Universidad del Valle de México

LAUREATE INTERNATIONAL UNIVERSITIES®

Actividad 3
Reporte de investigación

PROGRAMACIÓN CONCURRENTE

Docente. Cesar Garcia Martinez

José Emiliano Jauregui Guzmán

Por siempre responsable de lo que se ha cultivado

Introducción

En este reporte de investigación, se investigarán y se pondrán a prueba aspectos en la programación concurrente junto a la teoría de autómatas finitos deterministas, como principal objetivo comprender cómo ciertos problemas pueden afectar el rendimiento y la veracidad de los sistemas. Veremos en primer lugar, la condición de carrera con un ejemplo práctico de código en el que se evidencie su ocurrencia en una sección crítica, además explicaremos su posible consecuencia y la opción para evitarla. Posteriormente, se analizará el principio de interbloqueo, mostrando un caso en el que se ilustre cómo dos hilos pueden quedar atrapados esperando indefinidamente recursos que nunca se liberan. Por último, se abordará el concepto de autómatas finitos deterministas (DFA), en el cual realizaremos un resumen del material señalado junto con dos ejercicios que nos ayudan a un entendimiento más profundo del tema. Este estudio tiene como fin proporcionar una visión integral de los retos y soluciones en la programación concurrente y en la teoría de autómatas.

a. **Ejemplo de código en el que se evidencie una condición de carrera en su sección crítica y otro en el que se muestre el principio de interbloqueo.**

Para estos ejemplos rescate códigos muy básicos únicamente con le propósito de tener un entendimiento mejor de mi parte, para ello la condición de carrera se presenta cuando la temporización de eventos o la programación de tareas puede afectar involuntariamente a un valor de salida o de datos. Para este ejemplo se utilizo un recurso compartido para dos hilos.

Condición de carrera:

Users > josejaureguiguzman > Documents > UVM > PROGRAMACIO

```python
import threading

# Recurso compartido
counter = 0

# Función que incrementa
def increment():
    global counter
    for _ in range(100000):
        # Sección crítica
        counter += 1


thread1 = threading.Thread(target=increment)
thread2 = threading.Thread(target=increment)

# Iniciar los hilos
thread1.start()
thread2.start()

# Esperar a que ambos hilos terminen
thread1.join()
thread2.join()

print("\nValor final:\n", counter)

```

```
josejaureguiguzman@MacBook-Air-de-Jose ~ % 
NCURRENTE/Act 3/carrera.py"

Valor final:
 200000
josejaureguiguzman@MacBook-Air-de-Jose ~ % 
NCURRENTE/Act 3/carrera.py"

Valor final:
 187162
josejaureguiguzman@MacBook-Air-de-Jose ~ % 
NCURRENTE/Act 3/carrera.py"

Valor final:
 200000
josejaureguiguzman@MacBook-Air-de-Jose ~ % 
NCURRENTE/Act 3/carrera.py"
```

Interbloqueo:

Para este código únicamente simularemos que un hilo esta realizando una tarea con algún recurso para que se mantenga en espera el segundo hilo y no le permite acceder a el, ya que esta situación ocurre cuando dos hilos se bloquean simultáneamente, estos dos procesos están esperando el proceso complementario para completar la tarea. Por ello utilice el simulador onlinegdb ya que da con mejor claridad este ejemplo.

main.py

```python
import threading

recurso1 = threading.Lock()
recurso2 = threading.Lock()

# Hilo 1 intenta obtener los dos recursos en orden
def thread1_task():
    with recurso1:
        print("Hilo 1: Adquirió recurso1")

        # Simular que hace algo con el recurso
        threading.Event().wait(1)
        with recurso2:
            print("Hilo 1: Adquirió recuerso2")

# Hilo 2 intenta obtener los mismos recursos pero en orden inverso
def thread2_task():
    with recurso2:
        print("Hilo 2: Adquirió recurso2")

        # Simular que hace algo con el recurso
        threading.Event().wait(1)
        with recurso1:
            print("Hilo 2: Adquirió recurso1")

# Crear los hilos
thread1 = threading.Thread(target=thread1_task)
thread2 = threading.Thread(target=thread2_task)
# Iniciar los hilos
thread1.start()
thread2.start()
# Esperar a que ambos hilos terminen
thread1.join()
thread2.join()

print("Ambos hilos han terminado")
```

input

```
Hilo 1: Adquirió recurso1
Hilo 2: Adquirió recurso2
```

b. Explicación detallada de cada código planteado, mencionando en qué punto ocurre la condición de carrera o el interbloqueo, según corresponda.

Para el codigo que presenta una **condición de carrera** se utilizaron los siguientes recursos:

Utilizamos la biblioteca **import threading** el cual nos permite crear hilos en python.

El recurso compartido entre los dos hilos será la variable **counter** lo que les permite modificarlo, el cual será el primer paso para que pueda ocurrir una condición de carrera.

La función def **increment**()

```
global counter
for _ in range(100000):
    # Sección crítica
    counter += 1
```

es la que incrementara la variable counter a la que estamos solicitando, si un hilo llama a la función lo que hará será leer el valor que tiene counter y lo incrementara a valor 1 escribiendo el nuevo valor. Aquí se presenta la **sección crítica** ya que thread1 y thread2 van a modificar la variable counter pero no tendrán nada de sincronización lo que da lugar a la **condición de carrera** por lo que podría resultar en un valor incorrecto en el resultado.

La creación de los hilos en las lineas de código:
thread1 = threading.Thread(target=increment)
thread2 = threading.Thread(target=increment)
Cada uno de ellos realizara la funcion simultaneamente.

El método para iniciar los hilos:
thread1.start()
thread2.start()
Este método permite que se ejecuten en paralelo simultáneamente los hilos, una vez se ejecute start comienza la función increment por lo que comienza la condición de carrera desde aquí ya que los dos hilos van a tratar de modificar la variable al mismo tiempo.

Las lineas de código **join()** se implementan para que el programa espere a que los hilos terminen de ejecutar la tarea antes de continuar ya que sin ellas se podría imprimir el valor de la variables antes de que termine.

Una vez que join nos permitió que los dos hilos terminaran la linea de **print**("\nValor final:\n", counter) nos arroja el valor final de counter.

Ahora bien, al momento de ejecutar el código en la mayoría de ocaciones nos arroja el valor de 200,000 como debería de ser, ya que se espera que cada hilo incrementa a la variable counter 100,000, pero debido a que presenta la condición de carrera los hilos pueden estorbar entre si cuando intentar darle el valor a counter, lo que puede ocasionar que los incrementos no sean los adecuados dando un resultado incorrecto.

Una forma de como **solucionar** este problema es utilizar algún tipo de sincronización en el cual no les permita a los dos hilos modificar a la vez la variable counter, por lo que un Lock sería una opción para este ejemplo. **counter_lock = threading.Lock()**

Con una linea de código vamos a integrar **with counter_lock:** el cual hará que solo un hilo pueda ejecutar a la vez **counter += 1** eliminando la condición de carrera.

Para el código que presenta un **interbloqueo** se utilizaron los siguientes recursos:

Comenzamos definiendo dos recursos para este ejemplo pero tendran un **Lock** que se utilizara para que solo un pueda acceder a una sección crítica del código a la vez.

```
recurso1 = threading.Lock()
recurso2 = threading.Lock()
```

def thread1_task(): nos ayudará a que el hilo1 obtenga los recursos en orden. Posteriormente haremos que el hilo 1 adquiera el recurso1 usando **with recurso1** lo que bloquea su acceso al recurso1 hasta que el hilo1 se libere. **threading.Event().wait(1)** con esta línea simularemos que el hilo1 esta realizando una acción con el recurso1 por un segundo. Luego vamos a volver con **with** pero ahora con el recurso2 realizando uno de los primeros pasos para el **interbloqueo** ya que si otro hilo tiene el recurso2 va a bloquear al hilo1 hasta que este disponble el recurso2.

```
def thread1_task():
    with recurso1:
        print("Hilo 1: Adquirió recurso1")

        # Simular que hace algo con el recurso
        threading.Event().wait(1)
        with recurso2:
            print("Hilo 1: Adquirió recurso2")
```

Realizaremos el mismo procedimiento **thread2_task** ahora con el segundo hilo para que adquiera los recursos a la inversa. Cuando tenga el recurso2 e intente adquirir el recurso1 este estará bloqueado hasta que el recurso1 este disponible.

```
def thread2_task():
    with recurso2:
        print("Hilo 2: Adquirió recurso2")

        # Simular que hace algo con el recurso
        threading.Event().wait(1)
        with recurso1:
            print("Hilo 2: Adquirió recurso1")
```

Creamos los hilos **thread1** y **thread2** las cuales les asignamos una tarea **thread1_task** y **thread2_task** correspondientes

```
thread1 = threading.Thread(target=thread1_task)
thread2 = threading.Thread(target=thread2_task)
```

Iniciamos los hilos con **start** y esperamos a que terminen con **join.**

Ahora bien, al momento de ejecutar el programa los dos hilos intentarán adquirir los recursos a la vez. Lo que ocasione un **interbloqueo** por el desarrollo que les dimos, los dos hilos se quedaran esperando el recurso que cada uno bloqueo. Por este motivo ningún hilo podrá continuar lo que ocasiona su interbloqueo.

Una forma de como **solucionar** este problema es hacer que los hilos adquieran los recursos en el mismo orden para que no estén esperando uno al otro. En este caso ambos adquieren el recurso1 y luego el recurso2.

```
def thread1_task():
    with recurso1:
        print("Hilo 1: Adquirió recurso1")

        threading.Event().wait(1)
        with recurso2:
            print("Hilo 1: Adquirió recurso2")

def thread2_task():
    with recurso1:  # Ahora adquiere recurso1 primero
        print("Hilo 2: Adquirió recurso1")

        threading.Event().wait(1)
        with recurso2:
            print("Hilo 2: Adquirió recurso2").
```

De esta manera ya no se producirá el interbloqueo.

```
Hilo 1: Adquirió recurso1
Hilo 1: Adquirió recuerso2
Hilo 2: Adquirió recurso1
Hilo 2: Adquirió recurso2
Ambos hilos han terminado


...Program finished with exit code 0
Press ENTER to exit console.
```

c. **Resumen acerca de los autómatas finitos deterministas y resuelve los ejercicios 1 y 2 contenidos en el material Autómatas Finitos Determinísticos. Esto ayudará a que se entienda de mejor forma el funcionamiento de las transacciones entre estados de manera etiquetada o rotulada y en los aspectos generales de una máquina de estados.**

Modelo de computación: es un modelo matemático que aproxima el funcionamiento de una computadora. Estudia sus capacidades y limitaciones tales como computabilidad y complejidad.
Ejemplos:

- **Máquinas de Turing** (1936)
- **Cálculo Lambda** (1936)
- **Autómatas Finitos** (1950)

Estos modelos tienen diferentes **niveles de poder computacional**.

Un modelo básico de máquina tiene tres componentes:
•Entrada (Input)
•Salida (Output)
•CPU y Memoria (la memoria afecta el poder computacional)
Sin memoria, tenemos Autómatas Finitos. Con memoria adicional, como una pila, se tienen Push Down Automata (PDA), y con acceso aleatorio a la memoria, se tienen Máquinas de Turing.

Autómata: Modelo matemático de un sistema con entradas y salidas discretas.
Finito: Tiene un número finito de estados.
Determinístico: En todo momento está en un solo estado.

Un DFA es una 5-tupla M = (Q,Σ,δ,q0,F) donde:
• Q es el conjunto finito de estados.
• Σ es el conjunto finito de símbolos.
• δ es la función de transición.
• q0 es el estado inicial.
• F es el conjunto de estados finales.

Un **diagrama de transición** se construye de la siguiente forma:
•Se agrega un nodo para cada estado
• Marcamos al nodo del estado inicial con una flecha que ingresa.
• Marcamos los estados finales con doble círculos.
• Agregamos un eje etiquetado de un nodo al otro nodo si existe una transición δ

Definiciones importantes:
Alfabeto (Σ): es un conjunto finito de símbolos.
Cadena: secuencia finita de símbolos.
ε**:** es la cadena vacía (sin símbolos).
Σ*: conjunto de todas las palabras sobre Σ.

Lenguaje: Un lenguaje L sobre un alfabeto Σ es un conjunto de palabras sobre el alfabeto Σ (L ⊆ Σ*).

Operaciones sobre Cadenas
Longitud de una cadena w **(**|w|**):** es el número de símbolos.
Subcadena: es una cadena incluida en una cadena. Ejemplo, si w = bbabaa, ab es subcadena de w, aab no.
Prefijo de una cadena w**:** es una subcadena de w que comienza con el primer símbolo de w. Ejemplo, si w = bbabaa, bba es un prefijo de w, ba no.
Sufijo de una cadena w**:** es una subcadena de w que termina con el último símbolo de w. Ejemplo, si w = bbabaa, baa es un sufijo de w, aba no.
Concatenación de dos cadenas w **y** x **(**wx**):** es la unión de las cadenas w y x.

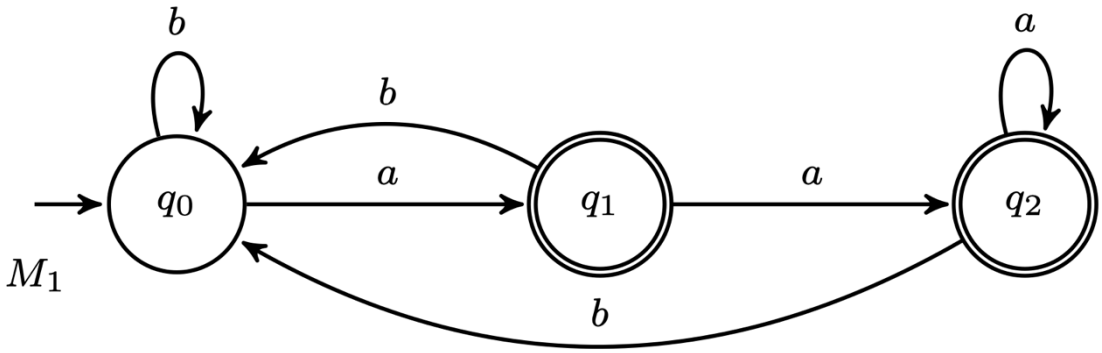
Lenguaje aceptado por un autómata
Sea M un DFA, el lenguaje aceptado por M (L(M)) es el conjunto de cadenas aceptadas por el autómata. Formulamente se expresa así:

$$L(M) = \{\alpha \in \Sigma * | \; existe \; p \in F \; tal \; que \; q_0 \underset{\rightarrow}{\alpha} p\}.$$

Equivalencia de Autómatas
Sean M y M′ dos DFA. Diremos que M y M′ son equivalentes si: L(M) = L(M′).

Ejercicio 1

Determine si las cadenas
abbaa abb aba abaaaaa abbbbbbaab
son aceptadas por el DFA definido por el siguiente diagrama



El lenguaje de M es:

$$L(M) = \{w \in \{a, b\} * | \; w \; termina \; en \; a\}$$

Por lo que no todas las cadenas son aceptadas.

Si la cadena ω es aceptada, toda subcadena de ω es aceptada? es aceptada la cadena ωω?

Esto va a depende del tipo de cadena que le demos a w ya que si tomamos la primera cadena que nos da “abbaa” la cadena si es aceptada porque termina en a, ahora vamos a considerar sus subcadenas de w.

Subcadena ab: esta subcadena no será aceptada ya que no termina en a.
Subcadena ba: esta subadena si será aceptada ya que si termina en a.

En el caso de si la cadena ww es aceptada, tomando en cuenta el mismo lenguaje de M que tenemos en el diagrama dado que la cadena w es aceptada en el ejemplo de la primera cadena “abbaa” ya que termina en a. Estamos hablando de que ww es la unión de w la parte final terminara en a por lo que sería aceptada siendo este “abbaaabbaa”

Ejercicio 2

Considere el autómata del ejercicio anterior. Justifique las siguientes afirmaciones:

1. **Si ω es aceptada, entonces termina en a.**

En el lenguaje que se presento en el ejercicio anterior afirmamos como verdadera esta oración ya que acepta una cadena si y solo si termina en a.
Si una cadena termina en un símbolo distinto de a, el autómata no podrá alcanzar un estado final. Esto es porque en nuestro autómata, las transiciones solo permiten llegar a un estado final cuando la última letra es a. Si la cadena w termina en b, no llegaríamos al estado final lo que no sería aceptada, ya que b nos regresaría al estado inicial.

2. **Si ω termina en a, entonces es aceptada. (Ayuda: si ω = αa, dónde termino después de recorrer α?)**

En este caso siempre el autómata va a partir desde el estado inicial por lo que al recorrer α el cual puede ser cualquier secuencia de símbolos puede llegar a cualquier estado, sin embargo al procesar el ultimo termino de de w que es a independientemente de donde este debe de llegar a un estado final. Por lo que afirmamos esta oración.

Conclusión

Para poder concluir esta actividad se destacan varios aspectos que relacionan a la programación concurrente debido a desafíos claves como la condición de carrera y el interbloqueo, que provienen cuando varios procesos acceden a recursos compartidos sin una sincronización. Por ello los autómatas finitos deterministas (DFA) proporcionan una herramienta my útil para diseñar y verificar las transiciones de estado en sistemas concurrentes lo que permite detectar posibles conflictos en el acceso a recursos. Estos conceptos están relacionados, ya que la gestión de la condición de carrera y el interbloqueo es importante para garantizar la fiabilidad y eficiencia de los sistemas concurrentes. Con lo desarrollado e investigado los modelos de autómatas ayudan a prevenir errores y optimizar los algoritmos de sincronización sobre estos problemas que se llegan a enfrentar. Las tendencias actuales en programación concurrente apuntan hacia una mayor integración de modelos formales y herramientas automatizadas, que nos ayuden con una facilidad en el desarrollo de aplicaciones estables y escalables que les permita adaptándose a la complejidad de las arquitecturas más modernas.

Referencia

Moltó, R., Alonso, J., Alvarruiz, F., Blanquer, I., Guerrero, D., Ibáñez, J. y Ramos, E. (2018). Ejercicios de programación paralela con OpenMP y MPI. Recuperado el 30 de enero del 2025, de https://gdocu.upv.es/alfresco/service/api/node/content/workspace/SpacesStore/557c25f2-e52a-49b8-86b3-

Sendoya, D., (Productor). (06 de Enero de 2015). 34 L8 Condiciones de carrera. Recuperado el 30 de enero del 2025, de https://www.youtube.com/watch?v=bxSpNpmmfXo

Empieza a Programar (Java). (Productor). (15 de Octubre de 2015). Bloque J Concurrente 3.3: Interbloqueos y Estrategia Buffer. Recuperado el 30 de enero del 2025, de https://www.youtube.com/watch?v=vY2ctd0oJA&ab_channel=EmpiezaAProgramar

Fervari, R. (2020). Autómatas Finitos Deterministicos (DFA) Haga clic para ver más opciones. Recuperado el 30 de enero del 2025, de https://cs.famaf.unc.edu.ar/~rfervari/icc16/slides/class-1-slides.pdf

GeeksforGeeks. (s/f). GeeksforGeeks. Recuperado el 30 de enero de 2025, de https://www.geeksforgeeks.org

Stack overflow - where developers learn, share, & build careers. (s/f). Stack Overflow. Recuperado el 30 de enero de 2025, de https://stackoverflow.com