Michael Chan - 18562
Laboratorio#3 Sistemas Operativos

1. ¿Qué es una race condition y por qué hay que evitarlas?
    a. Race condition es cuando la salida o estado de un proceso es dependiente de una secuencia de eventos que se ejecutan en un orden arbitrario y trabajan sobre un recurso compartido. En nuestro programa tenemos el mismo problema al tener diferentes procesos que trabajan sobre el mismo sudoku y debemos esperar a que todos hagan su verificación para dar el veredicto final.

2. ¿Cuál es la relación, en Linux,entre `pthreads` y `clone()`? ¿Hay diferencia al crear threads con uno o con otro?¿Qué es más recomendable?
    a. clone() es una instrucción en Linux de más bajo nivel y puede ser utilizado tanto para hacer multiprocessing y multithreading, mientras que los threads son de más alto nivel y solo se utilizan para multithreading. Si trabajáramos con clone() tendríamos mayor control de todo pero conlleva a más trabajo y más complejidad.

3. ¿Dónde,en su programa,hay paralelización de tareas, y dónde de datos?
    a. Tenemos paralización de tareas al momento de correr en el proceso hijo la verificación de Columnas y dentro de cada for tenemos paralelización gracias a OpenMP, por otra parte al todos trabajar con el mismo sudoku estamos trabajando paralelización de datos.
4. Al agregar los `#pragma`s a los ciclos `for`, ¿cuántos LWP's hay abiertos antes de terminar el `main()` y cuántos durante la revisión de columnas?¿Cuántos user threads deben haber abiertos en cada caso, entonces? Hint:recuerde el modelo de multithreading que usan Linux y Windows.
    a. Linux y Windows usan el modelo one to one, por lo cuál en cada caso se deben haber abierto 9 threads, uno por cada fila/ columna/ cuadro.
5. Al limitar el número de threads en `main()` a uno, ¿cuántos LWP's hay abiertos durante la revisión de columnas? Compare esto con el número de LWP's abiertos antes de limitar el número de threads en `main()`. ¿Cuántos threads(en general) crea OpenMP por defecto?
    a. Cuando se limita ya solo tenemos un LWP, OpenMP reconoce la cantidad de procesadores y crea la misma cantidad de threads por defecto. Sin embargo, nosotros podemos definir la cantidad de threads para funcionar según un for como lo es en el caso que usamos.

6. Observe cuáles LWP's están abiertos durante la revisión de columnas según `ps`. ¿Qué significa la primera columna de resultados de este comando? ¿Cuál es el LWP que está inactivo y por qué está inactivo?Hint: consulte las páginas del manual sobre `ps`.
    a. La primera columna "F" hace referencia a nivel de jerarquía, por lo cual indica si es un thread creado por el padre o por el hijo. Ya que las columnas solo corren en el proceso hijo, y en el padre existe un momento en el que debe esperar el resultado, existe un momento donde se está inactivo.
7. Compare los resultados de `ps` en la pregunta anterior con los que son desplegados por la función de revisión de columnas per se.¿Qué es un thread team en OpenMP y cuál es el master thread en este caso?¿Por qué parece haber un thread "corriendo", pero que no está haciendo nada? ¿Qué significa el término busy-wait?¿Cómo maneja OpenMP su thread pool?
    a. Con OpenMP podemos crear grupos de threads que se ejecutarán a la vez, la manera de agrupar se ve como un árbol y el master thread es sobre el cual se ejecutan todos. Que thread que aparece corriendo pero no hace nada se es aquél que está por encima de todos los otros y solo observa en qué momento el programa termina su ejecución. Busy-wait lo podemos utilizar para asegurarnos de que otro thread termina su ejecución antes de realizar otra acción.
8. Luego de agregar por primera vez la cláusula `schedule(dynamic)` y ejecutar su programa repetidas veces, ¿cuál es el máximo número de threads trabajando según la función de revisión de columnas? Al comparar este número con la cantidad de LWP's que se creaban antes de agregar `schedule()`, ¿qué deduce sobre la distribución de trabajo que OpenMP hace por defecto?
    a. Dentro de schedule nosotros pasamos como parámetro el cual asumo a la manera en la que asignaremos los threads, al hacerlo dinámicamente busca la manera óptima de hacerlo.
9. Luego de agregar las llamadas `omp_set_num_threads()` a cada función donde se usa OpenMP y probar su programa, antes de agregar `omp_set_nested(true)`, ¿hay más o menos concurrencia en su programa? ¿Es esto sinónimo de un mejor desempeño? Explique.
    a. Antes de implementar `omp_set_nested(true)` existe mayor concurrencia ya que no se define un orden , luego de `omp_set_nested(true)` ya tenemos a nuestros threads agrupados y se ejecutan de una manera más ordenada.
10. ¿Cuál es el efecto de agregar `omp_set_nested(true)`? Explique.

a. A pesar de tener un mejor ordenamiento de threads puede ser que probablemente tenga una mayor tiempo de ejecución, porque cuando no poseía cierto orden, ejecutaba cada thread en el momento que pudiera, mientras que si está en un grupo debe seguir cierto orden y no se posee la libertad de ejecutar cualquier thread en cualquier momento.