A picture containing icon

Description automatically generated

**Universidad Tecnológica Centroamericana**

Sede: Campus Unitec San Pedro Sula, Honduras

**Sistemas Operativos I**

*Actividad:* ***Proyecto #2: Sincronización de Procesos***

Ing. Román Arturo Pineda

Sección: 1853

**Estudiante:**

Angella Fernanda Falck Duran – 21941027

22 de junio de 2022

**Problema #1: El problema de la leche**

Los hilos deben garantizar la consistencia. De lo contrario, los subprocesos pueden dar lugar a una condición de carrera que conduzca a un resultado no determinista. Una carrera es cuando el resultado depende del orden en que terminan los hilos.

Por ejemplo:

El hilo A dice x = 1, y si (x == 2), entonces imprime '¿Qué?'

El hilo B dice x = 2.

¿Qué es x?

Los hilos requieren operaciones de sincronización para que esto sea respondido.

**Explicación del algoritmo:**

El problema Demasiadas botellas de leche modela a dos compañeros de alojamiento que comparten un refrigerador y que, como buenos compañeros, se aseguran de que el refrigerador este siempre bien abastecido de leche. Con compañeros de habitación tan responsables, el siguiente escenario es posible:

**Table

Description automatically generated**

El problema surge porque se deben cumplir las siguientes propiedades:

* Seguridad. Nunca más de una persona compra leche.
* Vivacidad. Si se necesita leche, eventualmente alguien la compra.

**Código**

Text

Description automatically generated

Text

Description automatically generated

**Explicación código:**

En este algoritmo podemos observar la manera en la que funciona el concepto de concurrencia:

Dentro del codigo, por medio de un enum podemos definir la cantidad de hilos/threads disponibles para nuestro buffer, el cual estaría trabajando en el problema, que en este caso es tener suficiente leche. "Too much milk" o demasiada leche, en español, se refiere a tener más que la necesaria (1 cartón de leche).

Dentro de este ejemplo podemos observar como el hecho de tener más trabajo que hilos puede crear un cuello de botella por falta de hilos para resolver el problema, y viceversa, si tenemos más hilos que trabajo estos intentaran trabajar en el mismo problema. Se imprime varias veces por esta misma razón.

**Simulación para 1 thread**

**Caso#1: Hay Suficiente Leche**

**Text

Description automatically generated**

**Caso#2: Hay Mucha Leche**

**Text

Description automatically generated**

**Caso #3: No Hay Leche**

**Text

Description automatically generated**

**Simulación para 2 threads**

**Caso#1: Prueba**

*Threads: 2, milkAvailable:1*

**Graphical user interface, text

Description automatically generated**

**Simulación para n threads -** *Utilizando 5*

**Caso #2: Prueba**

*Threads: 4, milkAvailable:3*

**Text

Description automatically generated**

**Problema #2: Productores y Consumidores**

En computación, el problema del productor-consumidor es un ejemplo clásico de problema de sincronización de multiprocesos.

**Explicación del algoritmo:**

El programa describe dos procesos, productor y consumidor, ambos comparten un buffer de tamaño finito. La tarea del productor es generar un producto, almacenarlo y comenzar nuevamente; mientras que el consumidor toma (simultáneamente) productos uno a uno. El problema consiste en que el productor no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío.

**Solución:**

La idea para la solución es la siguiente, ambos procesos (productor y consumidor) se ejecutan simultáneamente y se “despiertan” o “duermen” según el estado del buffer. Concretamente, el productor agrega productos mientras quede espacio y en el momento en que se llene el buffer se pone a “dormir”. Cuando el consumidor toma un producto notifica al productor que puede comenzar a trabajar nuevamente. En caso contrario, si el buffer se vacía, el consumidor se pone a dormir y en el momento en que el productor agrega un producto crea una señal para despertarlo. Se puede encontrar una solución usando mecanismos de comunicación de interprocesos, generalmente se usan semáforos. Una inadecuada implementación del problema puede terminar en un deadlock, donde ambos procesos queden en espera de ser despertados.

**Código**

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

**Explicación código:**

Hay uno o más productores que generan datos y estos los colocan un un búfer. Solo existe un solo consumidor que un elemento a la vez del búfer. Para la solución de este problema, el sistema es limitado para prevenir la superposición de operaciones del búfer. En otras palabras, solo un productor o consumidor puede acceder al búfer al mismo tiempo. En este problema se esta previendo tener un conflicto debido a tener múltiples productores, de esta manera evitamos que los 2 intenten escribir al mismo tiempo o insertar el objeto en el mismo espacio.

**Simulación para 1 thread**

Si se elimina 1 thread, por ejemplo, el de productores, el programa solo produciría y agregaría datos al buffer, esto no haría que se implementara el algoritmo de consumidores y productores correctamente, así como lo establece el problema.

Chart, text

Description automatically generated

Lo mismo pasaría si solo trabajamos con el thread de consumidores, el programa solo agarraría y consumiría datos.

Text

Description automatically generated

**Simulación para 2 threads**

**Graphical user interface

Description automatically generated with low confidence**

**Simulación para n threads**



La solución anterior funciona perfectamente cuando hay uno o más productores y un consumidor. Cuando múltiples consumidores comparten el mismo espacio de memoria para almacenar el buffer la solución anterior puede llevar a resultados donde dos o más procesos lean o escriban la misma región al mismo tiempo. Por ende, la simulación es apta solamente para 2 threads.

Cada semáforo funciona como un administrador de cierta función que es posible ejercer sobre nuestro buffer.

Tenemos un administrador que suma (crea espacio en el buffer o espacio al sacar un objeto), y tenemos un administrador que resta (que indica cuantos espacios disponibles quedan dentro de nuestro buffer o espacio).

Nuestro tercer semáforo es un administrador, pero no de nuestro buffer o espacio, sino de nuestros tipos semáforos o administradores, y la manera en la que funciona es que este se asegura de que en caso de que tengamos dos administradores de suma o de resta, estos no intenten actuar sobre el mismo lugar dentro de nuestro buffer ya sea para sumar o restar, así no intentan ejercer sobre el mismo lugar simultáneamente

Cuando hay múltiples productores:

Text

Description automatically generated

**Problema #3: Algoritmo de Peterson**

El algoritmo de Peterson, también conocido como solución de Peterson,1​ es un algoritmo de programación concurrente para exclusión mutua, que permite a dos o más procesos o hilos de ejecución compartir un recurso sin conflictos, utilizando sólo memoria compartida para la comunicación.

**Explicación del algoritmo:**

Mutex, o exclusión mutua, es un objeto en un programa que impide simultáneamente que varias personas accedan al mismo recurso.

* Durante la programación concurrente, las secciones críticas se utilizan para acceder a un recurso compartido por procesos o subprocesos.
* Solo un subproceso puede poseer el mutex durante el inicio del programa, por lo que se le asigna un nombre único.
* Si un subproceso tiene un recurso, debe bloquear el mutex de otros subprocesos para que otros subprocesos no puedan acceder a él simultáneamente. El subproceso desbloquea el mutex después de liberar el recurso.
* Los subprocesos múltiples entran en juego cuando dos subprocesos trabajan en los mismos datos simultáneamente. Como herramienta de sincronización, actúa como un candado.
* Si hay un mutex disponible, un subproceso puede adquirirlo; de lo contrario, se pone a dormir.

El algoritmo de Peterson se limita a dos procesos que se ejecutan alternativamente entre secciones críticas. Llamaremos a estos procesos Pi y Pj.

**Solución:**

La solución de Peterson necesita que se compartan dos elementos de datos entre los dos procesos:

* turno int: Indica su turno para entrar en su tramo crítico.
* Flag booleana[2]: Indica si un proceso está listo para entrar en su sección crítica. Da resultados como verdadero o falso.

**Código**

Text

Description automatically generated

Text

Description automatically generated

**Explicación código:**

Lo que representa el algoritmo de Peterson es que desbloquear el subproceso, ejecutar la acción, y volver a bloquear el subproceso siempre va a tardar el doble de tiempo que tomaría contar hasta x (realizar un proceso cualquiera).

Se crea una clase para implementar el algoritmo de Peterson. Dentro de la clase, existen dos variables: flag y turn. Después de eso, implementé dos funciones, una para bloquear el valor de la bandera (lock) como 1 y la otra para desbloquear (unlock) el valor de la bandera.

En la función de bloqueo (lock), cambie la prioridad del subproceso, espere a que se ejecute el subproceso y cambie el valor de la bandera. En la función de desbloqueo (unlock), marque el hilo que ya no tiene que ejecutarse en la sección crítica.

Ahora cree una función que reserve el valor para el cual cada subproceso intentará bloquear el algoritmo de Peterson, incrementar un valor entero ingresado por el usuario y luego desbloquear el subproceso. El proceso anterior se ejecutará el doble de x (dependiendo el valor) porque es muy independiente del tiempo y puede tomar algunas iteraciones para solucionar este problema.

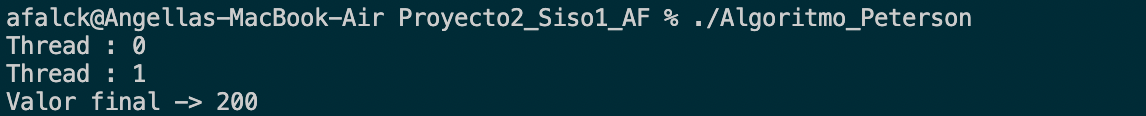
El proceso de asegurar la exclusión mutua puede ser realizado por dos procesos simultáneamente. Ambos procesos crean e inicializan variables compartidas antes de comenzar. Ninguno de los procesos está actualmente interesado en la sección crítica, por lo que los indicadores [0] y [1] se establecen en FALSO.

**Simulación para 1 thread**

****

El valor final será igual al valor inicial, ya que solo hay un thread, por ende no se bloqueara porque solo existe 1 subproceso.

**Simulación para 2 threads**

****

**Simulación para n threads -** *Utilizando 3 (no funciona correctamente)*

Text

Description automatically generated

**Problema #4: Algoritmo de Dekker**

El algoritmo de Dekker es un algoritmo de programación concurrente para exclusión mutua, que permite a dos procesos o hilos de ejecución compartir un recurso sin conflictos. Fue uno de los primeros algoritmos de exclusión mutua inventados, implementado por Edsger Dijkstra.

**Explicación del algoritmo:**

El algoritmo de Dekker fue la primera solución probablemente correcta al problema de la sección crítica. Permite que dos subprocesos compartan un recurso de un solo uso sin conflicto, utilizando solo la memoria compartida para la comunicación. Evita la alternancia estricta de un ingenuo algoritmo de toma de turnos y fue uno de los primeros algoritmos de exclusión mutua que se inventaron.

**Código**

**Text

Description automatically generated**

Graphical user interface, text

Description automatically generated

**Explicación código:**

Mas que mostrar la ejecución de la acción con las banderas (flags), lo que el algoritmo nos quiere demostrar es la concurrencia del programa, porque a pesar de que nuestra única variable inicializada es j, la inicialización completa de la misma depende del valor de 'i', que es nuestro proceso más pequeño si nos guiamos de la jerarquía de nuestros ciclos for.

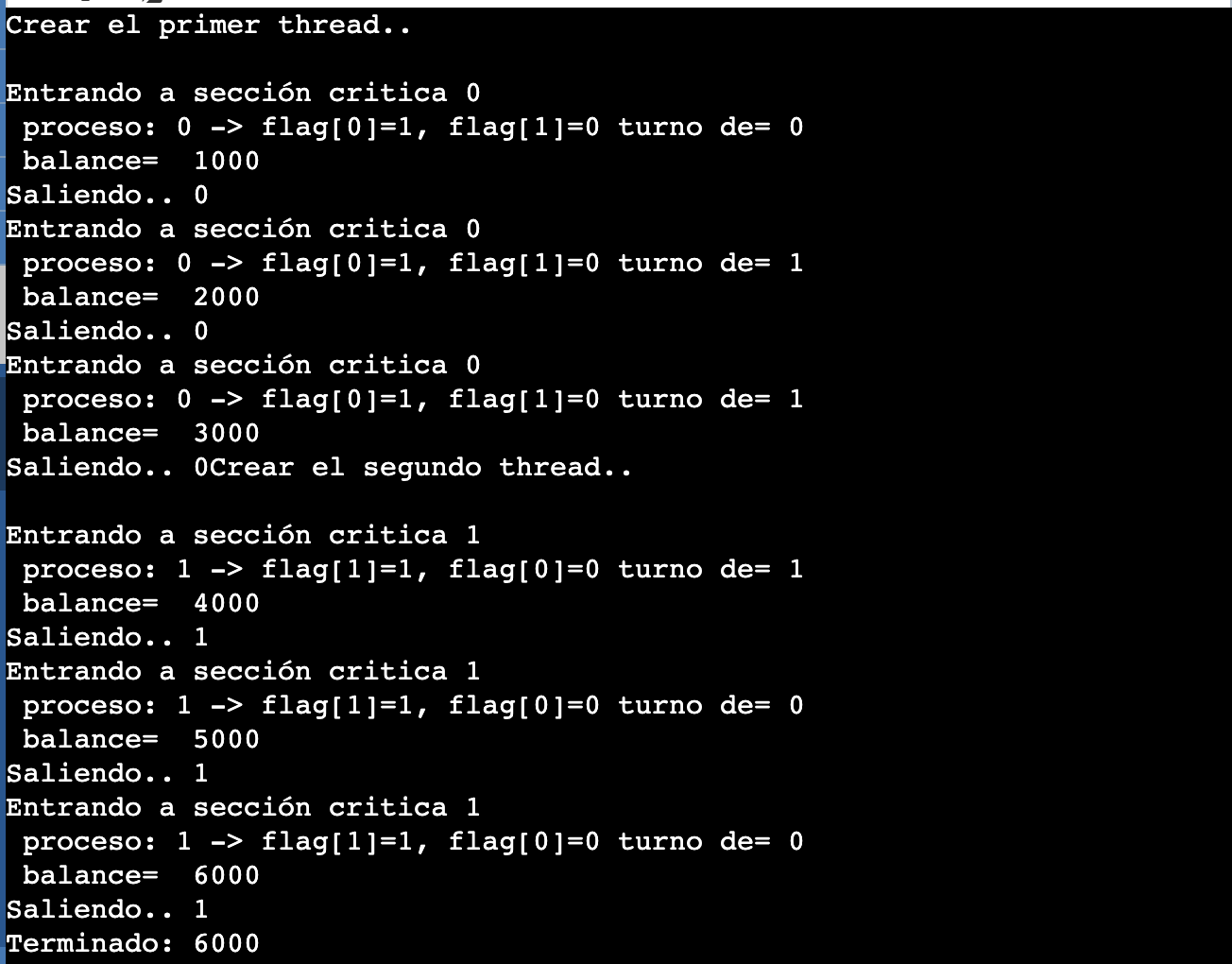
El ejercicio nos quiere comprobar que a pesar de ser dependiente el uno del otro, es que la declaración de j siempre va a tomar x cantidad de tiempo, y todo esto es posible debido a el uso de memoria compartida, la cual asegura que nuestro proceso, a pesar de tardarse un tiempo 'x', será realizado dentro de un margen controlado que no comprometerá la ejecución de nuestro proceso o programa. Aquí si aplica la exclusión mutua, todavía está en juego podemos decir por el hecho de que el tiempo que vamos a esperar sigue estando dentro de un margen aceptable. Sin embargo, este algoritmo solo puede manejar un máximo de 2 procesos.

**Simulación para 1 thread**

**Text

Description automatically generated**

**Simulación para 2 threads**

****

**Simulación para n threads**

Text

Description automatically generated

No se maneja correctamente

**Problema #5: Algoritmo de la Panadería**

El algoritmo de la panadería de Lamport es un algoritmo de computación creado por el científico en computación Lord Leslie Lamport, para implementar la exclusión mutua de N procesos o hilos de ejecución.

**Explicación del algoritmo:**

Todos los subprocesos del proceso deben tomar un número y esperar su turno para usar un recurso informático compartido o para ingresar a su sección crítica. El número puede ser cualquiera de las variables globales, y los procesos con el número más bajo se procesarán primero.

**Solución:**

En la vida real, el sistema de los boletos funciona perfectamente, pero en un sistema informático la obtención del boleto es problemática: varios hilos pueden obtener el mismo número de turno.

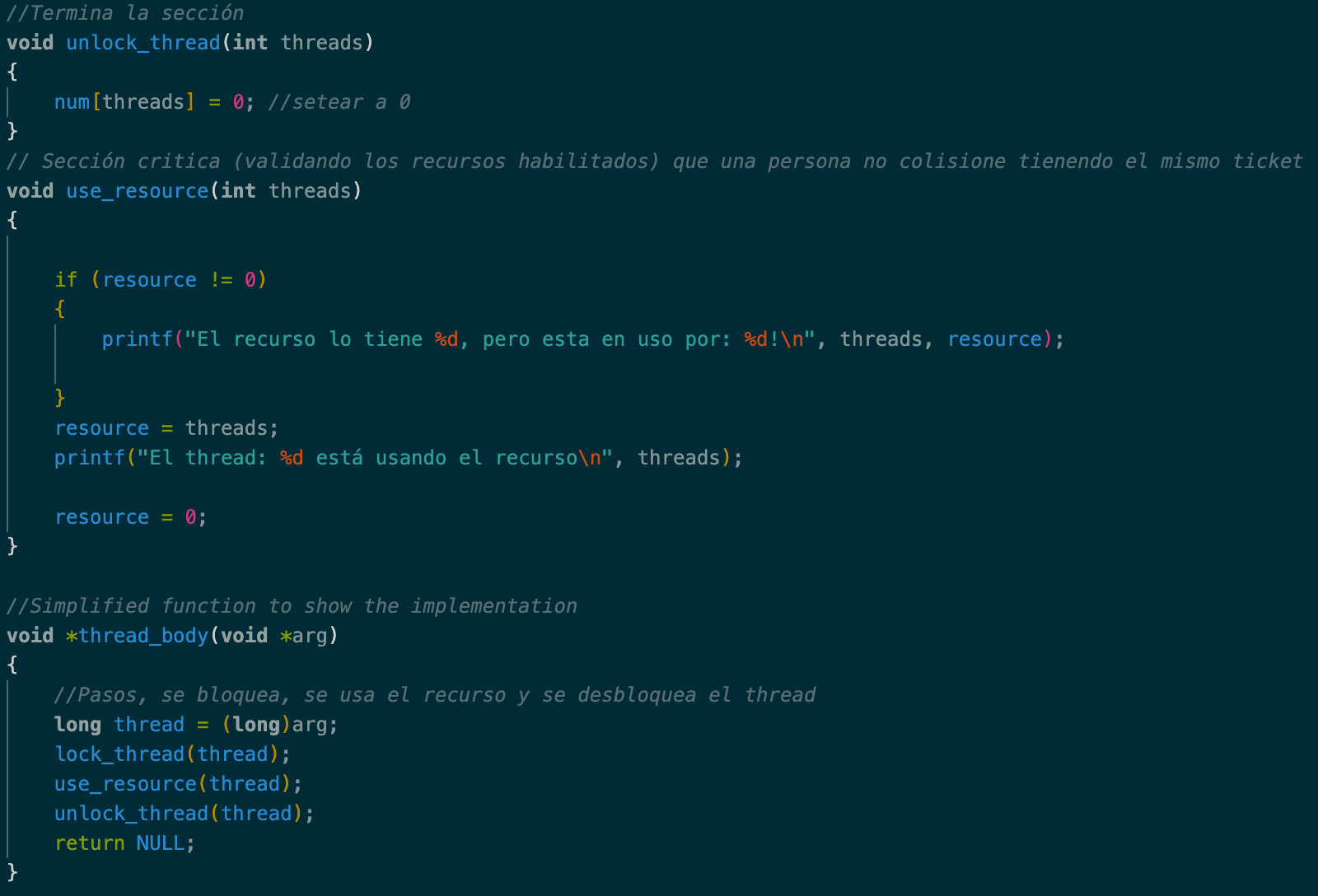
En el algoritmo de Lamport se permite que varios hilos obtengan el mismo número. En este caso, se aplica un algoritmo de desempate, que garantiza que sólo un hilo entra en sección crítica. El desempate se realiza así: si dos o más hilos tienen el mismo número de turno, tiene más prioridad el hilo que tenga el identificador con un número más bajo.

Como no puede haber dos hilos con el mismo identificador, nunca se da el caso de que dos hilos evalúen al mismo tiempo que tienen derecho a ejecutar su sección crítica.

**Código**

Text

Description automatically generated



Text

Description automatically generated

**Explicación código:**

Para comenzar, en la función de lock\_thread, el proceso establece su variable de "selección" en verdadero, lo que significa su intención de ingresar a la sección crítica.

Luego se le asigna el número de ticket más alto relacionado con otros procesos. La variable "seleccionar" se establece entonces en falso, lo que indica que ahora tiene un nuevo número de boleto. Con esto se propone cambiar el valor del boleto, pero no se debe permitir que otro proceso verifique el valor del boleto anterior, ya que este ya no esta disponible. Por eso antes de ver el valor del ticket dentro del ciclo for, se asegura que todos los procesos tengan la variable “selección” en falso.

Después, se examinan los valores del ticket de los procesos, siempre asegurándose que el proceso con el num de ticket de proceso mas bajo este incluido en la sección critica como se explicaba en el algoritmo anteriormente. El valor del boleto al final se asignará como 0 en la sección de salida.

**Simulación para 1 thread**

Text

Description automatically generated

Solo hay un thread simulando.

**Simulación para 2 threads**

**Text

Description automatically generated**

Solo hay dos thread simulando.

**Simulación para n threads -** *Utilizando 10*

