Universidad de Costa Rica

Escuela de Ciencias de la Computación e Informática

Programación Paralela y Concurrente

CI0117

Luis Eduardo Rojas Carrillo – B86875

Semana #6

a-)

Código C++:

Dado que el programa genera valores aleatorios con los cuales trabajar las salidas las partículas a crear pueden variar, crea un átomo de oxigeno o uno de hidrógeno según el “rand()”. Si hay dos átomos de hidrógeno y uno de oxigeno crea una molécula de agua. Cada hilo, o en el caso de la ejecución serial cada iteración crea una molécula y verifica si puede crear una molécula de agua. En el caso de la implementación con hilos, los procesos se regulan y evitan la condición de carrera a la memoria compartida con semáforos.

Una salida con N=10 con hilos es la siguiente:

Se creó un átomo de hidrogeno [7]

Se creó un átomo de hidrogeno [6]

Se creó un átomo de oxigeno [8]

Se creó un átomo de hidrogeno [5]

Molécula de agua creada por un O [8]

Se creó un átomo de oxigeno [4]

Se creó un átomo de oxigeno [9]

Se creó un átomo de oxigeno [10]

Se creó un átomo de hidrogeno [3]

Molécula de agua creada por un H [3]

Se creó un átomo de hidrogeno [2]

Se creó un átomo de oxigeno [1]

Destruyendo los recursos de memoria compartida.

Código Java:

1-) -Los hilos se sincronizan mediante los semáforos, de manera similar a la aplicación implementada en c++. Utiliza los siguientes métodos: “SignalH()” ,”SignalO()”, “WaitH()” y “WaitO()” para que según la molécula haga “signal” ó “wait” según corresponda.

2-) -La condición de carrera se podría dar en cH y cO, es por esta razón que se implementan mediante un “AtomicInteger()”, este es una implementación de biblioteca que nos permite crear “ints” o enteros que funciona eficazmente para incrementar contadores y evita la condición de carrera.

Código Python:

1-) -Los hilos se sincronizan mediante el uso de semáforos, el cuál utiliza los siguientes métodos: “def signalH(self)”, “def signalO(self)”, “def waitH(self)” y “def waitO(self)” para sincronizar el trabajo de los hilos según las moléculas que se crean.

2-) -La condición de carrera se evita mediante las variables: “self.lockH = threading.Lock()” y “self.lockO = threading.Lock()”. Las cuáles sirven para que los hilos entren ordenadamente a la memoria compartida e incrementen, decrementen o retornen las variables “self.cH” y “self.cO” sin condición de carrera.

b-)

1-) -Un monitor tiene como funcionamiento solucionar los problemas de concurrencia de los hilos, permite ejecutar códigos en paralelo sin tener problemas de sincronización. El monitor hace que sólo haya un hilo por método por lo que evita los problemas de concurrencia.

-Las variables de condición tiene la funcionalidad de generar “signal” o “wait” según un evento esperado. Esto permite generar sincronización de los procesos mediante eventos que otros hilos o subprocesos puedan generar.

2-) -Los métodos "SP" y "SV" realizan dos procesos simultáneos de ”wait” y “signal” correspondientemente sobre dos semáforos distintos.

3-) -La pantalla se manipula mediante un cuadro, dentro del cual las variables van cambiando según los hilos y su información con respecto al problema de los filósofos. La pantalla se crea mediante “displayTitle()” este crea mediante “WINDOW” el cuadro original, y la información varía con el método “displayPhilosopher()”.

-Las variables globales que se utilizan en el código son “Semaphore * table” y “Mutex * mutex” y “NUMWORKERS”. Este último define el número de trabajadores o hilos que va a ejecutar el programa.

-El programa utiliza los mutex para evitar la condición de carrera en el momento en que los hilos modifian las variables en “displayPhilosopher()” y la sincronización de los hilos se hace mediante los semáforos que están contenidos en “Semaphore * table” y se utilizan en el método “Philo()” el cuál se encarga de simular la ejecución de cada hilo.

4-) -Las variables globales se obviaron por la creación de un segmento de memoria compartida, dentro del “int main()” se inicializa esta memoria compartida, la cual es enviada como parámetro a “Philo()”. El programa elimina las variables globales pero dentro de los métodos se desarrollan los semáforos y mutex. Los semáforos se inicializan en “int()” junto con los mutex utilizados para evitar condición de carrera. Los philosofos ejecutan los métodos de signal y wait desarrollados en “semSignal()” y “semWait()” utilizando su propio id de filósofo.

-La sincronización de los procesos se hace mediante “semSignal()” y “semWait()” los cuales son métodos que hacen signal y wait doble lo cual garantiza la sincronización de los hilos.

-El programa resuelve de la misma manera el problema que el anterior, lo único que varía es la inexistencia de las variables globales.

5-) -La sincronización de los hilos en el programa se realiza mediante mutex y condiciones de variable definidas en “std::condition_variable self[5]” las cuáles permiten a los hilos saber cuando hacer wait y cuando hacer signal mediante a los estados en la mesa de los filósofos.

c-)

-Para la funcionalidad del programa de los filósofos se puede generar un “#pragma omp parallel” para que los hilos ejecuten el método “Philo()” con las mismas condiciones utilizadas para la sincronización. Además se puede utilizar un “#pragma omp critical” para evitar las condiciones de carrera en los display del programa.