Proyecto #1

Propósito:

Muchos programas para computadores de escritorio y servidores como: editores de texto, servidores web, y servidores de bases de datos son programas multi-threads. Este proyecto tiene la intención de fortalecer el entendimiento de los programas multi-threads haciendo énfasis en su funcionamiento, que ventajas poseen los entornos multi-threads sobre los entornos single-thread, y en que deben centran la atención los programadores que esperan realizar un programa multi-threads, donde los threads cooperen de manera correcta. Adicionalmente se espera que mediante la realización de este proyecto el estudiante se familiarice con diferentes funciones de la biblioteca *pthread* (ej: creación de threads, sincronización entre múltiples threads). Se recomienda al estudiante revisar las páginas *man* de: *tmpfile(), pthread_create(), pthread_join(), pthread_exit()*, además de otras funciones relevantes de la biblioteca *pthread*.

El proyecto consiste en la construcción de un programa que permitirá ordenar una gran cantidad de datos trabajando de manera paralela. Inicialmente usted debe asumir que los datos de entrada se entregaran por separado vía archivos independientes. Usted deberá ordenar cada uno de estos archivos mediante un thread (un thread por archivo). Luego de que cada uno de los threads complete su ejecución, deberá mezclar todos los archivos de entrada en un solo archivo, tomando dos archivos a la vez. Este proceso le ayudará a recrear el archivo original (de gran tamaño) que en un principio le fue dado en diferentes trozos. Este método de manipulación del archivo mejora de manera significativa el tiempo de ejecución, ya que los threads tienden a trabajar en paralelo.

Trasfondo:

La biblioteca *pthread* (*POSIX pthread*) es el estándar básico del API de threads para C/C++. Esta biblioteca permite crear o engendrar un nuevo flujo de ejecución concurrente en un proceso. Su uso es más efectivo en sistemas multi-procesadores o multi-core, ya que los flujos de instrucción pueden ser planificados para su ejecución en diferentes procesadores, mejorando el desempeño en cuanto a velocidad de procesamiento. Además de la ventajas a nivel de velocidad, la creación de un thread genera menos sobrecarga que la creación de un proceso (*fork()*), ya que el sistema operativo no debe inicializar un nuevo espacio en memoria virtual, o crear un ambiente adicional.

Actividades:

A continuación se describen en profundidad las actividades que deben realizar los estudiantes.

[Parte 1]

En este proyecto usted deberá construir un programa multithread de ordenamiento y clasificación. El thread principal de su programa recibirá como entrada vía la línea de comandos los nombres de los archivos a ordenar.
El número mínimo de archivos de entrada es dos (2); sin embargo, usted no debe realizar asunciones sobre el número máximo de archivos. Una vez procesados los nombres de los archivos, el thread principal creará un thread "trabajador" por archivo, al cual le pasará el nombre del archivo sobre el cual trabajará. Usted puede asumir que todos los nombres de archivos son válidos y diferentes.

[**Parte 2]**

Los archivos referenciados en **[Parte 1]** contienen una lista de *strings*. Cada línea en el archivo será un *string*; sin embargo, usted no puede realizar asunciones sobre la longitud del *string* . El número total de líneas en el archivo puede ser almacenado en el tipo de dato *unsigned int* definido en el estándar C.

Mientras el thread principal crea o engendra a los threads "trabajadores", los threads "trabajadores" leen los datos del archivo (pasado por parámetro) ordenándolo en forma lexicográfica decreciente (z va primero que a). El orden es *case insensitive* (usted puede usar *strcasecmp()* función que trabaja como *strcmp()* pero en *case insensitive*). El thread "trabajador" entonces escribe el conjunto de *strings* ordenados en un segundo archivo añadiendo la extensión ".sorted" (ej: file1.txt.sorted). Note que no se realizan modificaciones sobre el archivo original, por lo tanto los resultados se almacenan en un nuevo archivo. Las líneas vacías en el archivo original deben ser ignoradas, y no deben ordenarse o imprimirse en el archivo ordenado.

Como el archivo de entrada, el archivo de salida ordenado consta de un *string* por línea. Usted está obligado a usar la función estándar de C *qsort()*. Si usted no está familiarizado con *qsort()* consultes las páginas *man*. Una vez que ha finalizado el ordenamiento del archivo, el thread "trabajador" termina.

- Cuando un nuevo thread es creado, el thread principal debe pasar además del nombre del archivo a ordenar, un apuntador a un struct "stat". El thread debe salvar las siguientes estadísticas en la estructura:
    - Número de líneas ordenadas.
    - La línea más larga ordenada por el thread.
    - La línea más corta ordenada por el thread (excluyendo las líneas vacías).

- Después de terminar, el thread también debe imprimir el número de líneas que ha ordenado, y el nombre del archivo que ha generado, en el siguiente formato:

Este hilo trabajador escribió XXXX líneas en "YYYY"

[Parte 3]

Los múltiples archivos ordenados creados en la **[Parte 2]** ahora deben ser unidos en un único archivo ordenado. Se deben tomar dos archivos ordenados, y unirlos manteniendo el orden de los *strings* iniciando un thread para unir cada par de archivos. Se debe repetir este paso de forma iterativa hasta que se tenga un único archivo como resultado. Cuando se unan dos archivos, si una línea aparece en ambos archivos, debe aparecer sólo una vez en el archivo resultado. Se deben eliminar todos los archivos intermedios creados durante este procedimiento. (La función *tmpfile()* será de utilidad aquí; esta devuelve un puntero a un archivo temporal con permiso de acceso "w+" y se asegura que el archivo se elimine al finalizar el programa). Sólo debe existir un único archivo de salida que contenga todos los *strings* de los distintos archivos ordenados, sin duplicados, y debe ser llamado "sorted.txt".

Para unir cada par de archivos, se debe generar un nuevo thread. En cada nivel particular del árbol de unión (ver Figura 1), se debe recordar esperar a que retornen los threads del nivel previo antes de generar nuevos threads. El thread encargado de la unión debe mostrar el número total de líneas de cada uno de los dos archivos a unir, así como el número total de líneas luego de la unión.

Fusionado XXXX líneas y YYYY líneas en ZZZZ líneas

Finalmente, antes de que el programa termine, debe imprimir una última línea:

Un total de XXXX strings fueron pasados como entrada,
String más largo ordenado: LLLLLLLLLLLLLLL
String más corto ordenado: SSSSS

[Ejemplo a considerar]

Considere el siguiente ejemplo demostrativo de la situación anteriormente descrita:

En primer lugar **[Parte 1]** un usuario introduce ejecuta su código con los siguientes archivos:

./pf1 a1.txt a2.txt a3.txt a4.txt a5.txt a6.txt a7.txt

En segundo lugar **[Parte 2]** se ordenan cada uno de los archivos pasados por parámetro, sin modificar los archivos originales:

"a1.txt.sorted" es una copia ordenada de "a1.txt"
"a2.txt.sorted" es una copia ordenada de "a2.txt"
"a3.txt.sorted" es una copia ordenada de "a3.txt"
"a4.txt.sorted" es una copia ordenada de "a4.txt"
"a5.txt.sorted" es una copia ordenada de "a5.txt"
"a6.txt.sorted" es una copia ordenada de "a6.txt"
"a7.txt.sorted" es una copia ordenada de "a7.txt"

Adicionalmente cada thread al culminar imprime el número de líneas que ha ordenado, y el nombre del archivo que ha generado.

Este hilo trabajador escribió XXXX líneas en “YYYY”

Su salida en este momento debería ser similar a esta:

Este hilo trabajador escribió 10 líneas en "a7.txt.sorted".
Este hilo trabajador escribió 20 líneas en "a3.txt.sorted".
Este hilo trabajador escribió 30 líneas en "a2.txt.sorted".
Este hilo trabajador escribió 40 líneas en "a4.txt.sorted".
Este hilo trabajador escribió 100000 líneas en "a5.txt.sorted".
Este hilo trabajador escribió 2000000 líneas en "a6.txt.sorted".
Este hilo trabajador escribió 30000000 líneas en "a1.txt.sorted".

Es importante que note que los threads “trabajadores” pueden terminar su ejecución en un orden diferente al que fueron creados.

En tercer lugar **[Parte 3]** se debe dar inicio a la unión de los archivos que han sido ordenados en **[Parte 2]**, manteniendo el orden en el archivo final. Un ejemplo de este proceso se observa en la Figura 1.

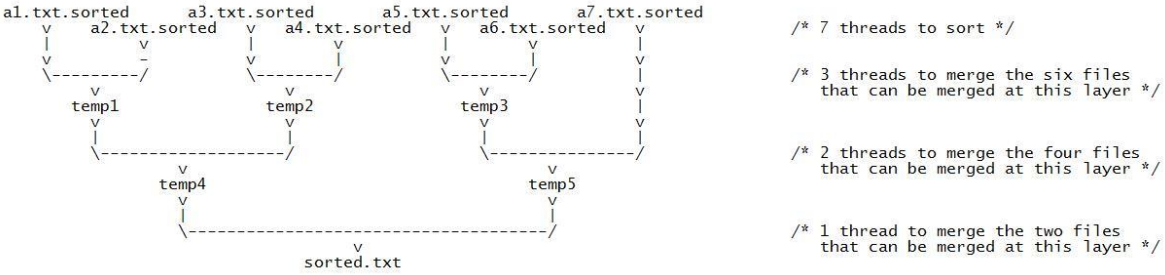

Figura 1. Ejemplo

En cada unión o mezcla el thread “trabajador” debe mostrar por pantalla el número de líneas del archivo de entrada y el número de líneas que se han escrito en el archivo de salida (archivo temporal o definitivo ordenado), después de remover las líneas duplicadas y las líneas vacías. Su salida debería ser similar a esta:

Fusionado 100 líneas y 1000 líneas en 1050 líneas.
Fusionado 10000 líneas y 300 líneas en 10300 lines.
Fusionado 10 líneas y 800 líneas en 801 lines.
Fusionado 1050 líneas y 10300 líneas en 10345 lines.
Fusionado 801 líneas y 1 líneas en 802 lines.
Fusionado 10345 líneas y 802 líneas en 11111 lines.

Al final el thread principal debe imprimir una información resumen similar a la siguiente:

Un total de 12531424 strings fueron pasados como entrada,
String más largo ordenado: hippopotomonstrosesquippedaliophobia
String más corto ordenado: bee

Una vez finalizada la ejecución de su programa, el directorio de trabajo actual sólo debe contener los nuevos archivos ordenados que ha creado, además de los archivos originales sin ningún tipo de modificación.

[Notas]

- Este proyecto puede realizar de manera individual o en grupo que no excedan dos (2) personas.
- El proyecto debe ser codificado en el lenguaje C, y debe correr bajo sistemas UNIX, específicamente Linux (Ubuntu 16.04).
- Usted encontrará utiles las siguientes funciones:
  - Para manipulación de archivos: *fopen(), fscanf(), fprintf(), tmpfile(), fclose(), fgets()*.

  - Para manipulación de *strings*: *strcasecmp(), strcat()*. Para manejo de threads: *pthread_create(), pthread_join()*.
- Usted debe usar la función *qsort()* para ordenar los archivos de entrada. Note que la unión de dos archivos ordenados no requiere una llamada a *qsort()*.
- Usted debe recordar que las copias serán severamente sancionadas.

[Compilación y ejecución]

Su programa debe poder compilarse haciendo uso del comando make desde un intérprete de comando. Su archivo Makefile debe sorportar las siguientes instrucciones:

$> make clean
$> make

La ejecución de su programa debe seguir la siguiente sintaxis:

$> ***pf1*** *<File #1> <File #2> <File #3> [...]*

[Entrega y entregables]

La fecha de entrega: **11 de Mayo hasta las 11:59pm**
Todos los archivos *.c*, *.h* u otros, tienen que estar comprimidos en formato **ZIP** (.zip)

GDSO/dp