dict__)
    else: #sí no hay datos, capturarlos
        estado_actual = EstadoEjecucion()
        estado_actual.Nombre = input("Ingrese su nombre: ")
        estado_actual.Fecha = int(input("Ingrese su fecha de cumpleaños (día): "))
        #guardar datos en un archivo de nombre "checkpoint"
        guardar_estado(estado_actual, "checkpoint.pkl")
        print("Datos guardados:", estado_actual.__dict__)
```

Ejecución

```
PS C:\Users\52332\Desktop\Tolerante a Fallas> python -
No hay datos previos en el archivo.
Ingrese su nombre: Kevin
Ingrese su fecha de cumpleaños (día): 31
Datos guardados correctamente.
Datos guardados: {'Nombre': 'Kevin', 'Fecha': 31}
PS C:\Users\52332\Desktop\Tolerante a Fallas> █
```

Ilustración 1. Primer caso, donde no existe un archivo previo (se crea).


Nombre	Fecha de modificación	Tipo	Tamaño
 checkpoint.pkl	05/02/2024 12:11 p. m.	Archivo PKL	1 KB

Ilustración 2. Archivo creado

```
PS C:\Users\52332\Desktop\Tolerante a Fallas> python -u
Datos cargados correctamente.
Datos encontrados: {'Nombre': 'Kevin', 'Fecha': 31}
PS C:\Users\52332\Desktop\Tolerante a Fallas> █
```

Ilustración 3. Segunda ejecución, con los datos recuperados.

CONCLUSIÓN

Yo concluyo que el checkpointing desempeña un papel muy importante en la creación de aplicaciones resistentes a fallos. La capacidad de guardar y recuperar estados de ejecución mediante la serialización de objetos con *pickle* nos da una solución muy eficaz. Sin embargo, es importante adaptar estas técnicas a las necesidades específicas de cada aplicación, considerando la complejidad de los estados internos y la gestión de errores