Laboratorio #3

Fecha de Entrega: 16 de marzo, 2022.

Descripción: en este laboratorio se empleará multithreading por medio de pthreads y OpenMP para desarrollar un verificador de soluciones para sudokus de nueve por nueve. Los entregables serán todo el código escrito, así como un documento que responda a las preguntas planteadas al final. Se recomienda auxiliar sus respuestas con screenshots de la ejecución de su programa.

Materiales: una máquina virtual con Linux o Windows, pero que tenga GCC, y documentación sobre OpenMP.

Contenido:

OpenMP busca paralelizar (no sólo ejecutar de forma concurrente), por lo que su funcionamiento requiere de más de un procesador. Si usará una máquina virtual, comience por asegurarse de que su máquina cuente con cuatro procesadores (en VirtualBox, ventana Settings, menú System, tab Processor). La cantidad se especifica para uniformizar las respuestas a las preguntas que se plantean al final.

Cree un programa en C llamado SudokuValidator.c. En él escriba tres funciones que se encarguen de revisar que todos los números del uno al nueve estén:

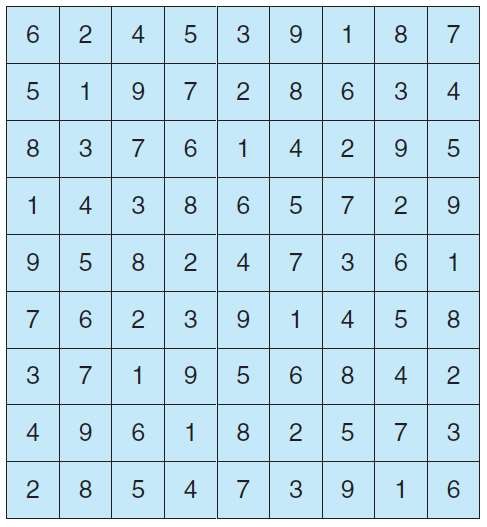
* En cada columna de un arreglo de nueve por nueve.
* En cada fila de un arreglo de nueve por nueve.
* En un subarreglo de tres por tres dentro de un arreglo de nueve por nueve.

Las funciones para verificación de filas y columnas serán iguales exceptuando un intercambio de índices al recorrer el arreglo. La función de revisión de subarreglos debe recibir una fila y una columna para ubicar la esquina superior izquierda de un cuadrado de tres por tres, donde iniciará la revisión dentro del arreglo de nueve por nueve. Todas estas funciones se deben basar en ciclos for obligatoriamente.

Este programa recibirá, en terminal, la ubicación de un archivo (sólo el nombre, si está en el mismo directorio que SudokuValidator.c) que contiene una solución a un sudoku de nueve por nueve. El formato de las soluciones debe ser un único string de ochenta y un dígitos, en la primera línea, comenzando por la celda superior izquierda del sudoku y avanzando de izquierda a derecha, por filas. Para este laboratorio se provee una solución de ejemplo en el archivo “sudoku”.

Lo primero que su main() deberá hacer es abrir el archivo usando open() y mappearlo a su memoria usando mmap(). Luego debe ejecutar un for en el que se copie cada símbolo del string en el archivo de solución a un arreglo bidimensional de nueve por nueve, de modo que le quede una grilla lógica como la que se muestra en la página siguiente.

Se recomienda que su arreglo bidimensional sea global (es decir, que esté declarado fuera del main()) para que sea accesible por varios threads. Luego de llenar la grilla, escriba un for que haga la revisión, con su función, de los subarreglos de tres por tres que conforman el arreglo de nueve por nueve (nota: revise los subarreglos de tres por tres cuya primera posición (si comenzamos desde 1) sea [𝑖, 𝑖] para 𝑖 ∈ {1,4,7}).



Grilla lógica ejemplar

Luego de lo anterior, obtenga el número de proceso (no el de thread) y ejecute un fork(). En el proceso hijo convierta el número del proceso padre (no el de thread) a texto, y ejecute por medio de execlp() el siguiente comando:

# ps –p <#proc> -lLf

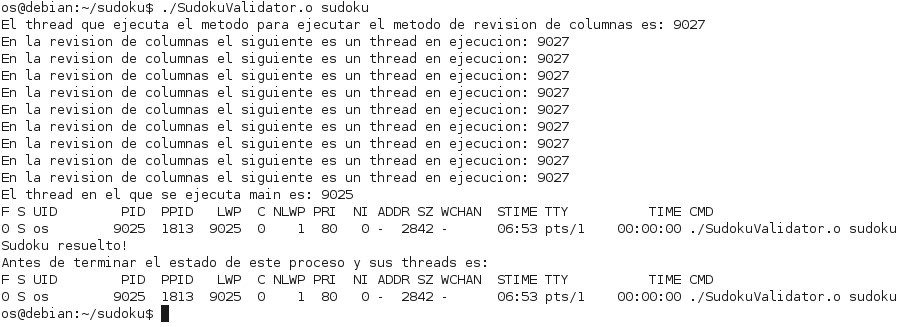
donde <#proc> es el número del proceso padre. Este comando permite ver información relacionada al proceso <#proc> que incluye los lightweight processes que tenga asociados.

En el proceso padre:

* Cree un pthread que haga su revisión de columnas.
* Ejecute pthread\_join() y luego despliegue el número de thread en ejecución. Para lograrlo debe #incluir <sys/syscall.h> en su programa y ejecutar syscall(SYS\_gettid) (el resultado de esta llamada de sistema es el id del thread).
* Espere a que concluya el hijo que está ejecutando ps.
* Realice su revisión de filas.
* Despliegue si la solución al sudoku es válida o no.
* Ejecute un nuevo fork() y ejecute el comando ps en el proceso hijo, tal como se describe en instrucciones anteriores. Esto servirá para comparar el número de LWP’s asociados al proceso padre cuando se está realizando la revisión de columnas y cuando (el padre) está a punto de terminar.
* Espere al hijo y retorne 0.

Observe que la creación de un thread que ejecute la revisión de columnas implica la creación de una función que sea asignable a un thread en el cual, a su vez, se ejecute su función de revisión de columnas. Es decir, una función que tenga tipo de retorno void\* y que termine con pthreads\_exit(0). En esa función tipo void\* también despliegue el número de thread en ejecución.

El siguiente es un ejemplo de cómo podría verse el output de su programa hasta este momento:

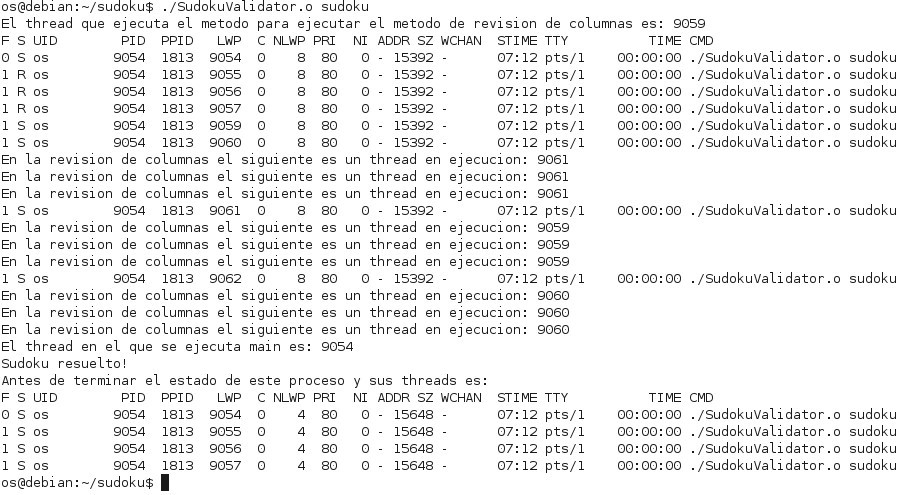


Como siguiente paso deberá paralelizar todos los ciclos for que pueda (vea la nota importante) usando OpenMP. Para ello simplemente es necesario que la siguiente línea preceda inmediatamente a la del for en cada caso:

# #pragma omp parallel for

Importante: evite las race conditions. Investigue el uso de la directiva private de OpenMP para auxiliarse en este aspecto. No todos los ciclos for deberán ser precedidos por la directiva.

Ejecutar su programa ahora deberá resultar en un output similar al siguiente:



Anote el número de LWP’s que se tienen durante la revisión de columnas y antes de terminar el programa.

Agregue la siguiente instrucción al principio de main():

# omp\_set\_num\_threads(1);

Ejecute su programa y note el resultado de las ejecuciones de ps. También anote los números de thread desplegados durante la revisión de columnas.

Ahora, agregue la siguiente directiva a todas las líneas #pragma que incluyó anteriormente:

# schedule(dynamic)

Ejecute su programa varias veces y observe los números de thread que se despliegan durante la revisión de columnas. Compárelos con el resultado de ps que se despliega durante la ejecución del pthread y anote sus observaciones.

Como siguiente paso, agregue una llamada a omp\_set\_num\_threads() al inicio de cada función donde se ejecute un for paralelo, determinando el número de threads adecuados (e.g., si su función ejecuta un for paralelo de nueve iteraciones, posiblemente el número de threads deba ser nueve). Ejecute su programa varias veces y anote los efectos sobre los threads en los resultados de ps. Repita el procedimiento comentando la cláusula schedule() en el primer for paralelo de su revisión de columnas. Finalmente agregue la siguiente instrucción al principio de cada función que use OpenMP:

# omp\_set\_nested(true);

Ejecute su programa y anote los efectos sobre el resultado.

Responda las siguientes preguntas:

1. **¿Qué es una race condition y por qué hay que evitarlas?**

La condición de carrera (race condition) ocurre cuando dos o más procesos acceden un recurso compartido sin control, de manera que el resultado combinado de este acceso depende del orden de llegada. Es muy probable que el otro thread haya leído el valor previo a ser modificado, esto hará que haga el proceso con el valor previo y obtendremos resultados no esperados. Si las race conditions no son manejadas debidamente pueden causar errores el programa, como el antes mencionado.

1. **¿Cuál es la relación, en Linux, entre pthreads y clone()? ¿Hay diferencia al crear threads con uno o con otro? ¿Qué es más recomendable?**

Al realizar pthread créate se usa clone para crear un nuevo thread. Pthread utiliza subprocesos. Clone es una llamada al sistema de Linux para crear procesos y threads. No se recomienda usar clone ya que complica la situación al no configurar subprocesos de POSIX. No hay diferencia al crear threads con pthreds o clone ya que la implementación de pthreads en Linux usa clone.

1. **¿Dónde, en su programa, hay paralelización de tareas, y dónde de datos?**

A nivel de tareas en mi programa hay paralelización en los ciclos for hay en los pthreads y hay de datos en OpenMP

1. **Al agregar los #pragmas 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.**

l terminar el main() hay 4 LWP’s, sin embargo, durante la revisión de columnas tenemos 8. Esto quiere decir que hay 4 y 8 user threads abiertos respectivamente.

1. **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?**

Son 4 threads de usuarios abiertos y 5 durante la evaluación. Antes de terminar es 1

1. **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.**

La columna S indica el estatus del proceso, sabiendo esto, podemos decir que los 4 LWP están activos (R) y uno inactivo (S). En este caso, el LWP inactivo es el proceso que dio origen a los otros 4 procesos, el proceso padre. Esta inactivo ya que este proceso espera a que los hijos terminen sus tareas para poder continuar.

1. **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?**

El master thread es el thread que esta corriendo sin hacer nada sobre las operaciones, esto se debe a que este thread se encarga de controlar que los otros threads hayan realizado sus tareas. OpenMP abre un thread pool, donde los threads estan corriendo. Una vez que los threads de kernel se hayan creado y terminan su trabajo se vuelven a un dock para esperar por tareas nuevas.

1. **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?**

Un máximo de 4 por los procesadores asignados a la máquina virtual. Se concluye que OMP implementa un balance de cargas equitativas para cada thread.

1. **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.**

En el programa hay más concurrencia. Esto no significa mejor desempeño directamente ya que al tomar en cuenta el overhead y el tiempo aumentaría por la organización de los threads

1. **¿Cuál es el efecto de agregar omp\_set\_nested(true)? Explique.**

Esto permite una paralelización anidada. Esto permite que los ciclos anidados creen threads propios, esto crea más LWP’s