Gómez Luna Alejandro

Sistemas Operativos
Grupo: 2

Ejercicio #6

1.- Los hilos se implementan dependiendo de cuánto saben el Kernel y programa de aplicación acerca de los hilos. Se tienen tres métodos para implementar hilos:

- Los hilos en el nivel Kernel: La creación, terminación y comprobación de estatus de los hilos en este nivel se realizan a través de llamados al sistema.
  - Cuando un proceso crea un hilo, el Kernel le asigna un id y un bloque de control de hilos (TCB), que contiene un apuntador al PCB del proceso.
  - Cuando un evento ocurre, el Kernel salva el estado del CPU del hilo interrumpido en su TCB para que después del evento, considerando los TCB de todos los hilos, seleccione un hilo listo. En caso de que el hilo listo seleccionado sea de un proceso distinto al que se encontraba en ejecución, se salva el contexto del proceso al que pertenece el hilo interrumpido y carga el del proceso al que pertenece el hilo seleccionado.
- Los hilos en el nivel de usuario: Se implementan mediante una biblioteca de hilos que se enlaza con el código de un proceso.
  - Cuando un proceso crea un hilo, la función de biblioteca crea un TCB para el hilo nuevo y lo considera en la planificación.
  - Cuando se requiere sincronización entre hilos, y conmutación, la función de biblioteca se encarga de realizarlo sin intervención del Kernel. También se encargará de el bloque del proceso si ningún hilo está disponible.
  - El código de la biblioteca del hilo es una parte de cada proceso.
- Los hilos híbridos: Tiene ambos hilos en el nivel del usuario y en el nivel del Kernel, y un método para asociar unos con otros. Los tres métodos son:
  - Asociación uno a uno: Cada hilo en el nivel de usuario se mapea permanentemente a un hilo en el nivel del Kernel.
  - Asociación muchos a uno: Todos los hilos creados en un proceso por la biblioteca del hilo son asociados con un solo hilo a nivel de Kernel.
  - Asociación muchos a muchos: Un hilo en el nivel del usuario puede mapearse en cualquier hilo en el nivel del Kernel.

Por otra parte, el concepto de Multithreading o Hyperthreading se refiere a la tecnología a nivel HW que permite a un mismo procesador poder ejecutar varios hilos a la vez.

Un procesador cuenta con diversos núcleos físicos con los que puede administrar diferentes hilos. Con esta nueva tecnología, los núcleos físicos se dividen en núcleos lógicos, lo que permite que se puedan ejecutar más hilos paralelamente.

Bibliografía

- Dhamdhere, D. M. (2008). Sistemas Operativos: Un enfoque basado en conceptos (2a. ed.). ProQuest Ebook Central.
- intel.la/content/www/xl/es/gaming/resources/hyper-threading.html consultado por última vez el 8/07/21 a las 13:21

2.- La condición de carrera se observa cuando
- Se realiza una reservación de asientos para alguna función cinematográfica, teatral, etc., puesto que ver si existen más usuarios queriendo reservar los mismos asientos, se produce una condición de carrera, ya que se está queriendo acceder a un recurso compartido y la reservación dependerá de quien acceda primero
- En una transacción bancaria, en donde se modifique el saldo de una cuenta. Si se realizan dos transacciones o más transacciones a la vez, se tiene una condición de carrera, ya que las transacciones acceden al mismo recurso y dependiendo del orden de las transacciones el saldo se modificará de distinta manera para las otras transacciones.
- En cualquier partido de futbol, si la pelota se encuentra suelta, existen varios jugadores queriendo controlar la pelota, entonces se tiene una condición de carrera, pues varios jugadores están queriendo acceder al recurso compartido que es el balón y quien llegue primero utilizará el recurso.

3.- En la solución de Peterson, cuando un proceso quiere entrar a su región crítica, primero se tiene la variable "otro", en la cual se almacena el id del otro proceso y servirá para conocer si el otro proceso ya se encuentra en su región crítica, también con ayuda del arreglo interesado, el cual es modificado por cada proceso. Este arreglo se modifica poniendo en 1 el elemento con índice igual al id del proceso que se encuentra modificándola, pues eso significa que ese proceso está interesado en acceder a su región crítica y se modifica con 0 cuando el proceso actual entró y salió de su región crítica

Por último se tiene la variable turno, la cual solamente funciona para que, si no había proceso alguno en su región crítica y dos procesos quieren entrar simultáneamente a su región crítica, el proceso que llegó al final se quede ejecutando el while hasta que el otro proceso salga de su región crítica mediante la función salir_region

Esta solución es mejor, ya que en la solución de alternancia de la sección crítica, si ningún proceso quiere acceder a su región crítica, los procesos se bloquean hasta que algún proceso entre a su región crítica

4. Se utiliza para que solo un proceso pueda utilizar las operaciones a la vez, ya que, cuando un proceso ha accedido al semáforo, la variable s se encuentra en 0, así que si otro proceso quiere acceder al semáforo, tendrá que esperar a que el proceso que accedió al semáforo anteriormente utilice la operación V y de esta forma acceder al semáforo.
Si ningún proceso accede al semáforo, la variable s tiene valor de 1

5. El semáforo contador se utiliza cuando el recurso compartido tiene elementos en los cuales se lleve una cuenta de la cantidad de recursos disponibles, por lo que su valor estará entre cero y un número entero N, mientras que el semáforo binario solamente indica si un recurso está ocupado (0) o está disponible (1).

6. Semáforo
   - prodCons0.c
     - En este caso se tiene un programa donde se tiene un productor y consumidor, definidos mediante las funciones producer() y consumer().
     - Para el caso del productor, se utiliza el método wait() del semáforo para entrar en su región crítica, y producirá un elemento, por lo que la variable full, que lleva el registro de los elementos producidos, se aumenta en 1. Asimismo, la variable empty, que lleva el registro de los espacios disponibles, disminuye en 1.
     - Por último se actualiza el buffer y se utiliza el método signal() del semáforo para liberarla
     - Para el caso del consumidor, se realizará el mismo procedimiento, pero cambiando que, como se consume un elemento, la variable full disminuye en 1 y la variable empty aumenta en 1.

- Este programa no utiliza hilos, por lo que solo es un ejemplo demostrativo de cómo se utiliza un semáforo sin embargo, al no implementarse con hilos no hay necesidad de utilizarlo, pues no existen recursos compartidos siendo modificados simultáneamente.

prodCons1-c (Semáforo binario)

- Este programa es bastante parecido al anterior con la excepción de que este sí implementa hilos.
- En la función main() se declaran las variables del semáforo que serán ; empty, para indicar que un espacio ha sido ocupado; full, para indicar que un producto se produjo; sm, variable de control del semáforo. Se crean los hilos correspondientes al productor y consumidor. Cuando estos terminen finaliza la ejecución del main.
- En la función producer() se tiene al productor. Se comienza por mostrar que se creó el productor y el id del hilo asociado a este. Luego se tiene un for loop, en donde para cada iteración esperará a que las variable empty y sm estén en 1, con la finalidad de que tenga un espacio para producir y acceder al semáforo, respectivamente. Posteriormente, modifica la variable compartido data y finalmente modifica la variable sm y full a 1, indicando que el semáforo está disponible y existe un producto disponible para el consumidor, respectivamente.
- En la función consumer() se tiene al consumidor. Se comienza por mostrar que se creó el consumidor y el id del hilo asociado a este. Sigue el mismo patrón que el productor, solo que esperará a que las variables full y sm estén en 1, con la finalidad de que tenga un producto para consumir y acceder al semáforo, respectivamente. Posteriormente, modifica la variable total, la cual irá decrementando el número de producto usando la variable compartido data y finalmente modifica la variable sm y empty a 1, indicando que el semáforo está disponible y existe un espacio disponible para el productor, respectivamente.
- Este programa permite visualizar de mejor manera el uso de semáforos, observando que los productos se van produciendo y consumiendo de manera dinámica, llevando el control a través de un semáforo, para utilizar y modificar los recursos compartidos adecuadamente.

▪ prodCons2-c (Semáforo contador)

- Este programa es una mejora al anterior, implementando un buffer y un mayor número tanto de productores como de consumidores
- En la función main() se declaran las variables del semáforo, que serán: empty, para indicar la cantidad de espacios disponibles; full, para indicar la cantidad de productos disponibles.
  Se crea una estructura mutex y se crean cinco hilos que serán los productores y cinco hilos que serán los consumidores. Cuando estos terminen, se elimina la estructura mutex y finaliza la ejecución del main
- En la función producer() se tiene al productor. Cada productor producirá un máximo de 5 productos esperando a que existan espacios disponibles, a través de la variable empty. Una vez que pueda producir un elemento, se utiliza el mutex para que solo un productor pueda colocar el producto en el buffer. Los productos se irán colocando consecutivamente, variando el nilo que agrega el elemento en un instante dado
  Finalmente, cuando un productor ha insertado su producto en el buffer, libera el mutex e indica que existe un producto, a través de la variable full
- En la función consumer() se tiene al consumidor. Cada consumidor consumirá un máximo de 5 productos, esperando a que existan productos disponibles, a través de la variable full. Se utiliza el mutex para que cada consumidor pueda ir consumiendo un producto a la vez, consecutivamente, variando el nilo que consume en un instante dado
  Finalmente, un consumidor libera el mutex y agrega un espacio disponible, a través de la variable empty.
- Aquí se observa el uso de semáforos y estructuras mutex, para el control adecuado de recursos entre varios hilos

Paso Mensajes

- Este programa también utiliza un productor y consumidor, pero desde una perspectiva de paso de mensajes
- En la clase principal se define al productor y al consumidor, a través de las clases Producer y Consumer

* En la clase Productor se define el método run(), que se ejecuta en cuanto el hilo se inicia. Aquí solamente se usará el método putMessage(), el cual una vez realizado se repite infinidad de veces con un intervalo de 1 segundo.
  El método putMessage() irá añadiendo mensajes, que contienen la fecha y hora actual, en el atributo messages. Una vez que se llegue al máximo de mensajes en el atributo messages, que es un vector, el productor esperará hasta que el consumidor lea un mensaje.
* En la clase Consumer se asocia a un determinado productor y en su método run() se usará el método getMessage() de su productor asociado y lo mostrará en pantalla, repitiendo esto infinidad de veces con un intervalo de 2 segundos.
  El método getMessage() irá retirando mensajes de el vector de mensajes, definido en el atributo messages del Productor. Una vez se quede el vector vacío, el consumidor esperará hasta que haya nuevos mensajes por leer.
* El método wait() se utiliza para que un hilo se quede en espera hasta que otro hilo utilice el método notify().
* Este programa sirve para comprender de mejor manera la utilización de monitores, pues solamente un hilo puede acceder al atributo messages del objeto creado a partir de Productor, por lo que se sincroniza el acceso a un objeto por parte de dos hilos.

Barriers

* SquareQueue.java
  - Para este programa se utilizan dos BlockingQueue, los cuales sirven para ir almacenando y retirando elementos de ellos, haciendo que un determinado hilo espere a que exista un espacio o elemento, dependiendo el caso.
  - Se tiene la clase principal SquareQueue donde se crean dos instancias BlockingQueue que serán asociadas con una instancia de la clase Squarer.
  - En la clase Squarer se utiliza una BlockingQueue para almacenar todas las peticiones y la otra para almacenar todos los resultados.
    Con el método take() se provoca que el hilo pause su ejecución hasta que exista un elemento y put() también pausa la ejecución de un hilo hasta que exista un espacio disponible. Estos métodos pertenecen a la clase BlockingQueue.

- Este es un programa simple, por lo que solo habrá una petición y un resultado. La cola de peticiones solo tendrá el número 42, al que se le elevará al cuadrado y el resultado se almacenará en la cola de resultados

* Tester.java
- Este programa implementa una CyclicBarrier, la cual es una manera en la que se sincronizan los hilos, haciendo que todos los hilos deban llegar a esta barrera para continuar su ejecución
- En la clase Tester se define una CyclicBarrier con un número de 3, lo que indica que para sobreposar esa barrera, 3 hilos deben llegar a ese punto. Posteriormente se crea un hilo y se inicia.
  En ésta clase se define un metodo run, el cual creará dos hilos hijos a partir de las clases Computation1 y Computation2. Una vez iniciados estos hilos, el hilo padre espera en la barrera
- En la clase Computation1 se implementa también el metodo run(), el cual realiza la multiplicación entre 3 y 2, y ya que la realiza, el hilo asociado a esta clase espera en la barrera
- En la clase Computation2 se hace de igual forma el metodo run(), en donde se verifica si la barrera ya se ha roto. Se realiza la suma de 10 y 20 para finalmente hacer que el hilo asociado a esta clase espere en la barrera.
- Una vez que los tres hilos llegan a la barrera, esta barrera se rompe y se muestran en pantalla los resultados calculados. Finalmente se reinicia la barrera.
- El programa nos muestra otro elemento de sincronización, el cual es la barrera, que hace que una cantidad determinada de hilos espere a esa barrera y no puedan seguir ejecutándose hasta que un cierto número de hilos llegue a la barrera