Sistemas Operativos 1/2018
Laboratorio 3

Profesores:
Cristóbal Acosta (cristobal.acosta@usach.cl)
Miguel Cárcamo (miguel.carcamo@usach.cl)
Fernando Rannou (fernando.rannou@usach.cl)
Ayudantes:
Esteban Alarcón (esteban.alarcon.v@usach.cl
Marcela Rivera (marcela.rivera.c@usach.cl))

I. Objetivos Generales

Este laboratorio tiene como objetivo aplicar los conceptos de hebras, mediante la construcción de una aplicación simple de procesamiento de imágenes. La aplicación debe ser escrita con el lenguaje de programación C sobre un sistema operativo Linux.

II. Objetivos Específicos

1. Conocer y usar las funcionalidades de `getopt()` como método de recepción de parámetros de entradas.
2. Construir funciones de lectura y escritura de archivos binarios usando `open()`, `read()`, y `write()`.
3. Construir funciones de procesamiento de imágenes.
4. Practicar técnicas de documentación de programas.
5. Conocer y practicar uso de makefile para compilación de programas.
6. Crear una aplicación concurrente utilizando librería `pthread.h`.
7. Crear hebras a través del uso de `pthread_create()`.
8. Utilizar técnicas para sincronizar hebras.

III. Conceptos

III.A. Concurrencia y Sincronización

Cuando dos o más tareas (procesos o hebras) comparten algún recurso en forma concurrente o paralela, debe existir un mecanismo que sincronice sus actividades.

De no existir una sincronización, es posible sufrir una corrupción en los recursos compartidos u obtener soluciones incorrectas.

III.B. Sección Crítica

Porción de código que se ejecuta de forma concurrente y podría generar conflicto en la consistencia de datos debido al uso de variables globales.

III.C. Mutex

Provee exclusión mutua, permitiendo que sólo una hebra a la vez ejecute la sección crítica.

III.D. Hebras

Los hilos POSIX, usualmente denominados pthreads, son un modelo de ejecución que existe independientemente de un lenguaje, además es un modelo de ejecución en paralelo. Éstos permiten que un programa controle múltiples flujos de trabajo que se superponen en el tiempo.

Para poder utilizar hebras, es necesario incluir la librería **pthread.h**. Por otro lado, dentro de la función main, se debe instanciar la variable de referencia a las hebras, para esto se utiliza el tipo de dato **pthread_t** acompañado del nombre de variable.

Luego el código que ejecutarán las hebras se debe construir en una función de la forma:

```
void * function (void * params)
```

La cual recibe parámetros del tipo void, por lo cual es necesario castear el o los parámetros de entrada para así poder utilizarlos sin problemas.

Algunas funciones para manejar las hebras son:

- `pthread_create`: función que crea una hebra. Recibe como parámetros de entrada:
  - La variable de referencia a la hebra que desea crear.
  - Los atributos de éste, los cuales no es obligación de modificar, por lo que en caso de no querer hacerlo, simplemente se deja NULL.
  - El nombre de la función que la hebra ejecutará (la cual debe cumplir con la descripción antes mencionada)
  - Por último, los parámetros de entrada (de la función que se ejecutará) previamente casteados.

  Un ejemplo sería:

  ```
  while(i < numeroHebras)
  {
      pthread_create(&hilo[i], NULL, escalaGris, (void *) &structHebra[i].id);
      i++;
  }
  ```

- `pthread_join`: función donde la hebra que la ejecuta, espera por las hebras que se ingresan por parámetro de entrada.

  Un ejemplo sería:

  ```
  while(i < numeroHebras)
  {
      pthread_join(hilo[i],NULL);
      i++;
  }
  ```

- `pthread_mutex_init`: función que inicializa un mutex, pasando por parámetros la referencia al mutex, y los atributos con que se inicializa.

  Para inicializar la estructura se utiliza:

```
while(i < numeroHebras)
{
    pthread_mutex_init(&MutexAcumulador, NULL);
    i++;
}
```

Donde `MutexAcumulador`, es una variable (de tipo `pthread_mutex_t`) que representa un mutex, el cual permitirá implementar exclusión mutua a una sección crítica que será ejecutada por varias hebras.

- **pthread_mutex_lock:** entrega una solución a la sección crítica. Ésta recibe como parámetros la variable que se desea bloquear para el resto de hebras. Un ejemplo de uso sería:

```
pthread_mutex_lock(&MutexAcumulador);
```

Cabe destacar que la primera hebra en ejecutar **pthread_mutex_lock** podrá ingresar a la sección crítica, el resto de hebras quedarán bloqueadas a la espera de que se libere el mutex.

- **pthread_mutex_unlock:** permite liberar una sección crítica. Ésta recibe como parámetro de entrada, la variable que se desea desbloquear. Su implementación es la siguiente:

```
pthread_mutex_unlock(&MutexAcumulador);
```

Por otro lado, existen casos donde se debe esperar que se completen varias tareas antes de que pueda continuar una tarea general, para solucionar esto se puede usar la sincronización de barrera . Los hilos **POSIX** especifican un objeto de sincronización llamado barrera, junto con funciones de barrera.

Las funciones crean la barrera, especificando el número de hebras que se sincronizan en ésta, luego se configuran para realizar tareas y esperan en la barrera hasta que todos las hebras alcanzan la barrera. Cuando llega el último subproceso a la barrera, todos los subprocesos reanudan la ejecución. Para hacer uso de esto, se necesitan las siguientes funciones:

- **pthread_barrier_init:** permite asignar recursos a una barrera e inicializar sus atributos. Su implementación es:

```
pthread_barrier_init(&rendezvous, NULL,NTHREADS);
```

Donde **rendezvous** es una variable de tipo pthread_barrier_t, **NULL** indica que se utilizan los atributos de barrera predeterminados (se pide usar estos atributos, no debiese ser necesario modificarlos) y **NTHREADS** es la cantidad de hebras que deben esperar en la barrera.

- **pthread_barrier_init:** permite sincronizar los hilos en una barrera especificada. El hilo de llamada bloquea hasta que el número requerido de hilos ha llamado a pthread_barrier_wait() especificando la barrera. Un ejemplo de su uso:

```
funcion1();
pthread_barrier_wait(&rendezvous);
funcion2();
```

Donde **funcion1** y **funcion2** son funciones que deben ejecutar las hebras pero de manera sincronizada, es decir, no se debe ejecutar funcion2 hasta que las n hebras (indicadas en barrier_init) ejecuten la funcion 1.

- **pthread_barrier_destroy:** cuando ya no se necesita una barrera, se debe destruir. SU implementación es:

```
pthread_barrier_destroy(&rendezvous);
```

IV. Definición de etapas

La aplicación consiste en un *pipeline* de procesamiento de imágenes astronómicas. Cada imagen pasará por tres etapas de procesamiento tal que al final del *pipeline* se clasifique la imagen como satisfaciendo o no alguna condición a definir. El programa procesará varias imágenes, una a la vez.

Las etapas del *pipeline* son:

1. Lectura de imagen RGB
2. Conversión a imagen en escala de grises
3. Binarización de imagen
4. Análisis de propiedad
5. Escritura de resultados

IV.A. Lectura de imágenes

Las imágenes estarán almacendas en binario con formato bmp. Habrá n imágenes y sus nombres tendrán el mismo prefijo seguido de un número correlativo. Es decir, los nombres serán imagen_1, imagen_2, ... , imagen_n.

Para este laboratorio se leerá una nueva imagen cuando la anterior haya finalizado el *pipeline* completo. Es decir, en el *pipeline* sólo habrá una imagen a la vez.

IV.B. Conversión de RGB a escala de grises

En una imagen RGB, cada pixel de la imagen está representado por tres números, que representan el componente de color rojo (Red), el de color verde (Green) y el de color azul (Blue). Para convertir una imagen a escala de grises, se utiliza la siguiente ecuación de luminiscencia:

$$Y = R * 0.3 + G * 0.59 + B * 0.11 \tag{1}$$

donde R, G, y B son los componentes rojos, verdes y azul, respectivamente.

IV.C. Binarización de una imagen

Para binarizar la imagen basta con definir un umbral, el cual define cuáles píxeles deben ser transformados a blanco y cuáles a negro, de la siguiente manera. Para cada pixel de la imagen, hacer

```
si el pixel > UMBRAL
    pixel = 1
sino
    pixel = 0
```

Los valores 0 y 1 no son representados por un bit sino son valores enteros (`int`) Al aplicar el algoritmo anterior, obtendremos la imagen binarizada.

IV.D. Clasificación

Se debe crear una función que concluya si la imagen es *nearly black* (casi negra). Esta característica es típica de muchas imágenes astronómicas donde una pequeña fuente puntual de luz está rodeada de vacio u oscuridad. Entonces, si el porcentaje de píxeles negros es mayor o igual a un cierto umbral la imagen es clasificada como *nearly black*.

IV.E. Escribir resultados

El programa debe imprimir por pantalla la clasificación final de cada imagen y escribir en disco la imagen binarizada resultante.

V. Enunciado

Se pide construir una aplicación que implemente las etapas previamente descritas mediante el uso de hebras. Se deberán crear n hebras, en donde la distribución de trabajo será la siguiente:

- La primera hebra creada, debe ir a leer la imagen correspondiente.
- Las n hebras deben ejecutar la siguiente etapa (pasar a gris). Para lograr sincronizar las hebras, es decir, que todas las hebras comiencen a grisear, deben esperar a que la primera hebra termine su tarea, para conseguir esto se pide hacer uso de de `pthread_barrier_init`, `pthread_barrier_wait` y `pthread_barrier_destroy`.
- Se pide hacer el paso anterior para la etapa de binarizar.
- Para la etapa nearly black, se pide que las n hebras comiencen la ejecución una vez que se haya binarizado la imagen. Cabe destacar que cada hebra debe determinar por fila la cantidad de pixeles negros, una vez determinado el valor, debe escribir en una variable global su respectivo resultado.

  Como pista, deberá usar algún método de sincronización para asegurar la consistencia de datos en la variable que representa la acumulación de pixeles negros. Como método de sincronización puede usar la que estime conveniente.
- Finalmente, la hebra padre (la hebra main) deberá hacer la etapa E, es decir, debe escribir la imagen binarizada y escribir por consola la conclusión (indicar si es nearly black).
- Repetir los anteriores pasos para n imágenes.

Por último, el programa debe recibir la cantidad de imágenes a leer, cantidad de hebras a crear, el valor del umbral de binarización, el valor del umbral de negrura y una bandera que indica si se desea mostrar o no la conclusión final hecha por la tercera función. Por lo tanto, el formato getopt es:

- -c: cantidad de imágenes.
- -h: cantidad de hebras.
- -u: UMBRAL para binarizar la imagen.
- -n: UMBRAL para clasificación.
- -b: bandera que indica si se deben mostrar los resultados por pantalla, es decir, la conclusión obtenida al leer la imagen binarizada.

Por ejemplo, la salida por pantalla al analizar 3 imágenes sería lo siguiente:

```
$ ./pipeline -c 3 -h 7 -u 50 -n 80 -b -pthread
|     image    |     nearly black    |
|--------------|---------------------|
|   imagen_1   |         yes         |
|   imagen_2   |         no          |
|   imagen_3   |         no          |
```

Donde sólo la imagen_1 fue clasificada como *nearly black*.
Cabe destacar que la forma de compilar un código que hace uso de la librería phtread es la siguiente:

```
gcc programa.c -o pipeline -pthread
```

VI. Otros conceptos

Como ayuda, la manera de leer una imagen es la siguiente:

- Abrir el archivo con el uso de open(). Recuerde que este archivo debe contener la matriz (imagen).
- Leer la imagen con read().

Con los pasos previamente descritos, se puede leer la imagen y manipularla de tal forma que se pueda crear la matriz. Recuerde que la imagen tiene una estructura para poder obtener el valor de los pixeles, por lo que se pide investigar al respecto.

Para el caso de escribir la imagen resultante:

- Se debe escribir con write().
- Finalmente se debe cerrar el archivo con close().

VII. Entregables

Debe entregarse un archivo comprimido que contenga al menos los siguientes archivos:

1. *Makefile*: archivo para make que compile los programas.
2. dos o mas archivos con el proyecto (*.c, *.h)

El archivo comprimido debe llamarse: RUTESTUDIANTE1_RUTESTUDIANTE2.
Ejemplo: 19689333k_189225326.zip

VIII. Fecha de Entrega

Jueves 7 de Junio hasta las 23:55 hrs.