Proyecto #2

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 (al cual llamará pf1.c). 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 trabajar 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 arreglo "stats", definido en pf1.c. El thread debe salvar las siguientes estadísticas en estructura del tipo *type stats_t* definida en pf1.h:
    - 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:

This worker thread writes XXXX lines to "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.

Merged XXXX lines and YYYY lines into ZZZZ lines

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

A total of XXXX strings were passed as input,
longest string sorted: LLLLLLLLLLLLLL
shortest string sorted: 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" is a sorted copy of "a1.txt"
"a2.txt.sorted" is a sorted copy of "a2.txt"
"a3.txt.sorted" is a sorted copy of "a3.txt"
"a4.txt.sorted" is a sorted copy of "a4.txt"
"a5.txt.sorted" is a sorted copy of "a5.txt"
"a6.txt.sorted" is a sorted copy of "a6.txt"
"a7.txt.sorted" is a sorted copy of "a7.txt"

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

This worker thread writes XXXXX lines to "YYYYY"

Su salida en este momento debería ser similar a esta:

This worker thread writes 10 lines to "a7.txt.sorted".
This worker thread writes 20 lines to "a3.txt.sorted".
This worker thread writes 30 lines to "a2.txt.sorted".
This worker thread writes 40 lines to "a4.txt.sorted".
This worker thread writes 100000 lines to "a5.txt.sorted".
This worker thread writes 2000000 lines to "a6.txt.sorted".
This worker thread writes 30000000 lines to "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.

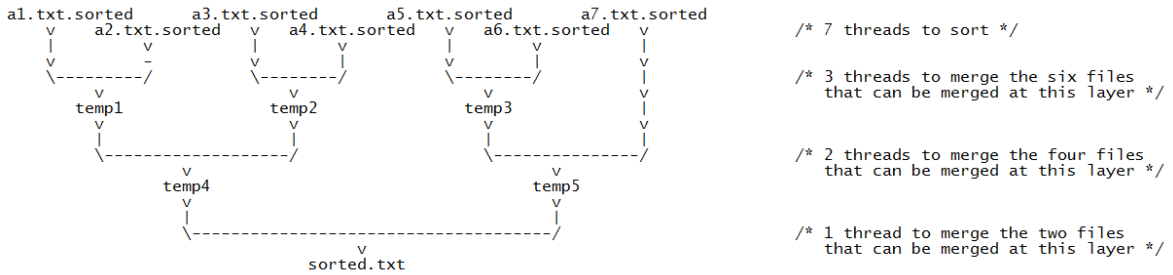

Figura 1. Ejemplo

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:

Merged 100 lines and 1000 lines into 1050 lines.
Merged 10000 lines and 300 lines into 10300 lines.
Merged 10 lines and 800 lines into 801 lines.
Merged 1050 lines and 10300 lines into 10345 lines.
Merged 801 lines and 1 lines into 802 lines.
Merged 10345 lines and 802 lines into 11111 lines.

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

A total of 12531424 strings were passed as input,
longest string sorted: hippopotomonstrosesquippedaliophobia
shortest string sorted: 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 se realizará de manera grupal (mínimo 2 personas) que no excedan de tres (3) personas.
- El asunto del correo debe ser [SO]_[Proy2]_[Cedula1]_[Cedula2]_[Cedula3].
- Los entregables del proyecto deben ser enviados en un archivo .zip al correo laboratoriosisop@gmail.com.
- El proyecto debe ser codificado en el lenguaje C, y debe correr bajo sistemas UNIX, **específicamente Linux**. (**ojo con esta condición**)
- Comience su proyecto usando los archivos provistos por el grupo docente (pf1.c y pf1.h), y no renombre estos archivos. Usted puede agregar funciones y/o editarlos pero no cree otros archivos.
- Usted encontrará utiles las siguientes funciones:
    - Para manipulación de archivos: *fopen(), fscanf(), fprintf(), tmpfile(), fclose(), fgets()*.

  - Para manipulación de *strings*: *strcmp(), 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()*.
- Cada thread "trabajador" debe imprimir sus *strings* justo antes de terminar.
- 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> [...]

[Entrega y entregables]

La fecha tope 31 de Julio de 2024 (tomen previsiones, por razones obvias).

GDSO/dp