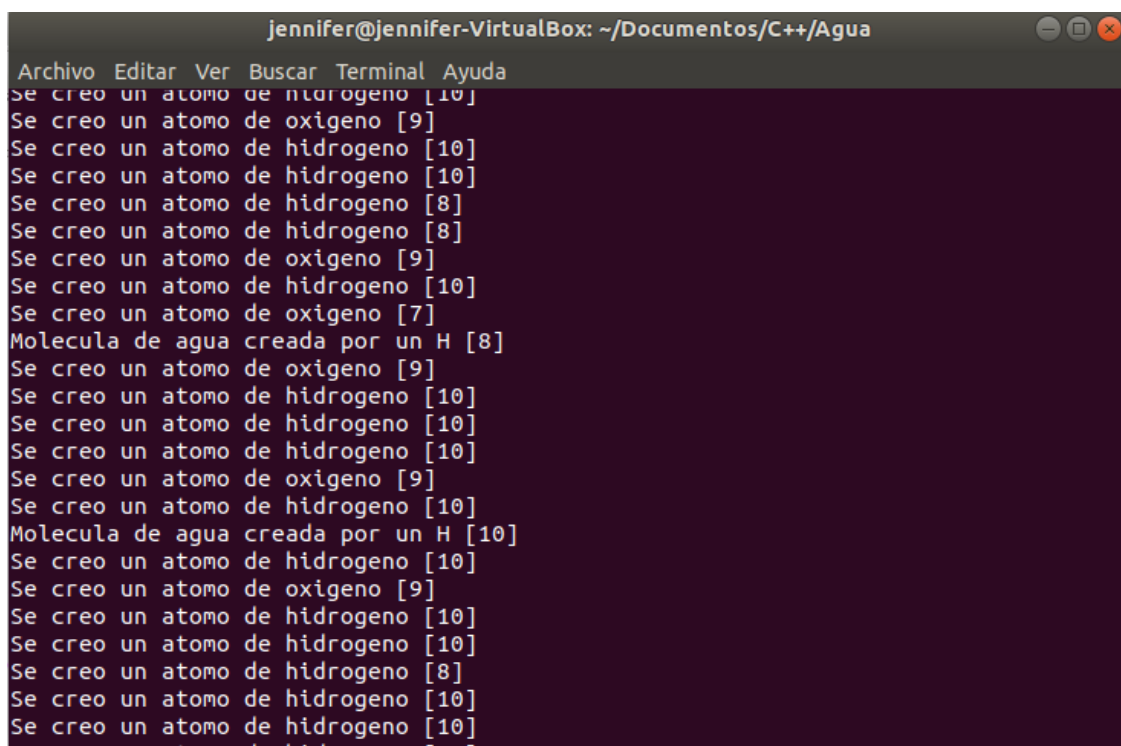


1. Problema del agua.

- Analice la solución en C++ y empleando "fork", corra este ejemplo y explique la salida: al ejecutar el programa, este presenta una condición de carrera notable, ya que no existe una condición que controle a los "procesos hijos", es decir, no existe un manejo de su inicio y su fin, los procesos continúan ejecutando el programa una y otra vez aunque su labor ya haya finalizado. Esto ocasiona que la ejecución del programa "se repita" muchísimas veces más de lo esperado, al continuar ejecutando la tarea encomendada más de una vez. Esto puede observarse al crear los átomos, ya que se crea una gran cantidad de ellos con el mismo id.



```
jennifer@jennifer-VirtualBox: ~/Documentos/C++/Agua
Archivo Editar Ver Buscar Terminal Ayuda
Se creo un atomo de hidrogeno [10]
Se creo un atomo de oxigeno [9]
Se creo un atomo de hidrogeno [10]
Se creo un atomo de hidrogeno [10]
Se creo un atomo de hidrogeno [8]
Se creo un atomo de hidrogeno [8]
Se creo un atomo de oxigeno [9]
Se creo un atomo de hidrogeno [10]
Se creo un atomo de oxigeno [7]
Molecula de agua creada por un H [8]
Se creo un atomo de oxigeno [9]
Se creo un atomo de hidrogeno [10]
Se creo un atomo de hidrogeno [10]
Se creo un atomo de hidrogeno [10]
Se creo un atomo de oxigeno [9]
Se creo un atomo de hidrogeno [10]
Molecula de agua creada por un H [10]
Se creo un atomo de hidrogeno [10]
Se creo un atomo de oxigeno [9]
Se creo un atomo de hidrogeno [10]
Se creo un atomo de hidrogeno [10]
Se creo un atomo de hidrogeno [8]
Se creo un atomo de hidrogeno [10]
Se creo un atomo de hidrogeno [10]
Se creo un atomo de hidrogeno [10]
```

(a) Ejecución del programa con condición de carrera.

- Cambie el código para controlar las condiciones de carrera y emplee su clase de semáforos, debe entregar este programa: el programa se encuentra dentro de la carpeta AguaSem. Captura de la ejecución:

```

jennifer@jennifer-VirtualBox:~$ cd ~/Documentos/C++/Agua
jennifer@jennifer-VirtualBox:~/Documentos/C++/Agua$ make
g++ -c -o Semaforo.o Semaforo.cpp
g++ -c -g Semaforo.cpp main.cpp
g++ -g Semaforo.o main.o -o agua
jennifer@jennifer-VirtualBox:~/Documentos/C++/Agua$ ./agua
Se creo un atomo de oxigeno [5]
Se creo un atomo de hidrogeno [6]
Se creo un atomo de oxigeno [4]
Se creo un atomo de hidrogeno [10]
Se creo un atomo de hidrogeno [7]
Se creo un atomo de hidrogeno [3]
Molecula de agua creada por un H [3]
Se creo un atomo de hidrogeno [8]
Molecula de agua creada por un H [8]
Se creo un atomo de oxigeno [2]
Se creo un atomo de oxigeno [9]
Se creo un atomo de oxigeno [1]
Destruyendo los recursos de memoria compartida
jennifer@jennifer-VirtualBox:~/Documentos/C++/Agua$

```

(b) Ejecución del programa sin condición de carrera y utilizando la clase propia de Semáforo.

Revise la solución en Java:

- Explique cómo se sincronizan los hilos: los hilos en este programa se sincronizan por medio de la clase Thread. La clase Partícula posee un "extends thread", comando que se encarga de crear una instancia de la clase Thread en Partícula. Esta clase utiliza el método run() para declarar el punto de entrada del nuevo hilo. Dentro de este método se llaman a los respectivos creadores de Oxígeno e Hidrógeno, los cuales poseen sus respectivos semáforos.
- Explique cómo es manejada las condiciones de carrera: además de los semáforos especificados en los métodos anteriores, la condición de carrera también es controlada por medio de las variables "atomic". Las cuáles pertenecen a los algoritmos encargados de comparar e intercambiar los datos para así asegurar su integridad. Estas variables poseen el método get() utilizado en el programa, que permite obtener el valor de memoria y así los cambios hechos por otros hilos son visibles.

Revise la solución en Python:

- Explique cómo se sincronizan los hilos: se logra utilizando el módulo thread, el cual instancia un objeto de la clase Thread. De esta manera en el programa se inicializan las variables compartidas. Para inicializar el proceso concurrente se implementa el comando start() como se realiza en el main, y se espera a que todos los hilos terminen su trabajo mediante join().
- Explique cómo es manejada las condiciones de carrera: se utilizan locks. Estos intervienen cuando se necesita realizar alguna operación en las variables compartidas, permitiendo que solo un hilo a la vez pueda "entrar" al método y modificar sus datos.

2. Problema de los filósofos comensales.

- Explique ¿Qué es un monitor? y ¿Qué son variables de condición?: Un monitor representa una estructura en la cual sus métodos son ejecutados con exclusión mutua, permitiendo el uso de ellas por más de un hilo. En palabras más sencillas, un monitor es "algo" que un hilo puede tomar y retener, evitando que todos los otros hilos también lo tomen, obligándolos a esperar hasta que este se libere. Las variables de condición también se encargan de la sincronización. Estas se encargan de que los hilos esperen hasta que ocurra una condición particular. Estas no se pueden compartir entre procesos. Las variables permiten que los hilos se liberen de un lock() y entren en un estado de suspensión (tipo sleep).

Revise la implantación de la clase "Semaphore":

- Explique el funcionamiento de los métodos "SP" y "SV": ambos realizan las operaciones de "semop" sobre dos semáforos. SP se encarga de aplicar la función Wait y SV la función Signal. Para esto se crea un vector tipo sembuf, cada uno de sus índices contiene el id de un semáforo.

Revise la solución con procesos "philofork-gv.cc":

- Explique la manera de manipular la pantalla y de desplegar la salida en el cuadro: primero se establece el encabezado de la "imagen" o forma que se quiere dar de salida. Ya en el método displayPhilosopher se termina de formar "el cuerpo" de la misma, este método recibe el id o número de un filósofo, así como su estado. Se definen las variables necesarias para la construcción de la ventana deseada. Los datos que se mostrarán en ese marco se definen en el switch. Cuando los datos están en su lugar se termina de formar el marco llamando a las variables anteriormente definidas. El método displayPhilosopher será llamada cada vez que un hilo culmine.
- Estudie la solución y explique cuáles son las variables globales y cómo se utilizan: el método Philo define dos variables, estas reciben un número random, el cual será el tiempo de espera que cada filósofo deberá permanecer en un mismo estado (comiendo o pensando). Lo primero que hace es pensar, espera su tiempo definido por la función rand() y se realiza un Wait() simultáneo a los dos semáforos (variables globales) para asegurar de que el filósofo actual no tiene a algún adyacente comiendo, es decir, que los palillos están libres. El estado pasa a comer, con su respectivo tiempo de espera (sleep) y luego se les hace un Signal() a los dos semáforos para que "liberen" los palillos, el estado nuevamente pasa a comer. El Mutex (variable global) se inicializa en el main para ser utilizado en la impresión en cola, así cada hilo o filósofo imprime su estado donde corresponde, sin que otro intervenga y tome el monitor para él.
- Explique la manera en que se realiza la sincronización de los procesos: se controla que los procesos hijos sean los encargados de realizar el método Philo, el cual por si mismo ya contiene semáforos para mantener el orden. Los procesos padres, luego de recibir la información de sus hijos, llaman al método displayPhilosopher, el cual también posee mutex para controlar una posible condición de carrera.

Revise la solución con procesos "philofork-ngv.cc":

- Estudie la solución y explique la manera en que se obviaron las variables globales: la solución del método Philo sigue siendo la misma, solo que ahora utilizando la estructura enviada por parámetro. Esta estructura posee un semáforo que representa cada palillo disponible en la mesa, además de otro semáforo que toma la función de un mutex, para controlar la impresión en consola. De esta manera ya no se necesitan las variables globales, ya que esta estructura, por medio de memoria compartida, podrá ser enviada y modificada por los hilos.
- Explique la manera en que se realiza la sincronización de los procesos: por medio de los múltiples semáforos. Display poseen a mutex, el método Philo a cada uno de los palillos como semáforos también, y en el main la memoria compartida se destruye apenas finaliza la función del proceso padre.
- ¿Resuelve el problema de la misma manera que la estrategia anterior?: tiene la misma lógica, pero no la misma solución.

Analice la versión con PThreads

- Explique la manera en que se realiza la sincronización de los hilos: cada filósofo, o hilo, posee su propia variable de condición, la cual se utiliza para comprobar el estado del filósofo, por medio del método test(). Cada método de la clase posee un lock a su inicio, llamado mediante el mutex de la clase diningPh. PhiloPT ejecuta el main creando un hilo para llamar al método Philo, genera la cantidad de hilos según la cantidad de filósofos existentes. Luego espera la finalización de cada uno de ellos mediante join().

3. Explique con palabras, cómo resolvería algunos de estos programas con OpenMP.

Utilizando la idea del programa "philoFork-gv.cc", utilizaría en lugar del fork() creador de los procesos hijos, el pragma parallel con el num threads como el número de filósofos disponibles. Para esperar al proceso que haya terminado, utilizaría un pragma barrier (de un solo hilo, para que no espere a todos). El display tendría un pragma critical, para que solo un hilo pudiese realizarlo.