UNIVERSIDAD DE CARABOBO
FACULTAD EXPERIMENTAL DE CIENCIAS Y TECNOLOGÍA
DEPARTAMENTO DE COMPUTACIÓN
SISTEMAS OPERATIVOS
PROFESORA PhD MIRELLA HERRERA
PROYECTO: SINCRONIZACIÓN ENTRE PROCESOS
AUTORES:
ADRIÁN RAMÍREZ C.I. 30.871.139
BRAYAN CEBALLOS C.I. 29.569.937
JOSHTIN MEJÍAS C.I. 31.071.016
ULISES RONDÓN C.I. 26.011.965

Problema 1

Resumen

El presente problema describe una situación de concurrencia de procesos: se tiene una central telefónica que cuenta con 100 teléfonos conectados con los cuales se puede realizar las acciones de “descolgar”, “marcar” y “colgar”, cada teléfono puede encontrarse en un estado derivado de dichas acciones “colgado y en espera”, “descolgado y marcando”, etc. Y a su vez, cada uno requiere de entrar en comunicación con la central y/o el usuario para pasar de un estado a otro (no en todos los casos).

Para la resolución del problema, se tuvo que realizar una simulación de la central operativa en un plazo de 24 horas. Dicha simulación fue realizada en lenguaje C haciendo uso de las librerías “Pthreads”, tomando a cada teléfono como un hilo independiente. Se utilizaron los semáforos como herramientas de sincronización para garantizar la exclusión mutua y así evitar cualquier inconveniente tanto en la comunicación con la central y el usuario como en la recolección de las estadísticas.

Se plantearon las siguientes asunciones:

- Modelado del proceso: Se modelaron dos tipos de hilos, el hilo que corresponde a la central telefónica, que se encarga de revisar constantemente el estado de los teléfonos para gestionar las conexiones (llamadas) y actualizar las estadísticas; y el de los teléfonos, cada uno fue considerado como un hilo independiente.
- Estados del teléfono: Los teléfonos pueden encontrarse en varios estados de acuerdo a las acciones que realice o no el usuario. A efectos prácticos se puede interpretar que los teléfonos pasan por un estado Inactivo, (COLGADO); un estado Espera, (LLAMANDO) y (RECIBIENDO_LLAMADA); un estado de acción por parte del usuario (DESCOLGADO_ESPERANDO_MARCAR) y (MARCANDO); y finalmente un estado de Comunicacion, (EN_LLAMADA) y (COMUNICANDO).
- Tiempo de simulación: Para efectos del problema se simula un segundo en representación de una hora.
- Comportamiento de los semáforos: La herramienta semáforo concibe una cola de llegada, comúnmente FIFO, para aquellos procesos que están en espera de hacer uso de un recurso crítico, de esta manera se asegura que todos los procesos solicitantes tendrán acceso al recurso y que el dicho acceso no pueda implicar condiciones de carrera.

Solución implementada

Se tuvo que identificar los recursos y secciones críticos del problema presentado para poder encontrar una solución que cumpliera con las especificaciones dadas.

- Recursos críticos: Los teléfonos (hilos), al ser un número limitado que, además, pueden variar el estado en el que se encuentran según las acciones del usuario y las respuestas de la central, fueron manejados como recursos críticos.
- Estructuras estadísticas: Fueron implementadas como estructuras, las que solamente el hilo de la central tiene acceso para modificar, también se establecieron como recurso crítico, esto debido a que, si bien el hilo de la central es único, éste se encuentra en una interacción constante con teléfonos cuyas operaciones son justamente las que determinan el valor de los estadísticos.
- Secciones críticas y sincronización:
    - Estado del teléfono: Como mecanismo de sincronización se utiliza semáforo para cada teléfono. El semáforo es binario, la sección crítica que protege es la operación de cambio del estado del teléfono. el sem_wait() decrementa el contador del semáforo bloqueando el acceso el acceso a cualquier otra solicitud para cambiar de estado. Una vez que el estado del teléfono ha sido cambiado el sem_post() vuelve a incrementar el contador del semáforo permitiendo que una nueva solicitud de cambio pueda entrar.
    - Llamadas: Existen estados que necesariamente involucran a dos teléfonos, (EN_LLAMADA), (COMUNICANDO) y etc. Es por ello que desde el hilo central se realiza una verificación del teléfono destino, se comprueba si se encuentra en un estado disponible(no bloqueado). En estos casos se utilizan dos semáforos binarios(emisor y receptor), la sección crítica corresponde al espacio entre los dos sem_wait() y sem_post(), en dicho espacio se encuentra la actualización de los estadísticos.
    - Actualización de los estadísticos: Nuevamente se hace uso de semáforos binarios, la sección crítica corresponde a toda operación de lectura y escritura de un estadístico. Se realiza un bloqueo sem_wait() y luego se libera de forma inmediata sem_post(). De esa forma se asegura la atomicidad de cada actualización y se evita las condiciones de carrera.

Escenario de concurrencia

La solución presentada resulta sólida debido a que resuelve los problemas comúnmente encontrados en la concurrencia. El uso de semáforos, en este caso binarios, permite evitar la condición de carrera ya que dos hilos no pueden acceder y, por ende, modificar al mismo tiempo el estado de un teléfono y el valor de un estadístico.

Los interbloques también son evitados ya que cuando la central necesita bloquear dos teléfonos a la vez (el de origen y el de destino), siempre los bloquea en un orden consistente basado en su ID. Esto asegura que dos intentos de conexión simultáneos no terminen esperándose el uno al otro indefinidamente.

La estructura del semáforo también impide que haya inanición. No existe una condición de prioridad que permita que nuevas solicitudes se atienden por sobre otras, de tal manera que, si no consideramos los parámetros timeout1 y timeout2 y las limitaciones del tiempo, todas las solicitudes en algún momento serían atendidas.

La espera activa es otro problema que se resuelve con el uso de los semáforos sem_wait() y sem_post() ya que estos hacen llamadas bloqueantes al sistema. Cuando un hilo (teléfono o central) debe esperar, el planificador del SO lo “duerme”, de forma tal que para que no consuma ciclos de CPU. Este “despierta” cuando el recurso por el que esperaba ya se encuentra disponible, de forma tal que esta manera tal de resolver la concurrencia resulta tremendamente eficiente.

Utilidad y aprendizaje

La resolución de este ejercicio trascendió considerablemente el ámbito de la programación convencional, representando una inmersión profunda en el diseño de sistemas concurrentes y distribuidos. Más allá de la implementación técnica, el mayor aprendizaje radicó en desarrollar una mentalidad arquitectónica capaz de anticipar escenarios de carrera, condiciones de bloqueo y estados transitorios en entornos donde múltiples entidades operan simultáneamente. Se tuvo que internalizar que en la concurrencia, el orden de los eventos es inherentemente impredecible, y por tanto, el diseño debe ser robusto frente a todas las posibles interleavings de ejecución. Esta perspectiva transforma por completo el approach de desarrollo: ya no se trata simplemente de escribir código funcional, sino de diseñar sistemas que se comporten correctamente bajo cualquier secuencia posible de operaciones.

El desafío fundamental consistió en abstraer las complejas especificaciones del mundo real en un modelo de estados manejable pero exhaustivo, donde cada componente (usuario, teléfono y central) operaba como un proceso independiente y coordinado. Esta experiencia fue particularmente valiosa para comprender los patrones de comunicación asíncrona y los mecanismos de sincronización que previenen condiciones de carrera sin caer en interbloqueos. El manejo de timeouts, por ejemplo, dejó claro cómo diseñar para la resiliencia: el sistema debía no solo manejar los casos de éxito, sino también recuperarse elegantemente de fallos y condiciones de tiempo expirado, manteniendo la consistencia global del estado del sistema.

Finalmente, el ejercicio demostró la crítica importancia del diseño de interfaces claras entre componentes y de estrategias de logging comprehensivas, que permitieran observar el comportamiento emergente del sistema sin alterar su funcionamiento. Esta capacidad de hacer visible el estado interno de un sistema distribuido resultó tan crucial como la implementación misma de la lógica de negocio. En esencia, el proyecto sirvió como un puente entre la teoría de la concurrencia y su aplicación práctica, desarrollando no solo habilidades técnicas específicas sino, más importante aún, una forma de pensar sistémica que es directamente transferible a cualquier escenario del mundo real que involucre coordinación entre procesos independientes, desde sistemas de telecomunicaciones hasta arquitecturas de microservicios en la actualidad.

Problema 2

Resumen

A continuación se detalla una solución de software para el problema de control de tráfico en un tramo de la autopista Maracay - La Victoria bajo mantenimiento. El objetivo es simular el comportamiento de múltiples vehículos (donde se encuentran carros y camiones con diferente peso) que transitan en ambos sentidos a través de cuatro subtramos con diferentes restricciones de capacidad. Utilizando exclusivamente el paradigma de programación concurrente.

Esta simulación se implementó en lenguaje C y se hace el uso de librerías “Pthreads", en representación de cada vehículo como un hilo de ejecución independiente. Se emplean mecanismos de sincronización como semáforos y mutex que permiten gestionar el acceso a los subtramos (recursos compartidos), prevenir condiciones de carrera en la recolección de las estadísticas exigidas y asegurar un flujo de tráfico ordenado y libre de interbloqueos.

Las asunciones planteadas fueron las siguientes:

- Modelado del proceso: Cada vehículo se modeló como un proceso en forma de hilo independiente, alineándose perfectamente con el paradigma de programación concurrente solicitado. El hilo principal actúa como “director de orquesta” de la simulación, ya que genera

vehículos a lo largo del tiempo y a su vez, recopila y presenta los resultados finales (estadísticas).

- Representación del hombrillo: Este se interpreta como el estado del hilo del vehículo cuando está bloqueado, esperando para acceder al siguiente subtramo. El sistema operativo gestiona esta espera a través de las llamadas bloqueantes a los semáforos, lo que previene la espera activa y hace un consumo eficiente de la CPU.
- Tiempo de simulación: Para efectos del código se toma un segundo de tiempo real como 1 hora del tiempo simulado, para ejecutar múltiples simulaciones en tiempos razonables.
- Creación y distribución de Vehículos: Para la creación de los vehículos a lo largo de la hora se aplica una distribución uniforme. El hilo principal se encarga de calcular un intervalo de sueño entre la creación de cada vehículo para distribuir los vehículos promedio por hora (Aproximadamente 500 para el caso del enunciado) a lo largo del segundo de tiempo real que representa la hora.
- Comportamiento de los semáforos: La implementación de semáforos de POSIX utiliza una cola de espera justa (aproximadamente FIFO), esto previene la inanición ya que la espera de un vehículo por tramo no será indefinidamente superado por vehículos que lleguen después.

Solución Implementada

Para esto se tuvo que identificar los recursos y secciones críticas para encontrar la mejor propuesta de solución, donde se especifican:

- Recursos críticos: Los subtramos se manejan como recursos contables, ya que al poseer capacidad limitada se convierten a su vez en recursos críticos. El más complejo de estos es el "cuello de botella" subtramo 2, ya que al depender del tipo de vehículo (carro o camión con sus respectivos pesos) adiciona una comparación para evaluar quién puede pasar.
- Estructuras estadísticas: Se implementaron como estructuras a la que todos los hilos de vehículos y el hilo principal acceden para leer y escribir. También se define como un recurso crítico. Estas se ven por medio de la consola (que es otro recurso compartido), a través de la salida estándar. Múltiples hilos escribiendo simultáneamente en ella pueden producir salida corrupta e ininteligible.
- Secciones críticas y sincronización:
    - Control de acceso para subtramos 1, 3 y 4: Como mecanismo de sincronización se utilizan semáforos para cada tramo, inicializados en los valores que corresponden a su capacidad (4, 1, 3). La sección crítica es el tiempo que un vehículo pasa dentro del tramo. Siendo protegida por un sem_wait(), que decrementa el contador del semáforo, ocupando un lugar y una vez que entra el protocolo de liberación se encarga de liberar el lugar por medio del sem_post().
    - Control de acceso al subtramo 2: Como mecanismo de sincronización se utiliza un único semáforo contador inicializado en 2, ya que el semáforo no cuenta vehículos sino pesos, entonces parece ser la solución más conveniente. Dependiendo del peso el vehículo ejecuta un sem_wait() (si es carro una vez y camión dos veces) para poder entrar a la sección crítica (recorrer el tramo) y en el protocolo de liberación realiza sem_post() la misma cantidad de veces que en el protocolo de negociación. Este enfoque garantiza la restricción de que se respete el peso máximo del tramo.
    - Actualización de las estadísticas: El mecanismo de sincronización empleado es mutex (stats.mutex_stats). En la sección crítica, todas las estructuras que modifican las estadísticas están protegidas. Antes de cualquier lectura o escritura en ella, se adquiere un bloqueo (pthread_mutex_lock()) y luego se libera de forma inmediata (ptread_mutex_unlock). Esto garantiza la atomicidad de las actualizaciones y evita las condiciones de carrera, asegurando la integridad de los datos.
    - Salida por consola: Para asegurar que la escritura vía terminal no se mezclen, se utiliza pthread_mutex_lock() (para solicitar escribir) y ptread_mutex_unlock() (para

avisar que ya dejó de escribir), generando un orden cronológico claro y legible de los eventos de la simulación.

Escenarios de Concurrencia

La robustez de la solución se demuestra en cómo se previenen los problemas clásicos de la concurrencia. Primeramente se hace referencia a las condiciones de carrera, estas son prevenidas por el uso de mutex_stats, esto nos permite que dos hilos no puedan leer e incrementar un contador en simultáneo en su memoria local, para luego resultar un resultado equívoco. Esto provocaría una pérdida en la actualización de las estadísticas.

De igual manera el diseño evita interbloqueos mediante una estrategia de adquisición ordenada de recursos. Los vehículos que solicitan acceso a los tramos, van en un orden estrictamente secuencial y ascendente (1, 2, 3, 4) o descendente (4, 3, 2, 1). Esto debido a que un vehículo nunca puede adquirir un tramo anterior en su ruta mientras posee un actual, evitando la espera circular que provoca interbloqueo.

Cómo se mencionó previamente en las asunciones, la inanición es prevenida por la cola de espera del semáforo. No hay un mecanismo de prioridad que permita que nuevos vehículos pasen por delante de los que ya están esperando en el hombrillo. Por ende, cada vehículo que llega a un tramo congestionado eventualmente obtendrá acceso.

Por último, tenemos la espera activa que se evita completamente con el uso de los semáforos sem_wait() y pthread_mutex_lock() que hacen llamadas bloqueantes al sistema. Cuando algún hilo (vehículo) debe esperar, el planificador del SO lo pone a “dormir”, para que no consuma ciclos de CPU. Este es despertado cuando el recurso que espera está disponible, haciendo esta forma la más eficiente de gestionar la espera en sistemas concurrentes.

Utilidad y aprendizaje

El presente ejercicio permitió realizar una aplicación práctica en la simulación y consolidar conocimientos teóricos fundamentales (vistos previamente en la asignatura). Va de la mano con múltiples materias que forman parte del cuerpo curricular de la carrera y facilita herramientas para que personal más especializado en el área (en este caso planificación urbana, tránsito, entre otros) puedan tomar medidas adecuadas para facilitar los “procesos” de la vida diaria en una parroquia, ciudad o país.

De igual manera se demuestra como el modelado por software puede predecir y analizar el comportamiento de este tipo de sistemas complejos del mundo real.

Además, el desarrollo permitió conocer análisis y propuestas a nivel teórico, llevando al entendimiento de las condiciones de Coffman para el interbloqueo y el diseño de estrategias para llevar a una adquisición ordenada de recursos para así evitarlo.

A nivel de programación, sirve para profundizar conceptos dentro del lenguaje C, como el manejo del ciclo de vida completo de los hilos POSIX , desde su creación (pthread_create) y paso de argumentos, hasta su finalización y la recolección de recursos (pthread_join), asegurando que el programa no termine de forma prematura.

Y por último, el problema llevó a los autores a un estilo de pensamiento defensivo. Buscando rutas viables que permitieran el buen uso de los recursos críticos, así como la prevención de condiciones de carrera, para garantizar la integridad y consistencia de los resultados (estadísticas) finales.