UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

SISTEMAS OPERATIVOS

Grupo 6

PROYECTO NO. 2

Una Situación Cotidiana Paralelizable

ALUMNOS

Abad Vásquez, Aldo

Rosales López, Luis André

PROFESOR

Ing. Gunnar Eyal Wolf Iszaevich

Ciudad Universitaria, Cd. Mx., 31 de marzo de 2022

Proyecto 2: Una Situación Cotidiana Paralelizable

Identificación y Descripción del Problema

1. Descripción

Un grupo de comensales acude a un comedor social. En este comedor es común que acuda un número indeterminado de comensales al mismo tiempo. El comportamiento de cada comensal es el siguiente: primero es necesario que obtengan su porción de comida. Posteriormente, cada uno de ellos deberá buscar un lugar donde sentarse a comer. Una vez sentado, procederá a comer y finalmente, retirarse.

Resulta que, en este comedor social se cuenta con una regla muy específica de buen comportamiento, que indica lo siguiente: "debe evitarse que un comensal se quede solo en la mesa". Esta regla genera como resultado que cada comensal pueda tener tres estados claros:

- Comiendo
- Haciendo sobre mesa (en espera / listo para retirarse)
- Retirándose

El problema entonces radica en lograr que la regla se cumpla toda vez que se puede tener un número aleatorio de comensales y que cada uno de ellos puede comenzar y terminar de comer en cualquier momento.

2. ¿Dónde pueden verse las consecuencias nocivas de la concurrencia? ¿Qué eventos pueden ocurrir que queramos controlar?

Las consecuencias nocivas de la concurrencia se pueden observar al intentar cumplir la regla, puesto que es necesario "coordinar" el comportamiento de los comensales para que pase lo que pase, no se dé el caso de que terminemos con un comensal en soledad.

El evento más importante para controlar es cuando un comensal se tiene la intención de retirarse, pero eso significa dejar solo a otro. Existen dos formas de resolver el evento:

- Otro comensal desea comer y sustituir al que se quiere retirar

- El comensal que quedaría solo termina de comer, por lo que ahora ambos se pueden retirar.

3. ¿Hay eventos concurrentes para los cuales el ordenamiento relativo no resulta importante?

   En este caso no identificamos ningún evento concurrente para el cual resulte relevante el ordenamiento relativo. No importa el orden en que se sienten los comensales ni el orden en el que se retiren, mientras al final tengamos al menos un par de comensales en la mesa.

Implementación del modelo

- ***Descripción de los mecanismos de sincronización empleados***

  Para la solución del problema planteado utilizamos dos semáforos. El primero de ellos lo utilizamos inicializado en uno para obtener un mutex que nos permita limitar el acceso a las variables compartidas.

  El segundo lo inicializamos en cero y nos sirve para limitar la retirada de comensales de uno en uno, de tal forma que no se retiren dos o más comensales al mismo tiempo y por consecuencia la regla se incumpla. Su patrón de uso corresponde con el de un torniquete.

- ***Lógica de operación***
  - ***Identificación del estado compartido (variables o estructuras globales)***

    El estado compartido se ve reflejado en el uso de dos variables de conteo. La primera de ellas es "comiendo" y nos indica el número de comensales que se encuentran en ese estado. Así mismo tenemos la variable “haciendoSobremesa” que nos indica cuantos comensales están en espera para retirarse.

    Estas dos variables nos ayudan a determinar el comportamiento de cada comensal.

- ***Descripción algorítmica del avance de cada hilo/proceso***

  Cada hilo comienza llamando a la función "obtenerComida(int)", que imprime en pantalla un mensaje relativo a dicha acción. Posteriormente adquiere el mutex que permite bloquear el acceso a nuestras variables globales.

  Una vez bloqueado el acceso, incrementa en uno el contador "comiendo" y verifica si es que existe algún otro hilo en espera de retirarse. De ser el caso, libera el torniquete para que dicho hilo se retire, decrementa el contador "haciendoSobremesa" y suelta el mutex.

  A continuación, llama a la función "comer(int)" que también imprime en pantalla un mensaje relativo a dicha acción. Hecho lo anterior, vuelve a adquirir mutex para cambiar de estado, pasando de "comiendo" a "esperando" (modificando las variables globales correspondientes).

  Es aquí cuando el hilo debe determinar su próxima acción con base en el estado de los contadores:

  - Si solo queda el hilo actual en espera y otro comiendo, entonces debe esperar a que dicho hilo termine. Suelta el mutex y adquiere el torniquete.
  - Si el hilo actual estaba siendo esperado entonces libera el torniquete, modifica el contador de hilos en espera y suelta el mutex.
  - En cualquier otro caso el hilo no se ve restringido a quedarse, por lo que puede decrementar el contador de hilos en espera y soltar el mutex.

  Finalmente, se llama a la función "retirarse(int)" para imprimir un mensaje relativo a dicha acción en pantalla.

- ***Descripción de la interacción entre ellos (sea mediante los mecanismos de sincronización o de alguna otra manera)***

  La interacción entre ellos se realiza mediante el torniquete. Este sirve para limitar la retirada de cada hilo de uno en uno. Cada que se genera un nuevo hilo se verifica si es posible liberar el torniquete, de tal forma que un hilo nuevo pueda sustituir a uno en espera. Esta interacción es la que permite que se cumpla la regla

de que no se quede solo ningún comensal durante la ejecución (aunque es más notorio justo al final de esta).

Descripción del entorno de desarrollo

- ***Lenguaje y versión utilizados***

El lenguaje empleado para la implementación es Python 3 (3.7.3). Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.

- ***Bibliotecas utilizadas***

1. ***threading — Paralelismo basado en hilos.*** Este módulo construye interfaces de hilado de alto nivel sobre el módulo de más bajo nivel «*_thread*».
    a. ***_thread — API de bajo nivel para manejo de hilos.*** Este módulo ofrece primitivas de bajo nivel para trabajar con múltiples *threads* o hilos (también llamados *light-weight processes o tasks*) – múltiples hilos de control compartiendo su espacio de datos global. Para sincronizar, provee «candados» simples (también llamados mutexes o *binary semaphores*).

    En CPython, debido al Candado de intérprete global, solo un hilo puede ejecutar código Python a la vez. Sin embargo, el subproceso sigue siendo un modelo apropiado si desea ejecutar varias tareas vinculadas a E/S simultáneamente.
2. ***time — Tiempo de acceso y conversiones.*** Este módulo proporciona varias funciones relacionadas con el tiempo.

    Aunque este módulo siempre está disponible, no todas las funciones están disponibles en todas las plataformas. La mayoría de las funciones definidas en este módulo llaman a la plataforma de biblioteca C funciones con el mismo nombre. A veces puede ser útil consultar la documentación de la plataforma, ya que la semántica de estas funciones varía entre plataformas.

3. ***random —Generar números pseudoaleatorios.*** Este módulo implementa generadores de números pseudoaleatorios para varias distribuciones.

   Para los enteros, existe una selección uniforme dentro de un rango. Para las secuencias, existe una selección uniforme de un elemento aleatorio, una función para generar una permutación aleatoria de una lista *in-situ* y una función para el muestreo aleatorio sin reemplazo.

   Para números reales, existen funciones para calcular distribuciones uniformes, normales (Gaussianas), log-normales, exponenciales negativas, gamma y beta. Para generar distribuciones angulares está disponible la distribución de von Mises.

   Casi todas las funciones del módulo dependen de la función básica *random()*, la cual genera uniformemente un número flotante aleatorio en el rango semiabierto [0.0, 1.0). *Python* utiliza Mersenne Twister como generador principal. Éste produce números de coma flotante de 53 bits de precisión y tiene un periodo de 2**19937-1. La implementación subyacente en C es rápida y segura para subprocesos. Mersenne Twister es uno de los generadores de números aleatorios más testados que existen. Sin embargo, al ser completamente determinístico, no es adecuado para todos los propósitos y es completamente inadecuado para fines criptográficos.

- ***Sistema operativo / distribución de desarrollo y/o pruebas***

El desarrollo se realizó y probó en Debian 10, con ayuda del editor de texto Vim y con ejecución en consola. Igualmente, se probó su funcionamiento en Windows 11 mediante el *IDLE* de *Python*. Dado que no hace uso de bibliotecas externas al lenguaje, es posible ejecutar el programa de manera directa siempre y cuando el sistema operativo cuente con Python 3 instalado, usando la instrucción:

```
python3 proyecto2.py
```

- ***Ejemplos o pantallazos de una ejecución exitosa***

```
debian@debian10:~/SO/sistop-2022-2/proyectos/2/AbadAldo-RosalesAndré$ python3 proyecto2.py
### Proyecto 2: Una situación cotidiana parelizable ###

 Bienvenido! Este programa modela el comportamiento en un comedor social en dónde se considera de mal
educación dejar a alguien solo en la mesa. Para más detalles favor de revisar la documentación adjunt


Por favor, selecciona alguna de las siguientes opciones:

1 .-  Indicar el número de comensales
2 .-  Salir

Opción elegida: 1
Dime, ¿Con cuántos comensales quieres que trabaje?: 5
Entendido, comenzando...

Comensal #0 consiguiendo comida...
Comensal #1 consiguiendo comida...
Comensal #2 consiguiendo comida...
Comensal #3 consiguiendo comida...
Comensal #4 consiguiendo comida...
Comensal #0 verificando si se puede sentar...
Comensal #0 sentandose a comer...
Comensal #1 verificando si se puede sentar...
Comensal #1 sentandose a comer...
Comensal #4 verificando si se puede sentar...
Comensal #4 sentandose a comer...
Comensal #2 verificando si se puede sentar...
Comensal #2 sentandose a comer...
Comensal #3 verificando si se puede sentar...
Comensal #3 sentandose a comer...
Comensal #0 no se ve obligado a quedarse... mejor irse
Comensal #0 retirándose...
Comensal #4 no se ve obligado a quedarse... mejor irse
Comensal #4 retirándose...
Comensal #3 no se ve obligado a quedarse... mejor irse
Comensal #3 retirándose...
Comensal #1 esperando a que otro comensal termine de comer...
Comensal #2 notificando que ha terminado al comensal que lo esté esperando...
Comensal #2 retirándose...
Comensal #1 retirándose...
Trabajo terminado...
```

Prueba 1. Intento inicial con 5 personas

```
Por favor, selecciona alguna de las siguientes opciones:

1 .-  Indicar el número de comensales
2 .-  Salir

Opción elegida: 1
Dime, ¿Con cuántos comensales quieres que trabaje?: 2
Entendido, comenzando...

Comensal #0 consiguiendo comida...
Comensal #1 consiguiendo comida...
Comensal #0 verificando si se puede sentar...
Comensal #0 sentandose a comer...
Comensal #1 verificando si se puede sentar...
Comensal #1 sentandose a comer...
Comensal #1 esperando a que otro comensal termine de comer...
Comensal #0 notificando que ha terminado al comensal que lo esté esperando...
Comensal #0 retirándose...
Comensal #1 retirándose...
Trabajo terminado...
```

Prueba 2. Con 2 personas

```
Por favor, selecciona alguna de las siguientes opciones:

1 .-  Indicar el número de comensales
2 .-  Salir

Opción elegida: 1
Dime, ¿Con cuántos comensales quieres que trabaje?: 1
Entendido, comenzando...

Comensal #0 consiguiendo comida...
Comensal #0 verificando si se puede sentar...
Comensal #0 sentandose a comer...
Comensal #0 no se ve obligado a quedarse... mejor irse
Comensal #0 retirándose...
Trabajo terminado...
```

Prueba 3. Con 1 persona

```
Por favor, selecciona alguna de las siguientes opciones:

1 .-  Indicar el número de comensales
2 .-  Salir

Opción elegida: 1
Dime, ¿Con cuántos comensales quieres que trabaje?: 8
Entendido, comenzando...

Comensal #0 consiguiendo comida...
Comensal #1 consiguiendo comida...
Comensal #2 consiguiendo comida...
Comensal #3 consiguiendo comida...
Comensal #4 consiguiendo comida...
Comensal #5 consiguiendo comida...
Comensal #0 verificando si se puede sentar...
Comensal #0 sentandose a comer...
Comensal #6 consiguiendo comida...
Comensal #7 consiguiendo comida...
Comensal #2 verificando si se puede sentar...
Comensal #2 sentandose a comer...
Comensal #7 verificando si se puede sentar...
Comensal #7 sentandose a comer...
Comensal #5 verificando si se puede sentar...
Comensal #5 sentandose a comer...
Comensal #2 no se ve obligado a quedarse... mejor irse
Comensal #2 retirándose...
Comensal #6 verificando si se puede sentar...
Comensal #6 sentandose a comer...
Comensal #4 verificando si se puede sentar...
Comensal #4 sentandose a comer...
Comensal #1 verificando si se puede sentar...
Comensal #1 sentandose a comer...
Comensal #1 no se ve obligado a quedarse... mejor irse
Comensal #1 retirándose...
Comensal #7 no se ve obligado a quedarse... mejor irse
Comensal #7 retirándose...
Comensal #3 verificando si se puede sentar...
Comensal #3 sentandose a comer...
Comensal #4 no se ve obligado a quedarse... mejor irse
Comensal #4 retirándose...
Comensal #5 no se ve obligado a quedarse... mejor irse
Comensal #5 retirándose...
Comensal #0 no se ve obligado a quedarse... mejor irse
Comensal #0 retirándose...
Comensal #6 esperando a que otro comensal termine de comer...
Comensal #3 notificando que ha terminado al comensal que lo esté esperando...
Comensal #3 retirándose...
Comensal #6 retirándose...
Trabajo terminado...
```

Prueba 4. Con 8 personas

Referencias

- Wolf, G., Ruiz, E., Bergero, F., & Meza, E. (2015). *Fundamentos de Sistemas Operativos*. Universidad Nacional Autónoma de México, Instituto de Investigaciones Económicas : Facultad de Ingeniería. http://sistop.org/pdf/sistemas_operativos.pdf
- Wolf, G., Ruiz, E., Bergero, F., & Meza, E. (s. f.). *Código ejemplo*. Fundamentos de sistemas operativos. http://sistop.org/codigo.html